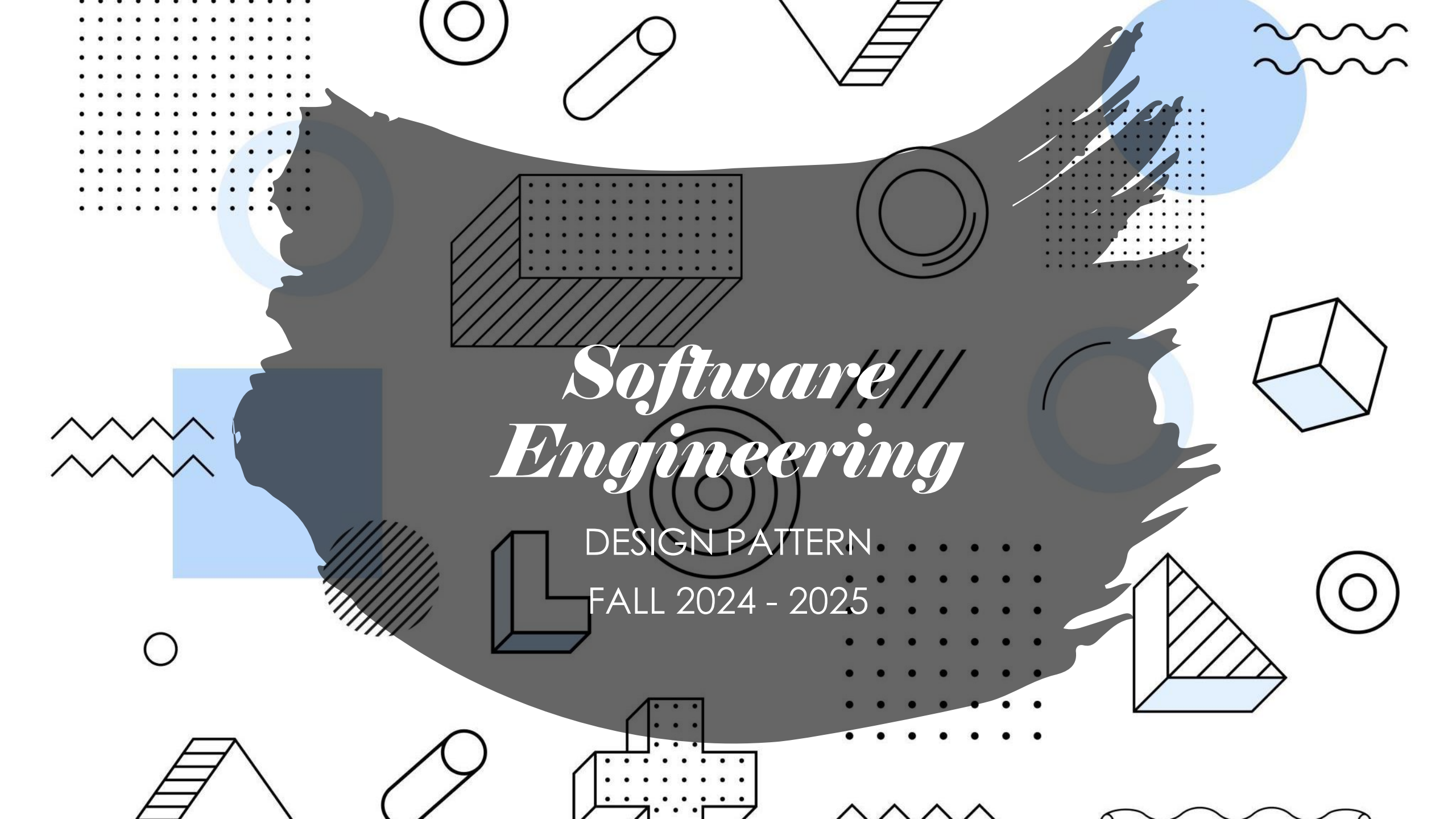

Software
Engineering
DESIGN PATTERN
FALL 2024 - 2025

Design Pattern

- **Design patterns are ready-made solutions to common problems in software design.**
- **They act like pre-made blueprints that can be customized to solve similar design issues in your code.**
- **You can't directly copy and paste a design pattern into your code like you would with a function or a library.**
- **Design patterns are more like general concepts or ideas to solve a specific problem, not specific lines of code.**
- **You need to understand the pattern and adapt it to fit the needs of your own program.**

Why use Design pattern?

- **Design patterns help save time and effort by providing proven solutions to recurring design challenges.**
- **They improve code reusability, maintainability, and overall code quality.**
- **Design patterns encourage best practices and standard approaches to solving problems in software development.**

What does the pattern consist of?

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Classification of patterns

- The architectural patterns are most universal and high-level patterns.
- They can be implemented in virtually any programming language.
- Architectural patterns guide the overall structure of an entire application.
- Unlike other patterns, they focus on high-level design and architecture.

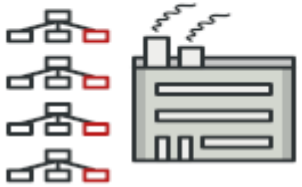
Classification of patterns

- **All patterns can be categorized by their intent, or purpose:**
  - **Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code.**
  - **Structural patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.**
  - **Behavioral patterns take care of effective communication and the assignment of responsibilities between objects.**

Creational Design Patterns

Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

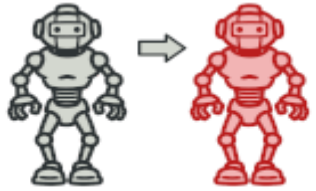


Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.

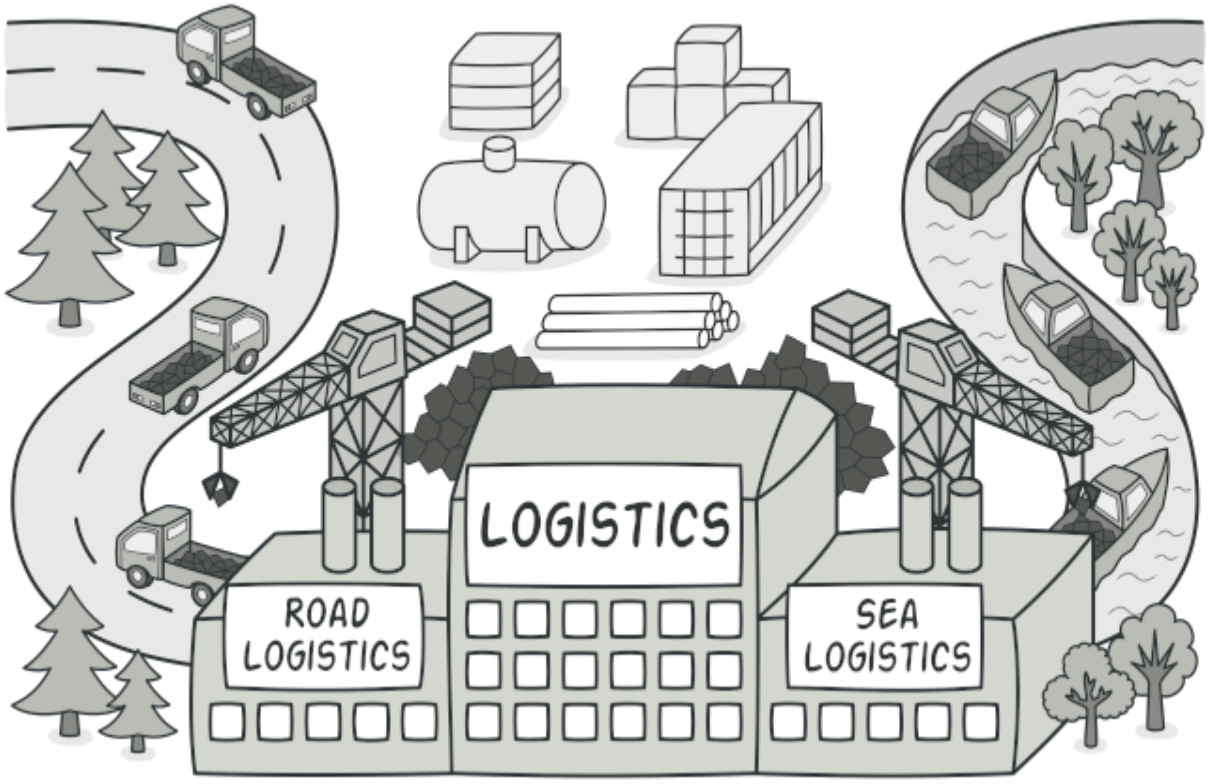
Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Factory Method

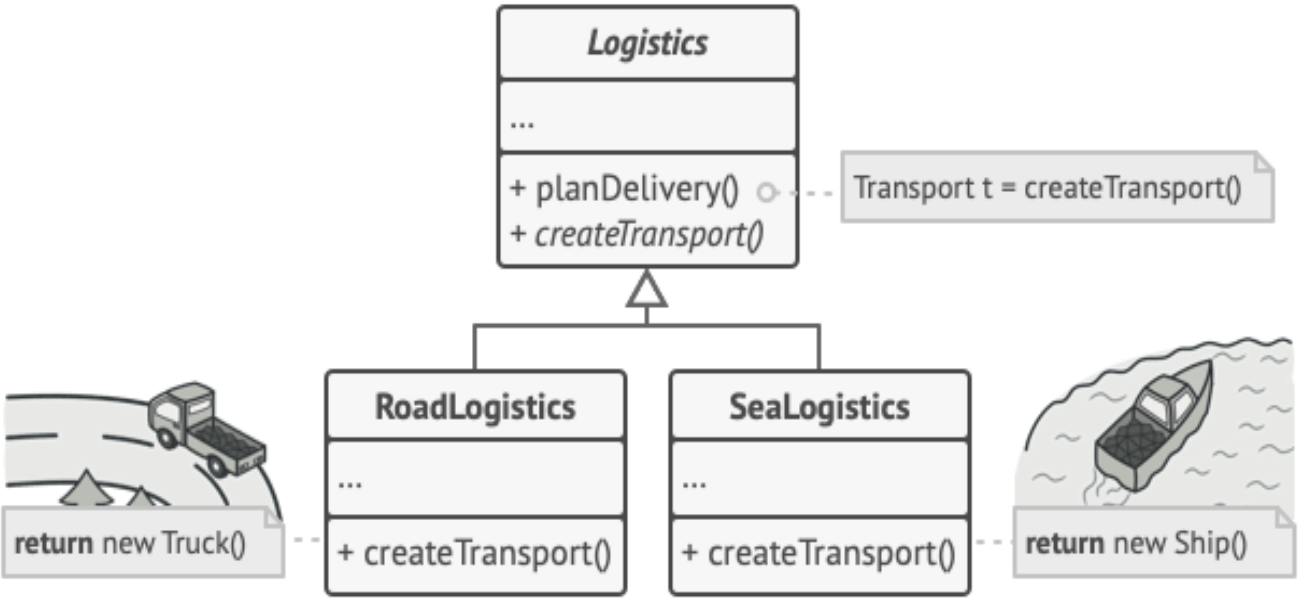
- Imagine that you're creating *a logistics management application*. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.

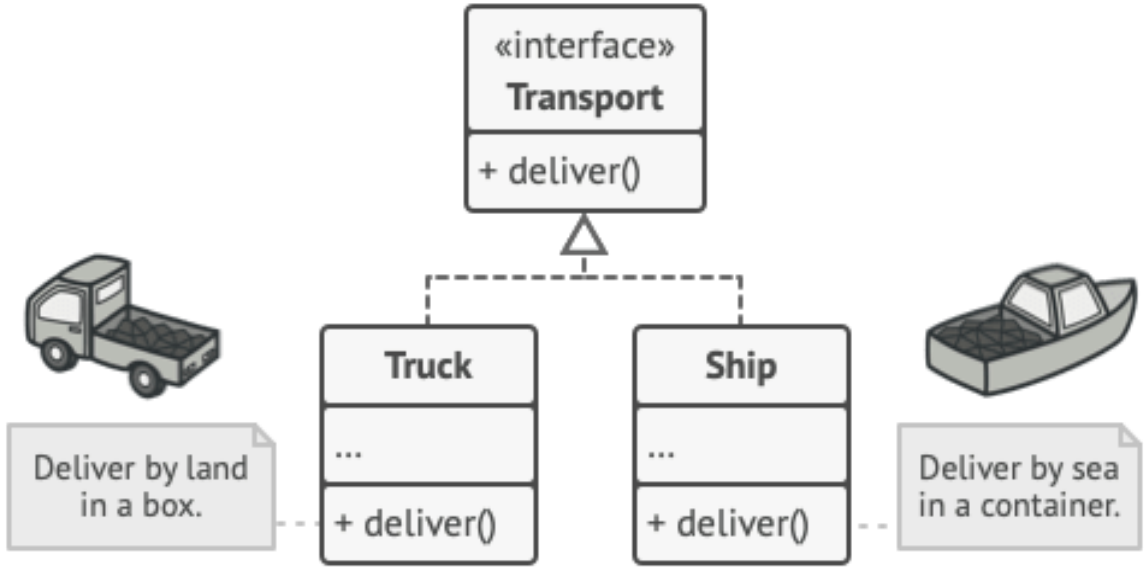
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.



Factory Method

- The **Factory Method pattern** suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. *Don't worry: the objects are still created via the new operator*, but it's being called from within the factory method. Objects returned by a factory method are often referred to as products.





Logistics Management Application Example Explanation

The Factory Method design pattern is *a creational pattern* that provides an interface for creating objects in a subclass. It allows a class to delegate the responsibility of instantiating objects to its subclasses. Here's a breakdown of the Factory Method pattern in the context of a logistics system.

1. **Logistics (Abstract Class):**
   - **This class defines a method planDelivering() that calls the createTransport() method.**
   - **It serves as the base class for different types of logistics.**
2. **Transport (Interface):This interface declares the method deliver(), which will be implemented by concrete transport classes.**
3. **Concrete Transport Classes (Truck and Ship):**
   - **Truck: Implements the deliver() method to define how delivery is executed by land.**
   - **Ship: Implements the deliver() method for sea delivery.**
4. **Subclasses of Logistics:**
   - **RoadLogistics: Overrides the createTransport() method to return a new Truck.**
   - **SeaLogistics: Overrides the createTransport() method to return a new Ship.**

Interface

Transport

Deliver()

Before

Class implement interface

Truck

```
Deliver(){
}
```

Class implement interface

Ship

```
Deliver(){
}
```

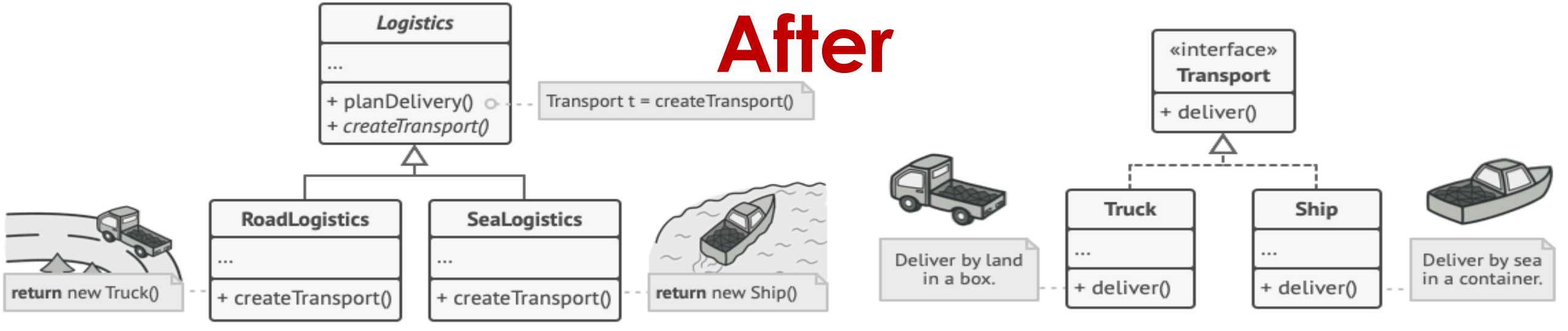
Public Class

Logistics

```
Plandelivery 🚚 (String DeliveryBy)
{
Transport transport;
If(DeliveryBy ==‘Sea’)
   transport=new Ship()
else if(DeliveryBy ==‘land’)
   transport=new Truck()
transport.Deliver();
}
```

How can you add bus Transport?

After



Here's an example of the Open/Closed Principle (OCP) applied to a logistics scenario, showing both the "before" and "after" implementations.

Before Applying the Factory Method

In this version, the `Logistics` class requires modification to handle new transport types.

python ▶ Run Copy

```python
class Transport:
    pass

class Truck(Transport):
    def deliver(self):
        return "Deliver by land in a truck."

class Ship(Transport):
    def deliver(self):
        return "Deliver by sea in a ship."

class Logistics:
    def plan_delivery(self, transport_type):
        if transport_type == "Truck":
            transport = Truck()
        elif transport_type == "Ship":
            transport = Ship()
        else:
            raise ValueError("Transport type not supported")
        return transport.deliver()

# Usage
logistics = Logistics()
print(logistics.plan_delivery("Truck"))  # Output: Deliver by land in a truck.
print(logistics.plan_delivery("Ship"))    # Output: Deliver by sea in a ship.
```

Summary

- In the **before** version, adding a new transport type required changes in the `Logistics` class.
- In the **after** version, new transport types can be added simply by creating a new class and updating the `TransportFactory`, aligning with the Open/Closed Principle.

After Applying the Factory Method

In this refactored version, new transport types can be added without modifying the `Logistics` class.

python ▶ Run Copy

```python
class Transport:
    def deliver(self):
        raise NotImplementedError("Subclasses must implement this method")

class Truck(Transport):
    def deliver(self):
        return "Deliver by land in a truck."

class Ship(Transport):
    def deliver(self):
        return "Deliver by sea in a ship."

class Logistics:
    def plan_delivery(self, transport: Transport):
        return transport.deliver()

class TransportFactory:
    @staticmethod
    def create_transport(transport_type):
        if transport_type == "Truck":
            return Truck()
        elif transport_type == "Ship":
            return Ship()
        else:
            raise ValueError("Transport type not supported")

# Usage
factory = TransportFactory()
truck = factory.create_transport("Truck")
ship = factory.create_transport("Ship")

logistics = Logistics()
print(logistics.plan_delivery(truck))  # Output: Deliver by land in a truck.
print(logistics.plan_delivery(ship))    # Output: Deliver by sea in a ship.

# Adding a new transport type (e.g., Airplane)
class Airplane(Transport):
    def deliver(self):
        return "Deliver by air in an airplane."

# Now we can add Airplane without modifying Logistics
airplane = factory.create_transport("Airplane")
print(logistics.plan_delivery(airplane))  # Output: Deliver by air in an airplane.
```

https://www.pythontutorial.net/python-oop/python-open-closed-principle/

Explanation

The Open/Closed Principle (OCP) states that software entities should be open for extension but closed for modification. This principle is crucial for maintaining code that is easy to extend and maintain without altering existing code.

An example illustrating the OCP before and after applying the Factory Method design pattern. *Before* Applying the Factory Method Consider a simple scenario where we have a Shape class and a ShapeAreaCalculator that calculates the area of different shapes. Initially, the ShapeAreaCalculator might look like this:

Before Applying the Factory Method

In this initial implementation, the `AreaCalculator` class must be modified every time a new shape is added.

python ▶ Run Copy

```python
class Shape:
    pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

class AreaCalculator:
    def calculate_area(self, shape):
        if isinstance(shape, Circle):
            return 3.14 * shape.radius * shape.radius
        elif isinstance(shape, Rectangle):
            return shape.width * shape.height
        else:
            raise ValueError("Shape not supported")

# Usage
circle = Circle(5)
rectangle = Rectangle(4, 6)
calculator = AreaCalculator()

print(calculator.calculate_area(circle))    # Output: 78.5
print(calculator.calculate_area(rectangle)) # Output: 24
```

After Applying the Factory Method

In this refactored implementation, new shapes can be added without modifying the `AreaCalculator`.

python ▶ Run Copy

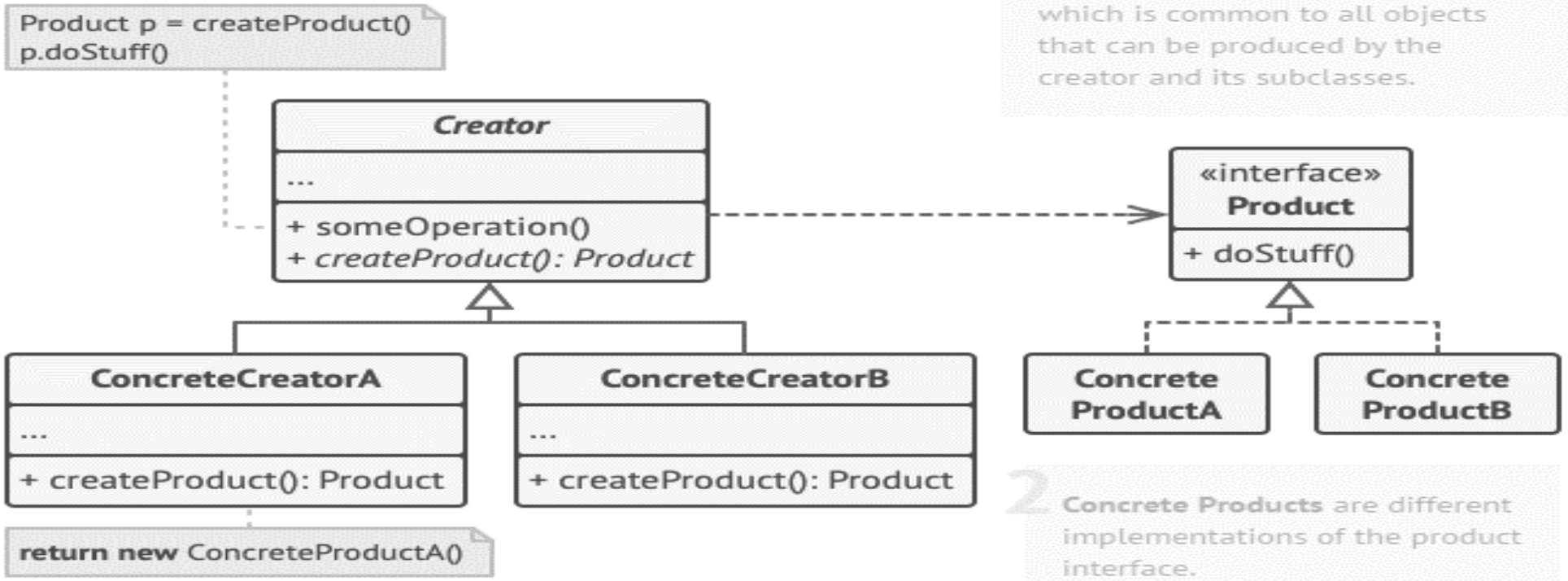
```python
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses must implement this method")

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height
```

```python
class ShapeFactory:
    @staticmethod
    def create_shape(shape_type, *args):
        if shape_type == "Circle":
            return Circle(*args)
        elif shape_type == "Rectangle":
            return Rectangle(*args)
        elif shape_type == "Triangle":
            return Triangle(*args)
        else:
            raise ValueError("Shape type not supported")

class AreaCalculator:
    def calculate_area(self, shape: Shape):
        return shape.area()

# Usage
factory = ShapeFactory()
circle = factory.create_shape("Circle", 5)
rectangle = factory.create_shape("Rectangle", 4, 6)
triangle = factory.create_shape("Triangle", 4, 8)

calculator = AreaCalculator()

print(calculator.calculate_area(circle))    # Output: 78.5
print(calculator.calculate_area(rectangle))  # Output: 24
print(calculator.calculate_area(triangle))   # Output: 16
```

Summary

- In the **before** version, adding a new shape required modifying the `AreaCalculator`.
- In the **after** version, new shapes can be added simply by extending the `Shape` class and updating the `ShapeFactory`, adhering to the Open/Closed Principle.

Factory Method



1 The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

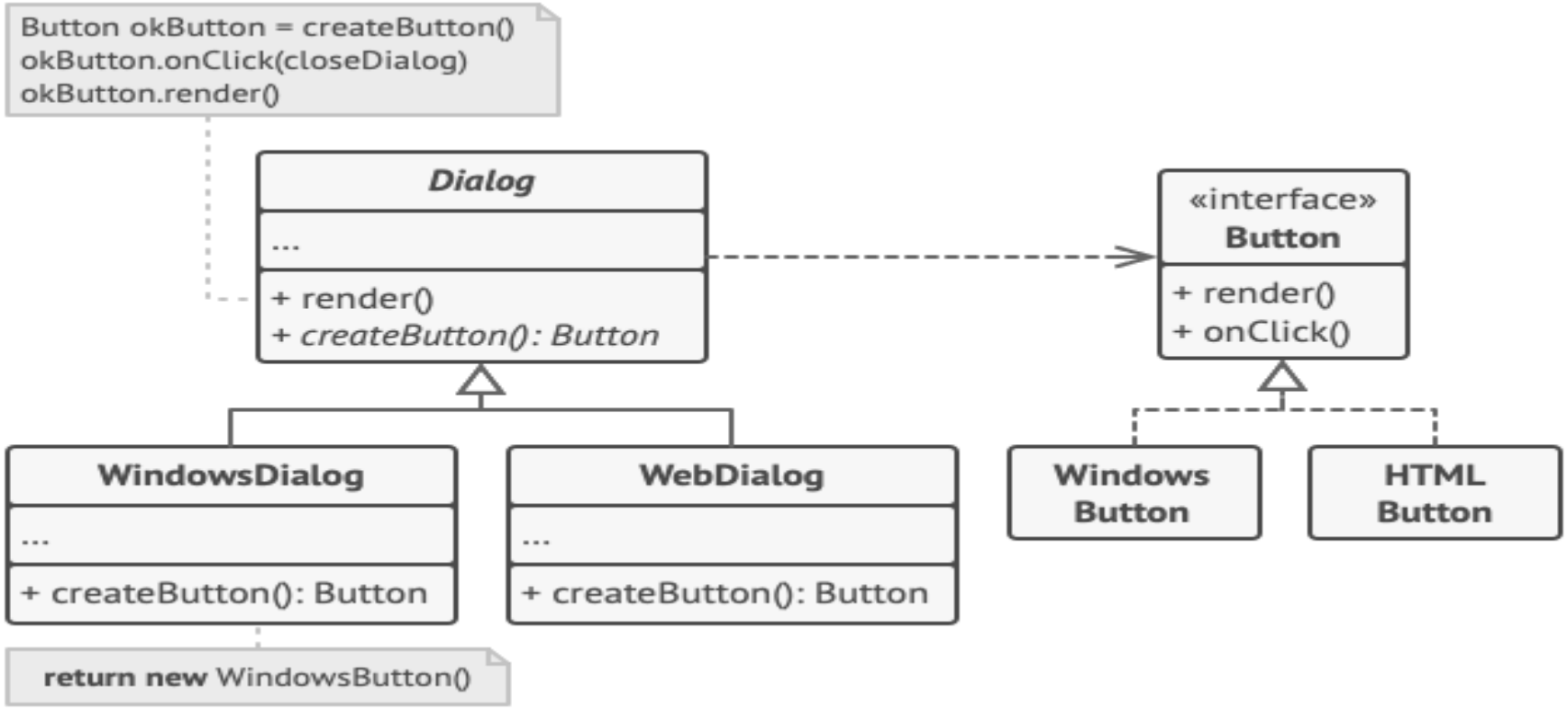
2 **Concrete Products** are different implementations of the product interface.

4 **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

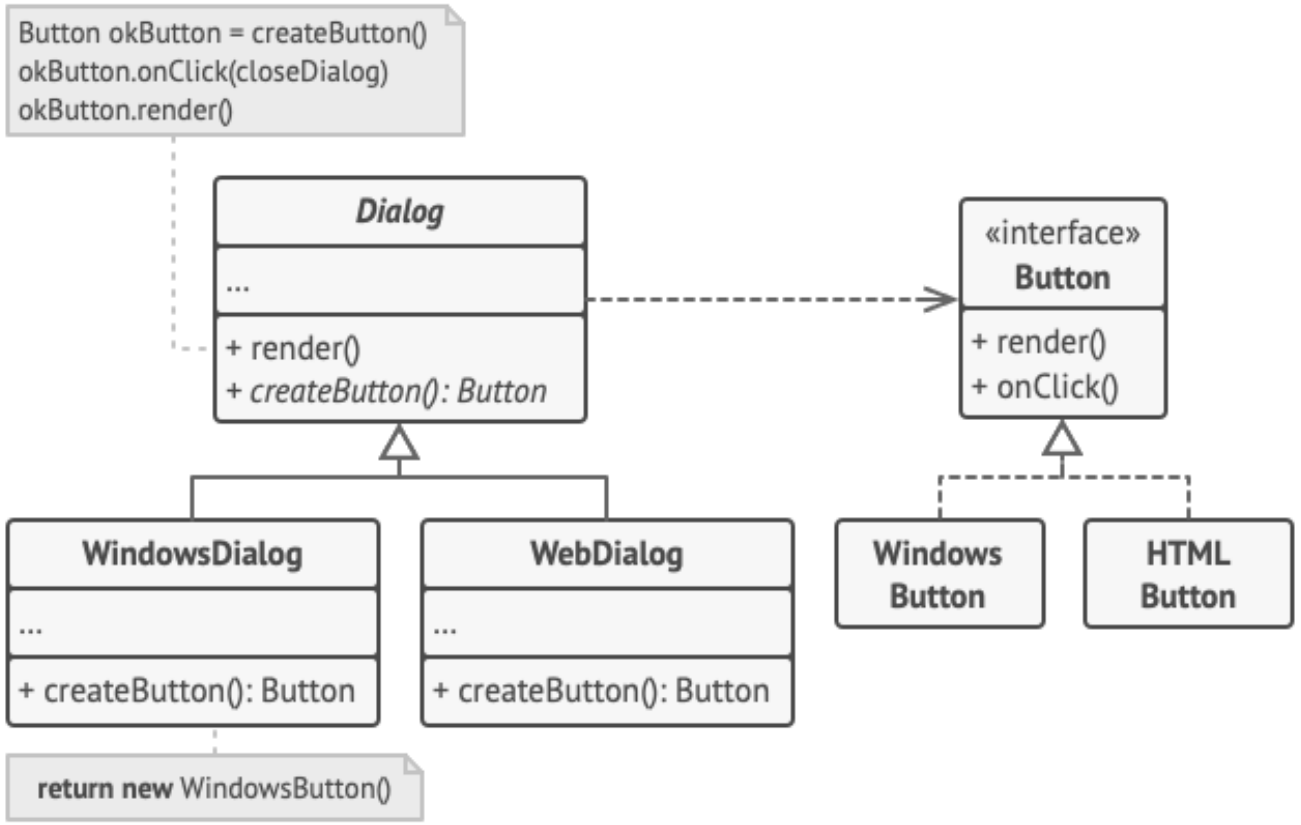
Factory Method

- This example illustrates how the Factory Method can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes.



Factory Method

- The base Dialog class uses different UI elements to render its window. Under various operating systems, these elements may look a little bit different, but they should still behave consistently. A button in Windows is still a button in Linux.
- When the factory method comes into play, you don't need to rewrite the logic of the Dialog class for each operating system. If we declare a factory method that produces buttons inside the base Dialog class, we can later create a subclass that returns Windows-styled buttons from the factory method. The subclass then inherits most of the code from the base class, but, thanks to the factory method, can render Windows-looking buttons on the screen.
- For this pattern to work, the base Dialog class must work with abstract buttons: a base class or an interface that all concrete buttons follow. This way the code within Dialog remains functional, whichever type of buttons it works with.



Window

Linux

Here's an explanation of the Open/Closed Principle (OCP) illustrated with a dialog and button scenario, detailing both the "before" and "after" implementations.

Before Applying the Factory Method

In the initial implementation, the `Dialog` class directly creates specific button types. This design violates the OCP because adding a new button type requires modifying the `Dialog` class.

plaintext Copy

```
class Button:
    def on_click(self):
        pass

class WindowsButton(Button):
    def on_click(self):
        return "Windows Button Clicked"

class HTMLButton(Button):
    def on_click(self):
        return "HTML Button Clicked"

class Dialog:
    def create_button(self):
        return WindowsButton()  # Directly creates a Windows Button

    def render(self):
        button = self.create_button()
        return button.on_click()

# Usage
dialog = Dialog()
print(dialog.render())  # Output: Windows Button Clicked
```

Issues with the Before Implementation

- **Tight Coupling**: The `Dialog` class is tightly coupled to the `WindowsButton` class.
- **Modification Required**: To add a new button type (e.g., `HTMLButton`), you must modify the `Dialog` class, which can lead to bugs and increased maintenance efforts.

After Applying the Factory Method

In the refactored implementation, the **`Dialog`** class is abstracted, allowing subclasses to create specific button types. This adheres to the OCP because the **`Dialog`** class does not need to change when a new button type is introduced.

plaintext Copy

```
class Button:
    def on_click(self):
        pass

class WindowsButton(Button):
    def on_click(self):
        return "Windows Button Clicked"

class HTMLButton(Button):
    def on_click(self):
        return "HTML Button Clicked"

class Dialog:
    def create_button(self):
        raise NotImplementedError("Subclasses must implement this method")

    def render(self):
        button = self.create_button()
        return button.on_click()

class WindowsDialog(Dialog):
    def create_button(self):
        return WindowsButton()

class WebDialog(Dialog):
    def create_button(self):
        return HTMLButton()

# Usage
windows_dialog = WindowsDialog()
print(windows_dialog.render())  # Output: Windows Button Clicked

web_dialog = WebDialog()
print(web_dialog.render())      # Output: HTML Button Clicked
```

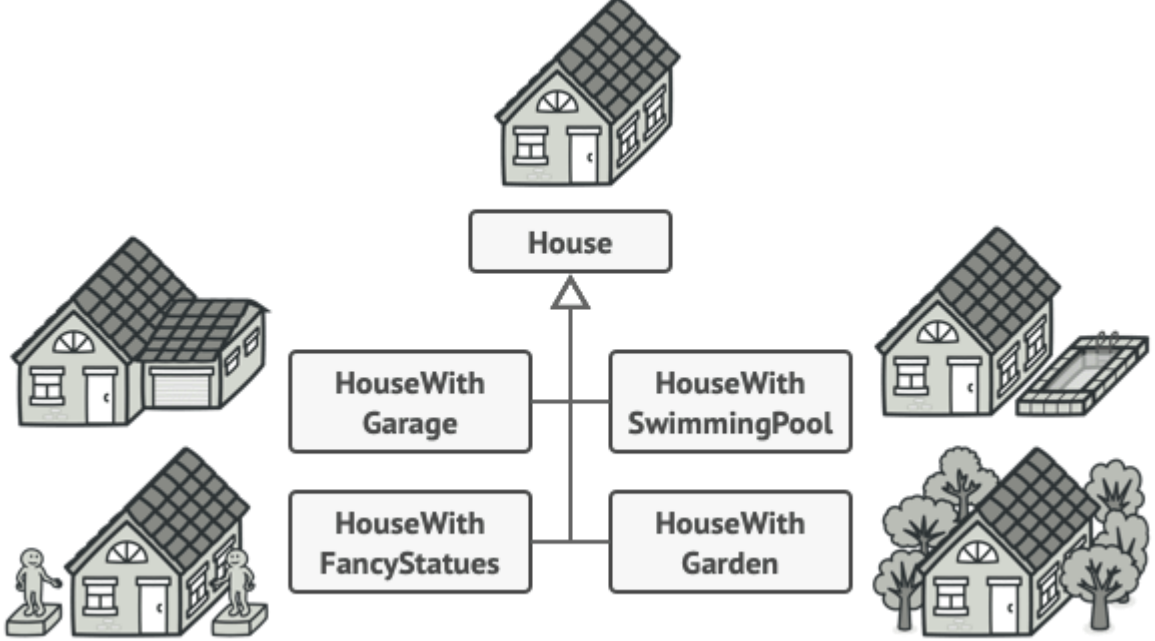
Benefits of the After Implementation

- **Loose Coupling**: The **`Dialog`** class is now an abstract class, promoting loose coupling. The implementation details of button creation are moved to the subclasses (**`WindowsDialog`** and **`WebDialog`**).
- **Easier to Extend**: New button types can be added by creating new classes (e.g., **`HTMLButton`**) and corresponding dialog classes (e.g., **`WebDialog`**) without modifying existing code.
- **Adheres to OCP**: The design follows the Open/Closed Principle, as it is open for extension (by adding new dialog types) but closed for modification (the base dialog class remains unchanged).

Summary

- **Before**: The **`Dialog`** class is responsible for creating button instances, which requires modification for any new button types.
- **After**: The **`Dialog`** class serves as a template, allowing subclasses to define how buttons are created, facilitating easier extensibility and maintenance.

Builder



- *For example*, let's think about how to create a House object. To *build a simple house*, you need to construct *four walls* and a *floor*, install a *door*, fit a *pair of windows*, and build a *roof*. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?
- The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.
- There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base House class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.

Builder

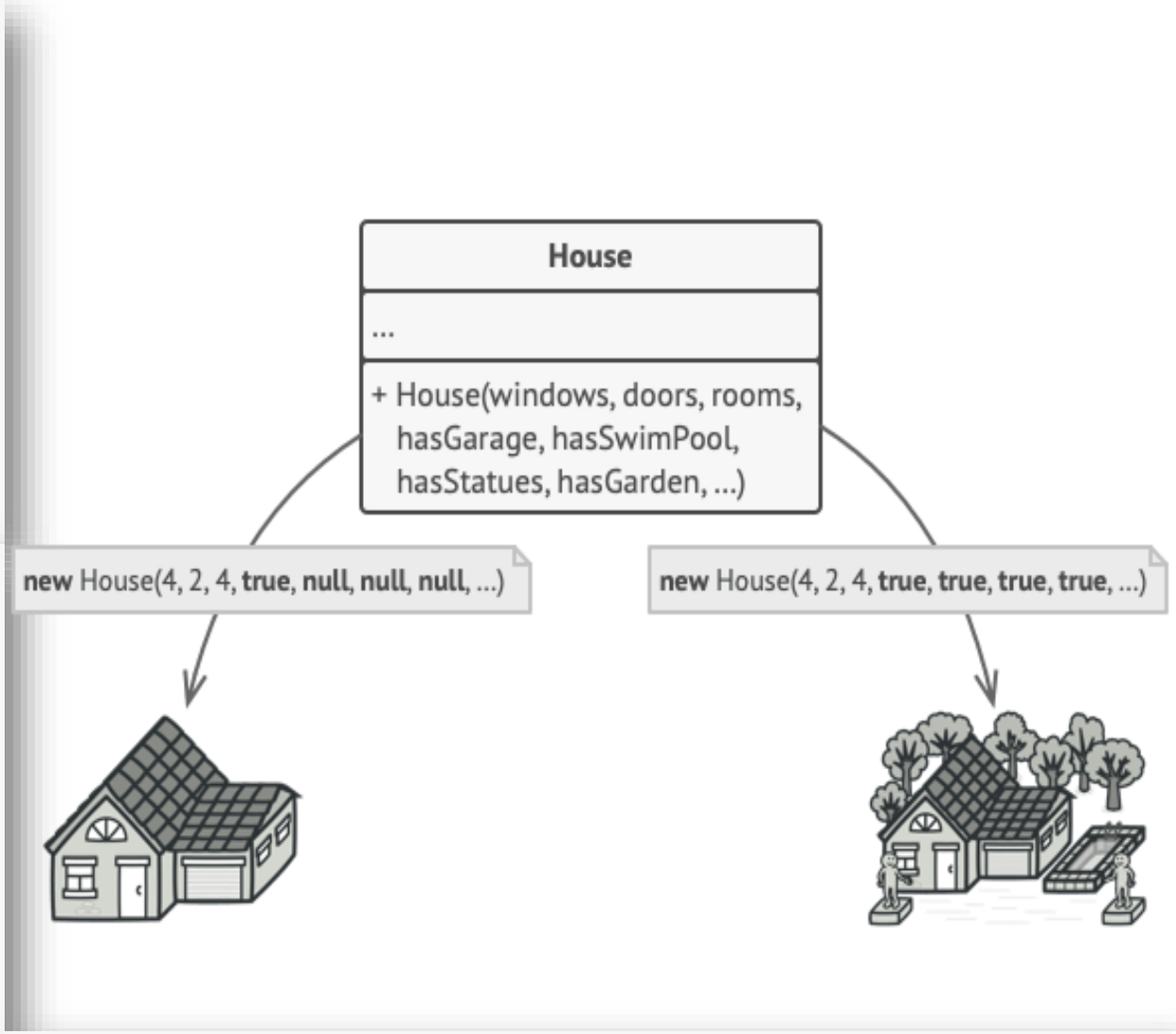
In most cases most of the parameters will be unused, making **the constructor calls pretty ugly** For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be useless nine times out of ten.

The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

Overview of the Builder Pattern

1. **Purpose**:
   - The Builder Pattern is intended to simplify the creation of complex objects by separating the construction process from the representation. It allows for the creation of objects with many optional parameters without requiring numerous constructors.

2. **Components**:
   - **Product**: In this case, the `House` class represents the complex object that is being built.
   - **Builder**: An intermediary class (not shown in the image) that provides methods to set various properties of the `House`.



Breakdown of the Diagram

1. **House Class**:
   - The diagram shows the `House` class, which has various attributes (or properties):
     - `windows`
     - `doors`
     - `rooms`
     - `hasGarage`
     - `hasSwimPool`
     - `hasStatues`
     - `hasGarden`
   - These attributes can have different types, including integers, booleans, and null values.

2. **Constructors**:
   - The image indicates two examples of constructing a `House` object:
     - `new House(4, 2, true, null, ...)`: This represents creating a `House` with specific values for windows, doors, and a boolean for having a garage. Other properties can be set to `null` or left unspecified.
     - `new House(4, 2, 4, true, true, true, true)`: This example shows a more detailed instantiation of a `House`, indicating all properties being set explicitly.

Benefits of Using the Builder Pattern

- **Flexibility**: The Builder Pattern allows for building objects with varying configurations and combinations of properties without needing multiple constructors.
- **Readability**: The code becomes more readable since it clearly indicates which property is being set at each step.
- **Maintainability**: If new properties are added to the `House`, the builder can be updated without affecting existing client code.

Example Implementation

Here's a simple example of how the Builder Pattern might be implemented for the **House** class in code:

php Copy

```php
class House {
    private $windows;
    private $doors;
    private $rooms;
    private $hasGarage;
    private $hasSwimPool;
    private $hasStatues;
    private $hasGarden;

    // Constructor is private to enforce the use of the builder
    private function __construct($builder) {
        $this->windows = $builder->windows;
        $this->doors = $builder->doors;
        $this->rooms = $builder->rooms;
        $this->hasGarage = $builder->hasGarage;
        $this->hasSwimPool = $builder->hasSwimPool;
        $this->hasStatues = $builder->hasStatues;
        $this->hasGarden = $builder->hasGarden;
    }

    public static function builder() {
        return new HouseBuilder();
    }
}
```

```php
class HouseBuilder {
    public $windows;
    public $doors;
    public $rooms;
    public $hasGarage = false;
    public $hasSwimPool = false;
    public $hasStatues = false;
    public $hasGarden = false;

    public function setWindows($windows) {
        $this->windows = $windows;
        return $this;
    }

    public function setDoors($doors) {
        $this->doors = $doors;
        return $this;
    }

    public function setRooms($rooms) {
        $this->rooms = $rooms;
        return $this;
    }

    public function setGarage($hasGarage) {
        $this->hasGarage = $hasGarage;
        return $this;
    }

    // Other setter methods...

    public function build() {
        return new House($this);
    }
}
```

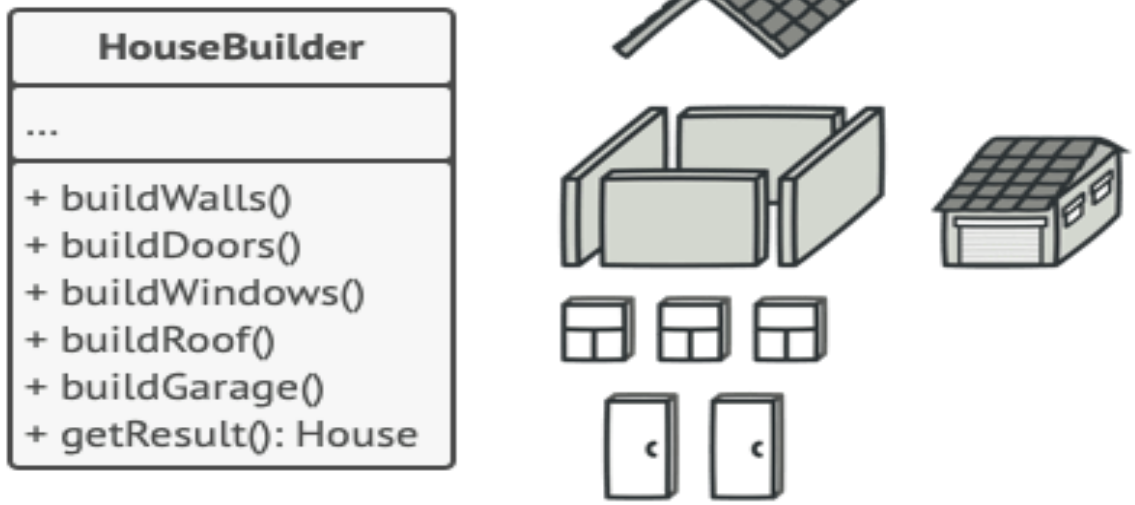
```php
    // Other setter methods...

    public function build() {
        return new House($this);
    }
}

// Usage
$house = House::builder()
    ->setWindows(4)
    ->setDoors(2)
    ->setGarage(true)
    ->build();
```

Summary

The Builder Pattern is a powerful design pattern that enhances the creation of complex objects by providing a flexible and readable way to set properties. The example of the **House** class illustrates how this pattern can be effectively applied, making it easier to manage and construct objects with various configurations.

Builder-Solution

- **The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.**

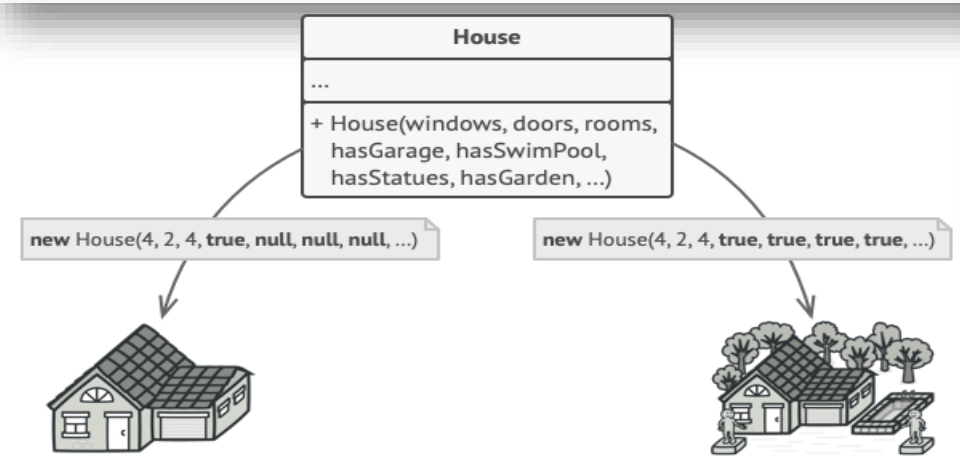


Before the Builder Pattern

1. **Complex Constructors**:
   - In the initial approach, a `House` may have a constructor that requires many parameters, making it cumbersome to create `House` objects. This can lead to constructor overloading or long parameter lists.
2. **Lack of Readability**:
   - When creating a `House`, it might look like this:

```php
$house = new House(4, 2, true, false, true, null, true);
```

   - This line is hard to read and understand, especially if the meaning of each parameter is not clear.
3. **Rigid Structure**:
   - If you want to change or add features (like a garage or swimming pool), you may need to modify the constructor and all places where the `House` is instantiated.

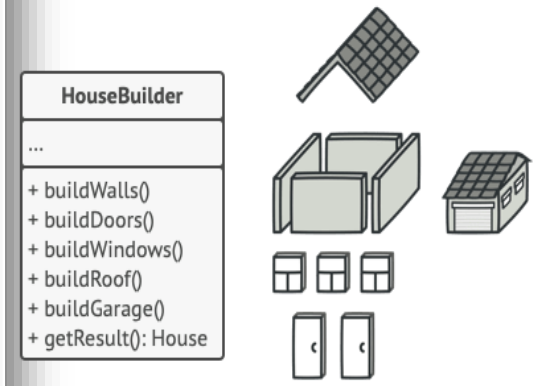


After Introducing the Builder Pattern

1. **Separation of Concerns**:
   - The Builder Pattern encapsulates the construction logic within a `HouseBuilder` class, allowing you to build a `House` step by step without needing to know all the details upfront.
2. **Method Chaining**:
   - The builder provides methods like `buildWalls()`, `buildDoors()`, `buildWindows()`, and `buildRoof()`, which can be called in sequence. This promotes a fluid interface and enhances readability:

```php
$houseBuilder = new HouseBuilder();
$houseBuilder->buildWalls()
             ->buildDoors()
             ->buildWindows()
             ->buildRoof();
$house = $houseBuilder->getResult();
```

3. **Modularity**:
   - Each method in the `HouseBuilder` can focus on a specific aspect of the `House` construction. This modular approach allows for easier modifications and expansions, such as adding new features without changing the client code.
4. **Clearer Intent**:
   - The use of named methods makes it clear what each step of the construction process is doing, improving code maintainability and reducing the likelihood of errors.



Breakdown of the Builder Methods

- `buildWalls()`:
  - Constructs the walls of the house, encapsulating the logic for wall creation.
- `buildDoors()`:
  - Handles the construction of doors, allowing for custom door configurations.
- `buildWindows()`:
  - Manages the addition of windows, including their placement and characteristics.
- `buildRoof()`:
  - Takes care of the roof construction, ensuring it fits the overall design of the house.
- `buildGarage()` (not shown but implied):
  - Can be added to include garage construction in the future.
- `getResult()`:
  - Finalizes the construction process and returns the fully constructed `House` object.

Summary

The transition from the traditional constructor approach to the Builder Pattern enhances the flexibility and readability of object construction. By separating the construction logic into a dedicated builder class, developers can easily create complex objects like `House` in a clear and maintainable way. This pattern is particularly beneficial when dealing with objects that have numerous optional attributes or require a specific construction sequence.

Here's an example illustrating the Builder Pattern, showcasing both the "before" and "after" implementations. The Builder Pattern is particularly useful for constructing complex objects step-by-step while keeping the construction process separate from the representation.

Before Applying the Builder Pattern

In this initial implementation, the `Car` class requires many parameters in its constructor, making it difficult to create instances, especially when some parameters are optional.

python ▶ Run Copy

```python
class Car:
    def __init__(self, make, model, year, color, engine_size, is_electric=False):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.engine_size = engine_size
        self.is_electric = is_electric

# Usage
car1 = Car("Tesla", "Model S", 2022, "Red", "Electric", True)
car2 = Car("Ford", "Mustang", 2021, "Blue", "V8")
```

Issues with the Before Implementation

- **Complexity**: The constructor has many parameters, making it hard to create a `Car` object without specifying all values.
- **Readability**: It's not clear what each parameter represents when calling the constructor, especially when many parameters are optional.
- **Error-Prone**: It's easy to mix up the order of parameters or forget to provide optional ones.

After Applying the Builder Pattern

In the refactored implementation, a `CarBuilder` class is introduced to construct `Car` objects step-by-step. This approach improves readability and maintainability.

python ▶ Run Copy

```python
class Car:
    def __init__(self, make, model, year, color, engine_size, is_electric=False):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
        self.engine_size = engine_size
        self.is_electric = is_electric

class CarBuilder:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.year = None
        self.color = None
        self.engine_size = None
        self.is_electric = False

    def set_year(self, year):
        self.year = year
        return self

    def set_color(self, color):
        self.color = color
        return self

    def set_engine_size(self, engine_size):
        self.engine_size = engine_size
        return self

    def set_is_electric(self, is_electric):
        self.is_electric = is_electric
        return self

    def build(self):
        return Car(self.make, self.model, self.year, self.color, self.engine_size, self.is_electric)

# Usage
car_builder = CarBuilder("Tesla", "Model S")
car1 = (car_builder
        .set_year(2022)
        .set_color("Red")
        .set_engine_size("Electric")
        .set_is_electric(True)
        .build())

car_builder = CarBuilder("Ford", "Mustang")
car2 = (car_builder
        .set_year(2021)
        .set_color("Blue")
        .set_engine_size("V8")
        .build())
```

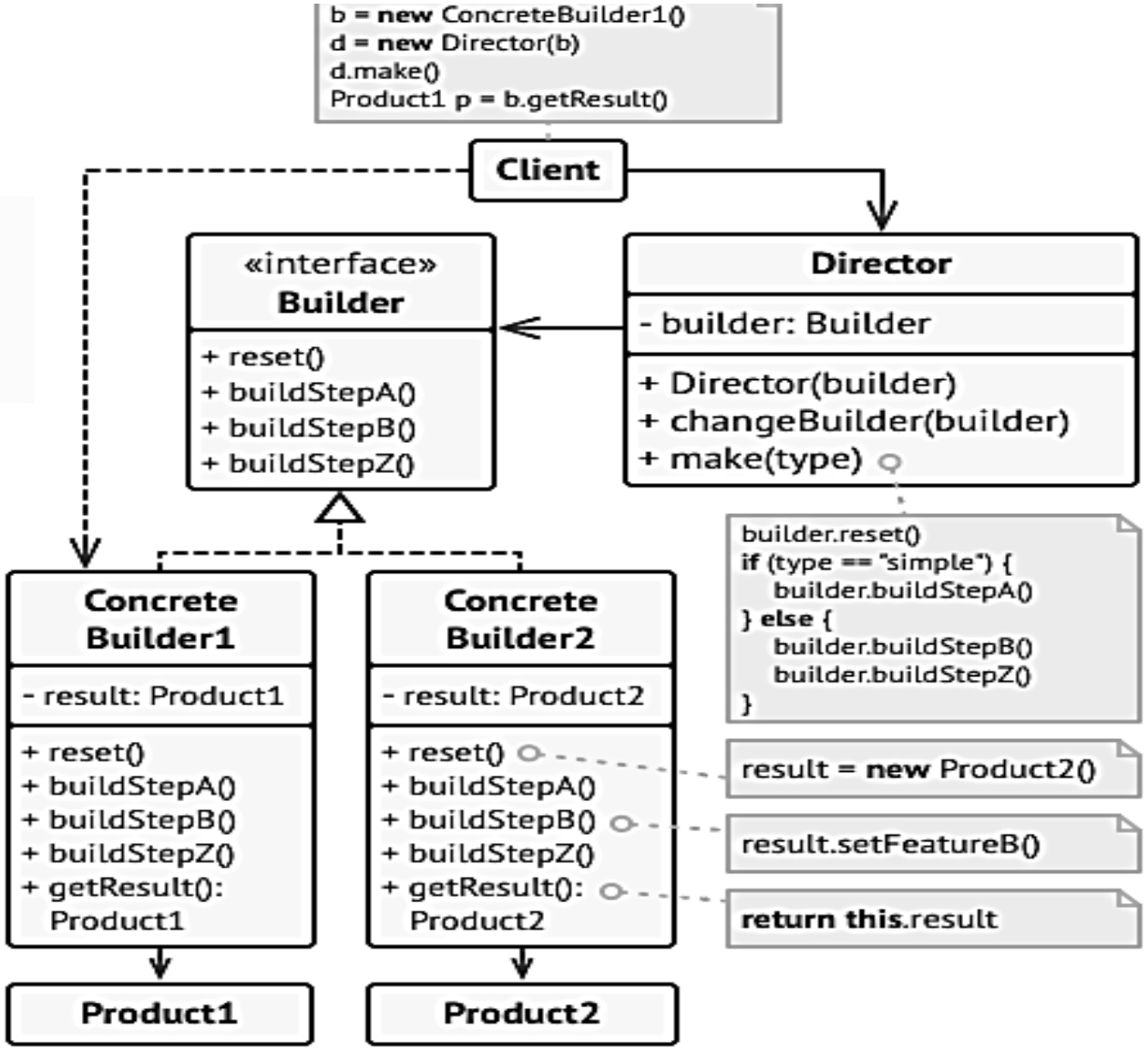
Benefits of the After Implementation

- **Fluent Interface**: The builder allows chaining methods, making the code more readable and expressive.
- **Optional Parameters**: It's easy to set only the parameters you need, improving flexibility.
- **Clear Intent**: Each method clearly indicates what it's setting, reducing the likelihood of errors.
- **Separation of Concerns**: The construction logic is separated from the `Car` class, adhering to the Single Responsibility Principle.

Summary

- **Before**: The constructor of the `Car` class had many parameters, making it complex and error-prone.
- **After**: The Builder Pattern simplifies the creation of `Car` objects, making the code more readable, flexible, and maintainable.

Builder-Solution



1 The **Builder** interface declares product construction steps that are common to all types of builders.

2 **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

3 **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.

4 The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

5 The **Client** must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

Builder Example

- **Builder**

```php
interface SQLQueryBuilder
{
    public function select(string $table, array $fields): SQLQueryBuilder;

    public function where(string $field, string $value, string $operator = '=')

    public function limit(int $start, int $offset): SQLQueryBuilder;

    // +100 other SQL syntax methods...

    public function getSQL(): string;
}
```

The `SQLQueryBuilder` interface is a design pattern used in software development to construct SQL queries programmatically. Here's a breakdown of its components:

Methods

1. **select($table, array $fields)**:
   - This method specifies the table from which to select data and the fields (columns) to retrieve.
   - Returns an instance of `SQLQueryBuilder` to allow method chaining.
2. **where(string $field, string $value, string $operator = '=')**:
   - This method adds a condition to filter results based on a specific field, value, and operator (default is =).
   - Also returns an instance of `SQLQueryBuilder` for chaining.
3. **limit(int $start, int $offset)**:
   - This method restricts the number of results returned by the query, starting from a specified point.
   - Returns an instance of `SQLQueryBuilder`.
4. **getSQL()**:
   - This method compiles the constructed query into a string format, ready for execution.

Purpose

The `SQLQueryBuilder` interface allows developers to build SQL queries dynamically and fluently, improving code readability and maintainability. It promotes a structured approach to SQL query creation, reducing the risk of syntax errors and SQL injection vulnerabilities.

Example Usage

php Copy

```php
$query = (new SQLQueryBuilder())
    ->select('users', ['id', 'name'])
    ->where('age', '18', '>=')
    ->limit(0, 10)
    ->getSQL();
```

In this example, a SQL query is being built to select user IDs and names for users who are 18 or older, limiting the results to the first 10 entries.

Builder Example

- Concrete Builder 1

```php
class MysqlQueryBuilder implements SQLQueryBuilder
{
    protected $query;

    protected function reset(): void
    {
        $this->query = new \stdClass();
    }

    /**
     * Build a base SELECT query.
     */
    public function select(string $table, array $fields): SQLQueryBuilder
    {
        $this->reset();
        $this->query->base = "SELECT " . implode(", ", $fields) . " FROM " .
        $this->query->type = 'select';

        return $this;
    }

    /**
     * Add a WHERE condition.
     */
    public function where(string $field, string $value, string $operator = '=
    {
        if (!in_array($this->query->type, ['select', 'update', 'delete'])) {
            throw new \Exception("WHERE can only be added to SELECT, UPDATE O
        }
```

```php
    public function limit(int $start, int $offset): SQLQueryBuilder
    {
        if (!in_array($this->query->type, ['select'])) {
            throw new \Exception("LIMIT can only be added to SELECT");
        }
        $this->query->limit = " LIMIT " . $start . ", " . $offset;

        return $this;
    }

    /**
     * Get the final query string.
     */
    public function getSQL(): string
    {
        $query = $this->query;
        $sql = $query->base;
        if (!empty($query->where)) {
            $sql .= " WHERE " . implode(' AND ', $query->where);
        }
        if (isset($query->limit)) {
            $sql .= $query->limit;
        }
        $sql .= ";";
        return $sql;
    }
}
```

The provided code snippet outlines a `MysqlQueryBuilder` class that implements the `SQLQueryBuilder` interface, specifically designed to construct SQL queries for a MySQL database. Here's a detailed explanation of its components:

Class Definition

```php
class MysqlQueryBuilder implements SQLQueryBuilder {
```

- This line indicates that `MysqlQueryBuilder` is a class that implements the `SQLQueryBuilder` interface, meaning it must provide definitions for all the methods declared in the interface.

Properties

```php
protected function reset() {
```

- This method likely resets the internal state of the query builder, preparing it to construct a new query.

Example Usage

Using this **`MysqlQueryBuilder`**, a developer might construct a SQL query like this:

```php
$query = (new MysqlQueryBuilder())
    ->select('users', ['id', 'name'])
    ->where('age', '18', '>=')
    ->limit(0, 10)
    ->getSQL();
```

Summary

The **`MysqlQueryBuilder`** class provides a structured and flexible way to construct SQL queries for MySQL databases. By implementing the **`SQLQueryBuilder`** interface, it ensures that all necessary methods are defined, allowing for method chaining and improving code readability. This approach reduces potential errors and enhances security by minimizing direct SQL string manipulation.

Methods

1. **select($table, array $fields)**:

```php
public function select($table, array $fields): SQLQueryBuilder {
    $this->query->type = 'select';
    $this->query->from = $table;
    $this->query->fields = $fields;
    return $this;
}
```

   - Sets the type of query to "select", specifies the table, and the fields to retrieve. It returns the current instance for method chaining.

2. **where(string $field, string $value, string $operator = '=')**:

```php
public function where(string $field, string $value, string $operator = '='): SQLQuer
yBuilder {
    // Logic to add a WHERE condition
}
```

   - Adds a filtering condition to the query. The default operator is equality (=). The specifics of how the condition is added would be implemented in the method body.

3. **limit(int $start, int $offset)**:

```php
public function limit(int $start, int $offset): SQLQueryBuilder {
    // Logic to set LIMIT clause
    return $this;
}
```

   - Implements the limit functionality to restrict the number of returned rows. It allows pagination of results.

4. **getSQL()**:

```php
public function getSQL(): string {
    // Logic to compile the final SQL query
}
```

   - Compiles the constructed query into a string that can be executed on a MySQL database. It likely checks for any errors or missing components before returning the final SQL statement.

Builder Example

- **Concrete Builder 2**

```php
class PostgresQueryBuilder extends MysqlQueryBuilder
{
    /**
     * Among other things, PostgreSQL has slightly different LIMIT syntax.
     */
    public function limit(int $start, int $offset): SQLQueryBuilder
    {
        parent::limit($start, $offset);

        $this->query->limit = " LIMIT " . $start . " OFFSET " . $offset;

        return $this;
    }

    // + tons of other overrides...
}
```

The provided code snippet defines a `PostgresQueryBuilder` class that extends the `MysqlQueryBuilder` class. This design allows the `PostgresQueryBuilder` to inherit all the functionality of the `MysqlQueryBuilder` while also providing specific implementations for PostgreSQL's syntax. Here's a breakdown of the key components:

Class Definition

```php
class PostgresQueryBuilder extends MysqlQueryBuilder {
```

- This line indicates that `PostgresQueryBuilder` is a subclass of `MysqlQueryBuilder`, inheriting its properties and methods.

Method Override

```php
public function limit(int $start, int $offset): SQLQueryBuilder {
    parent::limit($start, $offset);
    $this->query->limit = "LIMIT $start OFFSET $offset";
    return $this;
}
```

- **Purpose**: This method overrides the `limit` method from the `MysqlQueryBuilder`.
- **Parameters**: It takes two parameters: `$start` (the starting point) and `$offset` (the number of records to skip).
- **Calling Parent Method**: It first calls the parent class's `limit` method to maintain any base functionality that might be defined there.
- **PostgreSQL Syntax**: It sets the `$this->query->limit` property to the PostgreSQL-specific syntax for limiting results, which uses the keywords `LIMIT` and `OFFSET`.
- **Method Chaining**: Finally, it returns `$this`, allowing for further method chaining.

Additional Functionality

```php
// tons of other overrides...
```

- This comment suggests that the `PostgresQueryBuilder` class potentially includes many other methods that are overridden to accommodate PostgreSQL's unique SQL syntax and functionalities. This could include methods for handling `JOIN`, `INSERT`, `UPDATE`, `DELETE`, and even specific data types or functions unique to PostgreSQL.

Example Usage

Using the `PostgresQueryBuilder`, a developer might construct a SQL query like this:

```php
$query = (new PostgresQueryBuilder())
    ->select('users', ['id', 'name'])
    ->where('age', '18', '>=')
    ->limit(0, 10)
    ->getSQL();
```

Summary

The `PostgresQueryBuilder` class extends the `MysqlQueryBuilder`, allowing for easy adaptation of SQL query construction for PostgreSQL databases. By overriding methods like `limit`, it ensures the correct syntax is used while still leveraging the base functionality of the parent class. This approach promotes code reuse and adheres to the principles of object-oriented programming, allowing for cleaner and more maintainable code when dealing with different database systems.

Builder Example

- Client

```php
function clientCode(SQLQueryBuilder $queryBuilder)
{
    // ...

    $query = $queryBuilder
        ->select("users", ["name", "email", "password"])
        ->where("age", 18, ">")
        ->where("age", 30, "<")
        ->limit(10, 20)
        ->getSQL();

    echo $query;

    // ...
}
```

The provided code snippet illustrates a function named `clientCode` that utilizes a `SQLQueryBuilder` instance to construct an SQL query. Here's a detailed breakdown of its components:

Function Definition

```php
function clientCode(SQLQueryBuilder $queryBuilder) {
```

- This line defines a function called `clientCode`, which accepts a parameter named `$queryBuilder`.
- The parameter type is specified as `SQLQueryBuilder`, indicating that any object passed to this function must implement the `SQLQueryBuilder` interface (like `MysqlQueryBuilder` or `PostgresQueryBuilder`).

Query Construction

```php
$query = $queryBuilder
    ->select("users", ["name", "email", "password"])
    ->where("age", 18)
    ->where("age", 30)
    ->limit(10, 20)
    ->getSQL();
```

- **`$queryBuilder` Usage**: The function uses the `$queryBuilder` object to build an SQL query step-by-step through method chaining.

1. **Selecting Fields**:

```php
->select("users", ["name", "email", "password"])
```

   - This method call specifies that the query will select the **`name`**, **`email`**, and **`password`** fields from the **`users`** table.

2. **Adding WHERE Conditions**:

```php
->where("age", 18)
->where("age", 30)
```

   - These lines add filtering conditions to the query. However, the second **`where`** call seems to replace the first one unless it's designed to handle multiple conditions (like using **`AND`** or **`OR`**).

3. **Limiting Results**:

```php
->limit(10, 20)
```

   - This method sets a limit on the number of results returned by the query. The parameters indicate starting at record 10 and returning 20 records.

4. **Getting the Final SQL**:

```php
->getSQL();
```

   - This method compiles the constructed query into a valid SQL string that can be executed against a database.

Output

```php
echo $query;
```

- This line outputs the final SQL query string to the screen.

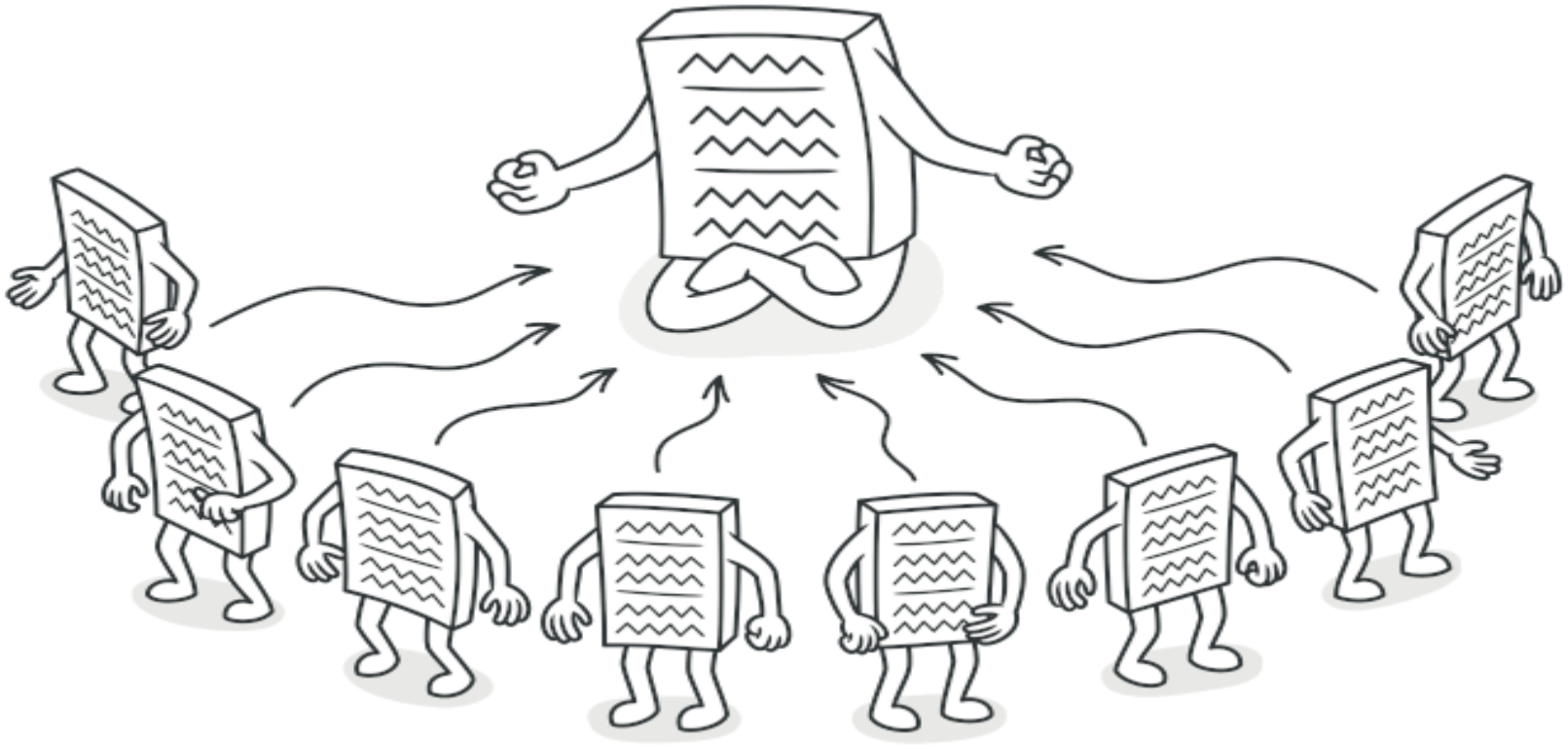
Summary

The **`clientCode`** function demonstrates how to use a **`SQLQueryBuilder`** to construct an SQL query in a flexible and readable manner. By utilizing method chaining, it allows developers to build complex queries in a straightforward way. The function can work with any implementation of the **`SQLQueryBuilder`** interface, making it adaptable for different database systems, such as MySQL or PostgreSQL.

This design promotes code reusability, maintainability, and adherence to the principles of abstraction in object-oriented programming.

Singleton

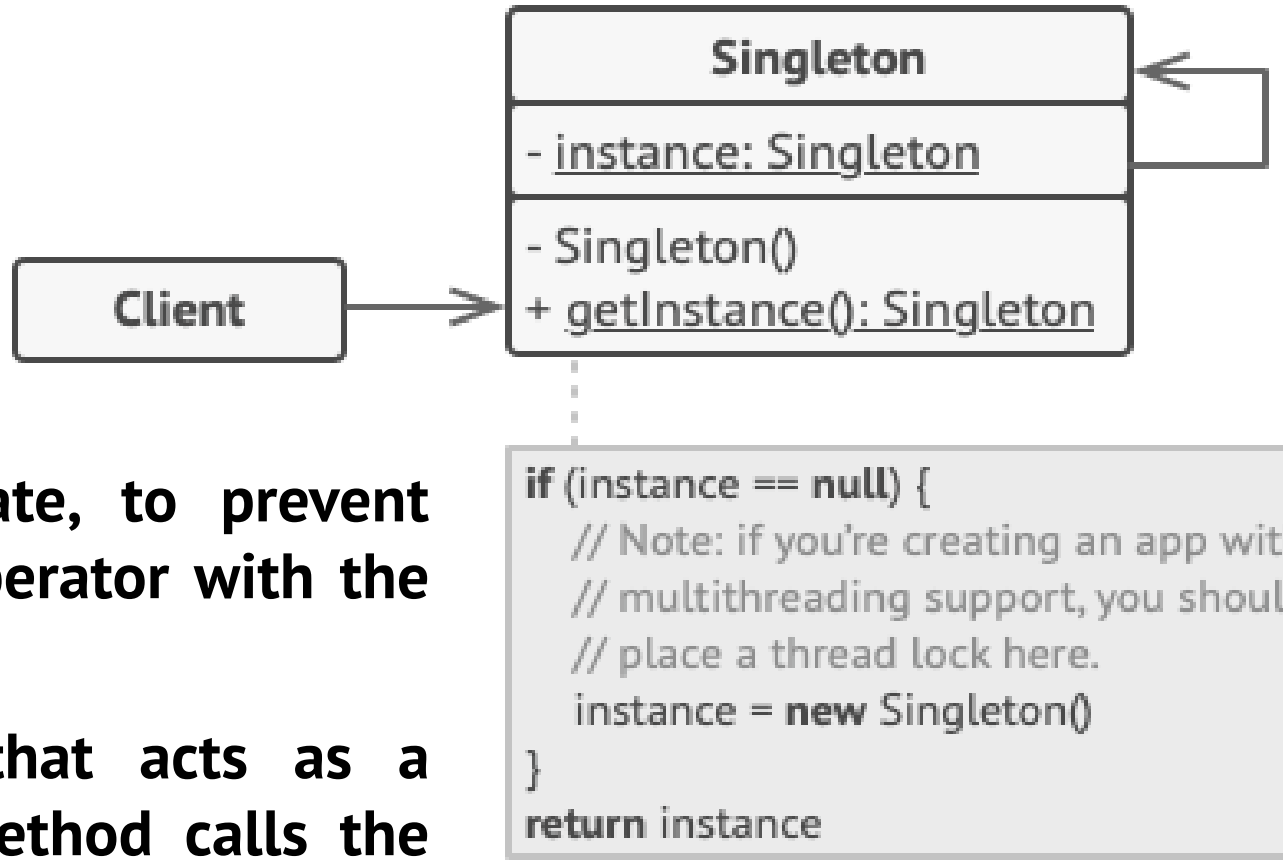
- **Singleton** is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.



Singleton - Problem

- **Ensure that a class has just a single instance**. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.
- **Provide a global access point to that instance**. Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.
- Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

Singleton - Solution



- **Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.**
- **Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.**

Singleton -Example

- Singleton class

```php
<?php
class Singleton
{
    private static $instances = [];
    protected function __construct() { }
    protected function __clone() { }
    public function __wakeup()
    {
        throw new \Exception("Cannot unserialize singleton");
    }
    public static function getInstance()
    {
        $subclass = static::class;
        if (!isset(self::$instances[$subclass])) {

            self::$instances[$subclass] = new static();
        }
        return self::$instances[$subclass];
    }
}
```

The provided code snippet is an implementation of the Singleton design pattern in PHP. Here's a breakdown of the key components:

Breakdown of the Code

1. **Private Static Property**:

```php
private static $instances = [];
```

This array holds the instances of the singleton class. It allows the class to manage its instances based on the subclass.

2. **Constructor**:

```php
protected function __construct() { }
```

The constructor is protected to prevent creating new instances of the class from outside the class itself.

3. **Cloning**:

```php
protected function __clone() { }
```

The __clone method is protected to prevent cloning of the instance.

4. **Waking Up**:

```php
public function __wakeup() {
    throw new Exception("Cannot unserialize singleton");
}
```

This method prevents the singleton instance from being unserialized, which would create a new instance.

```php
<?php
class Singleton
{
    private static $instances = [];
    protected function __construct() { }
    protected function __clone() { }
    public function __wakeup()
    {
        throw new \Exception("Cannot unserialize singleton");
    }
    public static function getInstance()
    {
        $subclass = static::class;
        if (!isset(self::$instances[$subclass])) {

            self::$instances[$subclass] = new static();
        }
        return self::$instances[$subclass];
    }
}
```

5. **Get Instance Method**:

```php
public static function getInstance() {
    $subclass = static::class;
    if (!isset(self::$instances[$subclass])) {
        self::$instances[$subclass] = new static();
    }
    return self::$instances[$subclass];
}
```

- This method checks if an instance of the class already exists in the $instances array.
- If it does not exist, it creates a new instance and stores it.
- It then returns the existing instance.

Usage

To utilize this Singleton class, you would typically extend it with another class, and then call getInstance() to get the singleton instance.

Example Usage

```php
class MySingleton extends Singleton {
    // Additional properties and methods
}

$instance = MySingleton::getInstance();
```

This ensures that no matter how many times you call MySingleton::getInstance(), you will receive the same instance.

Singleton -Example

- Log Singleton class

```php
class Logger extends Singleton
{
    private $fileHandle;
    protected function __construct()
    {
        $this->fileHandle = fopen('php://stdout', 'w');
    }
    public function writeLog(string $message): void
    {
        $date = date('Y-m-d');
        fwrite($this->fileHandle, "$date: $message\n");
    }

    public static function log(string $message): void
    {
        $logger = static::getInstance();
        $logger->writeLog($message);
    }
}
```

```php
class Logger extends Singleton
{
    private $fileHandle;
    protected function __construct()
    {
        $this->fileHandle = fopen('php://stdout', 'w');
    }
    public function writeLog(string $message): void
    {
        $date = date('Y-m-d');
        fwrite($this->fileHandle, "$date: $message\n");
    }

    public static function log(string $message): void
    {
        $logger = static::getInstance();
        $logger->writeLog($message);
    }
}
```

Let's break down the provided code snippet, which illustrates a `Logger` class implementing the Singleton design pattern. This Logger will manage logging messages to a file.

Explanation of the Code

1. Class Definition

```php
class Logger extends Singleton {
    private $fileHandle;

    protected function __construct() {
        $this->fileHandle = fopen('php://stdout', 'w');
    }
}
```

- **Inheritance**: The `Logger` class extends the `Singleton` class, meaning it inherits the Singleton behavior.
- **Private Property**: `private $fileHandle;` is used to hold the file handle for logging.
- **Protected Constructor**: The constructor is protected to prevent direct instantiation from outside the class. It initializes `$fileHandle` to write to standard output (or a specified file).

2. Logging Method

```php
public function writeLog(string $message): void {
    $date = date('Y-m-d');
    fwrite($this->fileHandle, "$date: $message\n");
}
```

- **Method Definition**: `writeLog` takes a string message as an argument.
- **Date Formatting**: The current date is formatted as `Y-m-d`.
- **Writing to File**: The `fwrite` function writes the formatted message along with the date to the file handle.

3. Static Log Method

```php
public static function log(string $message): void {
    $logger = static::getInstance();
    $logger->writeLog($message);
}
```

- **Static Method**: `log` is a static method that allows users to log messages without needing to instantiate the class.
- **Get Instance**: It retrieves the singleton instance using `getInstance()`.
- **Write Log**: It calls the `writeLog` method on the instance to log the message.

How to Use the Logger

Here's how you would use the Logger class within your application:

```php
Logger::log("This is a log message.");  // Logs the message to the output
```

Benefits of Using the Singleton Pattern for Logging

1. **Single Instance Management**: The Singleton ensures that there is only one instance of the logger throughout the application, avoiding the overhead of multiple loggers.
2. **Centralized Logging**: All log messages are managed through a single point, making it easier to maintain and control logging behavior.
3. **Global Access**: The static `log` method provides a simple interface to log messages from anywhere in the code without needing to manage instances.
4. **Consistency**: Since all log messages go through the same instance, it ensures consistent formatting and handling of log entries.

Before Using the Singleton Pattern

- **Multiple Instances**: Without the Singleton pattern, you might create multiple Logger instances, leading to scattered log messages and inconsistencies in how logs are handled.
- **Complexity**: Managing the state for multiple loggers can add unnecessary complexity to the code.

After Using the Singleton Pattern

- **Streamlined Logging**: The Singleton pattern streamlines the logging process, ensuring that all log messages are handled by a single instance.
- **Easier Maintenance**: Changes to the logging mechanism (e.g., output format or destination) can be done in one place, simplifying maintenance.

Summary

The Logger class, implementing the Singleton pattern, provides a centralized and efficient way to manage logging in your application. This design pattern enhances consistency, control, and simplicity, making it easier to maintain and use the logging functionality across your codebase.

Singleton -Example

- Config Singleton class

```php
class Config extends Singleton
{
    private $hashmap = [];

    public function getValue(string $key): string
    {
        return $this->hashmap[$key];
    }

    public function setValue(string $key, string $value): void
    {
        $this->hashmap[$key] = $value;
    }
}
```

Let's break down the provided code snippet that demonstrates a `Config` class implementing the Singleton design pattern. This class is used to manage configuration settings within an application.

Explanation of the Code

1. Class Definition

```php
class Config extends Singleton {
    private $hashmap = [];
}
```

- **Inheritance**: The `Config` class extends the `Singleton` class, meaning it inherits the Singleton behavior.
- **Private Property**: `private $hashmap = [];` is an array that will store configuration settings as key-value pairs.

2. Getting Configuration Values

```php
public function getValue(string $key) {
    return $this->hashmap[$key];
}
```

- **Method Definition**: `getValue` takes a string key as an argument.
- **Returning Values**: It returns the value associated with the specified key from the `$hashmap` array. If the key does not exist, it will trigger a notice.

3. Setting Configuration Values

```php
public function setValue(string $key, string $value): void {
    $this->hashmap[$key] = $value;
}
```

- **Method Definition**: `setValue` takes a key and a value as arguments.
- **Storing Values**: It stores the provided value in the `$hashmap` array under the specified key.

How to Use the Config Class

Here's how you would typically use the `Config` class in your application:

```php
// Set a configuration value
Config::getInstance()->setValue('database_host', 'localhost');

// Get a configuration value
$dbHost = Config::getInstance()->getValue('database_host');
echo $dbHost;  // Outputs: localhost
```

Benefits of Using the Singleton Pattern for Configuration Management

1. **Single Instance Management**: The Singleton ensures that only one instance of the configuration manager exists, preventing conflicts and ensuring consistency across the application.
2. **Centralized Configuration**: All configuration settings are managed through a single point, making it easier to maintain and update configurations.
3. **Global Access**: The static method to retrieve the instance allows settings to be accessed from anywhere in the code without needing to manage instances.
4. **Consistency**: Since all configuration values are stored in one instance, it ensures that the application uses consistent settings throughout its lifecycle.

Before Using the Singleton Pattern

- **Multiple Instances**: Without the Singleton pattern, you might create multiple `Config` instances, leading to different parts of the application using different configurations.
- **Complexity**: Managing configuration state across various instances can complicate the application structure.

After Using the Singleton Pattern

- **Streamlined Configuration Management**: The Singleton pattern streamlines how configuration settings are stored and accessed, ensuring that all parts of the application work with the same set of configurations.
- **Easier Maintenance**: Changes to configuration settings can be done in one place, making it easier to manage and maintain.

Summary

The `Config` class, implementing the Singleton pattern, provides a centralized and efficient way to handle application configurations. This design pattern enhances consistency, control, and simplicity, making it easier to manage configuration settings across your codebase.

Singleton -Example

- Client

```php
Logger::log("Started!");
$l1 = Logger::getInstance();
$l2 = Logger::getInstance();
if ($l1 === $l2) {
    Logger::log("Logger has a single instance.");
} else {
    Logger::log("Loggers are different.");
}

$config1 = Config::getInstance();
$login = "test_login";
$password = "test_password";
$config1->setValue("login", $login);
$config1->setValue("password", $password);
$config2 = Config::getInstance();
if ($login == $config2->getValue("login") &&
    $password == $config2->getValue("password")
) {
    Logger::log("Config singleton also works fine.");
}

Logger::log("Finished!");
```

Let's break down the provided code snippet that illustrates the use of the Singleton pattern with the `Logger` and `Config` classes. This example demonstrates how these Singleton classes can be used in practice.

Explanation of the Code

1. Logger Singleton Usage

```php
Logger::log("Started!");

// Get two instances of Logger
$l1 = Logger::getInstance();
$l2 = Logger::getInstance();

if ($l1 === $l2) {
    Logger::log("Logger has a single instance.");
} else {
    Logger::log("Loggers are different.");
}
```

- **Logging a Message**: The code starts by logging the message "Started!" using the `Logger` class.
- **Getting Instances**: Two instances of the `Logger` class are retrieved using `getInstance()`.
- **Instance Comparison**: The code checks if `$l1` and `$l2` refer to the same instance. Since both calls to `getInstance()` return the same instance, the message "Logger has a single instance." will be logged.

2. Config Singleton Usage

```php
$config1 = Config::getInstance();
$login = "test_login";
$password = "test_password";

// Set login and password values
$config1->setValue("login", $login);
$config1->setValue("password", $password);

// Get another instance of Config
$config2 = Config::getInstance();

// Check if the login and password match
if ($login === $config2->getValue("login") && $password === $config2->getValue("passwor
d")) {
    Logger::log("Config singleton also works fine.");
}
```

- **Getting Config Instance**: The code retrieves an instance of the `Config` Singleton.
- **Setting Values**: It sets the login and password values in the configuration using `setValue()`.
- **Retrieving Another Instance**: It retrieves another instance of the `Config` Singleton.
- **Value Comparison**: The code checks if the login and password values match the ones stored in the configuration. If they match, the message "Config singleton also works fine." is logged.

Summary of Singleton Behavior

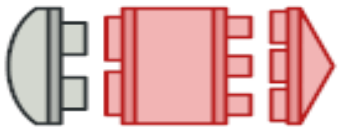
1. **Single Instance Guarantee**: Both the `Logger` and `Config` classes ensure that only one instance is created. This is confirmed by the checks in the code.
2. **Consistency Across the Application**: Any part of the application accessing the `Logger` or `Config` classes will interact with the same instance, ensuring consistent behavior and configuration.
3. **Centralized Management**: Both logging and configuration settings are managed centrally, making it easier to maintain and modify as necessary.

Conclusion

The provided code snippet effectively demonstrates the practical application of the Singleton design pattern with `Logger` and `Config` classes. This pattern ensures that these classes maintain a single instance throughout the application, promoting consistency and ease of management.

Structural Design Patterns

- **Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.**

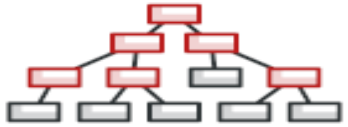


Adapter

Allows objects with incompatible interfaces to collaborate.

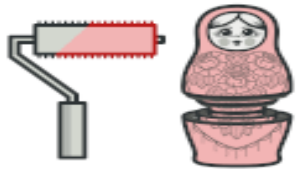
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Facade

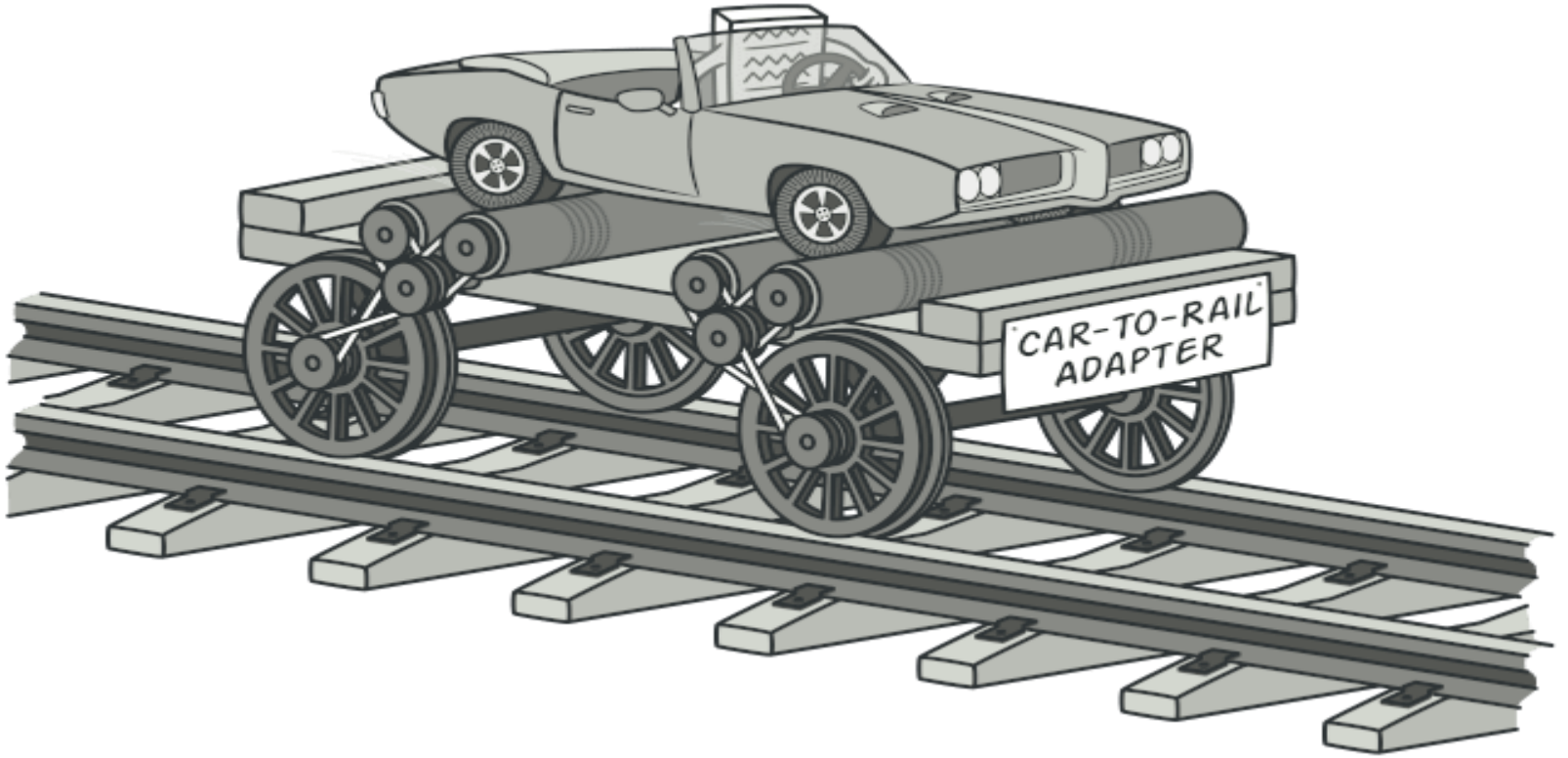
Provides a simplified interface to a library, a framework, or any other complex set of classes.

Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

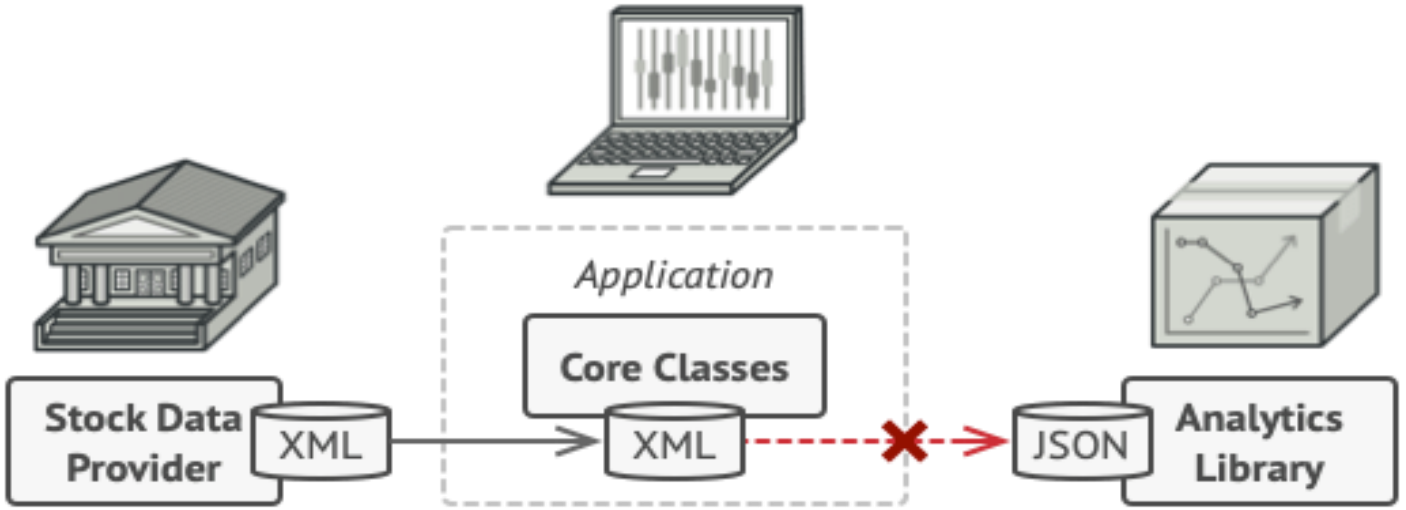
Adapter

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



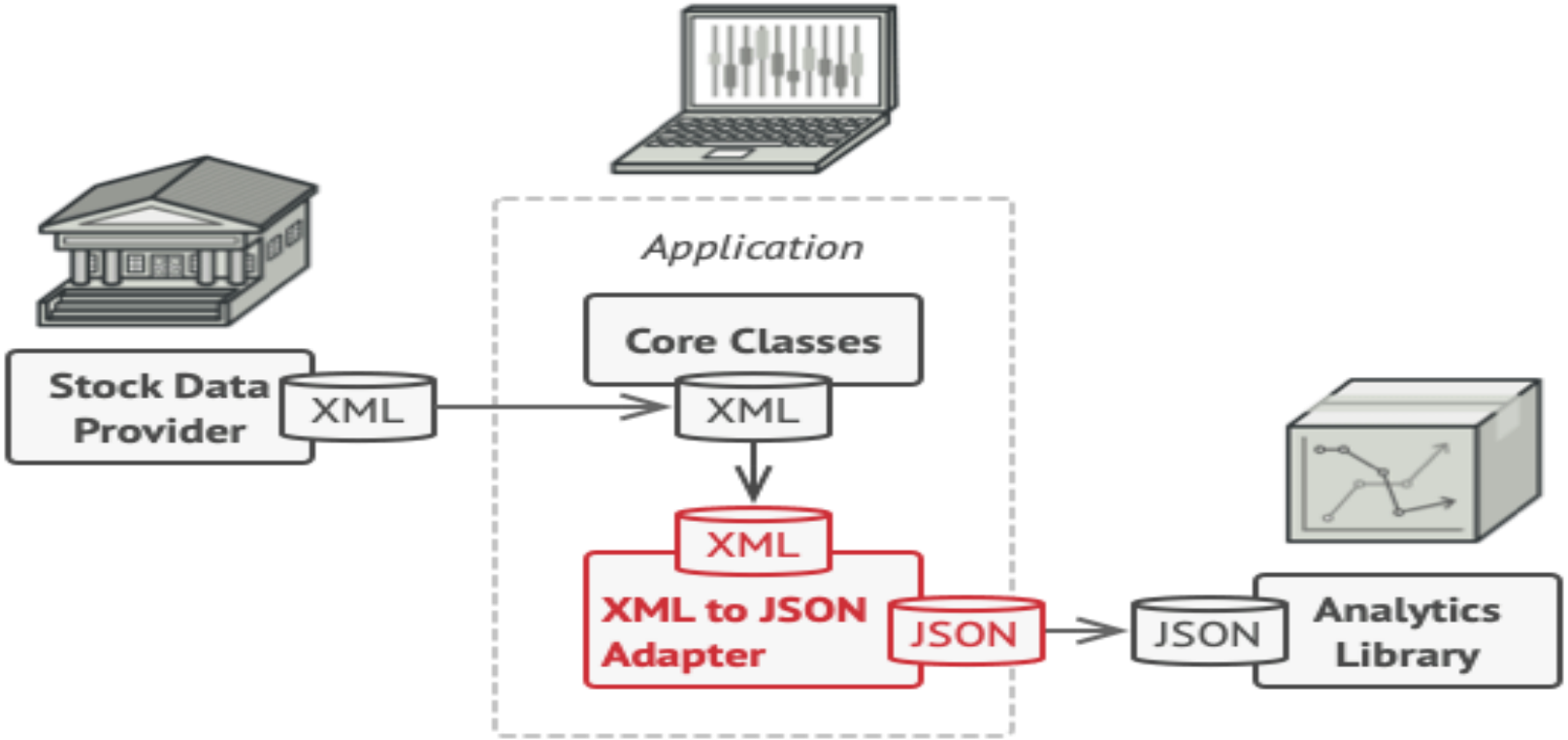
Adapter - Problem

- Imagine that you're creating a **stock market monitoring** app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



Adapter – Solution

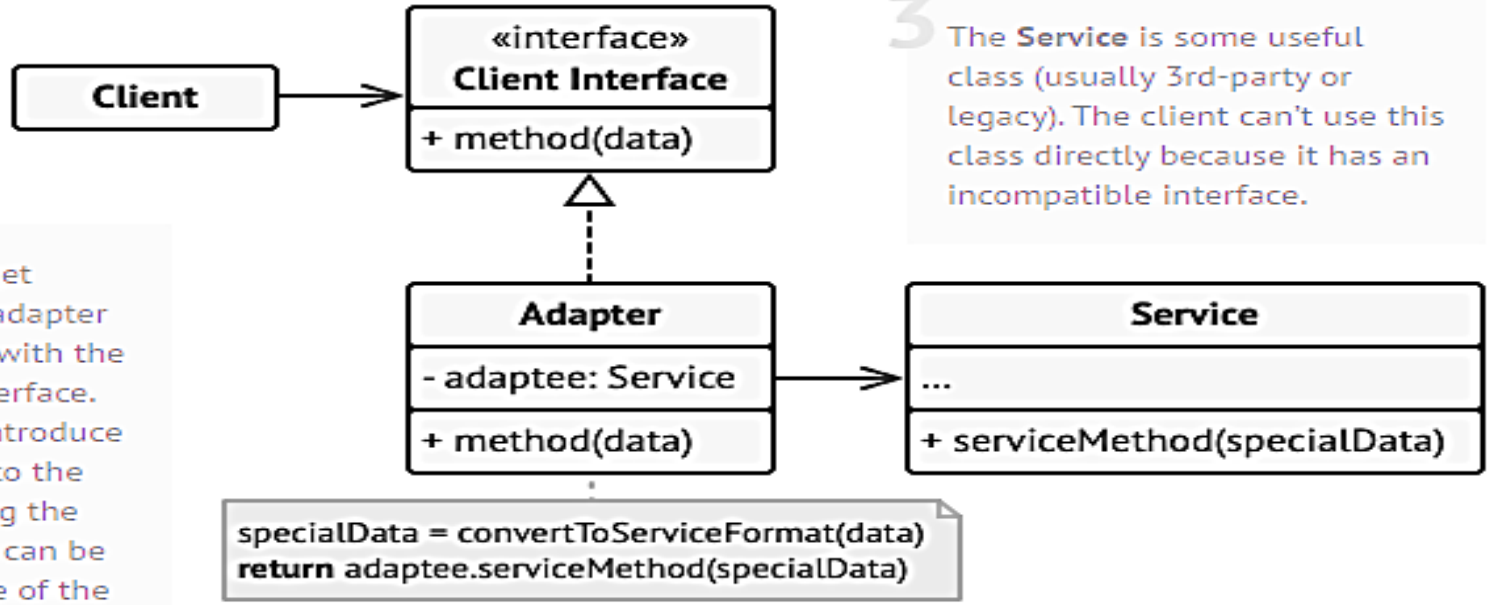
- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.



Adapter – Solution

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.



3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Adapter – Example

Interface
Concrete

```php
interface Notification
{
    public function send(string $title, string $message);
}

class EmailNotification implements Notification
{
    private $adminEmail;

    public function __construct(string $adminEmail)
    {
        $this->adminEmail = $adminEmail;
    }

    public function send(string $title, string $message): void
    {
        mail($this->adminEmail, $title, $message);
        echo "Sent email with title '$title' to '{$this->adminEmail}' that say
    }
}
```

Let's break down the provided code snippet that illustrates the Adapter design pattern using an interface and a concrete class. This example demonstrates how to create a notification system that can send messages via email.

Explanation of the Code

1. Notification Interface

```php
interface Notification {
    public function send(string $title, string $message);
}
```

- **Interface Definition**: The `Notification` interface defines a contract for sending notifications. It declares a method `send()` that takes two parameters: `$title` and `$message`.
- **Purpose**: Any class that implements this interface must provide a concrete implementation of the `send()` method, ensuring a consistent approach to sending notifications.

```php
class EmailNotification implements Notification {
    private $adminEmail;

    public function __construct(string $adminEmail) {
        $this->adminEmail = $adminEmail;
        echo "EmailNotification created for email: {$this->adminEmail}\n";
    }

    public function send(string $title, string $message): void {
        mail($this->adminEmail, $title, $message);
        echo "Sent email with title '$title' to '{$this->adminEmail}' that says: '$messag
e'\n";
    }
}
```

- **Class Definition**: The `EmailNotification` class implements the `Notification` interface, meaning it must provide a concrete implementation of the `send()` method.
- **Private Property**: `private $adminEmail;` stores the email address of the administrator who will receive notifications.
- **Constructor**:
  - `public function __construct(string $adminEmail)` initializes the `adminEmail` property.
  - It also outputs a message indicating that the `EmailNotification` has been created for the specified email.
- **send() Method**:
  - `public function send(string $title, string $message): void` implements the method from the `Notification` interface.
  - It uses the PHP `mail()` function to send an email with the specified title and message to the `adminEmail`.
  - After sending the email, it outputs a confirmation message.

How the Adapter Pattern Works in This Context

The Adapter pattern allows you to convert the interface of a class into another interface that clients expect. In this case, the `Notification` interface serves as the target interface, and the `EmailNotification` class acts as the adapter.

Benefits of Using the Adapter Pattern

1. **Flexibility**: By programming to an interface rather than a concrete implementation, you can easily switch out the notification method. For example, you could create another class, `SMSNotification`, that implements the `Notification` interface to send messages via SMS instead.
2. **Decoupling**: The client code that uses the `Notification` interface does not need to know about the specifics of how notifications are sent, promoting a cleaner separation of concerns.
3. **Ease of Maintenance**: If you need to change the way notifications are sent (e.g., switching from email to SMS), you only need to implement a new class that adheres to the `Notification` interface without altering the client code.

Example Usage

Here's how you might use the `EmailNotification` class in practice:

```php
// Create an instance of EmailNotification
$emailNotification = new EmailNotification("admin@example.com");

// Send a notification
$emailNotification->send("Test Title", "This is a test message.");
```

Summary

This example demonstrates the implementation of the Adapter design pattern through an interface (`Notification`) and a concrete adapter class (`EmailNotification`). By adhering to the interface, the `EmailNotification` class provides a specific way to send notifications via email while allowing for future extensions (like SMS notifications) without modifying existing code. This approach enhances code flexibility, maintainability, and scalability.

Adapter – Example

Third parity Service

```php
class SlackApi
{
    private $login;
    private $apiKey;

    public function __construct(string $login, string $apiKey)
    {
        $this->login = $login;
        $this->apiKey = $apiKey;
    }

    public function logIn(): void
    {
        echo "Logged in to a slack account '{$this->login}'.\n";
    }

    public function sendMessage(string $chatId, string $message): void
    {
        echo "Posted following message into the '$chatId' chat: '$message'.\n";
    }
}
```

Let's dive into the provided code snippet that illustrates the Adapter design pattern using a third-party service, specifically a Slack API integration. This example demonstrates how to create a class that can send messages to Slack while adhering to a defined interface.

Explanation of the Code

1. Class Definition: SlackApi

```php
class SlackApi {
    private $login;
    private $apiKey;

    public function __construct(string $login, string $apiKey) {
        $this->login = $login;
        $this->apiKey = $apiKey;
    }

    public function logIn(): void {
        echo "Logged into a Slack account '{$this->login}'.\n";
    }

    public function sendMessage(string $chatId, string $message): void {
        echo "Posted following message into the '{$chatId}' chat: '$message'\n";
    }
}
```

Example of How This Fits into an Adapter Pattern

To fully illustrate the Adapter pattern, you would define an interface and then create an adapter class that uses the `SlackApi`. Here's a simple example:

2. Notification Interface

```php
interface Notification {
    public function send(string $chatId, string $message);
}
```

3. SlackNotification Adapter Class

```php
class SlackNotification implements Notification {
    private $slackApi;

    public function __construct(SlackApi $slackApi) {
        $this->slackApi = $slackApi;
        $this->slackApi->logIn(); // Log in through the API
    }

    public function send(string $chatId, string $message): void {
        $this->slackApi->sendMessage($chatId, $message);
    }
}
```

Example Usage

Here's how you might use the `SlackNotification` adapter with the `SlackApi`:

```php
// Create an instance of SlackApi
$slackApi = new SlackApi("your_login", "your_api_key");

// Create an adapter for notifications
$slackNotification = new SlackNotification($slackApi);

// Send a notification
$slackNotification->send("general", "Hello, Slack!");
```

Summary

In summary, this example demonstrates how to use the Adapter design pattern with a third-party service like Slack. The `SlackApi` class interacts directly with the Slack API, while the `SlackNotification` adapter conforms to the `Notification` interface, allowing for seamless integration into a larger notification system. This approach enhances flexibility, maintainability, and adherence to the single responsibility principle, making your codebase easier to manage as it evolves.

- **Class Definition**: `SlackApi` represents a third-party service for sending messages to Slack.
- **Private Properties**:
  - `private $login;` stores the login information for the Slack account.
  - `private $apiKey;` stores the API key required to authenticate with the Slack API.
- **Constructor**:
  - `public function __construct(string $login, string $apiKey)` initializes the `login` and `apiKey` properties with the provided values.
- **logIn() Method**:
  - `public function logIn(): void` simulates logging into a Slack account and outputs a confirmation message that includes the login.
- **sendMessage() Method**:
  - `public function sendMessage(string $chatId, string $message): void` simulates sending a message to a specific chat identified by `chatId`.
  - It outputs a message indicating that the specified message has been posted to the given chat.

Adapter Pattern Context

In this context, the `SlackApi` class could be adapted to fit into a larger notification system that uses an interface for sending messages. The Adapter pattern allows you to use the `SlackApi` class without modifying its implementation, as long as you create a wrapper that conforms to a defined interface.

Benefits of Using the Adapter Pattern

1. **Interoperability**: The Adapter pattern allows different systems to work together. By adapting the `SlackApi` class to a common interface, other parts of your application can send messages without needing to know the specifics of how Slack integration works.
2. **Decoupling**: The client code can remain decoupled from the specifics of the `SlackApi` implementation, making it easier to maintain and modify. If you decide to switch to a different messaging service, you can create a new adapter without affecting the existing codebase.
3. **Single Responsibility**: The `SlackApi` class is focused solely on interacting with the Slack API, while the adapter class can handle the specifics of how that interaction fits into your application's broader messaging system.

Adapter – Example

Adapter

```php
class SlackNotification implements Notification
{
    private $slack;
    private $chatId;

    public function __construct(SlackApi $slack, string $chatId)
    {
        $this->slack = $slack;
        $this->chatId = $chatId;
    }
    public function send(string $title, string $message): void
    {
        $slackMessage = "#" . $title . "# " . strip_tags($message);
        $this->slack->logIn();
        $this->slack->sendMessage($this->chatId, $slackMessage);
    }
}
```

Let's break down the provided code snippet that illustrates the Adapter design pattern, specifically focusing on the `SlackNotification` class, which adapts the `SlackApi` class to a common `Notification` interface.

Explanation of the Code

1. SlackNotification Class

```php
class SlackNotification implements Notification {
    private $slack;
    private $chatId;

    public function __construct(SlackApi $slack, string $chatId) {
        $this->slack = $slack;
        $this->chatId = $chatId;
    }

    public function send(string $title, string $message): void {
        $slackMessage = "" . $title . " - " . strip_tags($message);
        $this->slack->logIn();
        $this->slack->sendMessage($this->chatId, $slackMessage);
    }
}
```

Breakdown of the Code

1. Class Definition

- **Class Declaration**: `class SlackNotification implements Notification` indicates that `SlackNotification` is implementing the `Notification` interface. This means it must provide an implementation for the methods defined in that interface.

2. Properties

- **Private Properties**:
  - `private $slack;`: This property holds an instance of the `SlackApi` class, which provides the methods for interacting with the Slack API.
  - `private $chatId;`: This property stores the ID of the chat where messages will be sent.

3. Constructor

```php
public function __construct(SlackApi $slack, string $chatId) {
    $this->slack = $slack;
    $this->chatId = $chatId;
}
```

- **Dependency Injection**: The constructor takes an instance of `SlackApi` and a chat ID as parameters. This is an example of dependency injection, which promotes flexibility and testability.
- **Property Initialization**: The constructor initializes the `$slack` and `$chatId` properties with the provided arguments.

4. Send Method

```php
public function send(string $title, string $message): void {
    $slackMessage = "" . $title . " - " . strip_tags($message);
    $this->slack->logIn();
    $this->slack->sendMessage($this->chatId, $slackMessage);
}
```

- **Method Signature**: The `send` method takes two parameters: `$title` and `$message`, both of which are strings.
- **Message Formatting**:
  - The method constructs the `slackMessage` by concatenating the title and the message, ensuring to strip any HTML tags from the message using `strip_tags()`. This helps prevent any unintended HTML from being sent in the message.
- **Logging In**:
  - The method calls `$this->slack->logIn()`, which simulates logging into Slack. This is necessary before sending a message, although in a real-world scenario, you might want to manage authentication differently to avoid logging in every time you send a message.
- **Sending the Message**:
  - Finally, it calls `$this->slack->sendMessage($this->chatId, $slackMessage)`, sending the constructed message to the specified chat.

The Adapter Pattern Context

The `SlackNotification` class serves as an adapter that allows the `SlackApi` class to fit into a broader notification system. By implementing the `Notification` interface, `SlackNotification` can be used interchangeably with other notification types, such as email or SMS, as long as they also implement the same interface.

Benefits of Using the Adapter Pattern

1. **Interoperability**: The Adapter pattern allows different systems (like Slack, SMS, email) to be used interchangeably as long as they conform to the same interface.
2. **Decoupling**: The client code can send notifications without needing to know the specifics of how those notifications are sent. This makes the codebase cleaner and easier to maintain.
3. **Flexibility**: If you decide to switch to a different messaging service or add another notification method, you can create a new adapter class that implements the `Notification` interface without modifying the existing code.
4. **Single Responsibility**: `SlackNotification` is responsible for handling Slack-specific logic, while the `SlackApi` deals with the actual API calls. This separation of concerns makes the code easier to test and maintain.

Example Usage

Here's how you might use the `SlackNotification` adapter in practice:

```php
// Create an instance of SlackApi
$slackApi = new SlackApi("your_login", "your_api_key");

// Create an instance of SlackNotification
$slackNotification = new SlackNotification($slackApi, "your_chat_id");

// Send a notification
$slackNotification->send("Alert", "This is a test message.");
```

Summary

The `SlackNotification` class is a clear example of the Adapter design pattern in action. It adapts the `SlackApi` to fit within a broader notification system by implementing the `Notification` interface. This allows for greater flexibility, maintainability, and clarity in the codebase, making it easier to manage and extend as needed. The Adapter pattern is particularly useful in scenarios where you need to integrate with third-party services while maintaining a consistent interface across different implementations.

Adapter – Example

Client

```php
function clientCode(Notification $notification)
{
    // ...

    echo $notification->send("Website is down!",
        "<strong style='color:red;font-size: 50px;'>Alert!</strong> " .
        "Our website is not responding. Call admins and bring it up!");

    // ...
}

echo "Client code is designed correctly and works with email notifications:\n";
$notification = new EmailNotification("developers@example.com");
clientCode($notification);
echo "\n\n";


echo "The same client code can work with other classes via adapter:\n";
$slackApi = new SlackApi("example.com", "XXXXXXXX");
$notification = new SlackNotification($slackApi, "Example.com Developers");
clientCode($notification);
```

Let's break down the provided code snippet that demonstrates how a client interacts with an adapter in the context of the Adapter design pattern. This example showcases how a client can utilize different notification methods (like email and Slack) interchangeably through a consistent interface.

Explanation of the Code

1. Client Function

```php
function clientCode(Notification $notification) {
    echo $notification->send("Website is down!", "<strong style='color:red;font-size: 50px;'>Alert!</strong> The website is not responding. Call admins and bring it up!");
}
```

Breakdown of the Code

- **Function Definition**:
  - `function clientCode(Notification $notification)` defines a function that takes a `Notification` object as its parameter. This function expects any object that implements the `Notification` interface, allowing for flexibility in the type of notification being sent.
- **Sending Notification**:
  - Inside the function, `$notification->send(...)` is called, passing a title and a message as arguments.
  - The message includes an HTML `<strong>` tag to style the alert, demonstrating that the message can contain formatted text.

Conclusion

This example showcases how the Adapter design pattern allows for flexible and interchangeable use of different notification methods through a consistent interface. The `clientCode` function remains unchanged regardless of the underlying notification implementation, demonstrating the power of abstraction and decoupling in software design. This approach not only enhances maintainability but also encourages a modular architecture that can easily adapt to new requirements.

2. Example Client Code

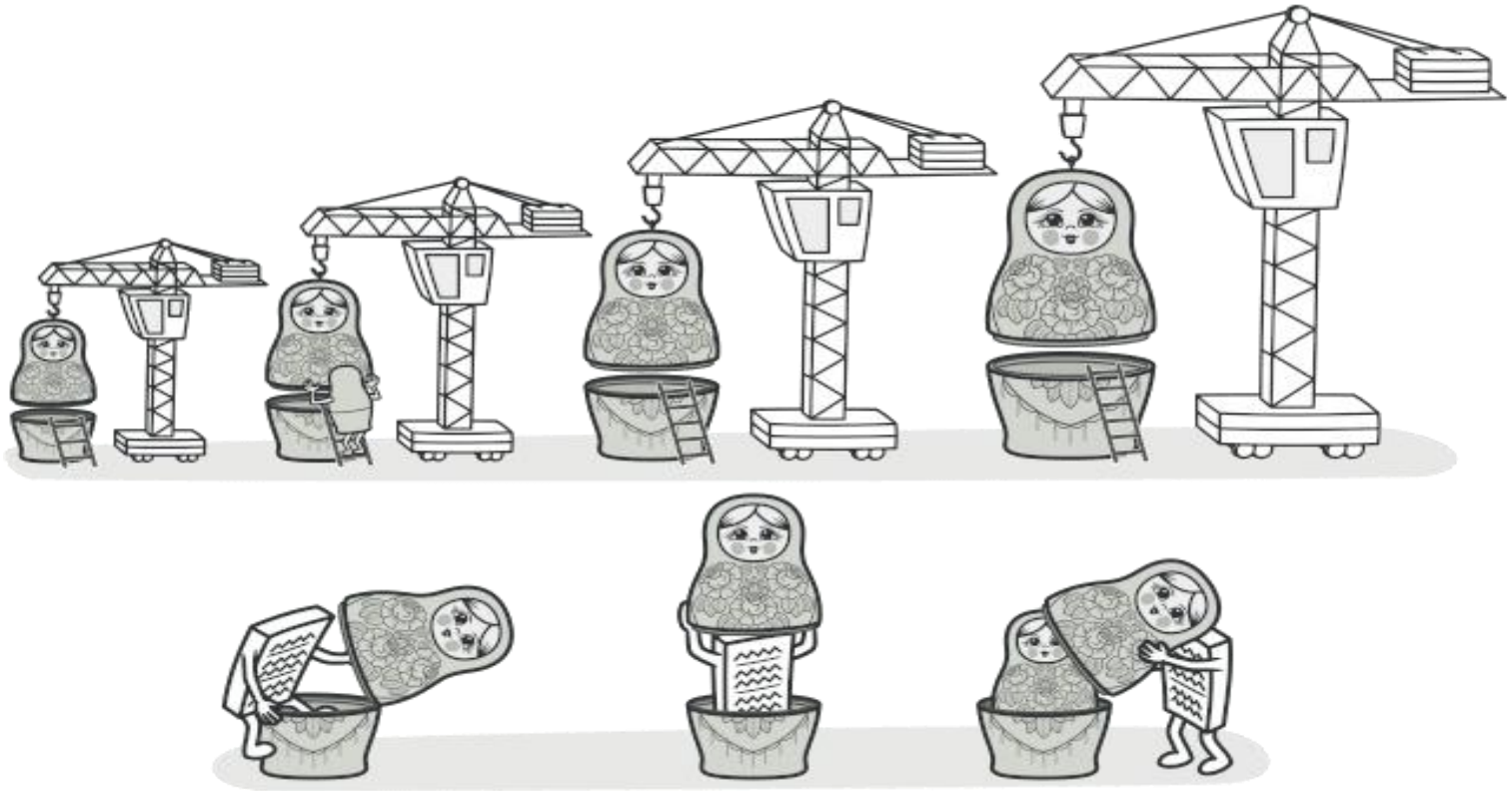
```php
echo "Client code is designed correctly and works with email notifications:\n";
$emailNotification = new EmailNotification(); // Assume this class implements Notification
clientCode($emailNotification);

echo "\nClient code can work with other classes via adaptation:\n";
$slackApi = new SlackApi("example.com", "your_api_key");
$notification = new SlackNotification($slackApi, "Example on Developers");
clientCode($notification);
```

Breakdown of the Example Client Code

1. **Email Notification**:
   - `echo "Client code is designed correctly and works with email notifications:\n";` outputs a message indicating that the client code will demonstrate email notifications.
   - `clientCode($emailNotification);` calls the `clientCode` function with an instance of `EmailNotification`, which is assumed to implement the `Notification` interface. This shows that the client code can work with email notifications seamlessly.
2. **Slack Notification**:
   - `echo "\nClient code can work with other classes via adaptation:\n";` indicates that the following example will demonstrate the use of a Slack notification.
   - `new SlackApi("example.com", "your_api_key");` creates an instance of the `SlackApi`, which is the third-party API for sending messages to Slack.
   - `new SlackNotification($slackApi, "Example on Developers");` creates an instance of the `SlackNotification` adapter, which adapts the `SlackApi` to the `Notification` interface.
   - `clientCode($notification);` calls the same `clientCode` function with the `SlackNotification` instance, demonstrating that the same client code can handle different types of notifications without modification.

Decorator

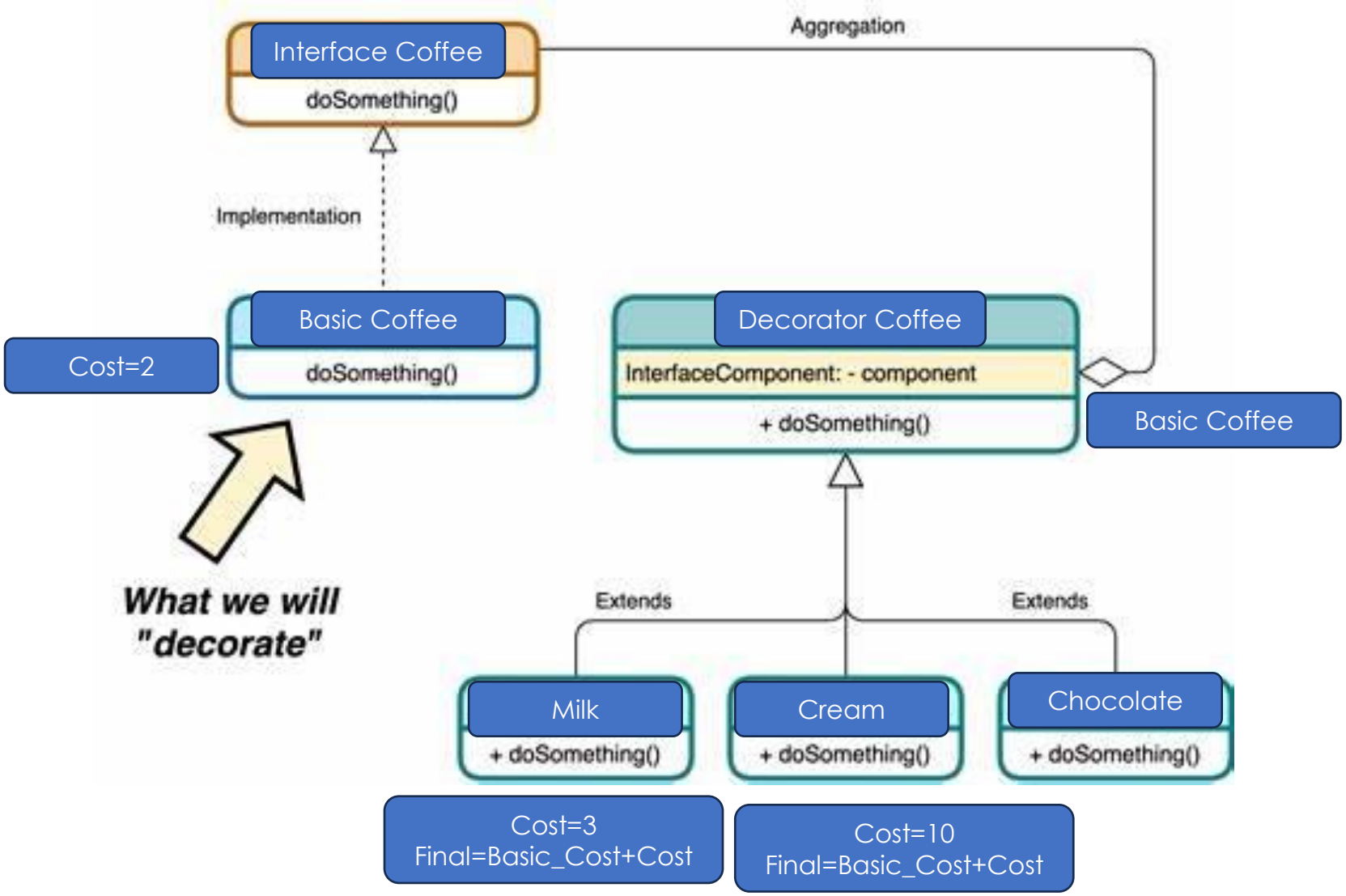
- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



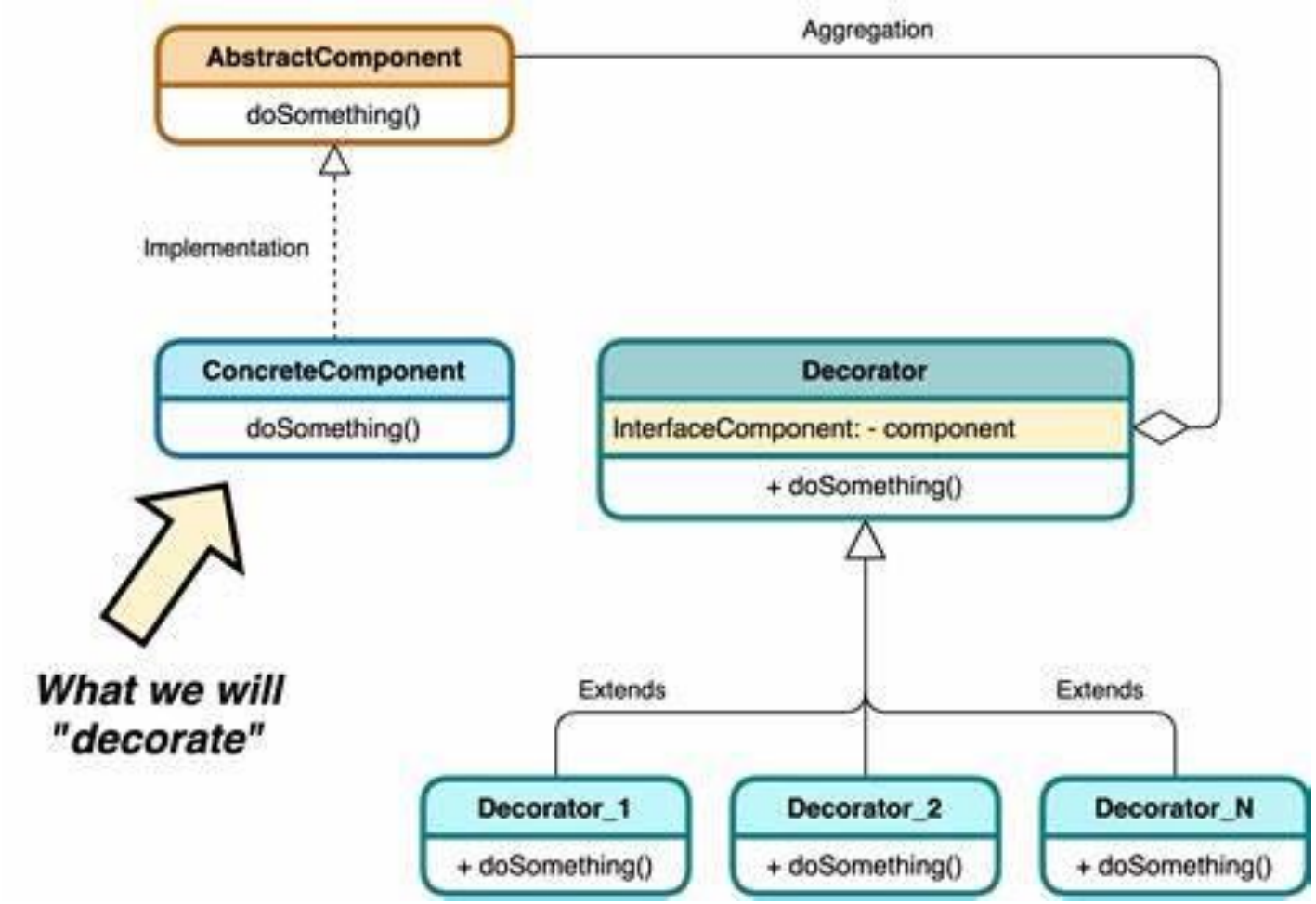
Decorator - Problem

- If you have a basic object, like a **cup of coffee**, and you **want to add different types of flavors to it**, such as *whipped cream* or *chocolate syrup*, the Decorator Design Pattern can be used.

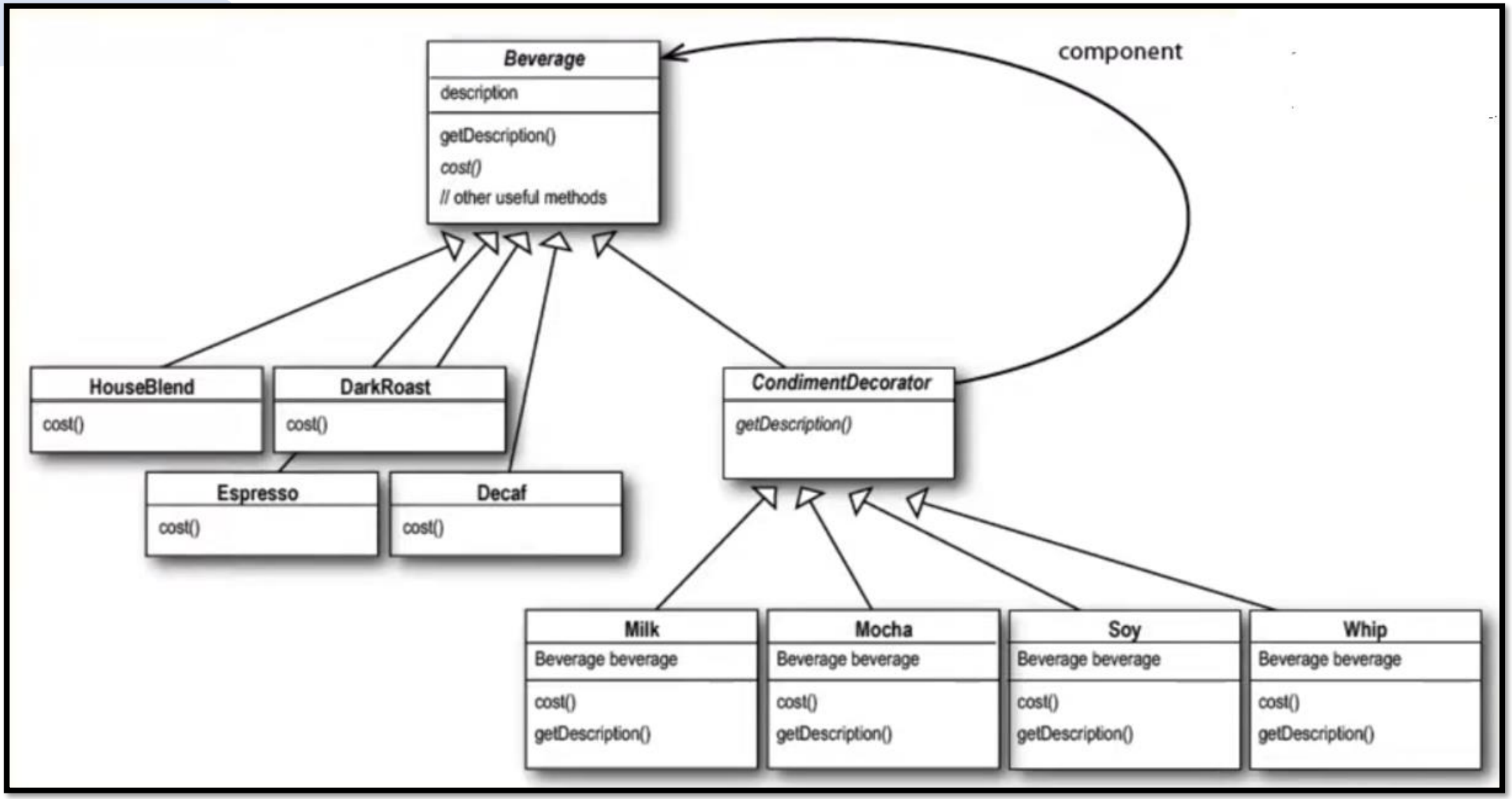
Decorator-Solution



Decorator-Solution



Decorator-Solution



Decorator-Solution

```php
interface Coffee
{
    public function getCost(): int;
    public function getDescription(): string;
}
```

```php
class SimpleCoffee implements Coffee
{
    public function getCost(): int
    {
        return 10;
    }

    public function getDescription(): string
    {
        return 'Simple Coffee';
    }
```

```php
abstract class CoffeeDecorator implements Coffee
{
    protected $decoratedCoffee;

    public function __construct(Coffee $decoratedCoffee)
    {
        $this->decoratedCoffee = $decoratedCoffee;
    }
}
```

Decorator-Solution

```php
class CreamCoffee extends CoffeeDecorator
{
    private const PRICE = 5;

    public function getCost(): int
    {
        return $this->decoratedCoffee->getCost() + self::PRICE;
    }

    public function getDescription(): string
    {
        return $this->decoratedCoffee->getDescription() . ', cream';
    }
}
```

```php
class MilkCoffee extends CoffeeDecorator
{
    private const PRICE = 2;

    public function getCost(): int
    {
        return $this->decoratedCoffee->getCost() + self::PRICE;
    }

    public function getDescription(): string
    {
        return $this->decoratedCoffee->getDescription() . ', milk';
    }
}
```

Decorator-Solution

```
$simpleCoffee = new SimpleCoffee();
echo $simpleCoffee->getCost();
echo $simpleCoffee->getDescription();

// Order coffee with milk
$milkCoffee = new MilkCoffee($simpleCoffee);
echo $milkCoffee->getCost();
echo $milkCoffee->getDescription();

// Order coffee with milk and cream
$mixCoffee = new CreamCoffee($milkCoffee);
echo $mixCoffee->getCost();
echo $mixCoffee->getDescription();
```

```
// Order simple coffee
$simpleCoffee = new SimpleCoffee();
echo $simpleCoffee->getCost(); // output: 10
echo $simpleCoffee->getDescription(); // out

// Order coffee with milk
$milkCoffee = new MilkCoffee($simpleCoffee);
echo $milkCoffee->getCost(); // output: 12 (
echo $milkCoffee->getDescription(); // outpu

// Order coffee with cream
$creamCoffee = new CreamCoffee($simpleCoffee
echo $creamCoffee->getCost();
echo $creamCoffee->getDescription();
```

Behavioral Design Pattern

Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

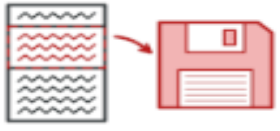
Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

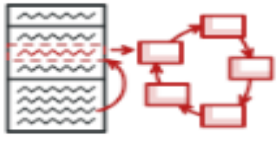


Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

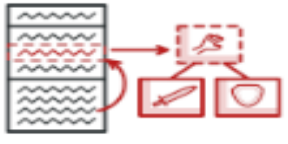
Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Strategy

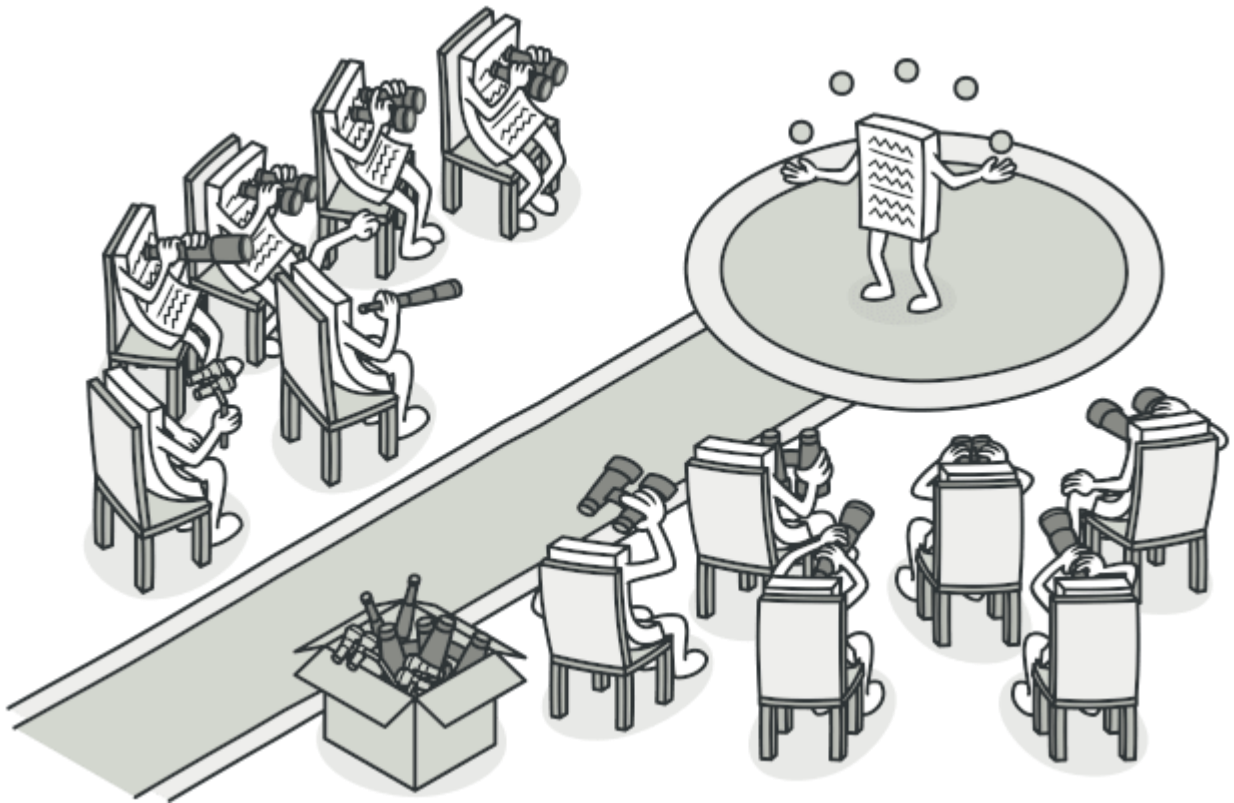
Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

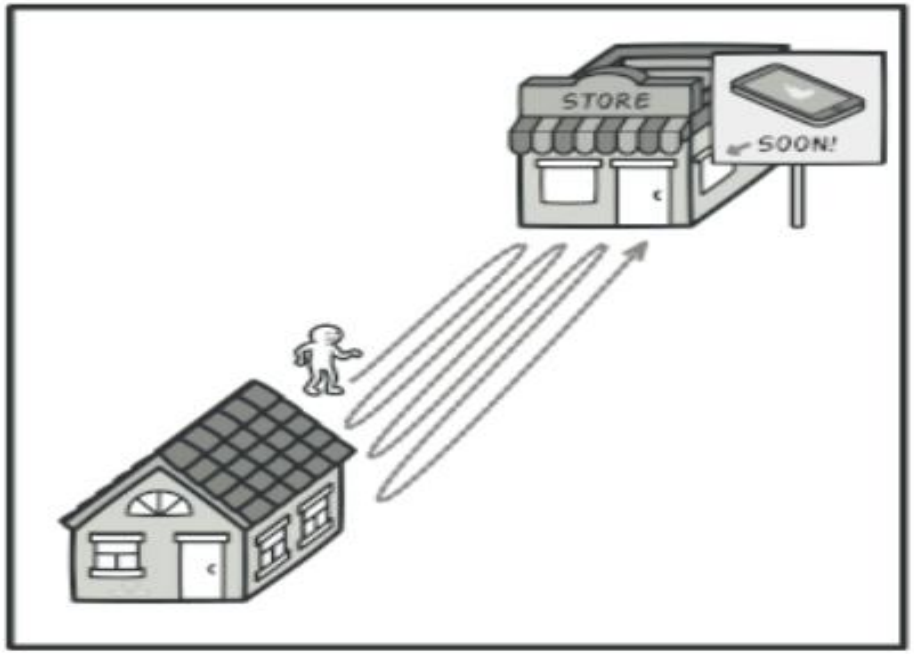
Observer

- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



Observer - problem

- Imagine that you have two types of objects: a *Customer* and a *Store*. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.
- The customer could visit the store every day and check product availability. But while the product is still in route, most of these trips would be pointless.



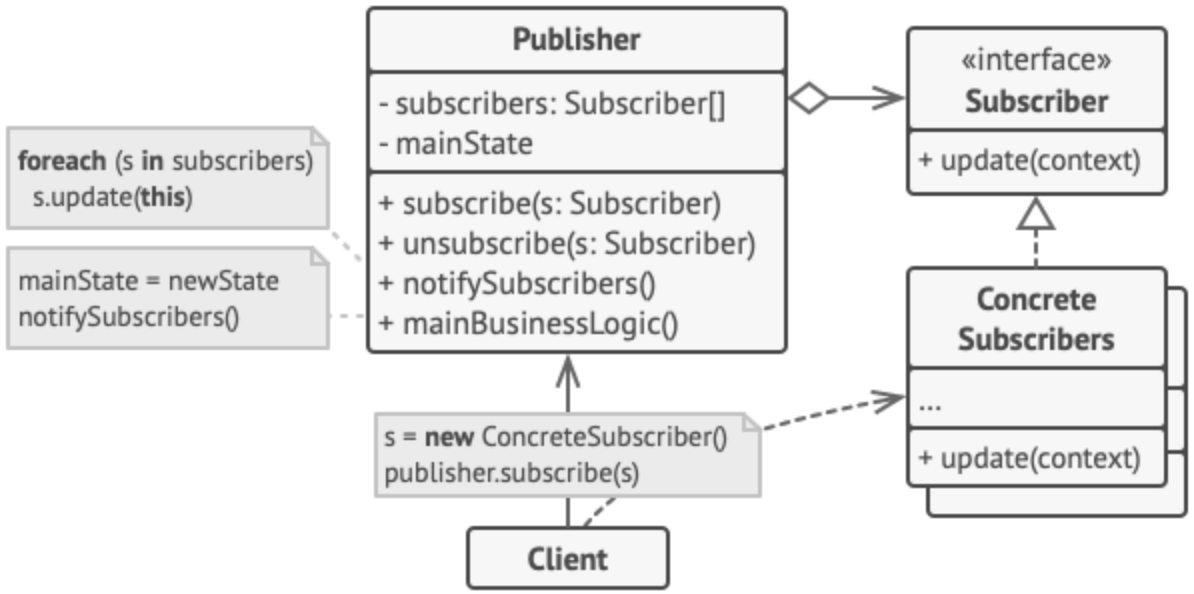
Observer - problem

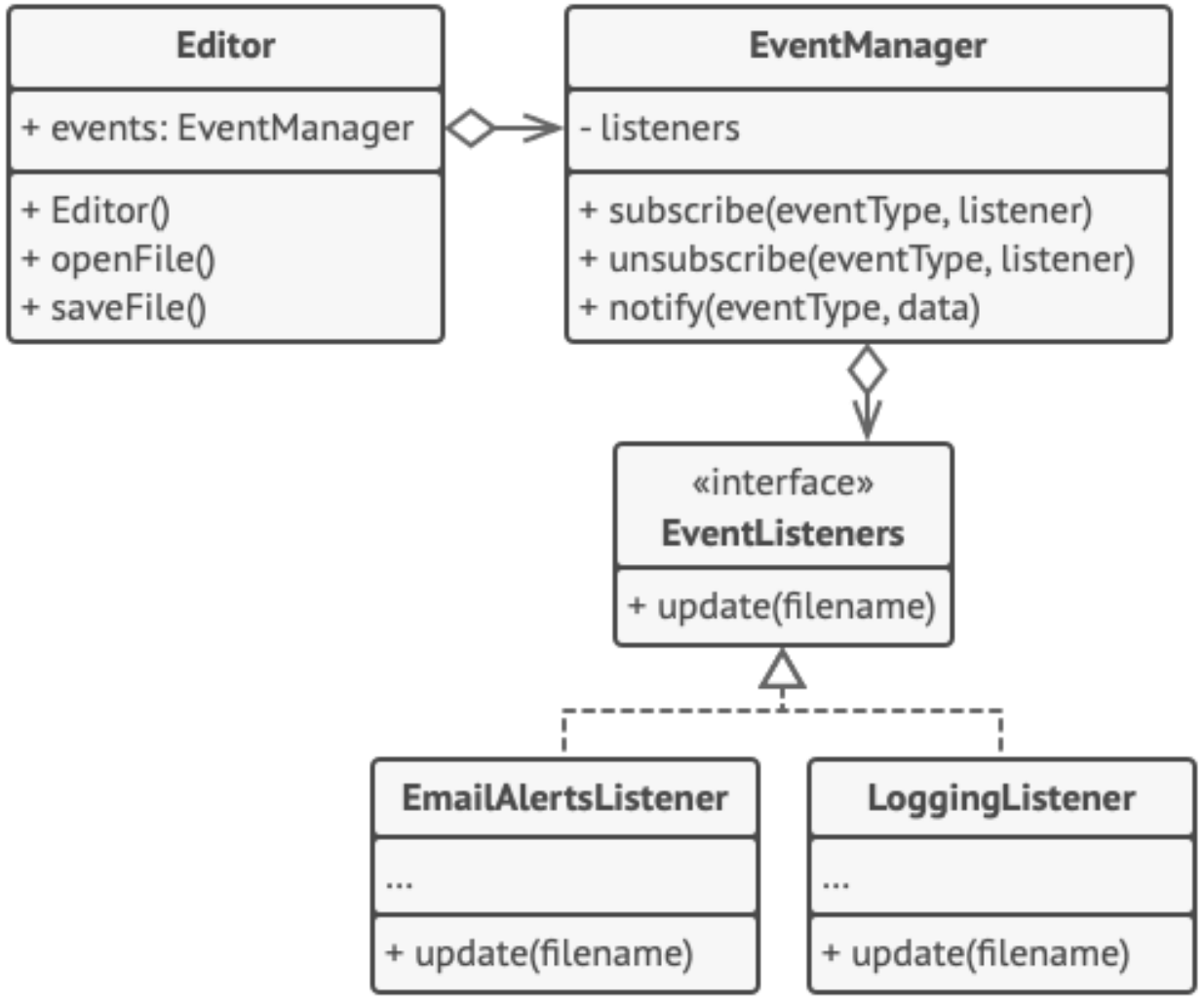
- On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.
- It looks like we've got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

Observer - solution

- The object that has some interesting state is often called ***subject***, but since it's also going to notify other objects about the changes to its state, we'll call it ***publisher***. All other objects that want to track changes to the publisher's state are called ***subscribers***.

Observer - solution





Observer - Pseudocode

Publisher

```
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)
```

Observer - Pseudocode

```
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Methods of business logic can notify subscribers about
    // changes.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

    // ...
```

Events - Observable

Observer - Pseudocode

Listener - Observer

```
interface EventListener is
    method update(filename)

// Concrete subscribers react to updates issued by the publisher
// they are attached to.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))
```

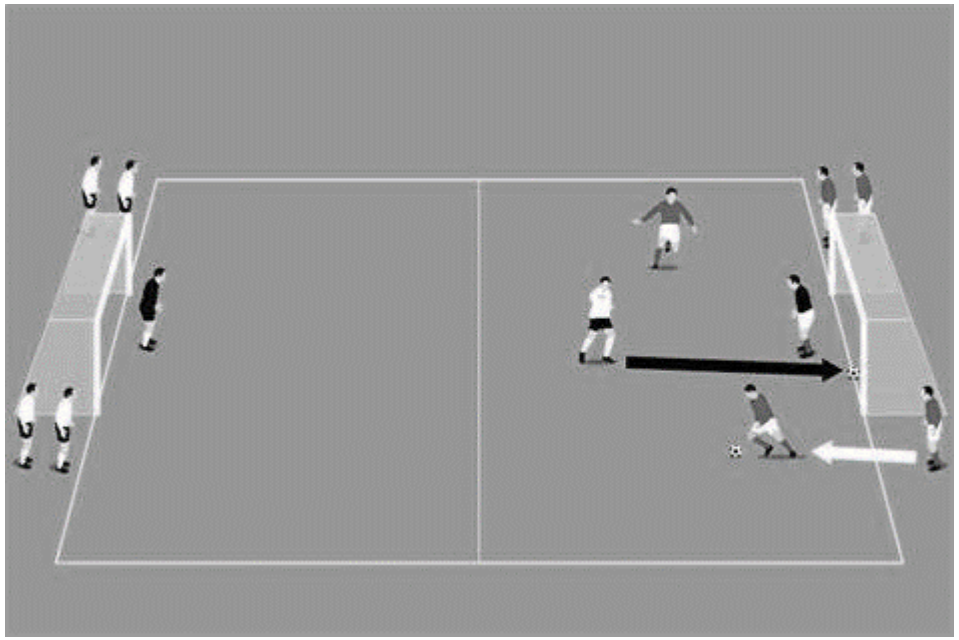
Observer - Pseudocode

Listener - Observer

```
class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))
```

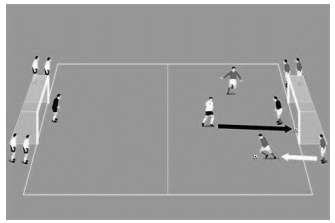
Strategy Design pattern

- Football match
  - Al Ahly (Attack strategy)
  - Al Zamalk (Defend strategy)
  - bonus attack +2
  - bonus Defend +1
  - Welcome attack (Attacker)
  - Welcome Defend (Defensive)
  - Match consists of two teams
  - Match has score



Class Match
-Score
-Team1
-Team2
+SetTeam1()
+SetTeam2()
+GetScore()
+Play()
{
If (Team1 is Ahly)
Ahly is attacker
Else if (Team1 is Zamalk)
Zamalk is defend
}
+ Welcome()
{
If (Team1 is Ahly)
Welcome (Ahly attacker)
Else if (Team1 is Zamalk)
Welcome (Zamalk is defensive)

}



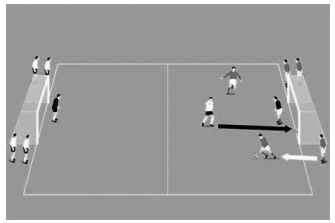
Example

Class Team
-name
-bonus
+set/get Name()
+set/get Bonus()

Class Attack
-name
+set/get Name()
+play()

Class defend
-name
+set/get Name()
+play()

```
Class Match
-Score, -Team1, -Team2
+SetTeam1()
+SetTeam2()
+Play()
{
        If (Team1 is Ahly)
                Ahly is attacker
        Else if (Team1 is Zamalk)
                Zamalk is defend
}
+ Welcome()
{
        If (Team1 is Ahly)
                Welcome (Ahly attacker)
        Else if (Team1 is Zamalk)
                Welcome (Zamalk is defensive)

}
+SetScore(winner)
{
        If (winner is Ahly)
                team1.bonus+2
        Else if (winner is Zamalk)
                team1.bonus+1
}
```



Example

Class Team
-name
-bonus
+set/get Name()
+set/get Bonus()

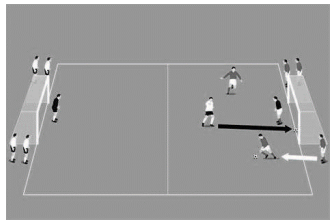
Abstract class
strategy
-name
+set/get Name()
+play()

Class Attack
extend strategy
+play()

Class defend
extend strategy
+play()

Violate OCP

Class Match
-Score, -Team1, -Team2
+SetTeam1(Strategy)
+SetTeam2(Strategy)
+Play()
{
Team1.play()
Team2.play()
}
+ Welcome()
{
Team1.welcome()
Team2.welcome()
}
+SetScore(winner)
{
Team.SetBonus()
Team2.SetBonus()
}



Class Team
-name
-bonus
-strategy
+set/get Name()
+set/get Bonus()
+set/get Strategy()
+Welcome()
{“hello”
+this.name
+strategy.name()
}
+SetBonus()
{strategy.SetScorePoint()}

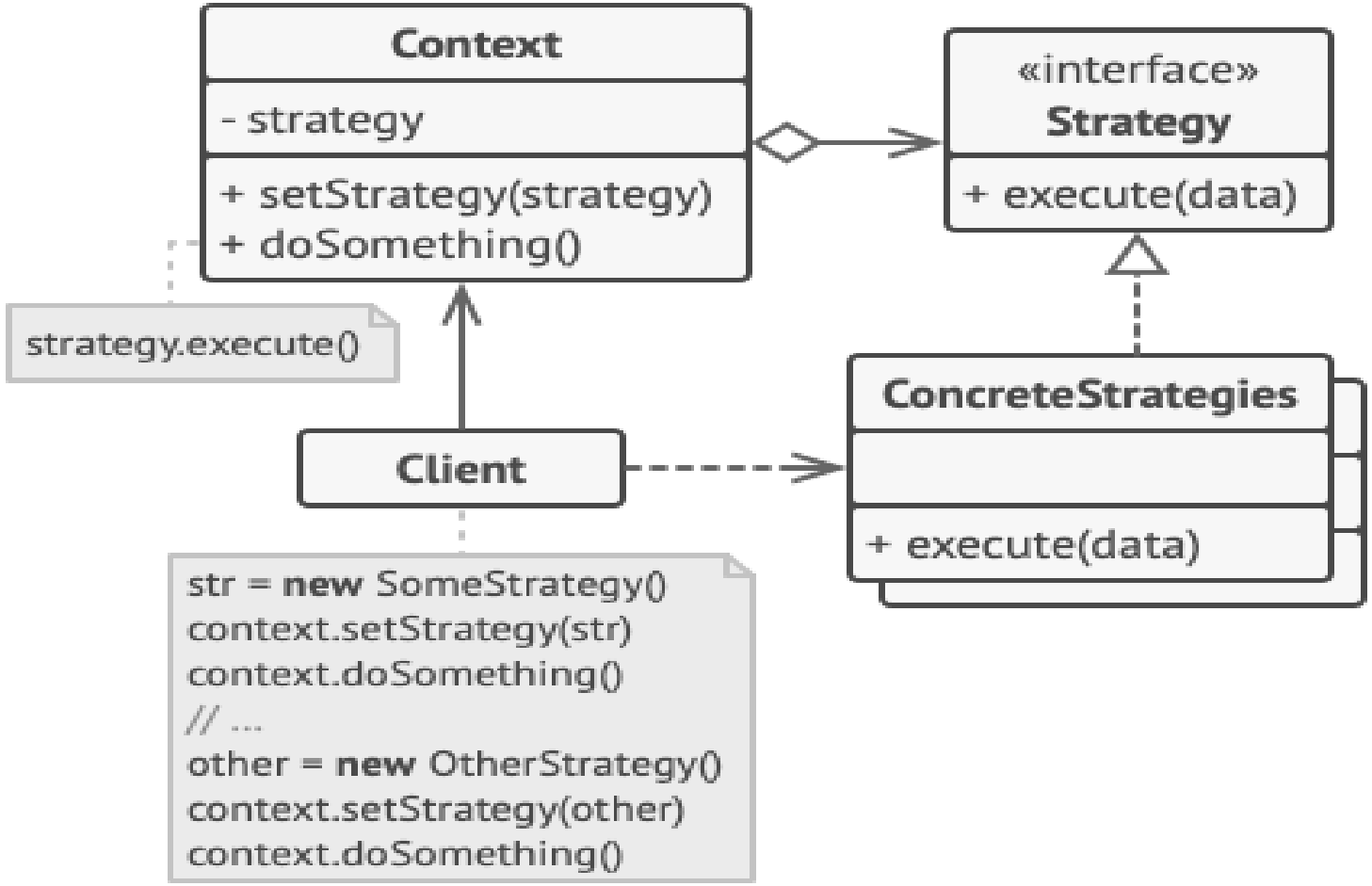
Solution

Abstract class strategy
-name
-ScorePoint
+set/get Name()
+set/get ScorePoint()
+play()

Class Attack
extend strategy
+play(){Attack}
SetScorePoint(){+2}

Class defend
extend strategy
+play() {defend}
SetScorePoint(){+1}

Solution Strategy Design pattern



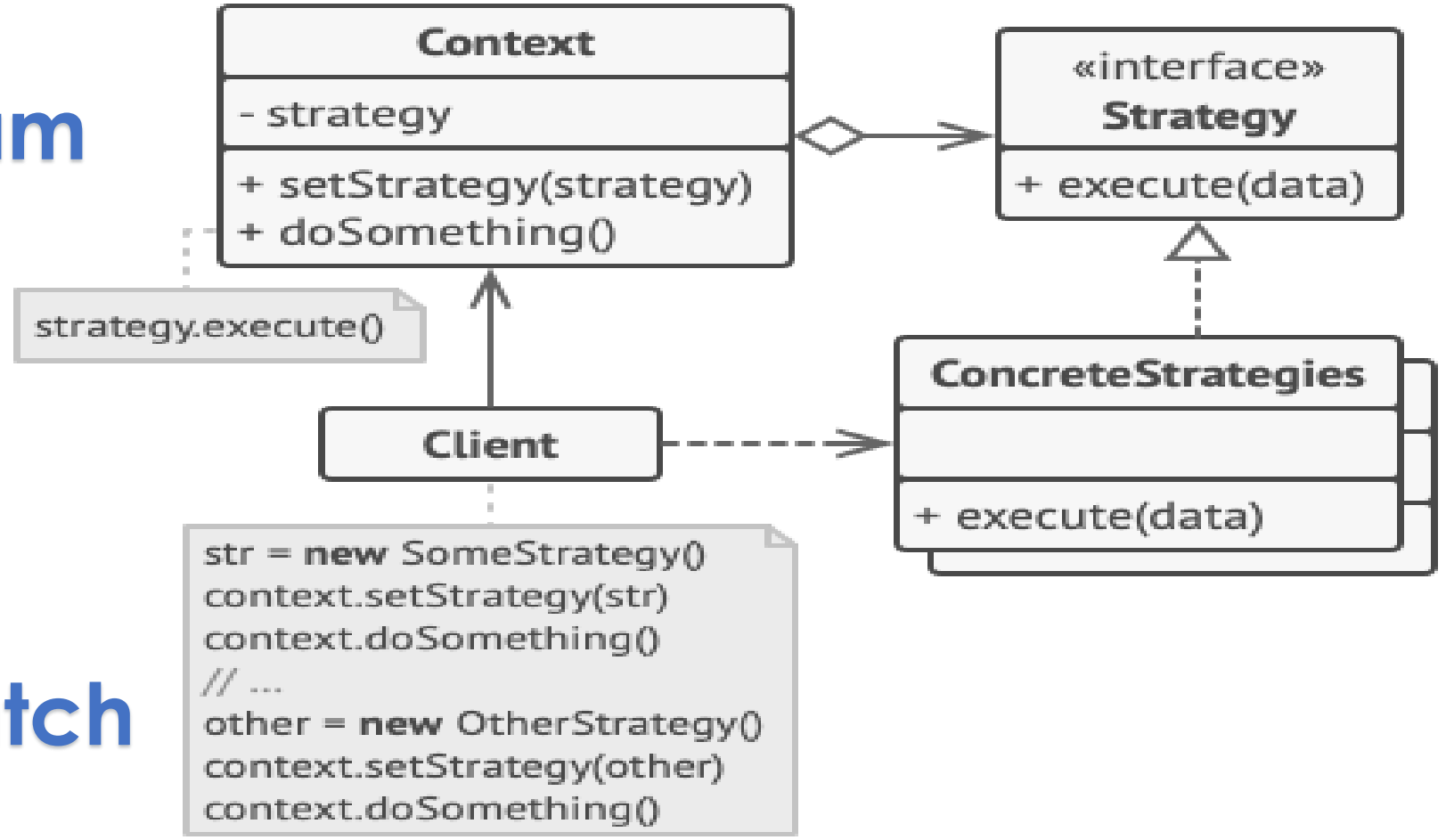
Strategy Design pattern

- The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.
- The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.
- **Concrete Strategies** implement different variations of an algorithm the context uses.

Solution Strategy Design pattern

Team

Strategy



Defend

Attack

Match