Behavioral Pattern: Strategy

Kevin Dockx
Architect

@KevinDockx https://www.kevindockx.com

Coming Up

Describing the strategy pattern

- Exporting an order to a certain format

Structure of the template method pattern

Alternative approach

Coming Up

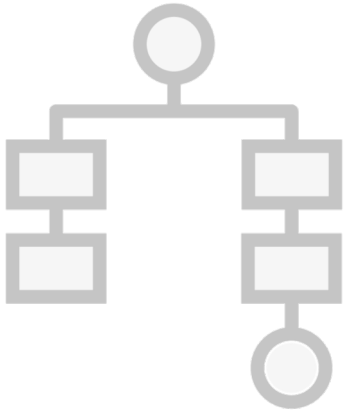
Use cases for this pattern

Pattern consequences

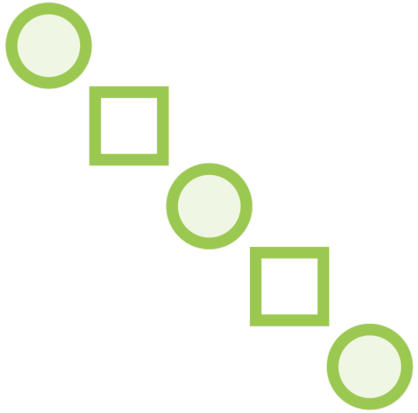
Related patterns

Describing the Strategy Pattern

Creational



Structural



Behavioral

Strategy

The intent of this pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

```
public class Order
{
     public void Export()
     {
          // code to export to CSV...
     }
}
```

Describing the Strategy Pattern

```
public class Order
{
     public void Export()
     {
          // code to export to CSV...
     }
}
```

Describing the Strategy Pattern

```
public enum Format {
     CSV,
     Json
}

public class Order
{
     public void Export(Format exportFormat)
     {
          // code to export to CSV or Json...
     }
}
```

Describing the Strategy Pattern

```
public enum Format {
     CSV,
     Json
}

public class Order
{
     public void Export(Format exportFormat)
     {
          // code to export to CSV or Json...
     }
}
```

Describing the Strategy Pattern

```
public enum Format {
     CSV,
     Json
}

public class Order
{
     public void Export(Format exportFormat)
     {
          // code to export to CSV or Json...
     }
}
```

Describing the Strategy Pattern

```
public enum Format {
     CSV,
     Json
}

public class Order
{
     public void Export(Format exportFormat)
     {
          // code to export to CSV or Json...
     }
}
```

Describing the Strategy Pattern

```
public enum Format {
     CSV,
     Json,
     XML
}

public class Order
{
     public void Export(Format exportFormat)
     {
          // code to export to CSV or Json or XML...
     }
}
```

Describing the Strategy Pattern

```
public enum Format {
     CSV,
     Json,
     XML
}

public class Order
{
     public void Export(Format exportFormat)
     {
          // code to export to CSV or Json or XML...
     }
}
```

Describing the Strategy Pattern

Order class is becoming complex
Changing / adding export logic requires changing the Order class

```
public class Order
{
    private JsonExportService _jsonExportService = new();
    private XMLExportService _xmlExportService = new();
    private CSVExportService _csvExportService = new();

    public void Export(Format exportFormat)
    {
        // export using service instances, depending on exportFormat
    }
}
```

Describing the Strategy Pattern

```
public class Order
{
    private JsonExportService _jsonExportService = new();
    private XMLExportService _xmlExportService = new();
    private CSVExportService _csvExportService = new();

    public void Export(Format exportFormat)
    {
        // export using service instances, depending on exportFormat
    }
}
```

Describing the Strategy Pattern

```
public class Order
{
    private JsonExportService _jsonExportService = new();
    private XMLExportService _xmlExportService = new();
    private CSVExportService _csvExportService = new();

    public void Export(Format exportFormat)
    {
        // export using service instances, depending on exportFormat
    }
}
```

Describing the Strategy Pattern

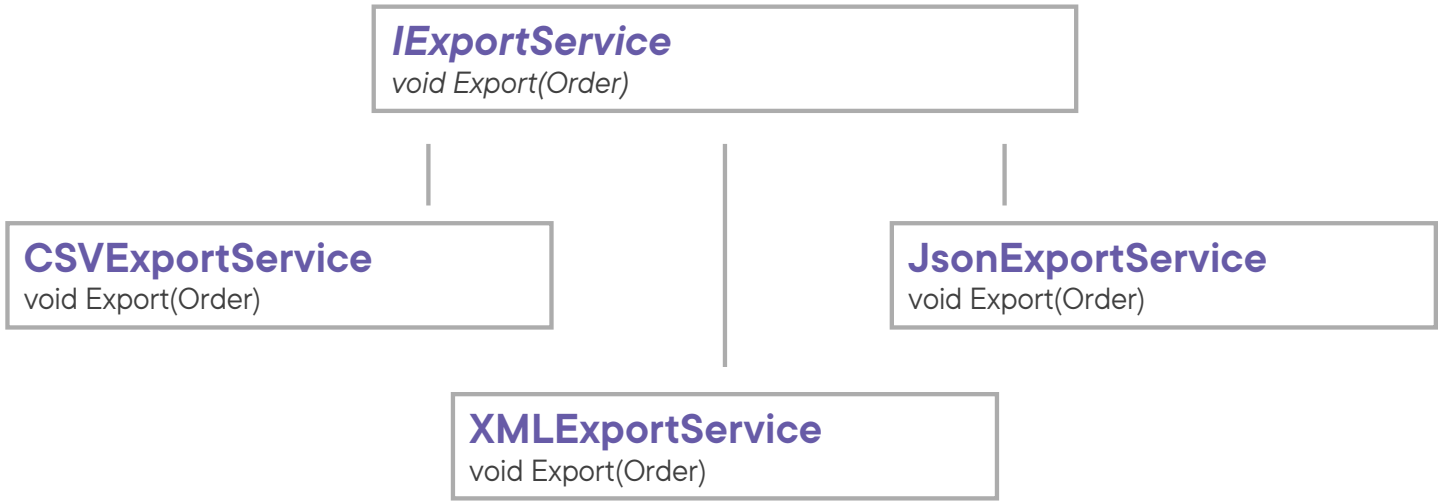
```
public class Order
{
    private JsonExportService _jsonExportService = new();
    private XMLExportService _xmlExportService = new();
    private CSVExportService _csvExportService = new();

    public void Export(Format exportFormat)
    {
        // export using service instances, depending on exportFormat
    }
}
```

Describing the Strategy Pattern

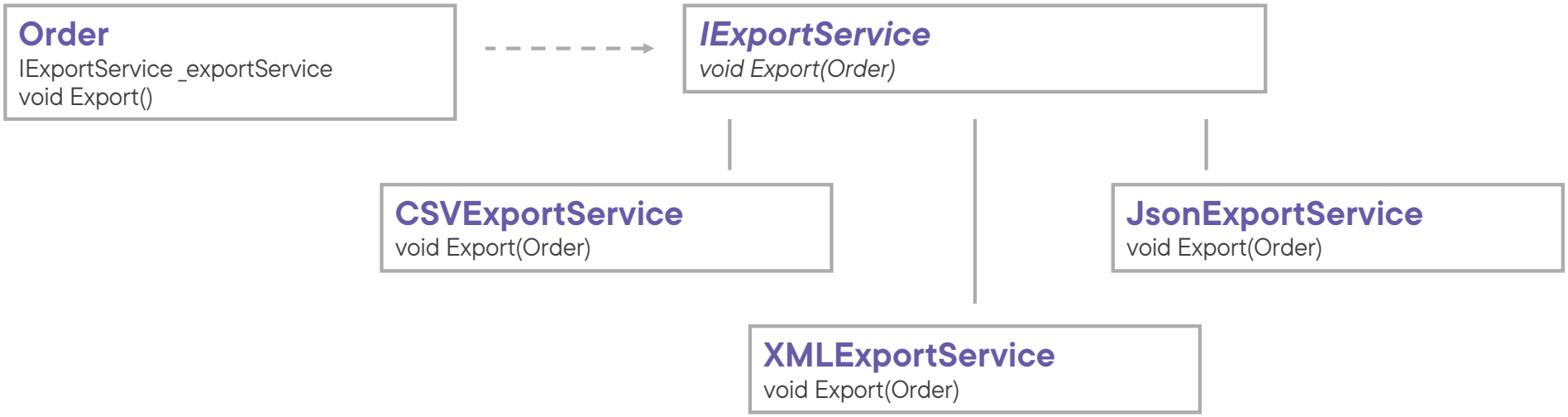
Order class is now responsible for export service lifetime management
Tight coupling is introduced

Describing the Strategy Pattern

Describing the Strategy Pattern



Describing the Strategy Pattern

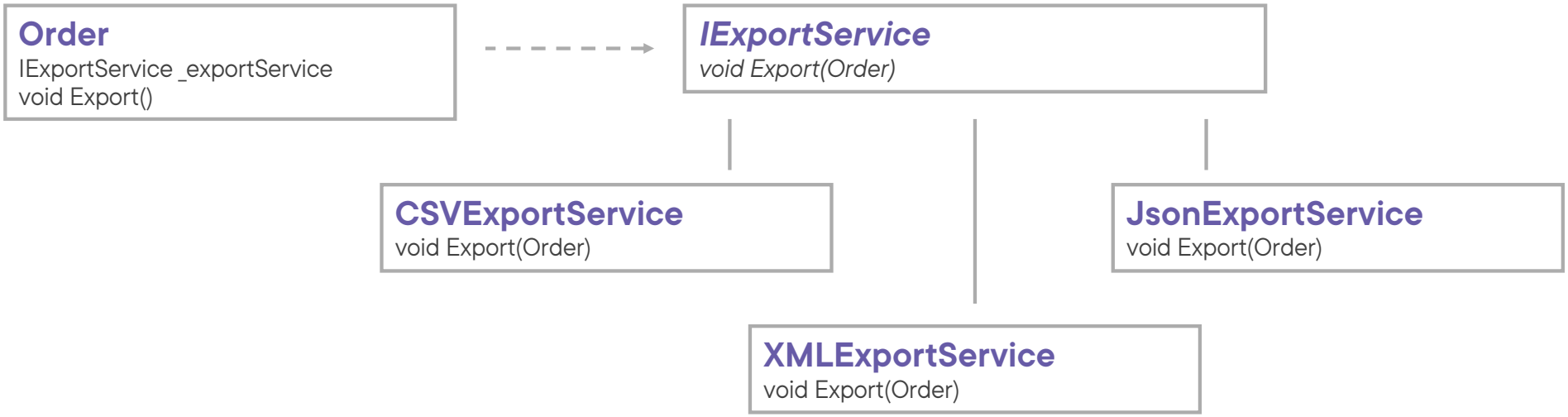


Describing the Strategy Pattern

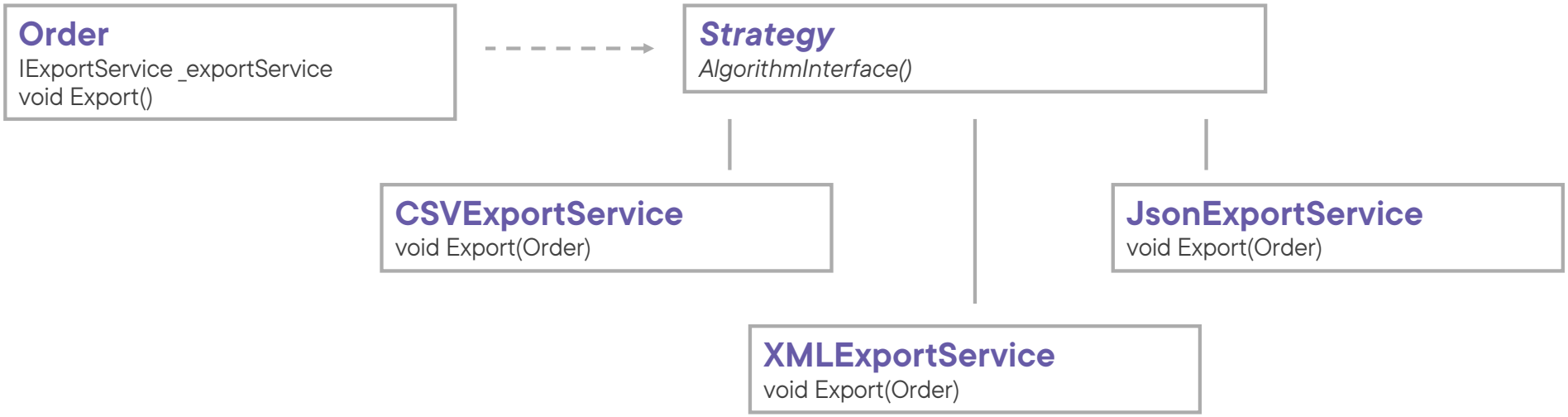
Client passes implementation to `Order`

- `Order` only knows about the interface
- Easy to add new implementations or modify existing implementations

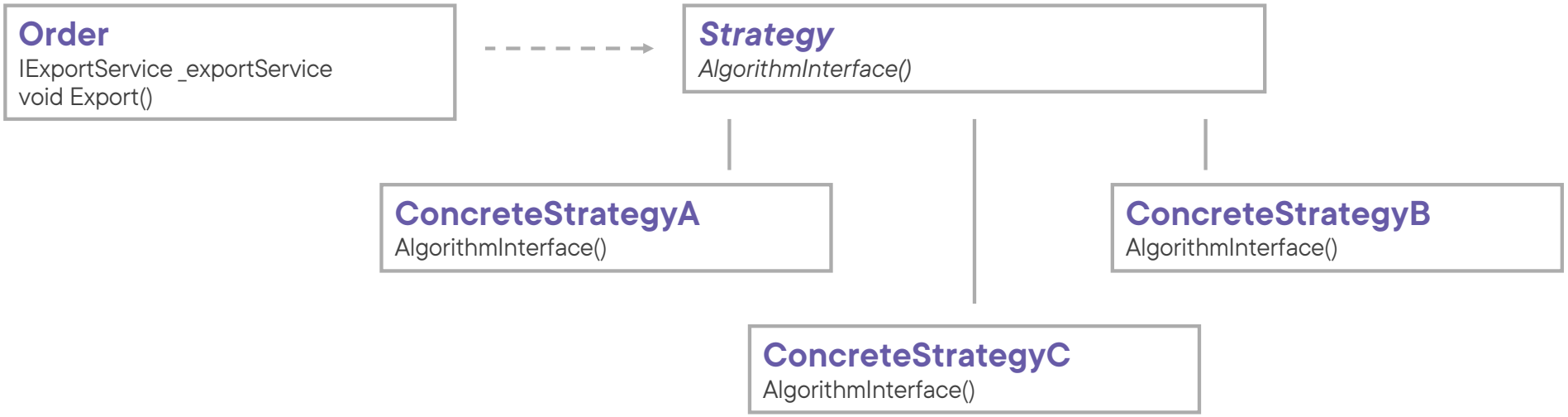
Structure of the Strategy Pattern



Structure of the Strategy Pattern

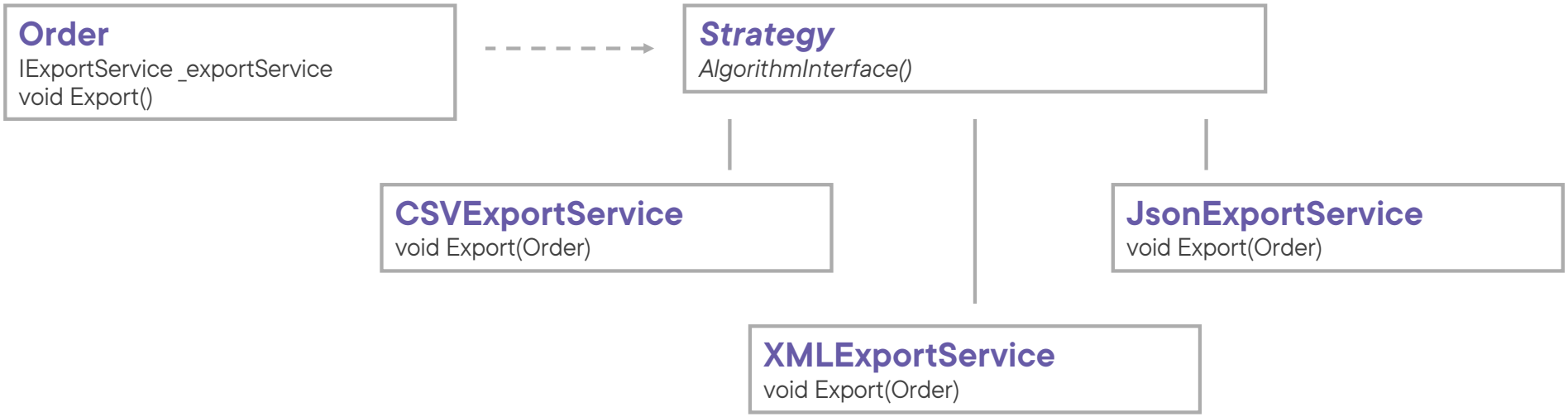


Structure of the Strategy Pattern

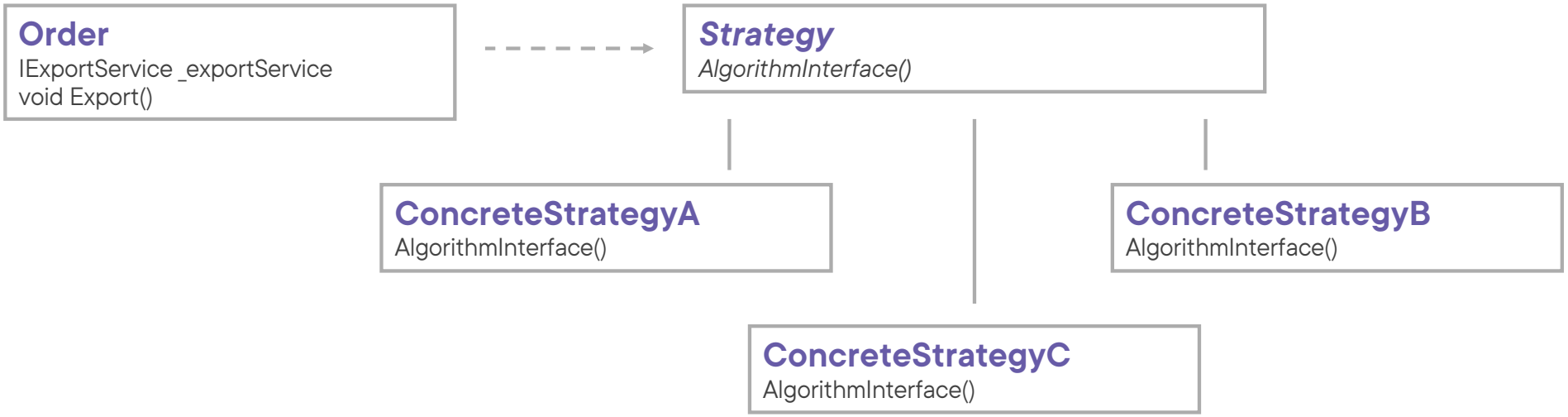


Strategy declares an interface common to all supported algorithms. **Context** uses it to call the algorithm defined a by **ConcreteStrategy**.

Structure of the Strategy Pattern

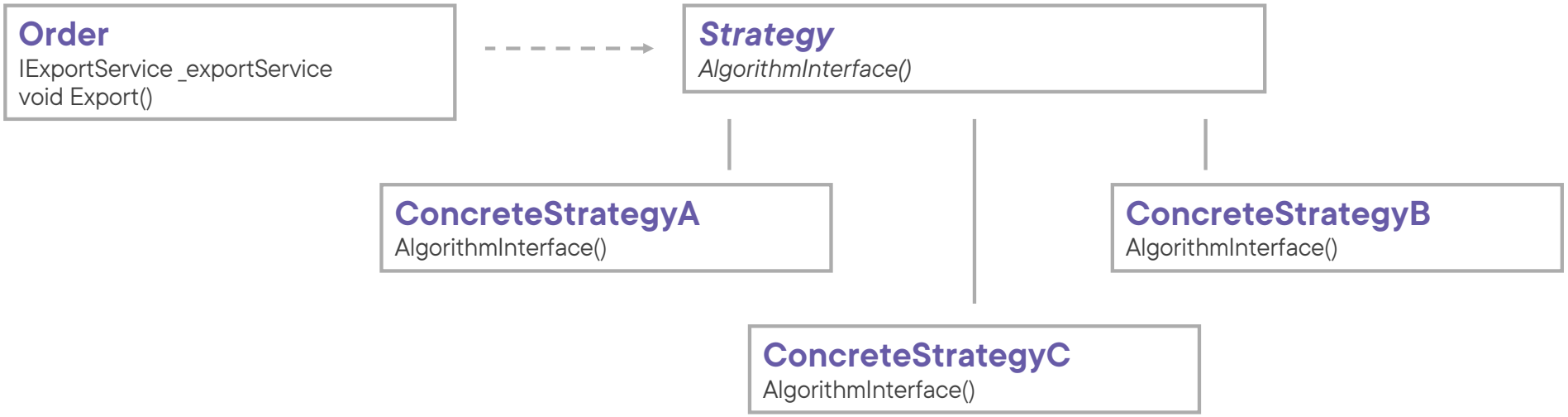


Structure of the Strategy Pattern

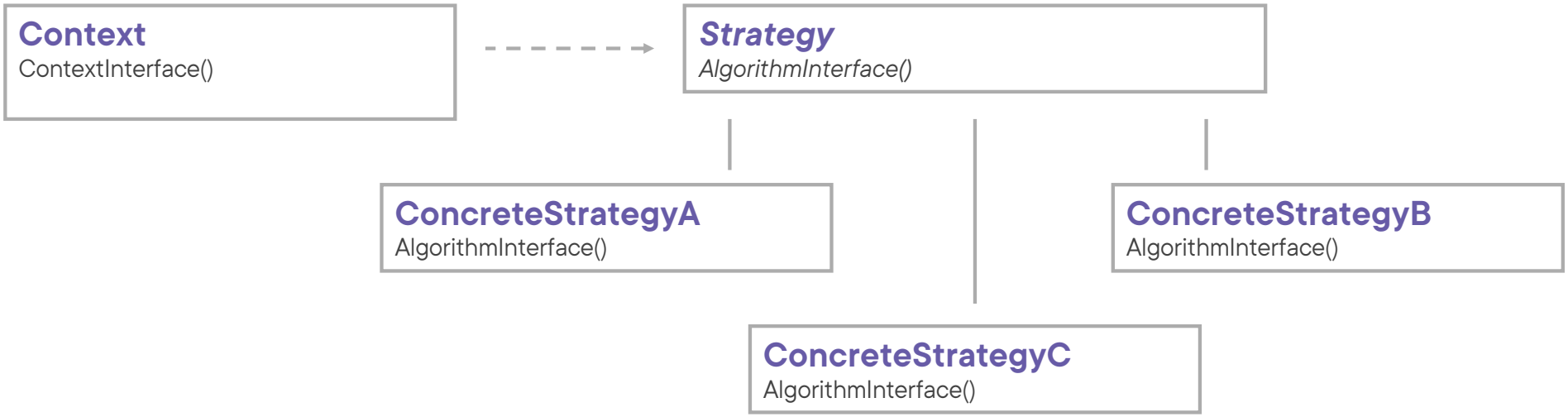


ConcreteStrategy implements the algorithm using the **Strategy** interface.

Structure of the Strategy Pattern

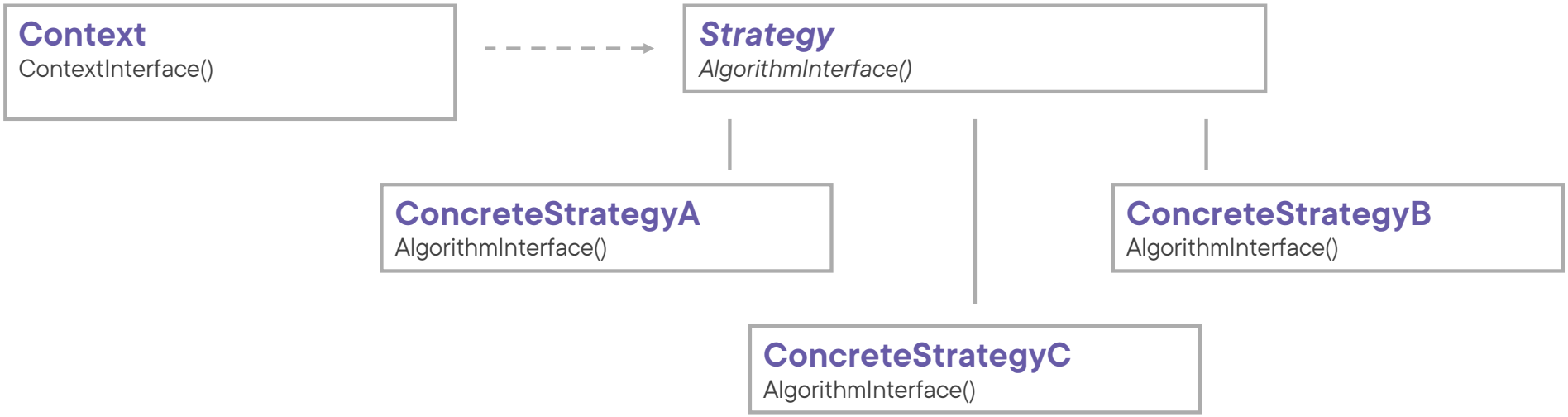


Structure of the Strategy Pattern

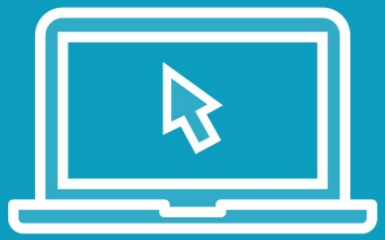


Context is configured with a **ConcreteStrategy** object and maintains a reference to a **Strategy** object

Structure of the Strategy Pattern

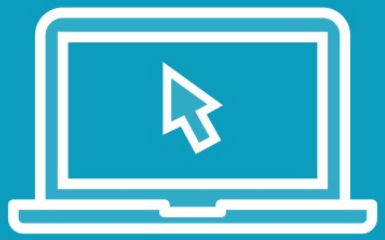


Demo



Implementing the strategy pattern

Demo



Implementing a strategy pattern variation with a method parameter

Use Cases for the Strategy Pattern

When many related classes differ only in their behavior

When you need different variants of an algorithm which you want to be able to switch at runtime

When your algorithm uses data, code or dependencies that the clients shouldn't know about

When a class defines many different behaviors which appear as a bunch of conditional statements in its method

Pattern Consequences

It offers an alternative to subclassing your context

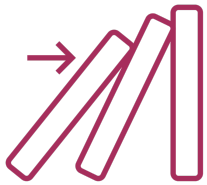
New strategies can be introduced without having to change the context: open/closed principle

It eliminates conditional statements

It can provide a choice of implementations with the same behavior

Pattern Consequences

If the client injects the strategy, it must be aware of how strategies differ



There's overhead in communication between the strategy and the context

Additional objects are introduced, which increases complexity

Related Patterns

Flyweight
Strategy objects make good flyweights

Bridge
Also based on composition, but solves a different problem

State
Also based on composition, but solves a different problem

Template method
Template method allows varying part of an algorithm through inheritance: a static approach. Strategy allows behavior to be switched at runtime, via composition: a dynamic approach.

Summary

Intent of the strategy pattern:

- To define a family of algorithms, encapsulate each one, and make them interchangeable

Common variation: concrete strategy injected via method parameter

Up Next:
Behavioral Pattern: Command