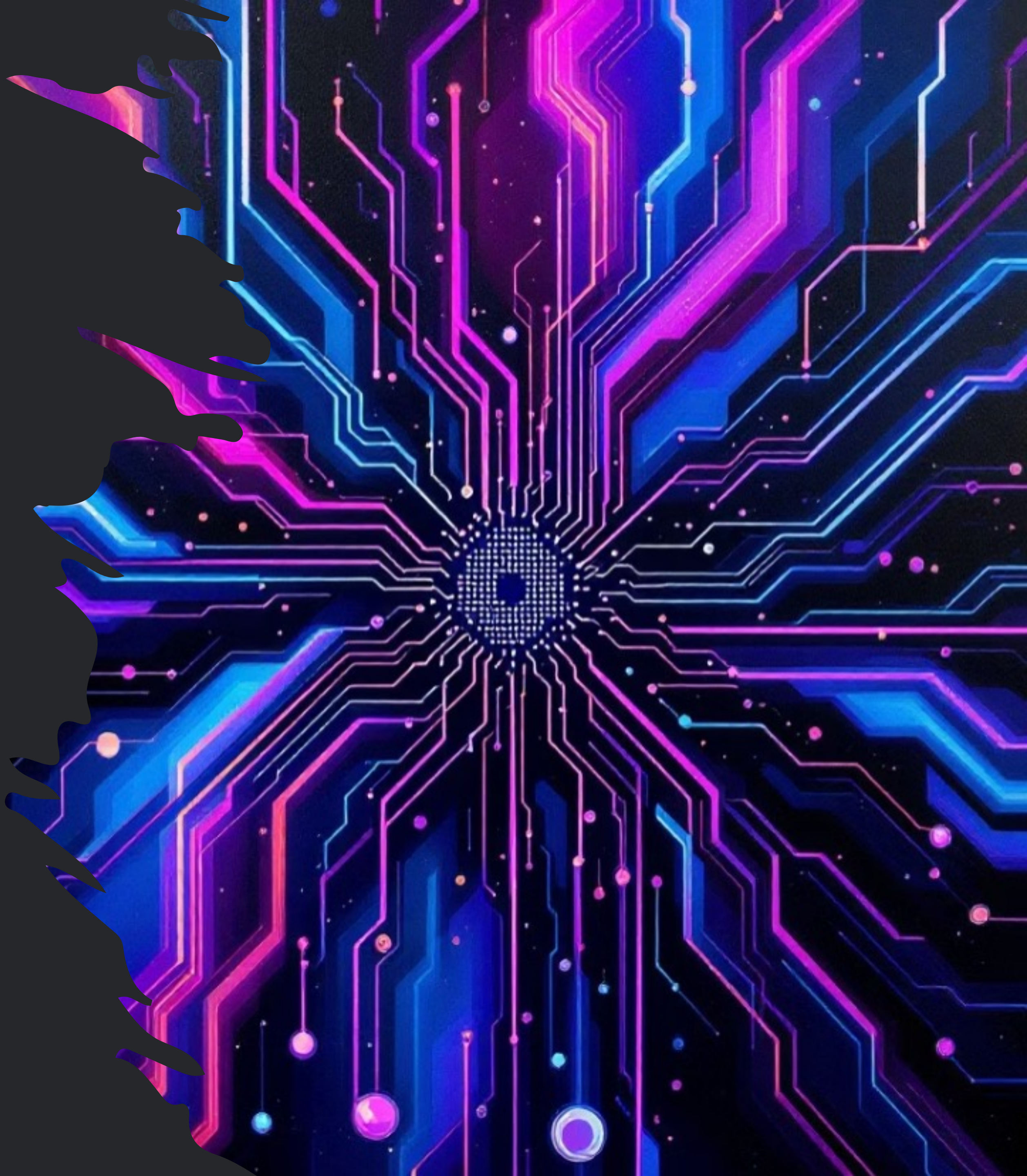
Design Patterns

Authors:

Iuliia Shestopalova

Jozef Hrdý



What Are Design Patterns?

Templates

Solve recurring problems in the design of software applications.

Object-Oriented

Describe interactions between classes and objects in object-oriented programming.

Flexible

Can be adapted to various programming languages and specific project needs.

Proven

These solutions have been tested and refined through real-world application.

A Brief History of Design Patterns

1

1960s: Architectural Roots

Christopher Alexander introduces pattern language for architectural design.

2

1987: OOPSLA Conference

First mention of design patterns in computing at Orlando conference.

3

1994: Gang of Four Book

Publication of "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma et al.

The Gang of Four (GoF) Patterns

23 Patterns

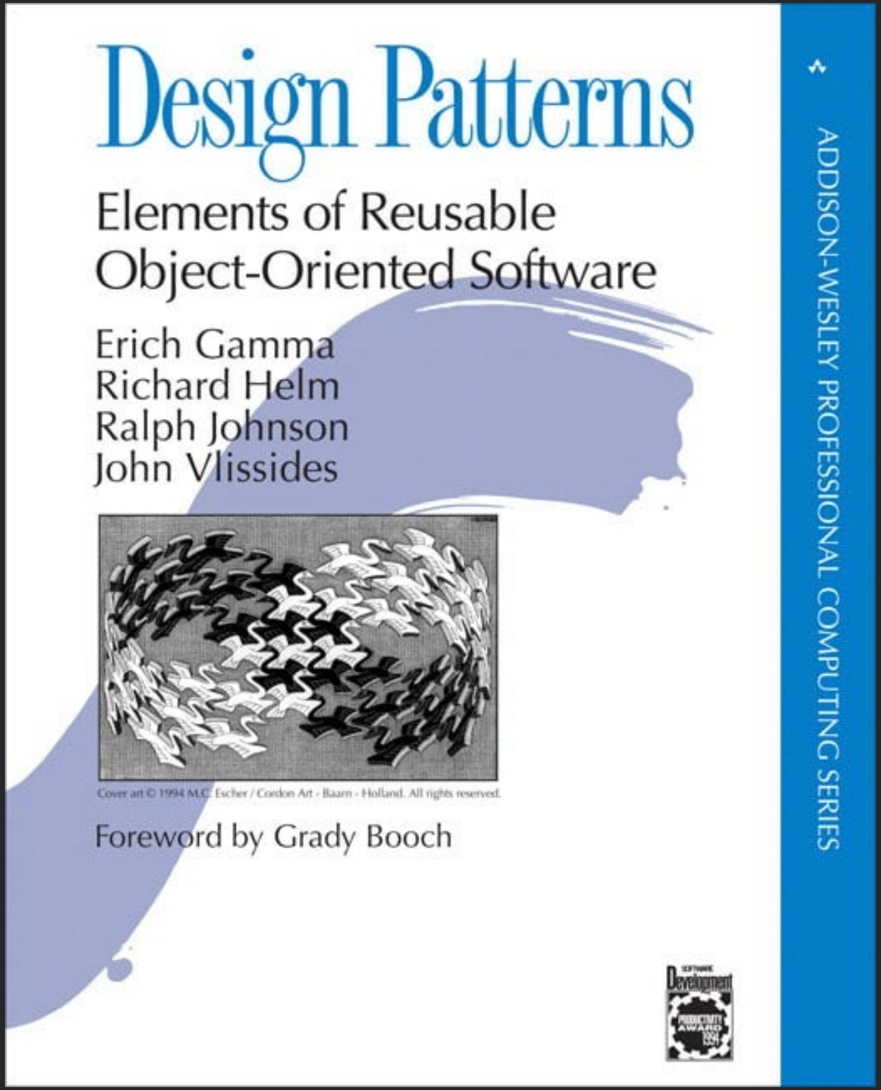
The GoF book introduced 23 fundamental design patterns.

Three Categories

Patterns are classified as **Creational, Structural, or Behavioral.**

Universal Application

These patterns have proven applicable across various programming languages and domains.



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns vs. Algorithms

Design Patterns

- Reusable solutions for common problems
- Provide blueprints for structuring code
- Examples: Singleton, Factory, Observer

Algorithms

- Step-by-step instructions for solving tasks
- Focus on efficiency and performance
- Examples: Sorting, Searching, Graph Traversal

JS

Benefits of Using Design Patterns

Reusability

Code reuse, reduced development time.

Maintainability

Easier to modify and update.

Scalability

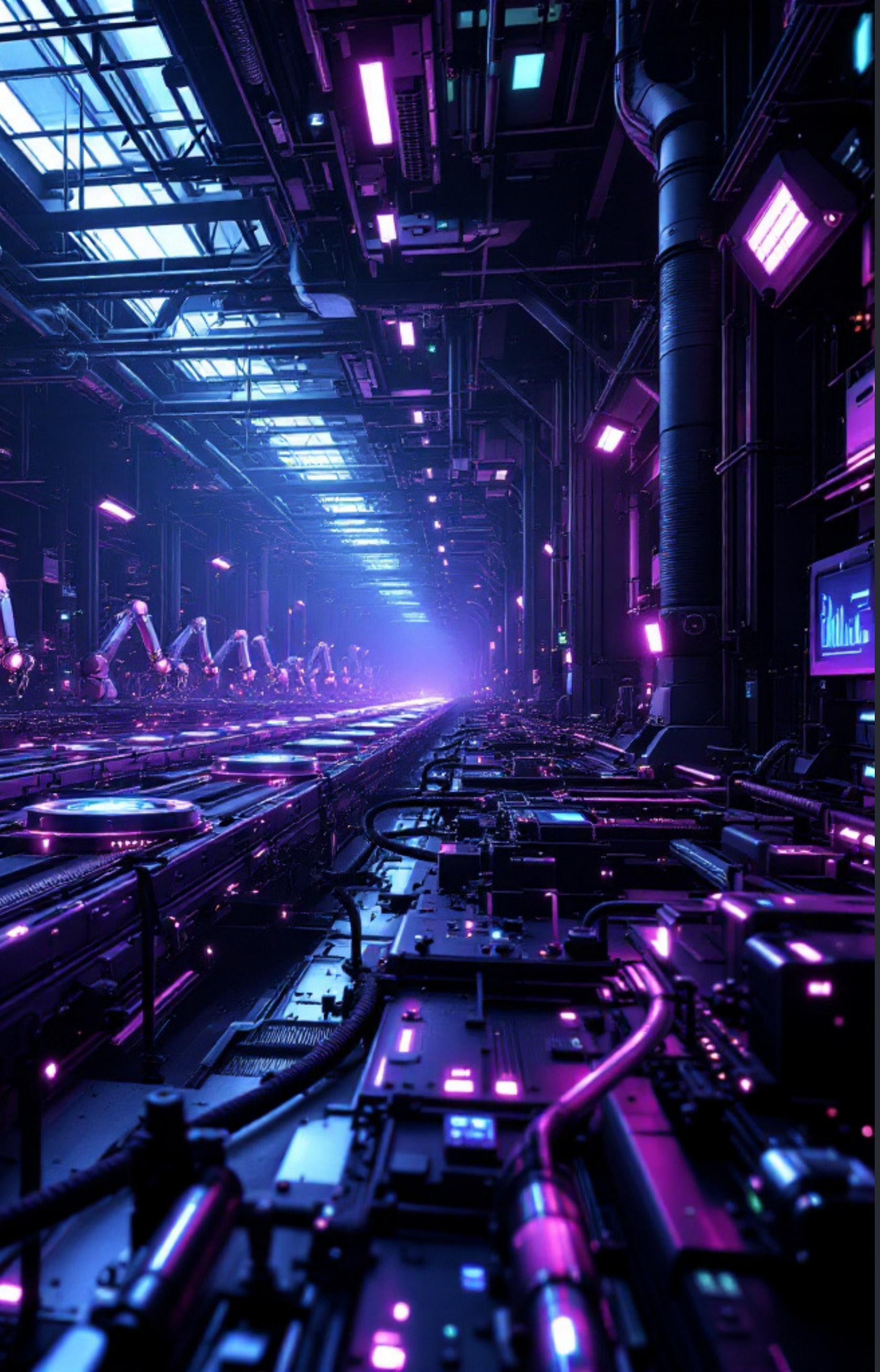
Adapt to growing needs.

Consistency

Unified design across project.

Efficiency

Improved performance and resource usage.



Creational Patterns in Software Engineering

Creational patterns are design patterns that provide a **standardized way to create objects**. They aim to simplify object creation, **improve code reusability**, and enhance flexibility in designing complex systems.

Introduction to Creational Patterns

1 **Object Creation**

Creational patterns address the problem of object creation, offering a structured approach to handle object instantiation in software applications.

2 **Flexibility and Reusability**

These patterns promote code reusability by abstracting object creation logic, allowing for flexible and adaptable object construction.

3 **Maintainability**

They improve the maintainability of code by providing a consistent and predictable way to create objects, simplifying code modifications and enhancements.

The Abstract Factory Pattern

Centralized Creation

The Abstract Factory pattern provides a **central point for creating families of related objects**, ensuring consistency and adherence to specific design requirements.

Interchangeable Factories

Different implementations of the abstract factory can be used interchangeably, **allowing for flexibility** in choosing the appropriate factory for a specific context.

Product Families

The pattern enables the creation of families of related objects, such as **UI elements, database connections, or different types of documents**, each with their own specific characteristics.

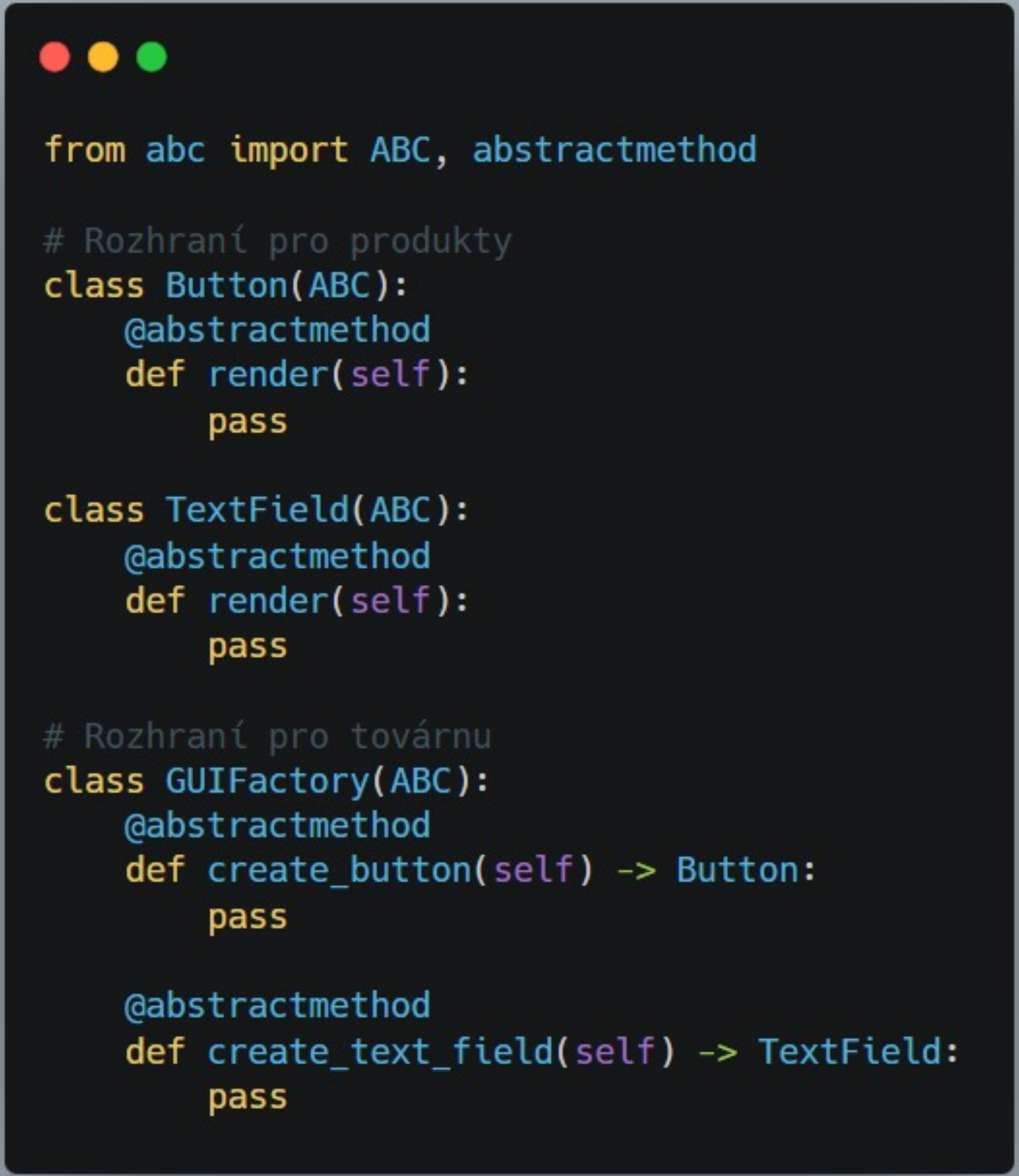
Abstract Factory: Cross platform UI app

The Abstract Factory can be used to create, for example, a user interface that needs to work across different operating systems (Windows, macOS, Linux). Each system can have its own specific implementations of buttons and text fields.

The Abstract Factory helps by defining an abstract interface for creating these elements, and then individual concrete factories ensure that the correct implementations are produced for the respective operating system.

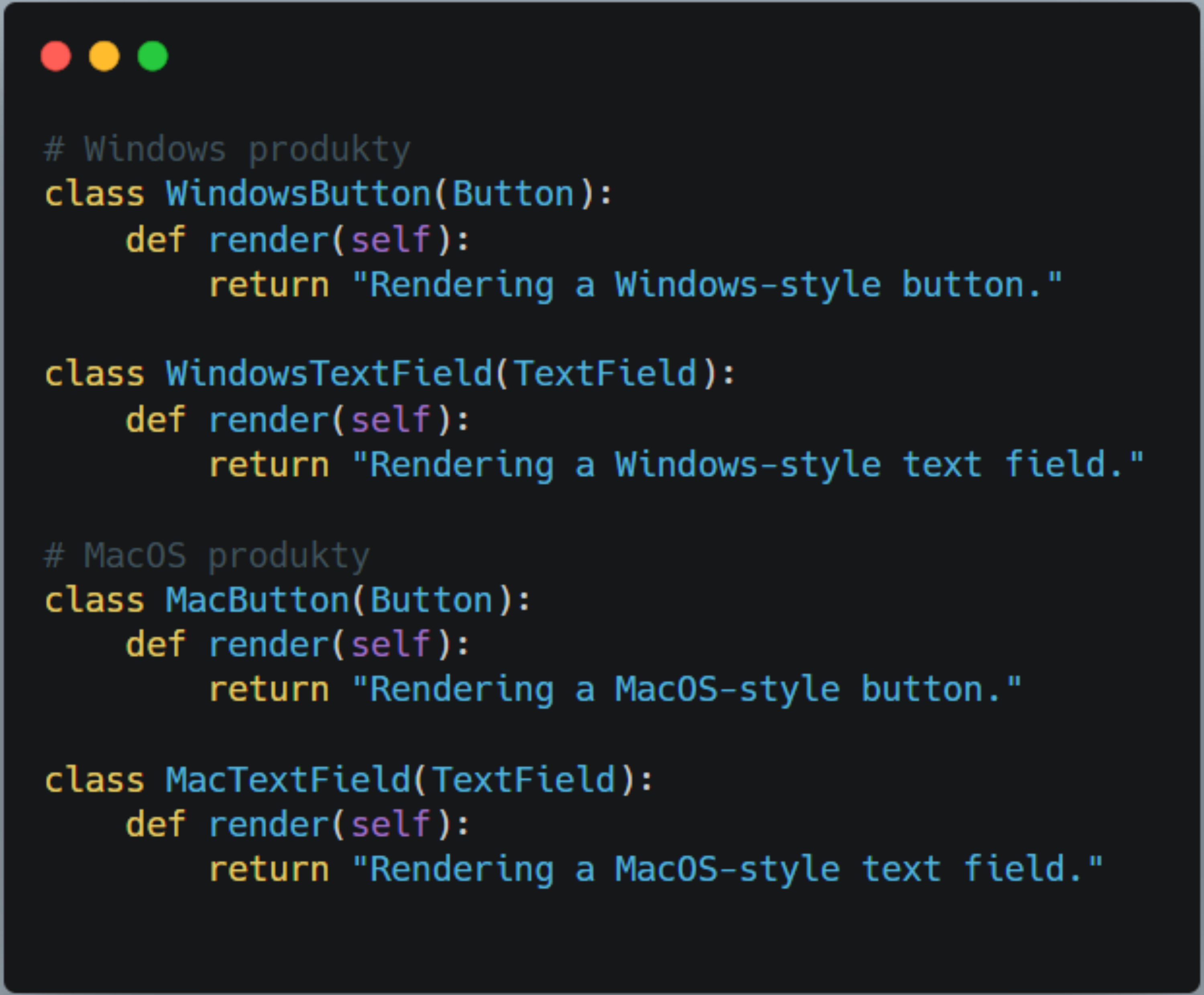
1

We define a common interface for factories as well as for products (e.g., buttons and text fields).



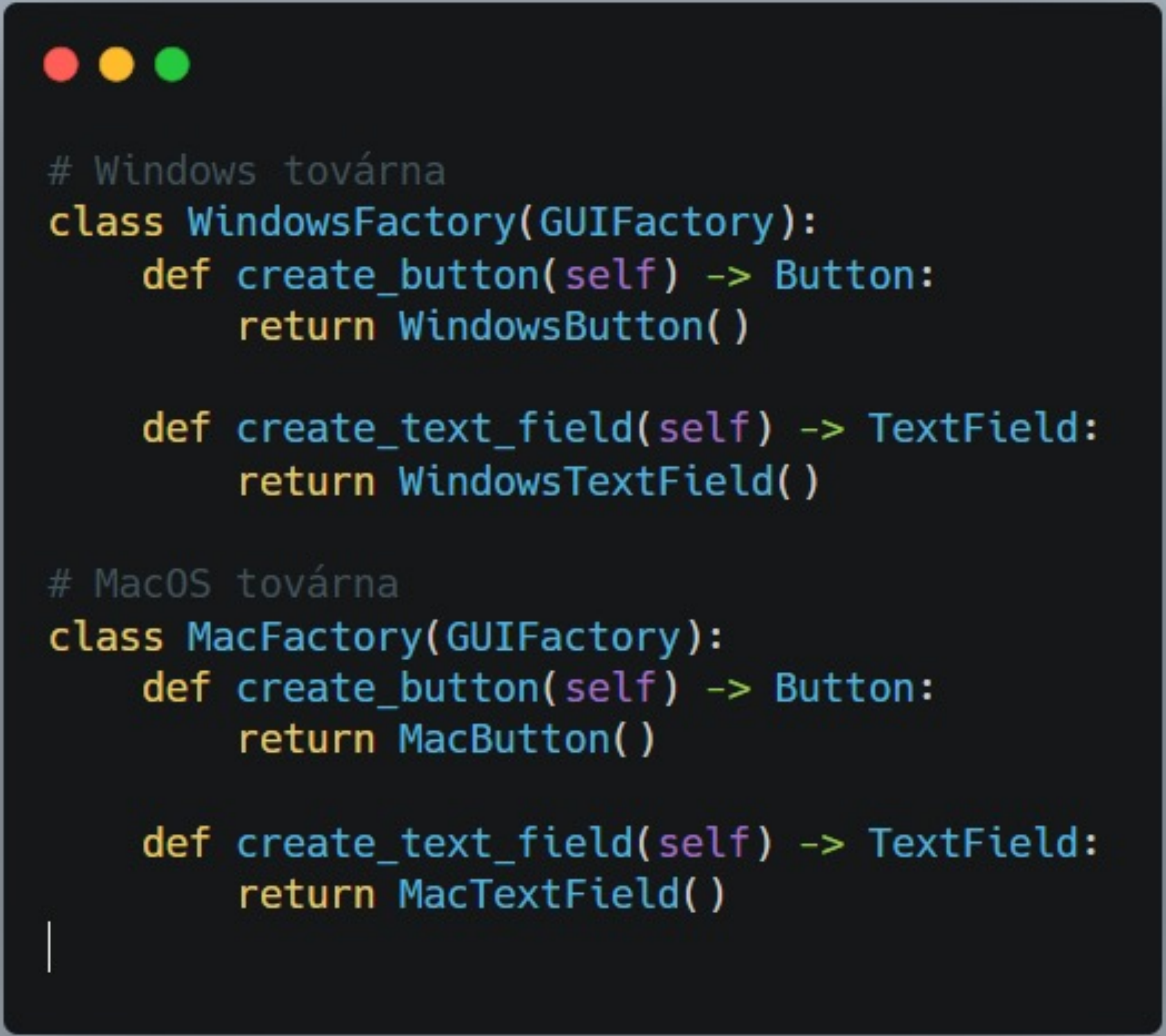
```python
from abc import ABC, abstractmethod

# Rozhraní pro produkty
class Button(ABC):
    @abstractmethod
    def render(self):
        pass

class TextField(ABC):
    @abstractmethod
    def render(self):
        pass

# Rozhraní pro továrnu
class GUIFactory(ABC):
    @abstractmethod
    def create_button(self) -> Button:
        pass

    @abstractmethod
    def create_text_field(self) -> TextField:
        pass
```

We define a common interface for factories as well as for products (e.g., buttons and text fields).



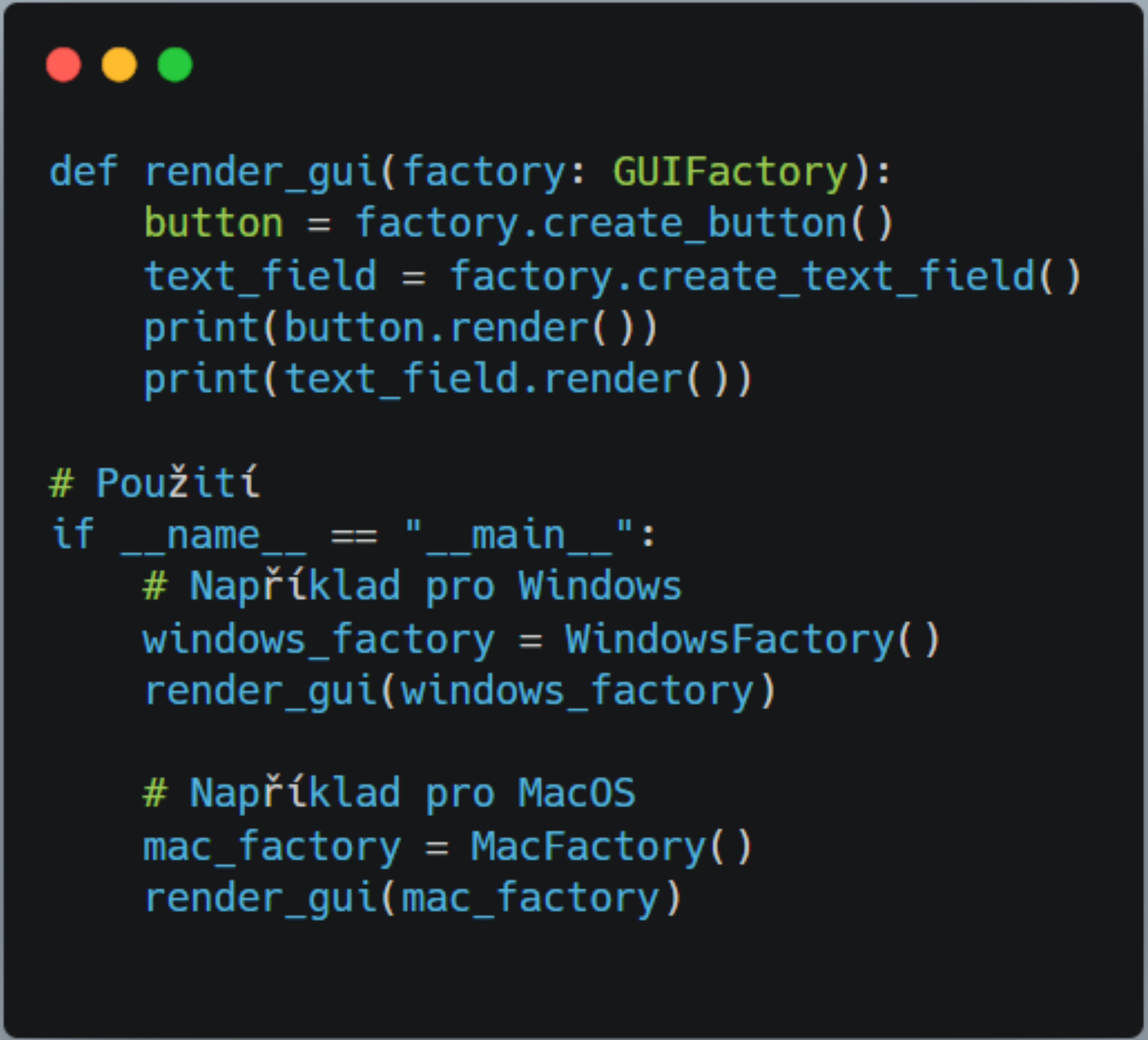
```python
# Windows produkty
class WindowsButton(Button):
    def render(self):
        return "Rendering a Windows-style button."

class WindowsTextField(TextField):
    def render(self):
        return "Rendering a Windows-style text field."

# MacOS produkty
class MacButton(Button):
    def render(self):
        return "Rendering a MacOS-style button."

class MacTextField(TextField):
    def render(self):
        return "Rendering a MacOS-style text field."
```

Each factory returns the appropriate products for the given operating system.



```python
# Windows továrna
class WindowsFactory(GUIFactory):
    def create_button(self) -> Button:
        return WindowsButton()

    def create_text_field(self) -> TextField:
        return WindowsTextField()

# MacOS továrna
class MacFactory(GUIFactory):
    def create_button(self) -> Button:
        return MacButton()

    def create_text_field(self) -> TextField:
        return MacTextField()
```

The client code works only with abstract factories and products, without knowing which specific products it is using.



```python
def render_gui(factory: GUIFactory):
    button = factory.create_button()
    text_field = factory.create_text_field()
    print(button.render())
    print(text_field.render())

# Použití
if __name__ == "__main__":
    # Například pro Windows
    windows_factory = WindowsFactory()
    render_gui(windows_factory)

    # Například pro MacOS
    mac_factory = MacFactory()
    render_gui(mac_factory)
```

Benefits of the Abstract Factory Pattern

Reduced Coupling

By abstracting object creation, the Abstract Factory pattern decouples the code that uses objects from the specific details of their creation.

Improved Testability

The separation of object creation logic makes it easier to test different implementations of the factory without affecting the code that uses them.

Enhanced Flexibility

The Abstract Factory pattern allows for easy switching between different product families by simply changing the factory implementation.

The Builder Pattern

1 **Step-by-Step Construction**

The Builder pattern allows for the creation of complex objects step-by-step, building them incrementally rather than all at once.

2 **Flexible Object Creation**

It provides a way to create objects with different configurations by specifying the construction steps in a flexible and adaptable manner.

3 **Separate Construction Logic**

The pattern separates the construction logic from the object itself, enhancing code modularity and making it easier to modify the construction process.

Key Components of the Builder Pattern

Component	Description
Builder	Defines the interface for constructing the object.
Concrete Builder	Implements the builder interface and provides specific methods for constructing the object.
Director	Orchestrates the construction process by calling methods on the concrete builder to assemble the object.
Product	Represents the complex object being constructed.

Builder: Computers

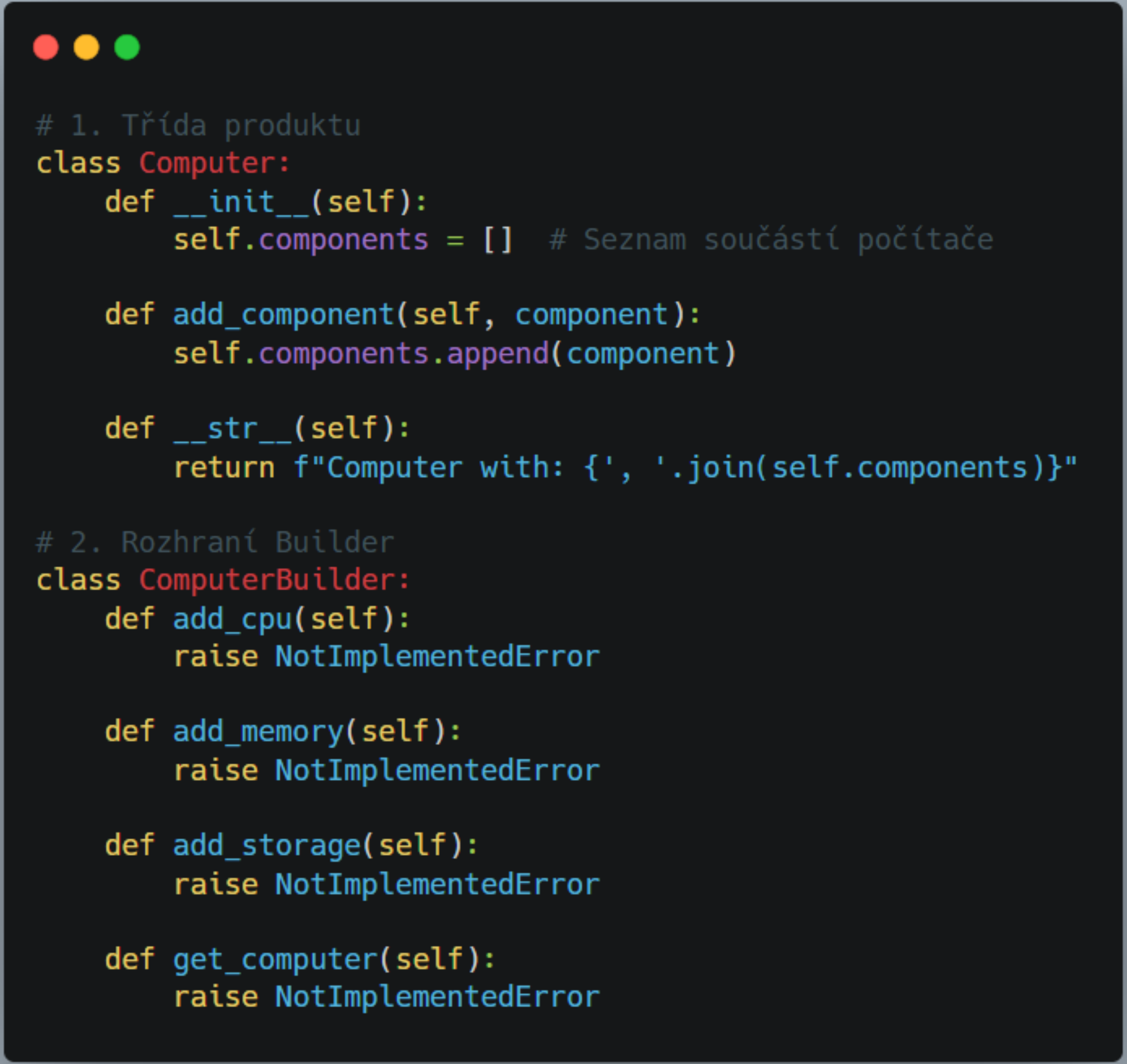
The Builder pattern is used when we want to create a complex object (e.g., a car, a house, a computer) that consists of many parts.

Instead of assembling the object directly within a single class, we break the process down into smaller steps. These steps are handled by a separate object (the Builder), which is responsible for the step-by-step construction of the object.

1

Computer – represents the final product that we are building.

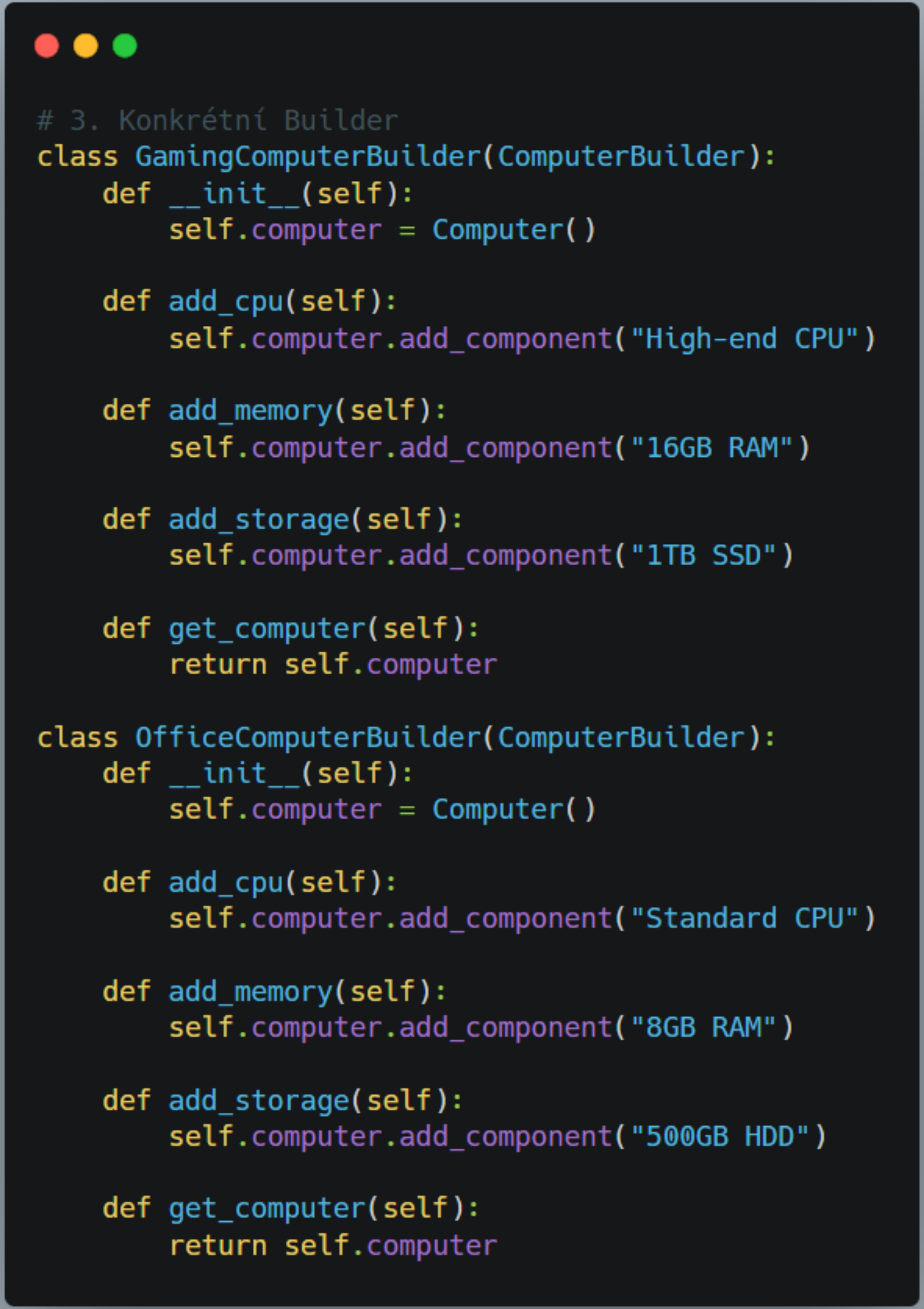
ComputerBuilder – defines the interface for constructing different parts of the product.



```python
# 1. Třída produktu
class Computer:
    def __init__(self):
        self.components = []  # Seznam součástí počítače

    def add_component(self, component):
        self.components.append(component)

    def __str__(self):
        return f"Computer with: {', '.join(self.components)}"

# 2. Rozhraní Builder
class ComputerBuilder:
    def add_cpu(self):
        raise NotImplementedError

    def add_memory(self):
        raise NotImplementedError

    def add_storage(self):
        raise NotImplementedError

    def get_computer(self):
        raise NotImplementedError
```

GamingComputerBuilder and OfficeComputerBuilder – different implementations of the Builder, each constructing a different type of computer.

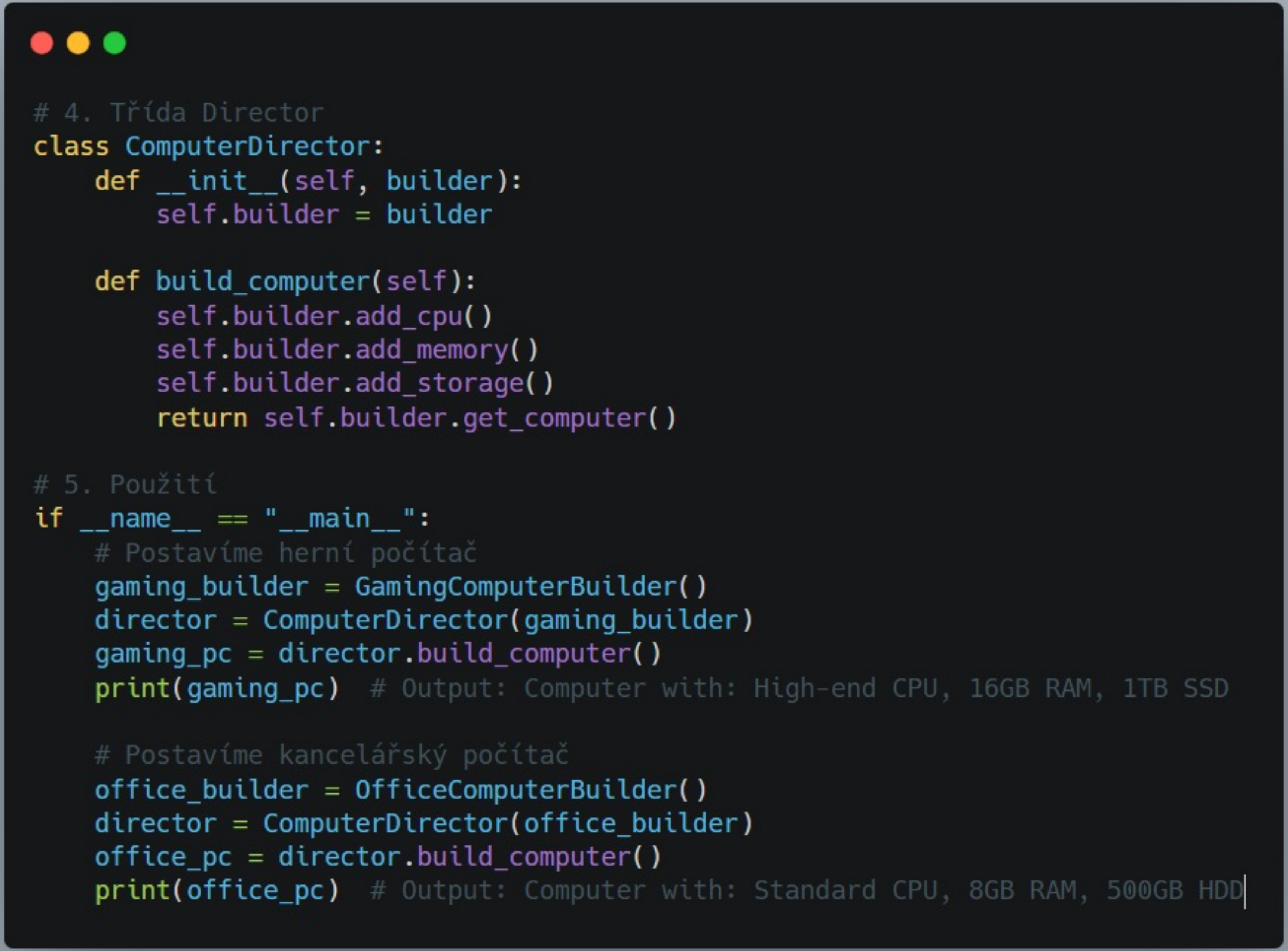
ComputerDirector – manages the process of how the individual parts are assembled together.



```python
# 3. Konkrétní Builder
class GamingComputerBuilder(ComputerBuilder):
    def __init__(self):
        self.computer = Computer()

    def add_cpu(self):
        self.computer.add_component("High-end CPU")

    def add_memory(self):
        self.computer.add_component("16GB RAM")

    def add_storage(self):
        self.computer.add_component("1TB SSD")

    def get_computer(self):
        return self.computer

class OfficeComputerBuilder(ComputerBuilder):
    def __init__(self):
        self.computer = Computer()

    def add_cpu(self):
        self.computer.add_component("Standard CPU")

    def add_memory(self):
        self.computer.add_component("8GB RAM")

    def add_storage(self):
        self.computer.add_component("500GB HDD")

    def get_computer(self):
        return self.computer
```

ComputerDirector – directs the process of assembling the individual parts of the computer in the correct order.

This approach makes it easy to create different types of computers using the same construction process.



```python
# 4. Třída Director
class ComputerDirector:
    def __init__(self, builder):
        self.builder = builder

    def build_computer(self):
        self.builder.add_cpu()
        self.builder.add_memory()
        self.builder.add_storage()
        return self.builder.get_computer()

# 5. Použití
if __name__ == "__main__":
    # Postavíme herní počítač
    gaming_builder = GamingComputerBuilder()
    director = ComputerDirector(gaming_builder)
    gaming_pc = director.build_computer()
    print(gaming_pc)  # Output: Computer with: High-end CPU, 16GB RAM, 1TB SSD

    # Postavíme kancelářský počítač
    office_builder = OfficeComputerBuilder()
    director = ComputerDirector(office_builder)
    office_pc = director.build_computer()
    print(office_pc)  # Output: Computer with: Standard CPU, 8GB RAM, 500GB HDD
```

The Prototype Pattern

Cloning Objects

The Prototype pattern allows for creating new objects by cloning existing prototype objects, reducing the need for repetitive object creation logic.

Faster Instantiation

Cloning existing objects can be significantly faster than using traditional object creation methods, particularly for complex objects.

Code Reusability

The Prototype pattern promotes code reusability by allowing you to reuse existing object instances to create new ones, reducing duplication of code.

Advantages of the Prototype Pattern

1. **Reduced Code Duplication**

   The Prototype pattern eliminates the need to write repetitive code for object creation, leading to more concise and maintainable code.

2. **Performance Optimization**

   Cloning existing objects can be faster than traditional object creation methods, especially for complex objects with many attributes.

3. **Flexible Object Creation**

   The Prototype pattern allows you to create objects with different configurations by modifying the prototype object before cloning it.

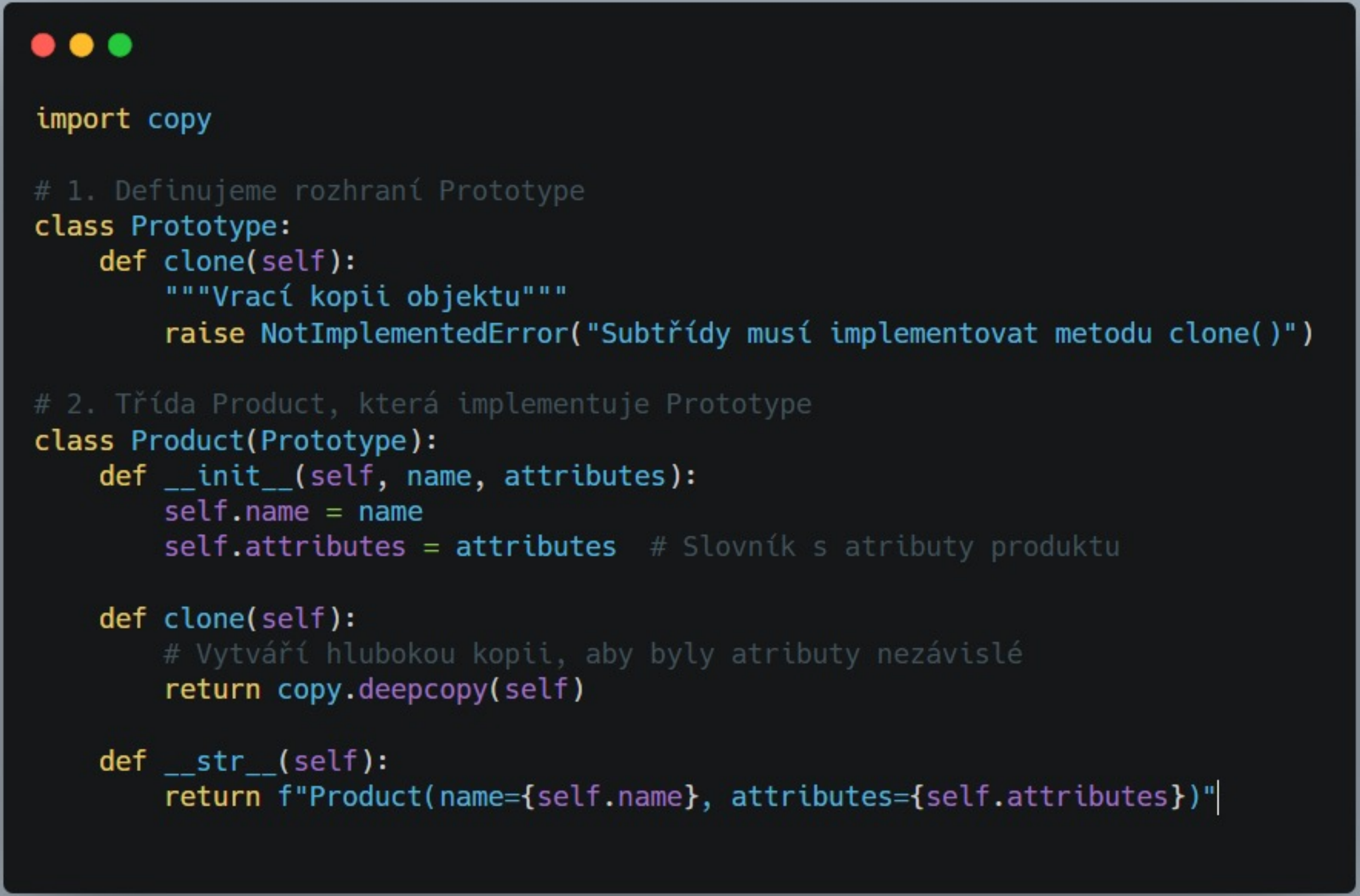
Prototype: Example – Creating multiple objects

Following code will solve the problem of creating objects with complex structures or configurations efficiently and independently. Instead of building new objects from scratch every time, it allows cloning predefined prototypes, ensuring that each cloned object can be modified without affecting the original.

This approach is particularly useful when object creation is resource-intensive or when multiple variations of an object are needed.

We define a common interface for factories as well as for products (e.g., buttons and text fields).

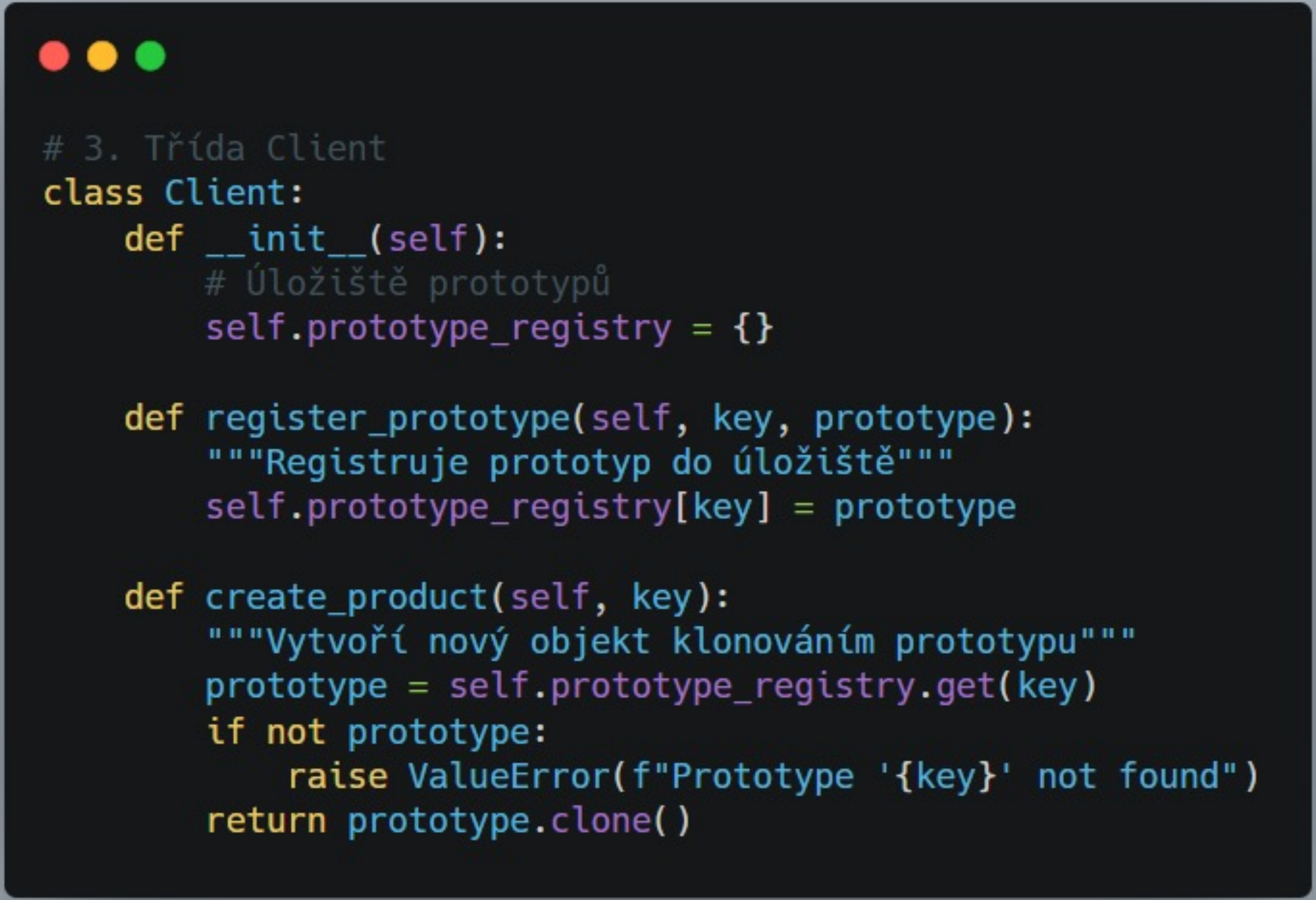
The clone method creates a deep copy of the object using the copy module to ensure that sub-objects are also copied, avoiding shared references.



```python
import copy

# 1. Definujeme rozhraní Prototype
class Prototype:
    def clone(self):
        """Vrací kopii objektu"""
        raise NotImplementedError("Subtřídy musí implementovat metodu clone()")

# 2. Třída Product, která implementuje Prototype
class Product(Prototype):
    def __init__(self, name, attributes):
        self.name = name
        self.attributes = attributes  # Slovník s atributy produktu

    def clone(self):
        # Vytváří hlubokou kopii, aby byly atributy nezávislé
        return copy.deepcopy(self)

    def __str__(self):
        return f"Product(name={self.name}, attributes={self.attributes})"
```

The Prototype class defines an abstract clone method that all concrete prototypes must implement.

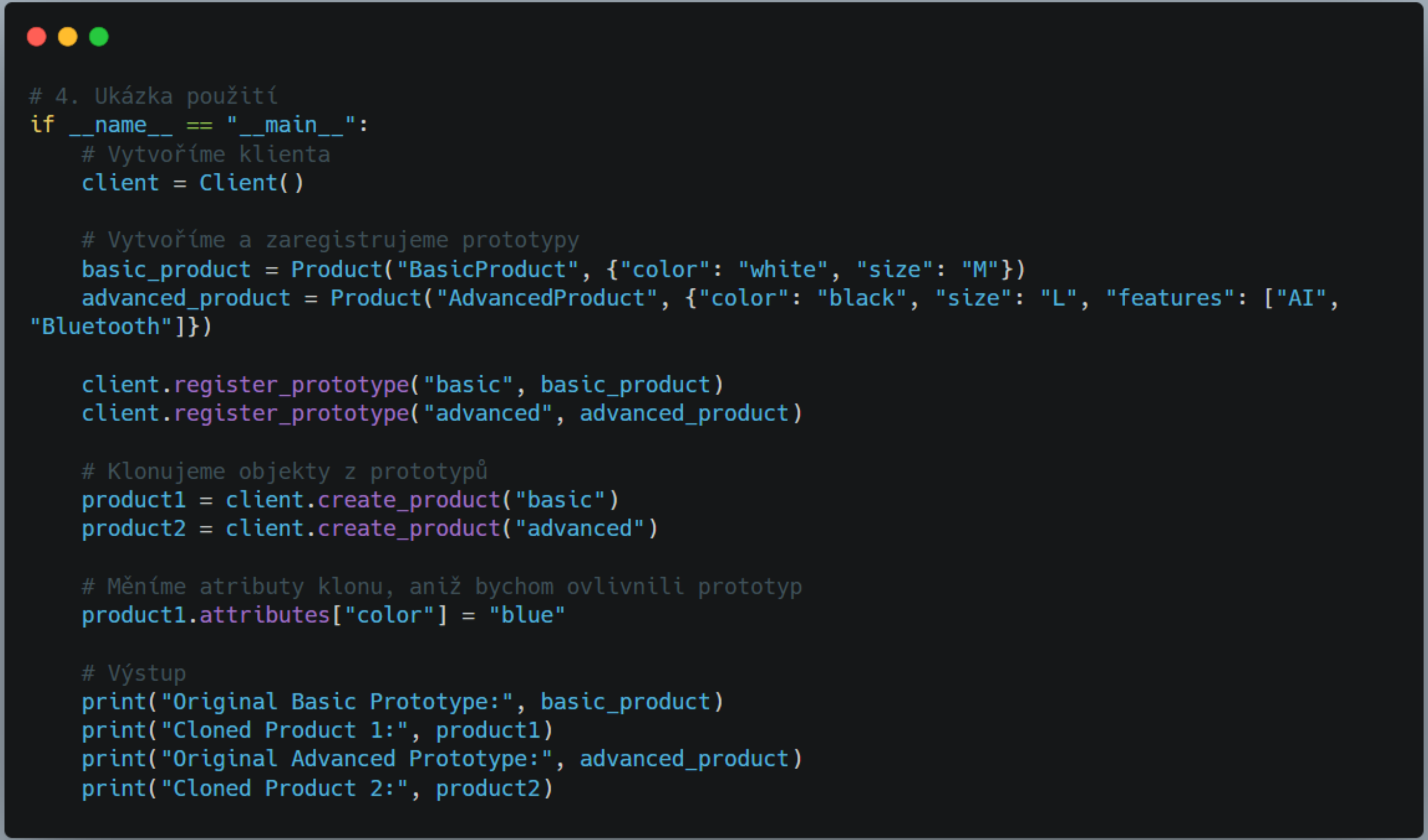
The clone method is responsible for creating a copy of the object.



```python
# 3. Třída Client
class Client:
    def __init__(self):
        # Úložiště prototypů
        self.prototype_registry = {}

    def register_prototype(self, key, prototype):
        """Registruje prototyp do úložiště"""
        self.prototype_registry[key] = prototype

    def create_product(self, key):
        """Vytvoří nový objekt klonováním prototypu"""
        prototype = self.prototype_registry.get(key)
        if not prototype:
            raise ValueError(f"Prototype '{key}' not found")
        return prototype.clone()
```

Two prototypes (basic_product and advanced_product) created

Two new products are created by cloning the prototypes using the create_product method. A cloned product's (product1) attribute is modified, which does not affect the original prototype (basic_product), as the copy is independent.



```python
# 4. Ukázka použití
if __name__ == "__main__":
    # Vytvoříme klienta
    client = Client()

    # Vytvoříme a zaregistrujeme prototypy
    basic_product = Product("BasicProduct", {"color": "white", "size": "M"})
    advanced_product = Product("AdvancedProduct", {"color": "black", "size": "L", "features": ["AI",
"Bluetooth"]})

    client.register_prototype("basic", basic_product)
    client.register_prototype("advanced", advanced_product)

    # Klonujeme objekty z prototypů
    product1 = client.create_product("basic")
    product2 = client.create_product("advanced")

    # Měníme atributy klonu, aniž bychom ovlivnili prototyp
    product1.attributes["color"] = "blue"

    # Výstup
    print("Original Basic Prototype:", basic_product)
    print("Cloned Product 1:", product1)
    print("Original Advanced Prototype:", advanced_product)
    print("Cloned Product 2:", product2)
```

The Singleton Pattern

1. **Controlled Instantiation**

   The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.

2. **Global Access**

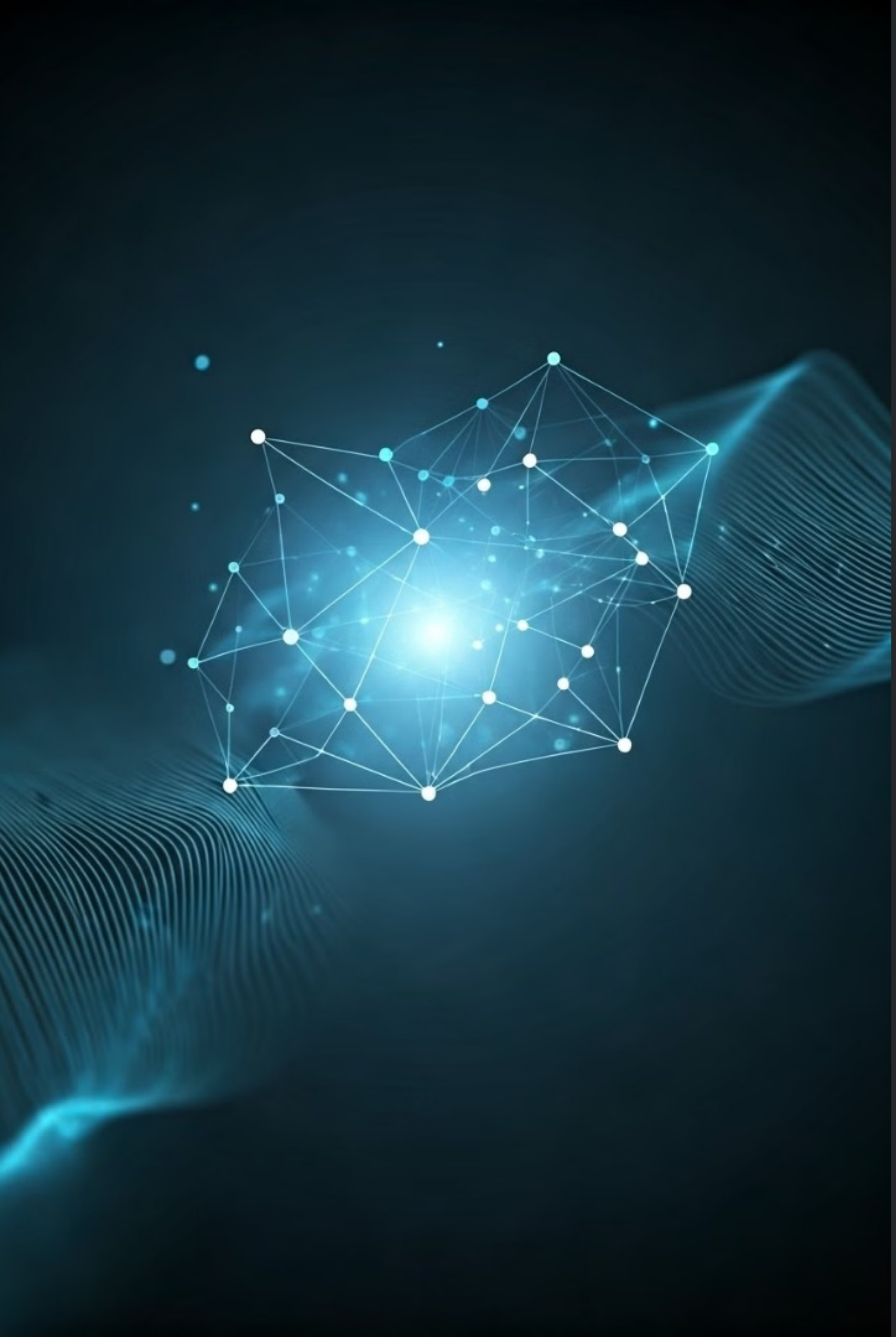
   The Singleton pattern provides a global point of access to the single instance, allowing different parts of the application to access it easily.

3. **Resource Management**

   The Singleton pattern can be used to manage resources like database connections or configuration settings efficiently.

Conclusion and Key Takeaways

Creational patterns are valuable tools for simplifying object creation and enhancing code reusability. By mastering these patterns, developers can build more flexible, maintainable, and efficient software applications.



Structural Patterns in Software Engineering

Structural patterns in software engineering focus on **organizing** and **composing** classes and objects to form larger, flexible structures. They enhance code maintainability by simplifying relationships, promoting reuse, and improving clarity in complex systems.

Adapter Pattern

Transforms a class interface into another expected by clients.

Implementation types

- **Object Adapter**
- **Class Adapter**

Key Components:

1. Target Interface
2. Adaptee
3. Adapter

Object Adapter: Structure

Key Components:

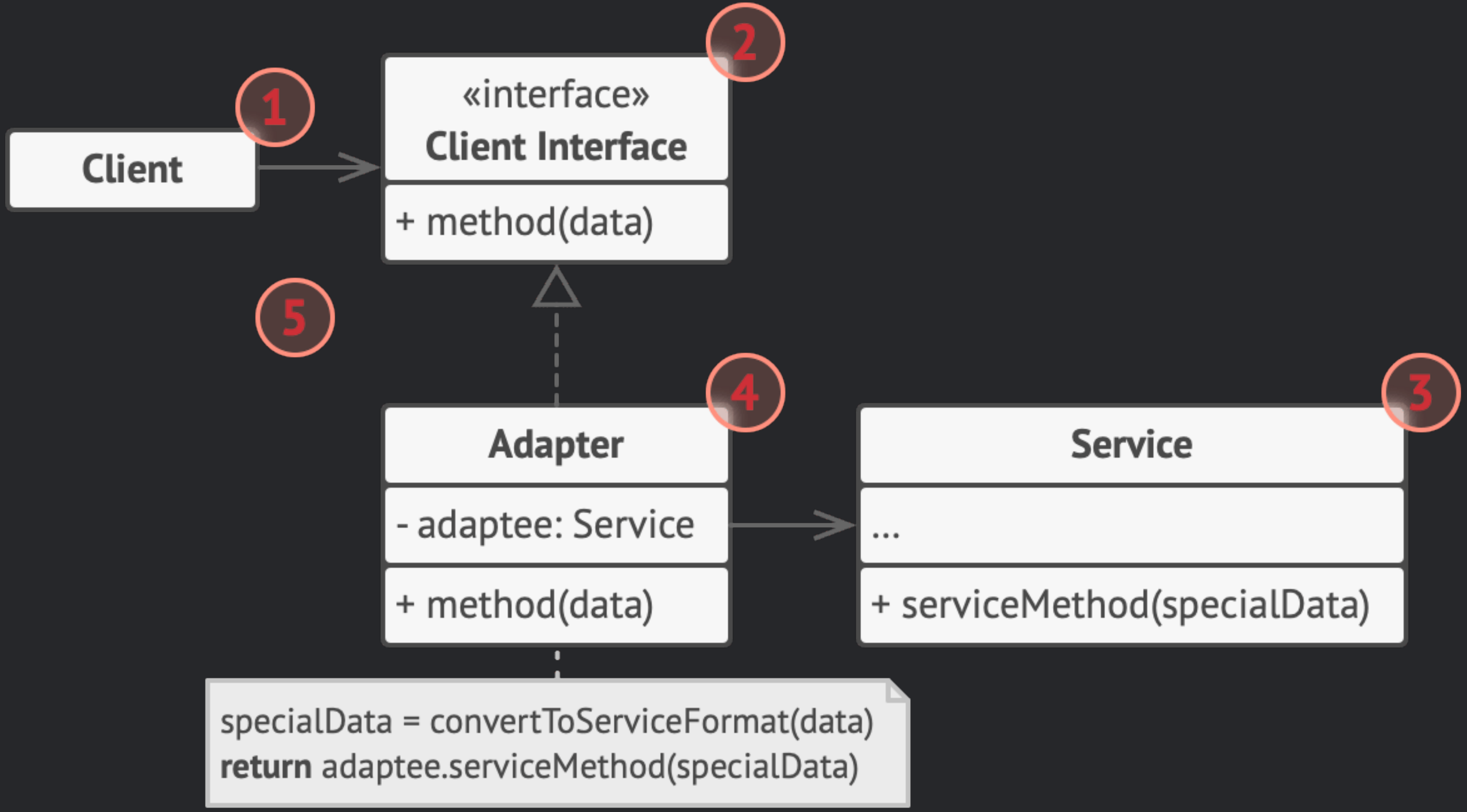
1 Client

2 Client Interface

3 Service

4 Adapter

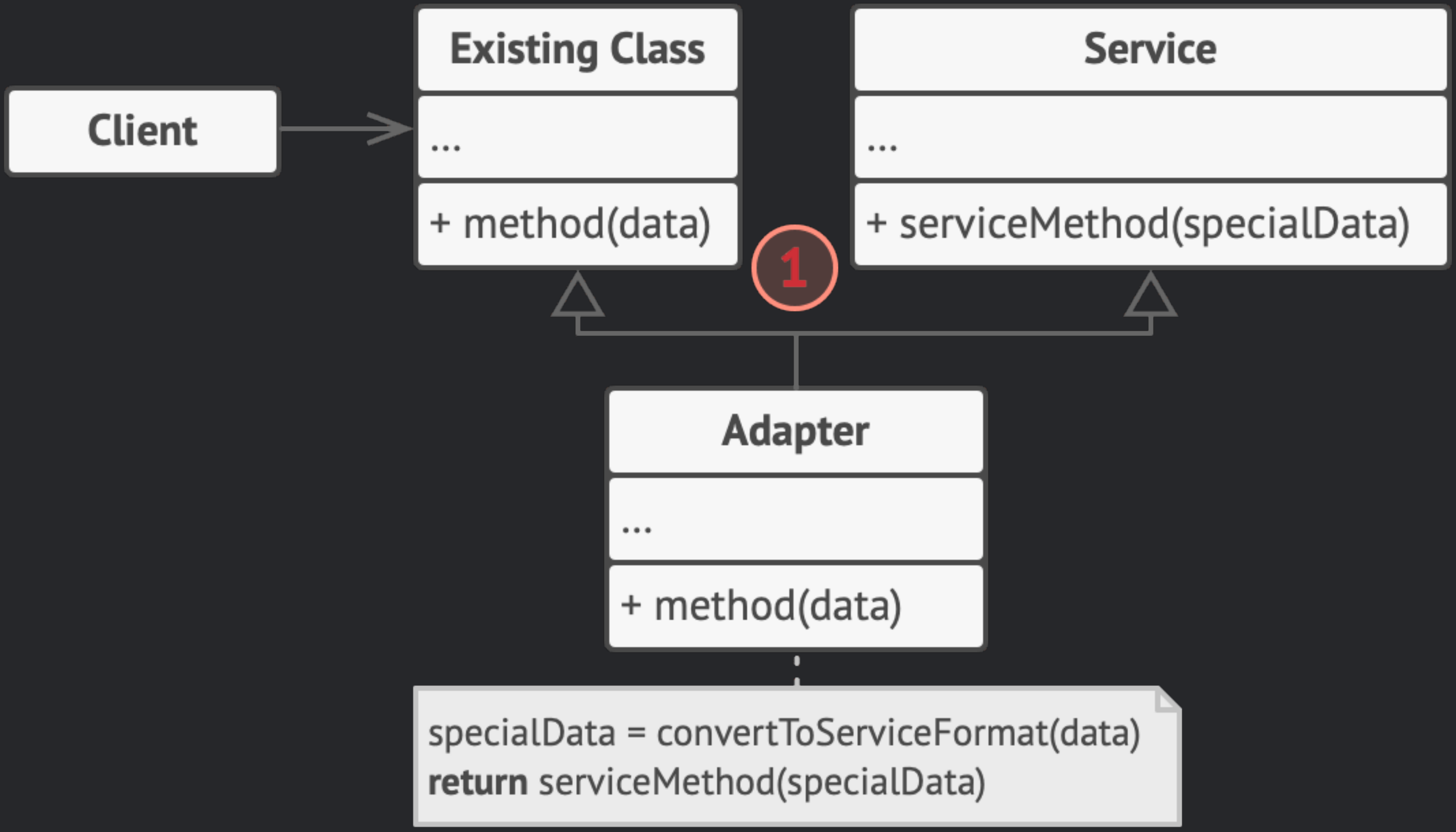
5 Decoupling



```
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)
```

Object Adapter design pattern structure diagram. Image from refactoring.guru

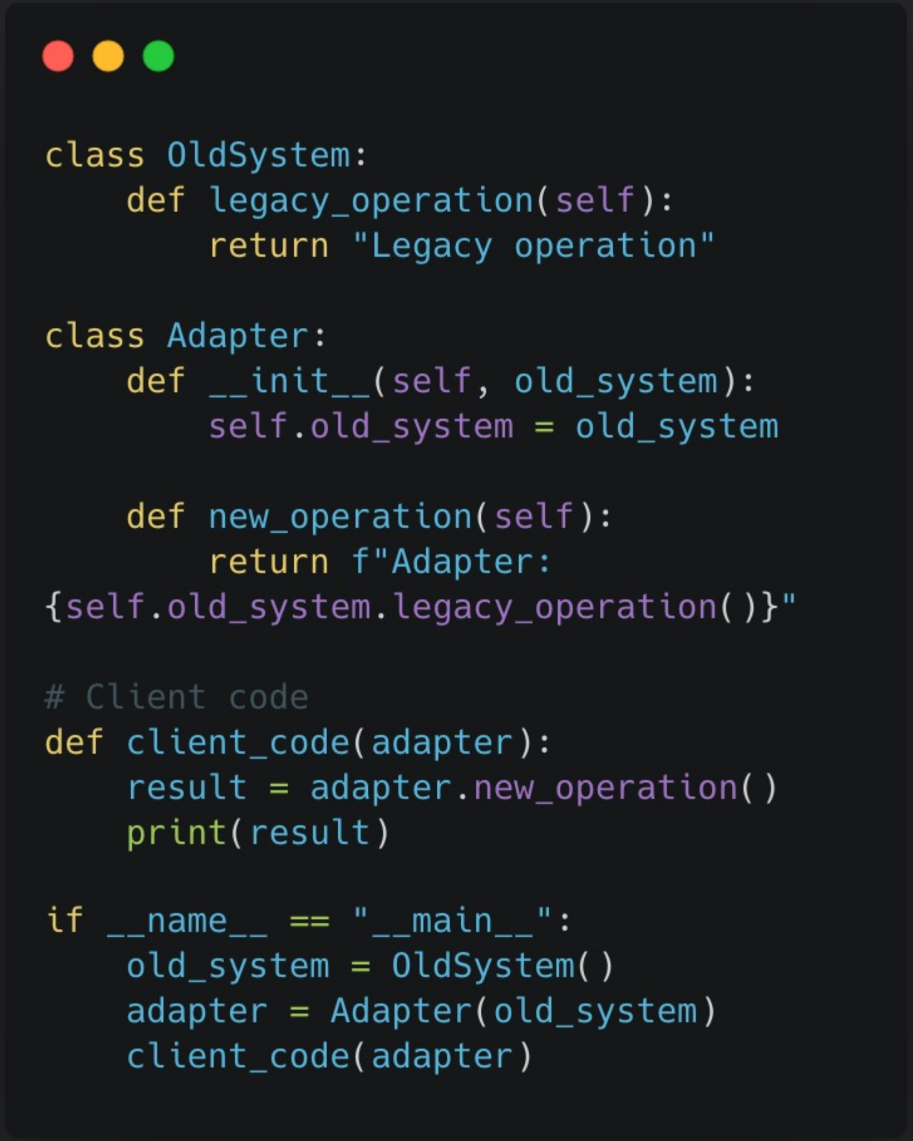
Class Adapter: Structure


Client
Existing Class
...
+ method(data)
Service
...
+ serviceMethod(specialData)
1
Adapter
...
+ method(data)
specialData = convertToServiceFormat(data)
return serviceMethod(specialData)

Class Adapter design pattern structure diagram. Image from refactoring.guru

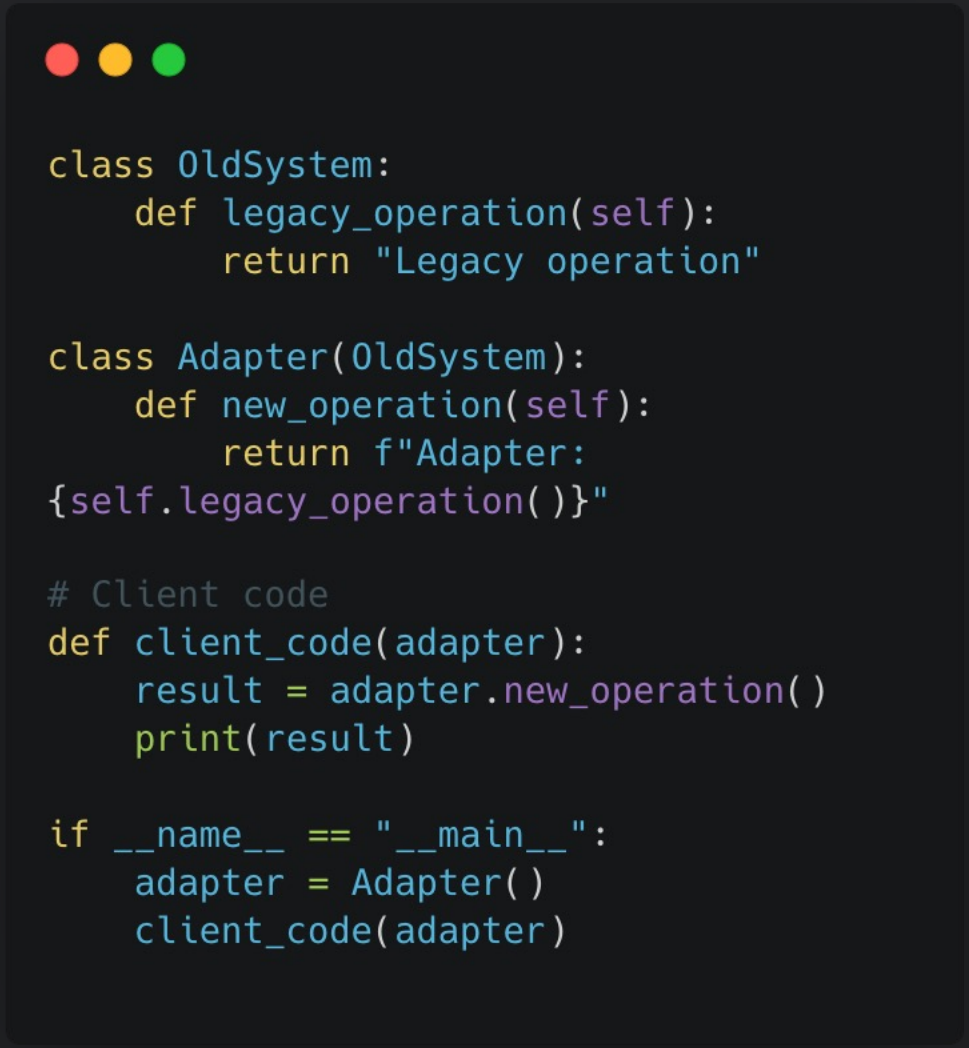
Adapter

Object Adapter Implementation



```python
class OldSystem:
    def legacy_operation(self):
        return "Legacy operation"

class Adapter:
    def __init__(self, old_system):
        self.old_system = old_system

    def new_operation(self):
        return f"Adapter:
{self.old_system.legacy_operation()}"

# Client code
def client_code(adapter):
    result = adapter.new_operation()
    print(result)

if __name__ == "__main__":
    old_system = OldSystem()
    adapter = Adapter(old_system)
    client_code(adapter)
```

Class Adapter Implementation

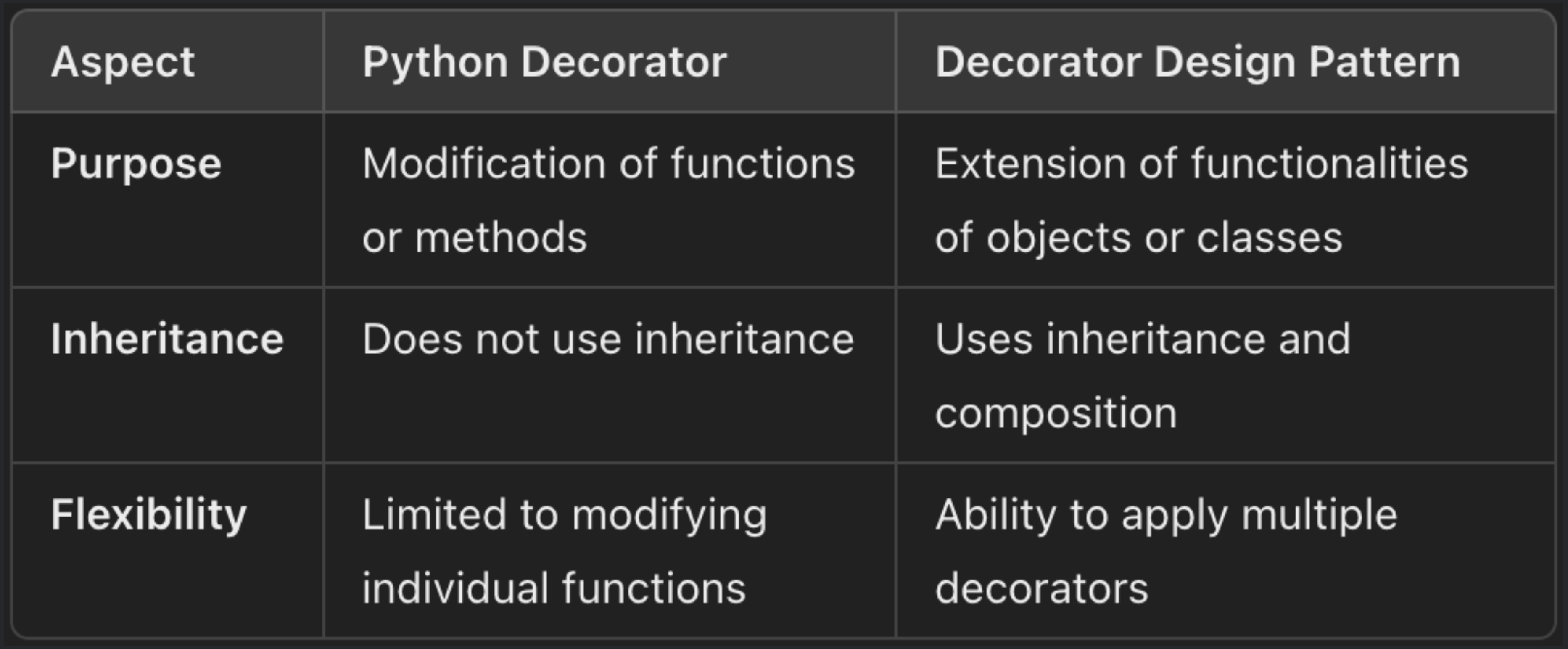


```python
class OldSystem:
    def legacy_operation(self):
        return "Legacy operation"

class Adapter(OldSystem):
    def new_operation(self):
        return f"Adapter:
{self.legacy_operation()}"

# Client code
def client_code(adapter):
    result = adapter.new_operation()
    print(result)

if __name__ == "__main__":
    adapter = Adapter()
    client_code(adapter)
```

Decorator Pattern

Dynamically adds functionality to objects without subclassing.

Python Decorator VS Decorator Design Pattern



Aspect	Python Decorator	Decorator Design Pattern
Purpose	Modification of functions or methods	Extension of functionalities of objects or classes
Inheritance	Does not use inheritance	Uses inheritance and composition
Flexibility	Limited to modifying individual functions	Ability to apply multiple decorators

Decorator: Structure

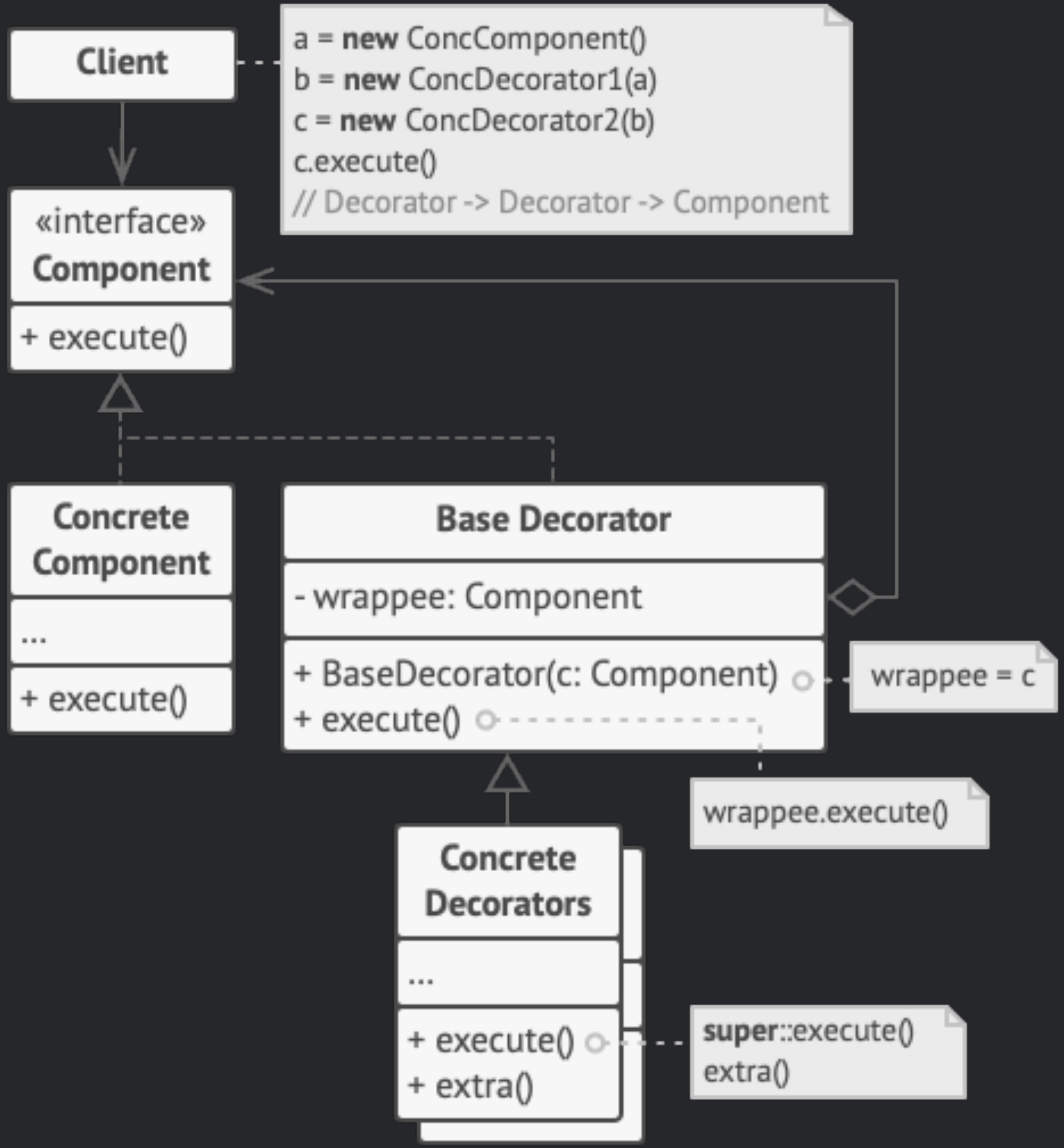
Key Components:

1 Component

2 Concrete Component

3 Decorator

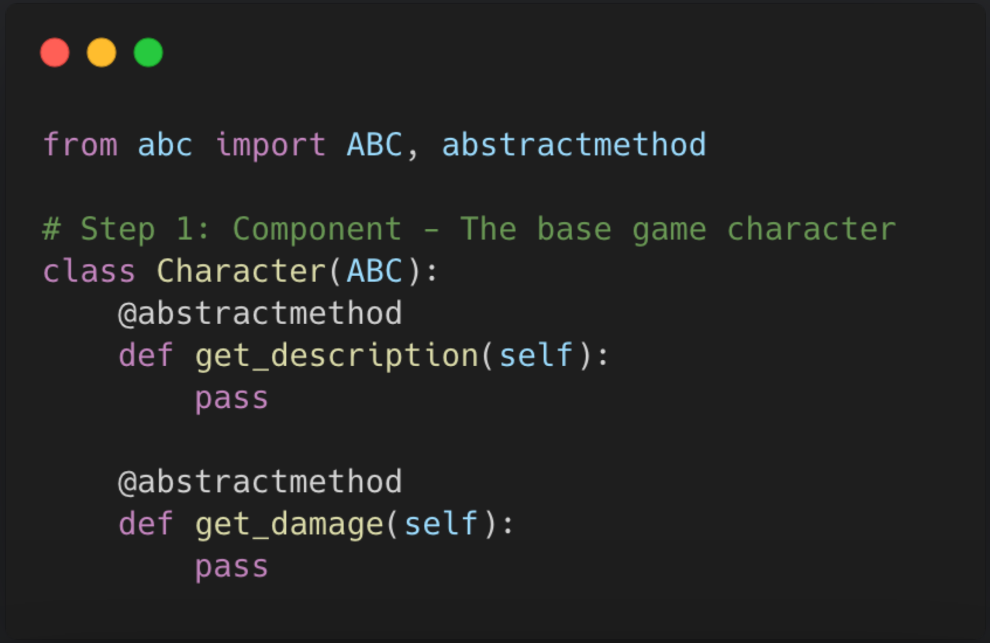
4 Concrete Decorator



```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

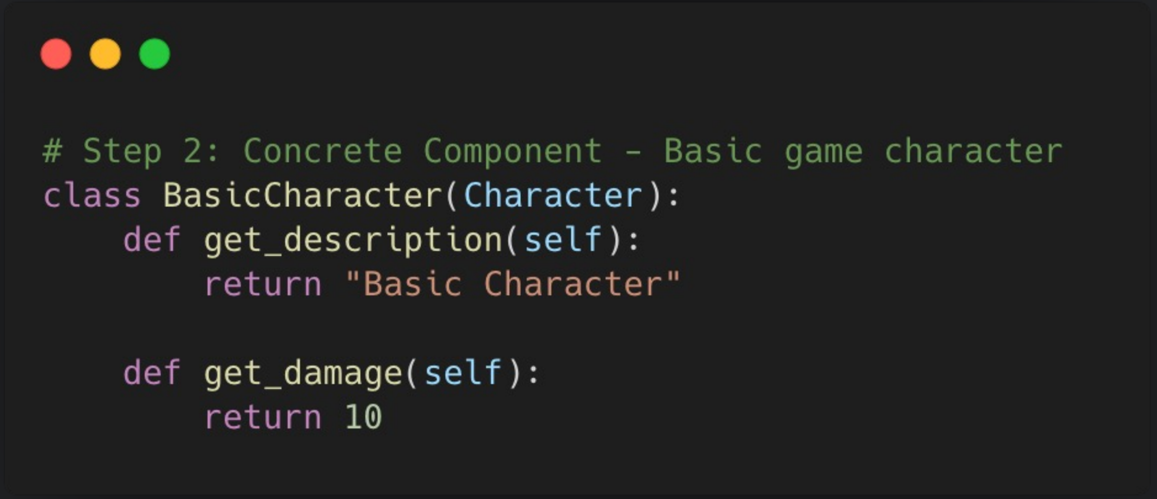
Decorator Implementation in Python

1 Component



```python
from abc import ABC, abstractmethod

# Step 1: Component - The base game character
class Character(ABC):
    @abstractmethod
    def get_description(self):
        pass

    @abstractmethod
    def get_damage(self):
        pass
```

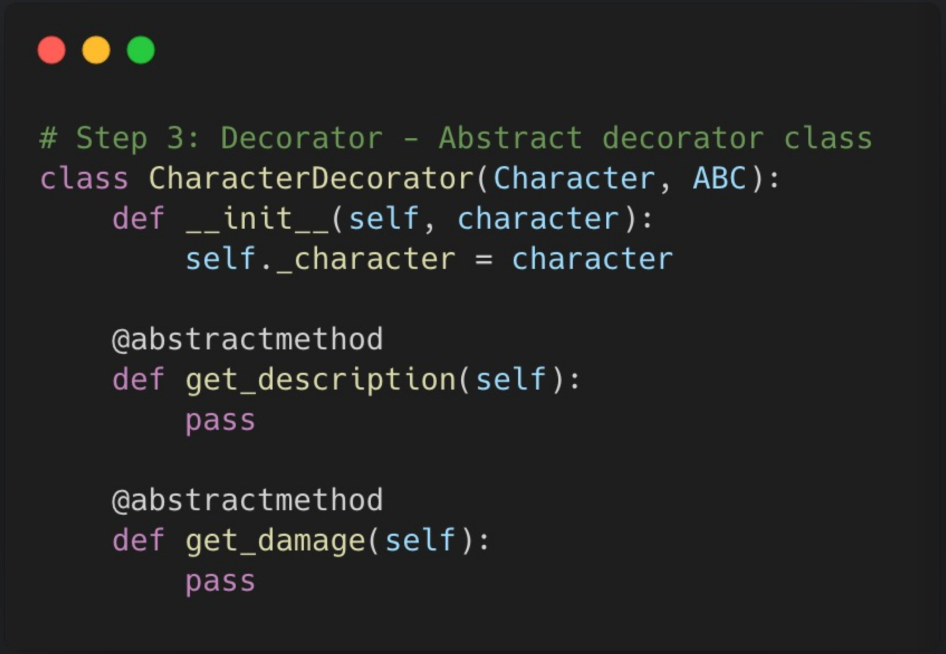
2 Concrete Component



```python
# Step 2: Concrete Component - Basic game character
class BasicCharacter(Character):
    def get_description(self):
        return "Basic Character"

    def get_damage(self):
        return 10
```

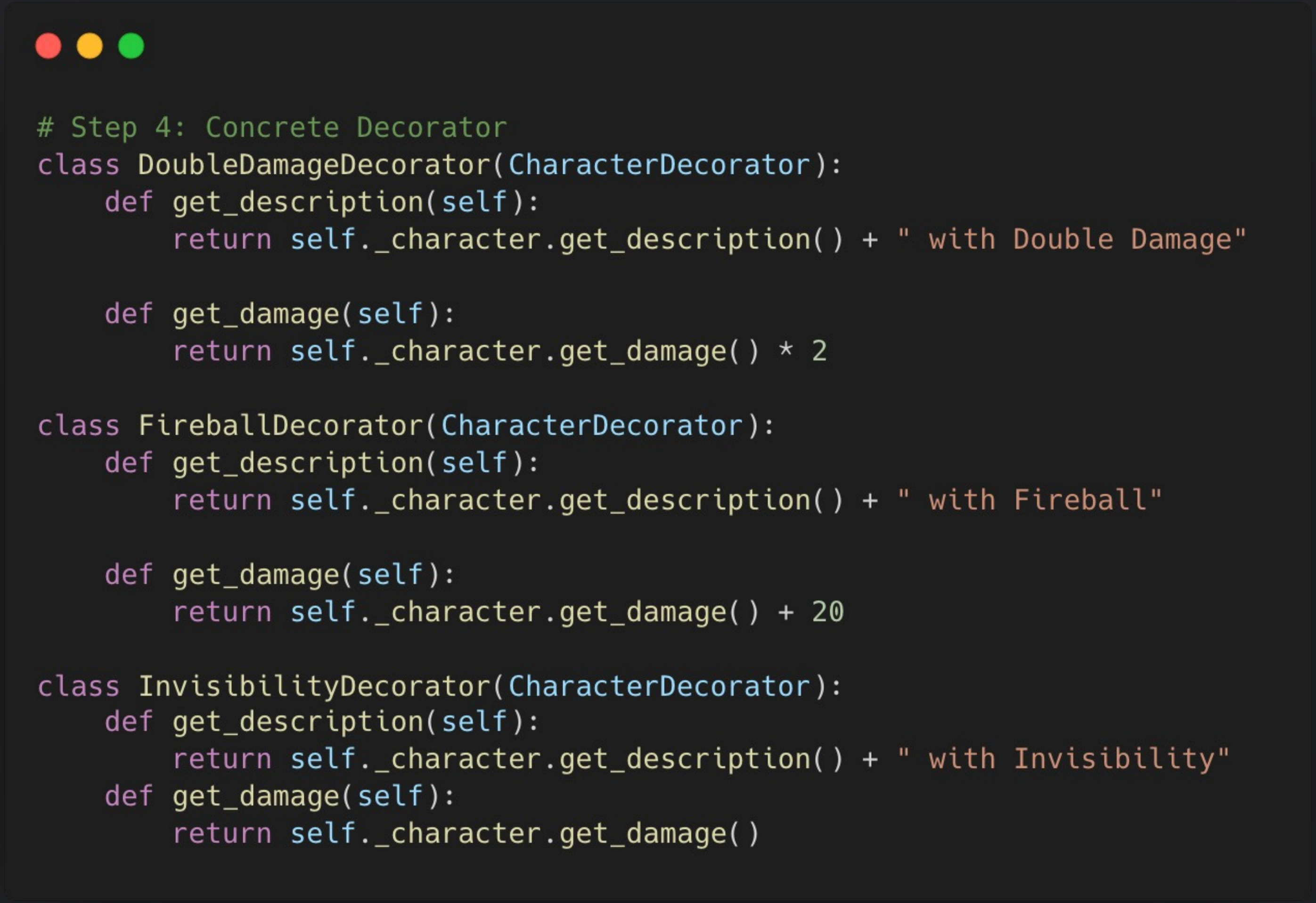
Decorator Implementation in Python

3 Decorator



```python
# Step 3: Decorator - Abstract decorator class
class CharacterDecorator(Character, ABC):
    def __init__(self, character):
        self._character = character

    @abstractmethod
    def get_description(self):
        pass

    @abstractmethod
    def get_damage(self):
        pass
```

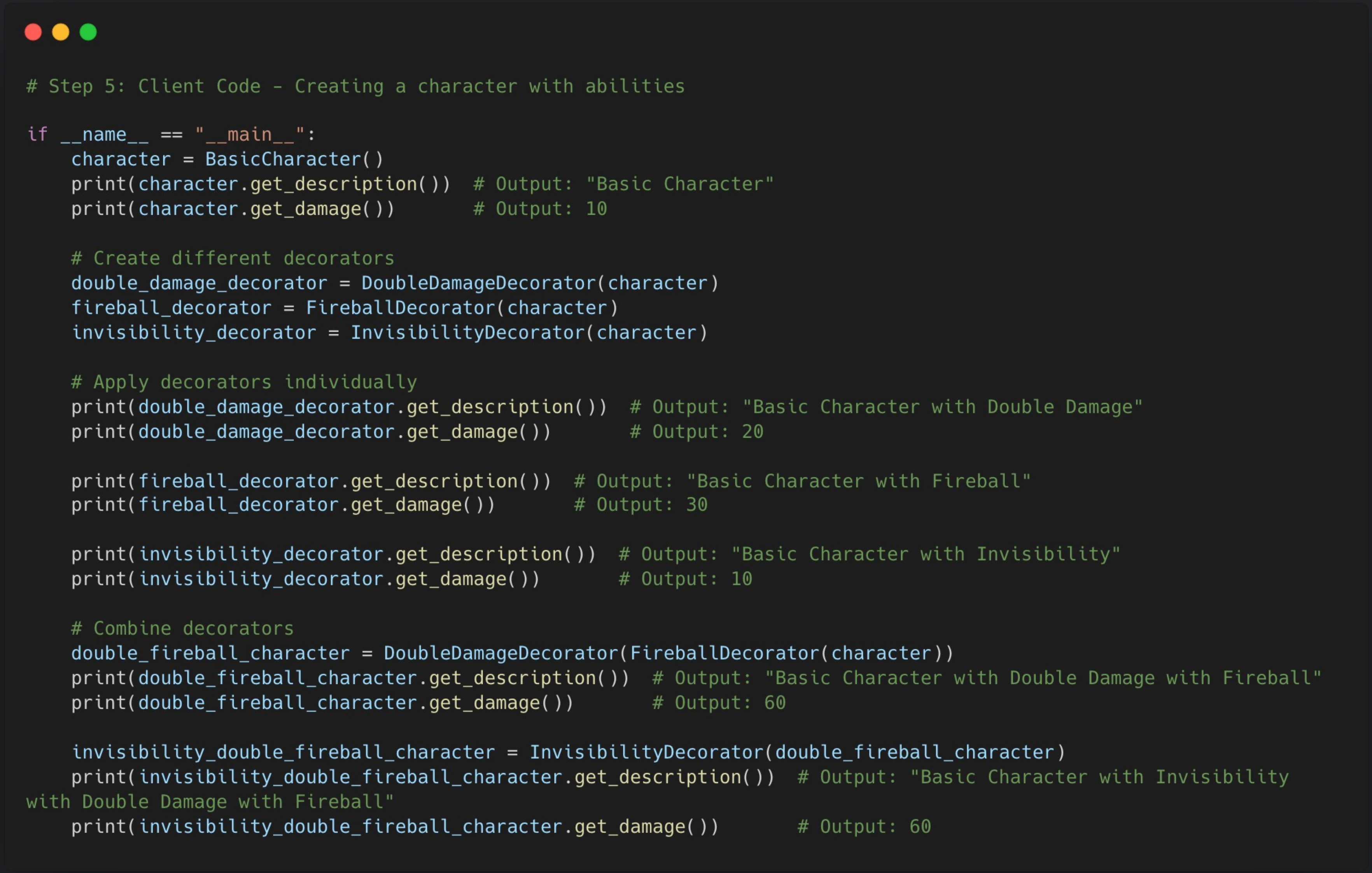
4 Concrete Decorator



```python
# Step 4: Concrete Decorator
class DoubleDamageDecorator(CharacterDecorator):
    def get_description(self):
        return self._character.get_description() + " with Double Damage"

    def get_damage(self):
        return self._character.get_damage() * 2

class FireballDecorator(CharacterDecorator):
    def get_description(self):
        return self._character.get_description() + " with Fireball"

    def get_damage(self):
        return self._character.get_damage() + 20

class InvisibilityDecorator(CharacterDecorator):
    def get_description(self):
        return self._character.get_description() + " with Invisibility"
    def get_damage(self):
        return self._character.get_damage()
```

Decorator Implementation in Python

Main Client Code



```python
# Step 5: Client Code - Creating a character with abilities

if __name__ == "__main__":
    character = BasicCharacter()
    print(character.get_description())  # Output: "Basic Character"
    print(character.get_damage())       # Output: 10

    # Create different decorators
    double_damage_decorator = DoubleDamageDecorator(character)
    fireball_decorator = FireballDecorator(character)
    invisibility_decorator = InvisibilityDecorator(character)

    # Apply decorators individually
    print(double_damage_decorator.get_description())  # Output: "Basic Character with Double Damage"
    print(double_damage_decorator.get_damage())       # Output: 20

    print(fireball_decorator.get_description())  # Output: "Basic Character with Fireball"
    print(fireball_decorator.get_damage())       # Output: 30

    print(invisibility_decorator.get_description())  # Output: "Basic Character with Invisibility"
    print(invisibility_decorator.get_damage())       # Output: 10

    # Combine decorators
    double_fireball_character = DoubleDamageDecorator(FireballDecorator(character))
    print(double_fireball_character.get_description())  # Output: "Basic Character with Double Damage with Fireball"
    print(double_fireball_character.get_damage())       # Output: 60

    invisibility_double_fireball_character = InvisibilityDecorator(double_fireball_character)
    print(invisibility_double_fireball_character.get_description())  # Output: "Basic Character with Invisibility
with Double Damage with Fireball"
    print(invisibility_double_fireball_character.get_damage())       # Output: 60
```

Composite Pattern

Organizes objects into tree structures.

When to Use

1. **Hierarchical Structures**

   Employ when creating tree-like systems where elements share common handling.

2. **Complex Relationships**

   Ideal for managing intricate connections among objects and simplifying software structures.

3. **Unified Element Management**

   The Singleton pattern can be used to manage resources like database connections or configuration settings efficiently.

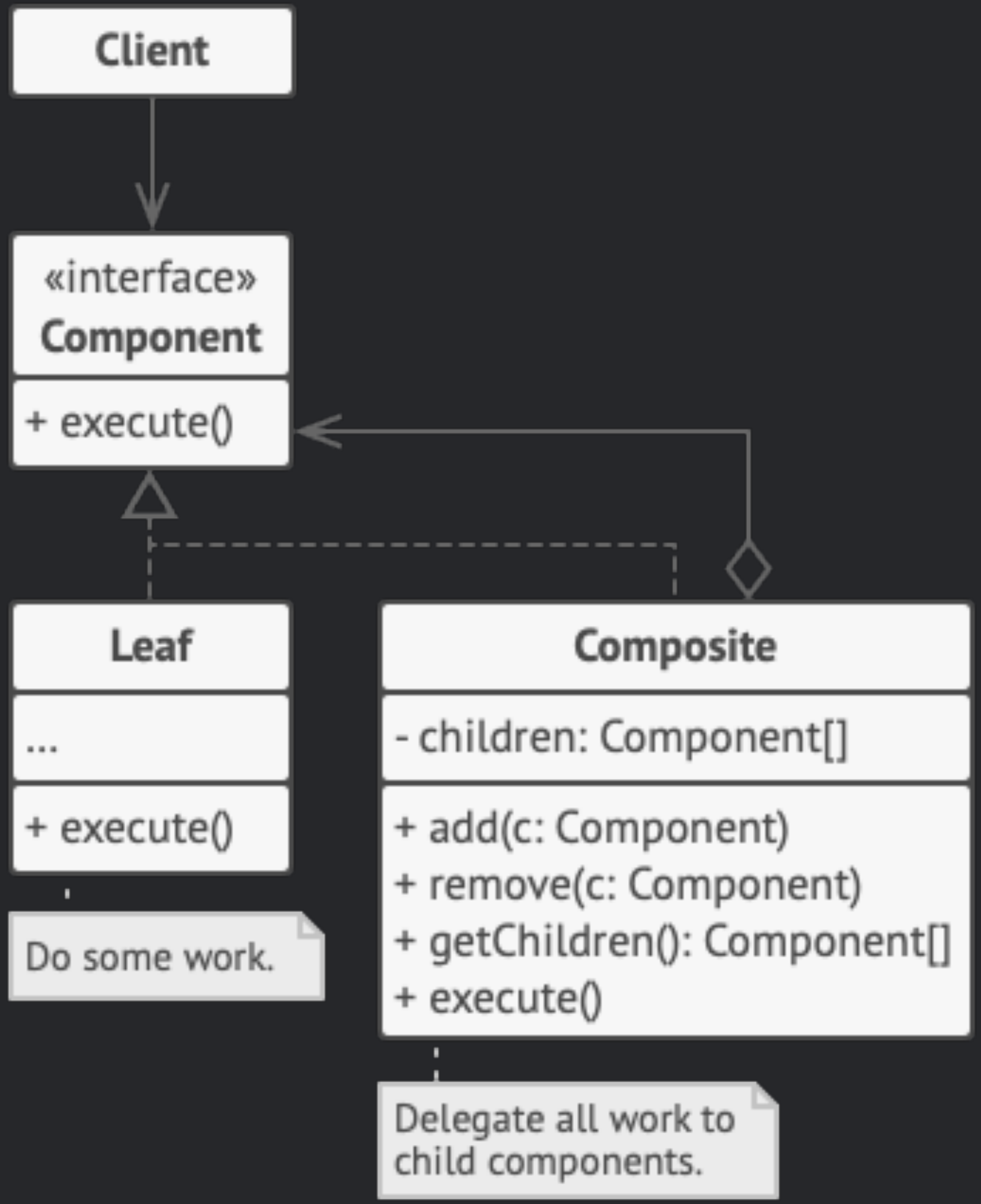
Composite: Structure

Key Components:

1 Component Interface

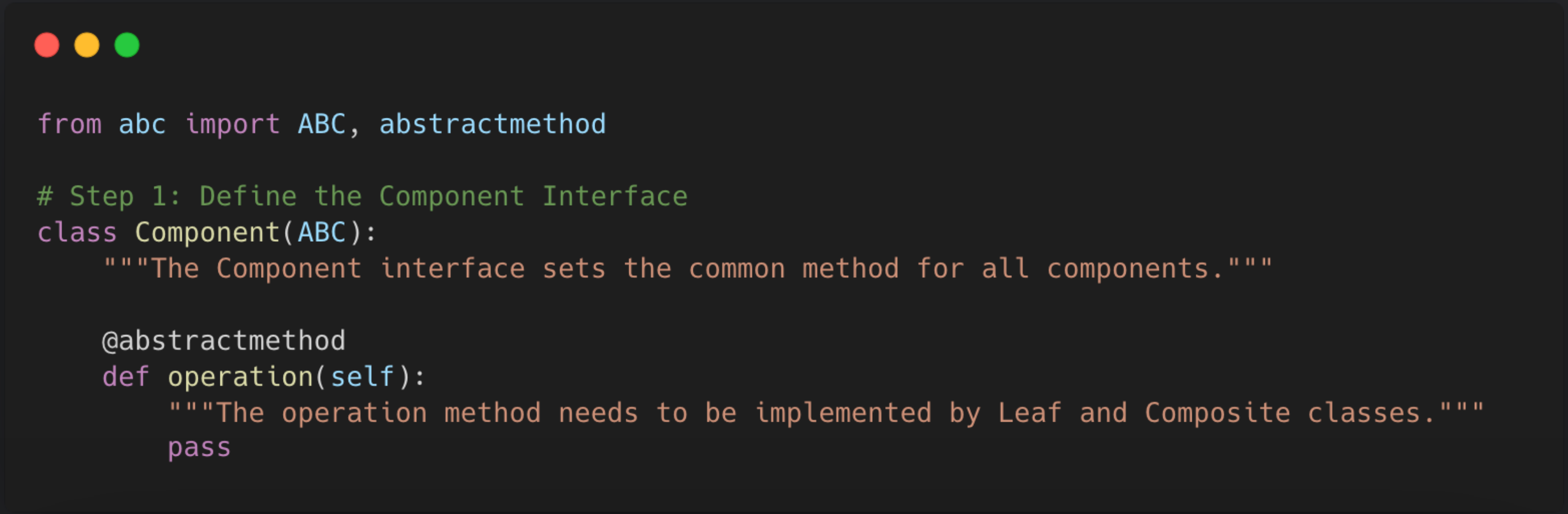
2 Leaf

3 Composite



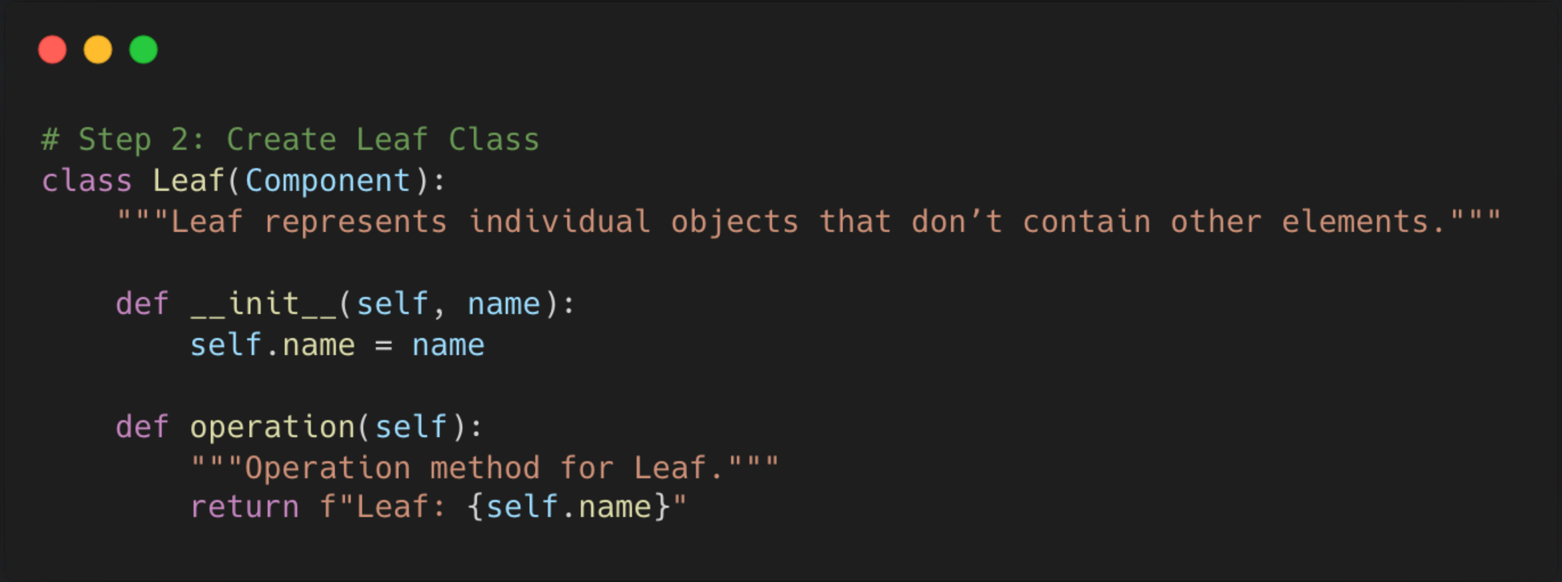
Composite Implementation in Python

1 Component Interface



```python
from abc import ABC, abstractmethod

# Step 1: Define the Component Interface
class Component(ABC):
    """The Component interface sets the common method for all components."""

    @abstractmethod
    def operation(self):
        """The operation method needs to be implemented by Leaf and Composite classes."""
        pass
```

2 Leaf



```python
# Step 2: Create Leaf Class
class Leaf(Component):
    """Leaf represents individual objects that don’t contain other elements."""

    def __init__(self, name):
        self.name = name

    def operation(self):
        """Operation method for Leaf."""
        return f"Leaf: {self.name}"
```

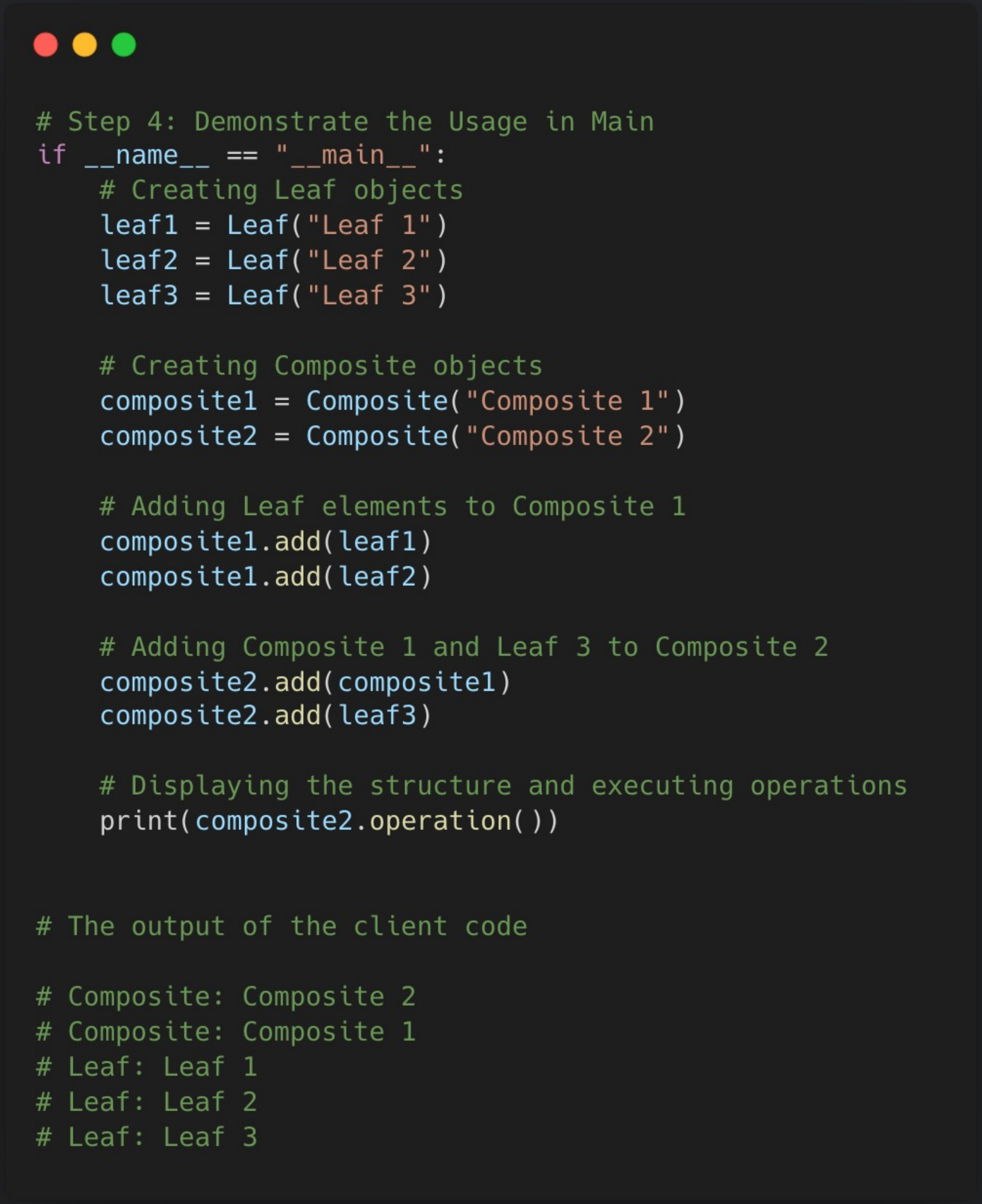
Composite Implementation in Python

3 Composite

```python
# Step 3: Create Composite Class
class Composite(Component):
    """Composite acts as a container that can hold both Leaf and other Composite instances."""

    def __init__(self, name):
        self.name = name
        self.children = []

    def add(self, component):
        """Method to add elements to the Composite."""
        self.children.append(component)

    def remove(self, component):
        """Method to remove elements from the Composite."""
        self.children.remove(component)

    def operation(self):
        """Operation method for Composite."""
        results = [f"Composite: {self.name}"]
        for child in self.children:
            results.append(child.operation())
        return "\n".join(results)
```

Composite Implementation in Python

Main Client Code



```python
# Step 4: Demonstrate the Usage in Main
if __name__ == "__main__":
    # Creating Leaf objects
    leaf1 = Leaf("Leaf 1")
    leaf2 = Leaf("Leaf 2")
    leaf3 = Leaf("Leaf 3")

    # Creating Composite objects
    composite1 = Composite("Composite 1")
    composite2 = Composite("Composite 2")

    # Adding Leaf elements to Composite 1
    composite1.add(leaf1)
    composite1.add(leaf2)

    # Adding Composite 1 and Leaf 3 to Composite 2
    composite2.add(composite1)
    composite2.add(leaf3)

    # Displaying the structure and executing operations
    print(composite2.operation())


# The output of the client code

# Composite: Composite 2
# Composite: Composite 1
# Leaf: Leaf 1
# Leaf: Leaf 2
# Leaf: Leaf 3
```

Facade Pattern

Simplifies access to complex subsystems with a unified interface.

When to Use

1. **Simplified Interface**

   Employ it for straightforward access to intricate subsystems, shielding users from complexities.

2. **Managing Complex Subsystems**

   Use it to streamline access to commonly used subsystem features, reducing client configuration and code.

3. **Layered Subsystem Structure**

   Apply it to create clear entry points in subsystem layers, reducing inter-subsystem coupling via centralized communication.

Facade: Structure

Key Components:

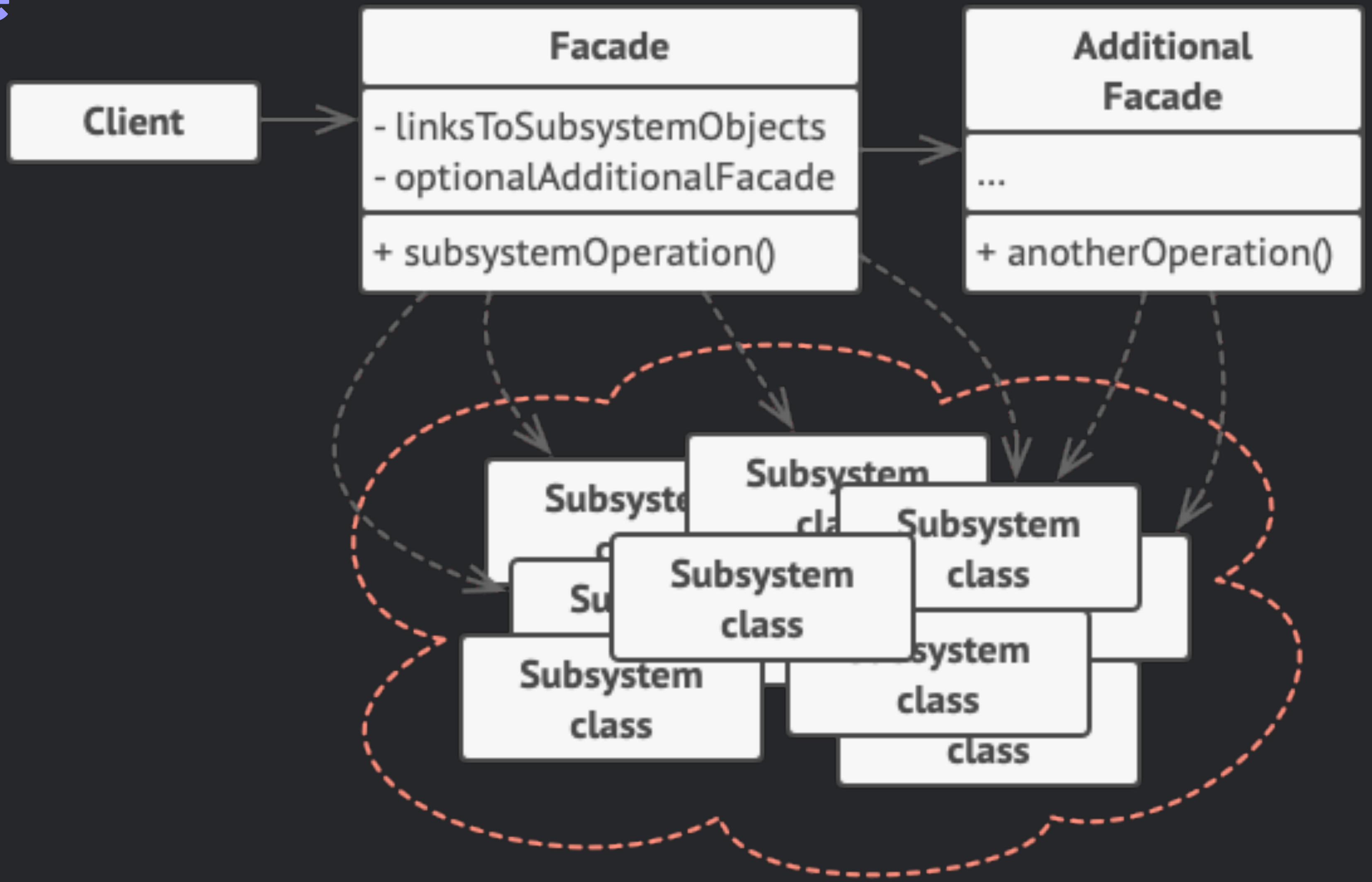
1 Facade Class

2 Additional Facades

3 Complex Subsystem

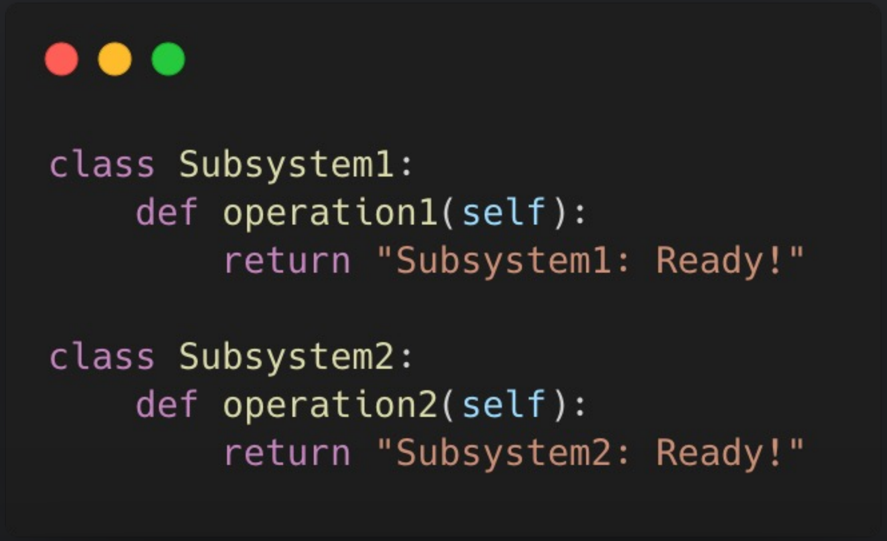
4 Subsystem Classes

5 Client



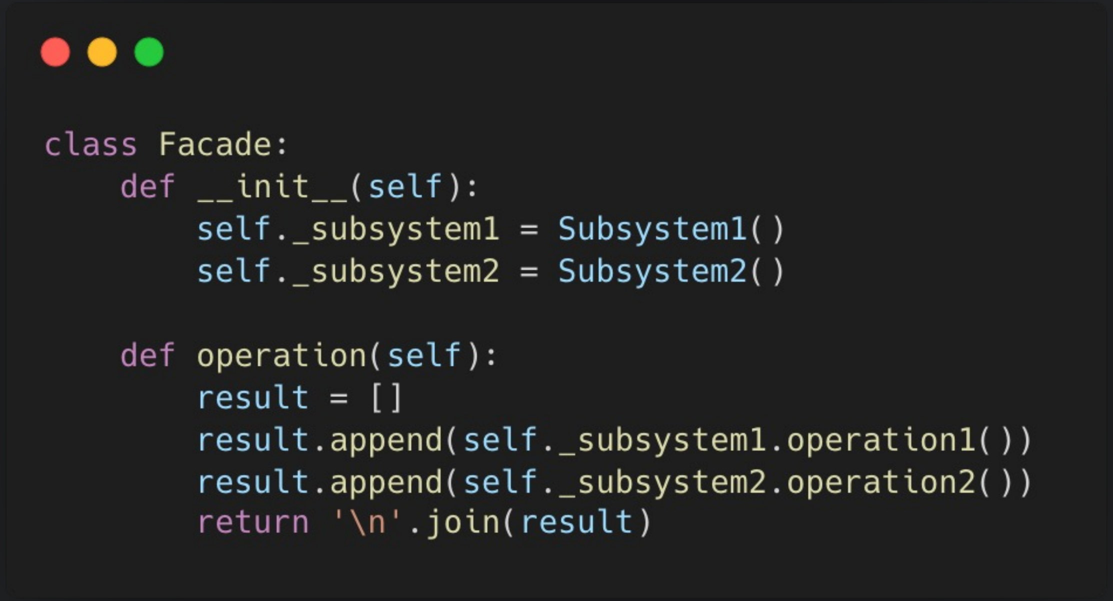
Facade Implementation in Python

1 Subsystem Classes



```python
class Subsystem1:
    def operation1(self):
        return "Subsystem1: Ready!"

class Subsystem2:
    def operation2(self):
        return "Subsystem2: Ready!"
```

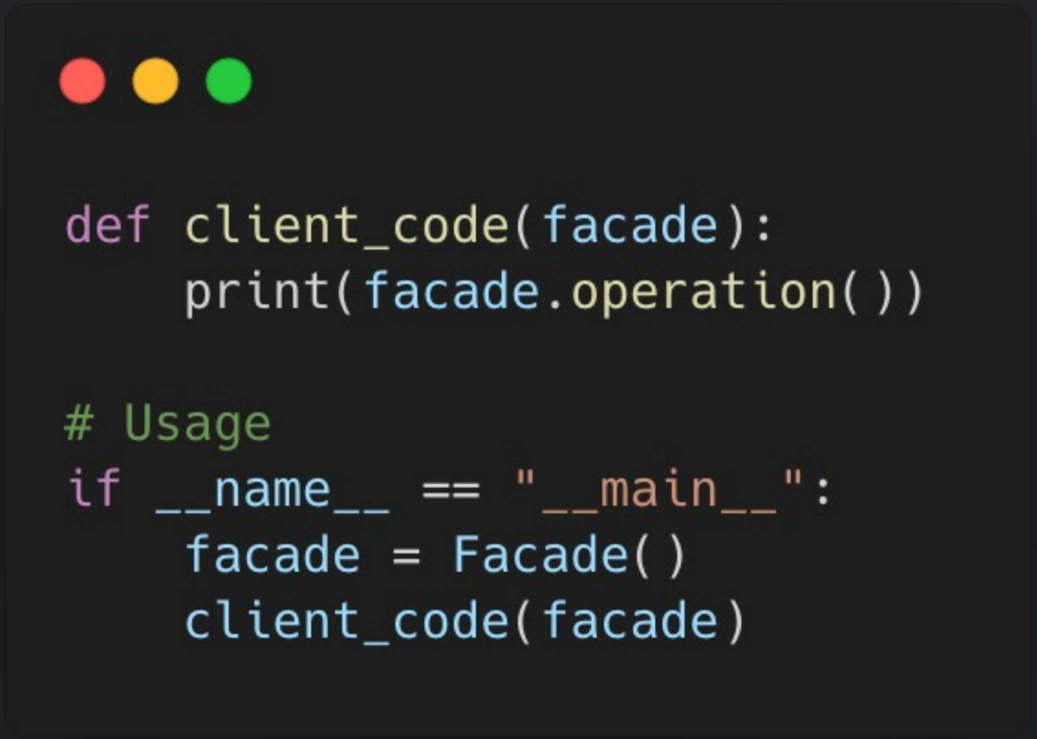
2 Implement Facade Class



```python
class Facade:
    def __init__(self):
        self._subsystem1 = Subsystem1()
        self._subsystem2 = Subsystem2()

    def operation(self):
        result = []
        result.append(self._subsystem1.operation1())
        result.append(self._subsystem2.operation2())
        return '\n'.join(result)
```

Facade Implementation in Python

3 Utilize Facade in Client Code



```python
def client_code(facade):
    print(facade.operation())

# Usage
if __name__ == "__main__":
    facade = Facade()
    client_code(facade)
```

Proxy Pattern

Acts as a placeholder to control access to another object.

When to Use

1 **Expensive Object Loading**

When dealing with complex or resource-intensive objects, consider the Virtual Proxy. It acts as a placeholder, loading the full object on demand, and optimizing resource usage.

2 **Remote Object Access**

If the original object is in a different address space and you need local-like interaction, opt for the Remote Proxy. It manages connection details, making remote objects appear local.

3 **Enhanced Security**

Employ the Proxy pattern for added security. The Protection Proxy enforces controlled access based on client rights, safeguarding sensitive resources.

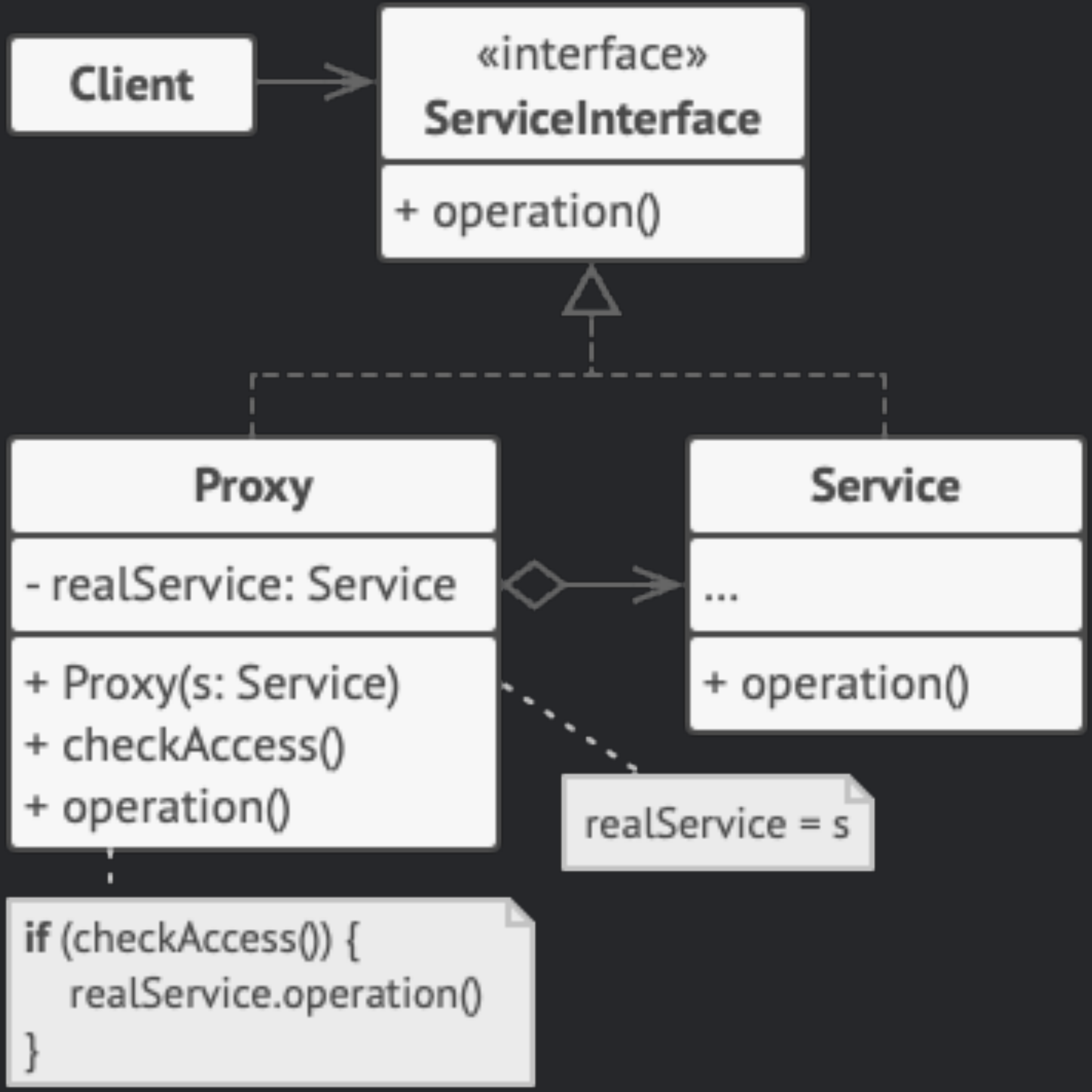
Proxy: Structure

Key Components:

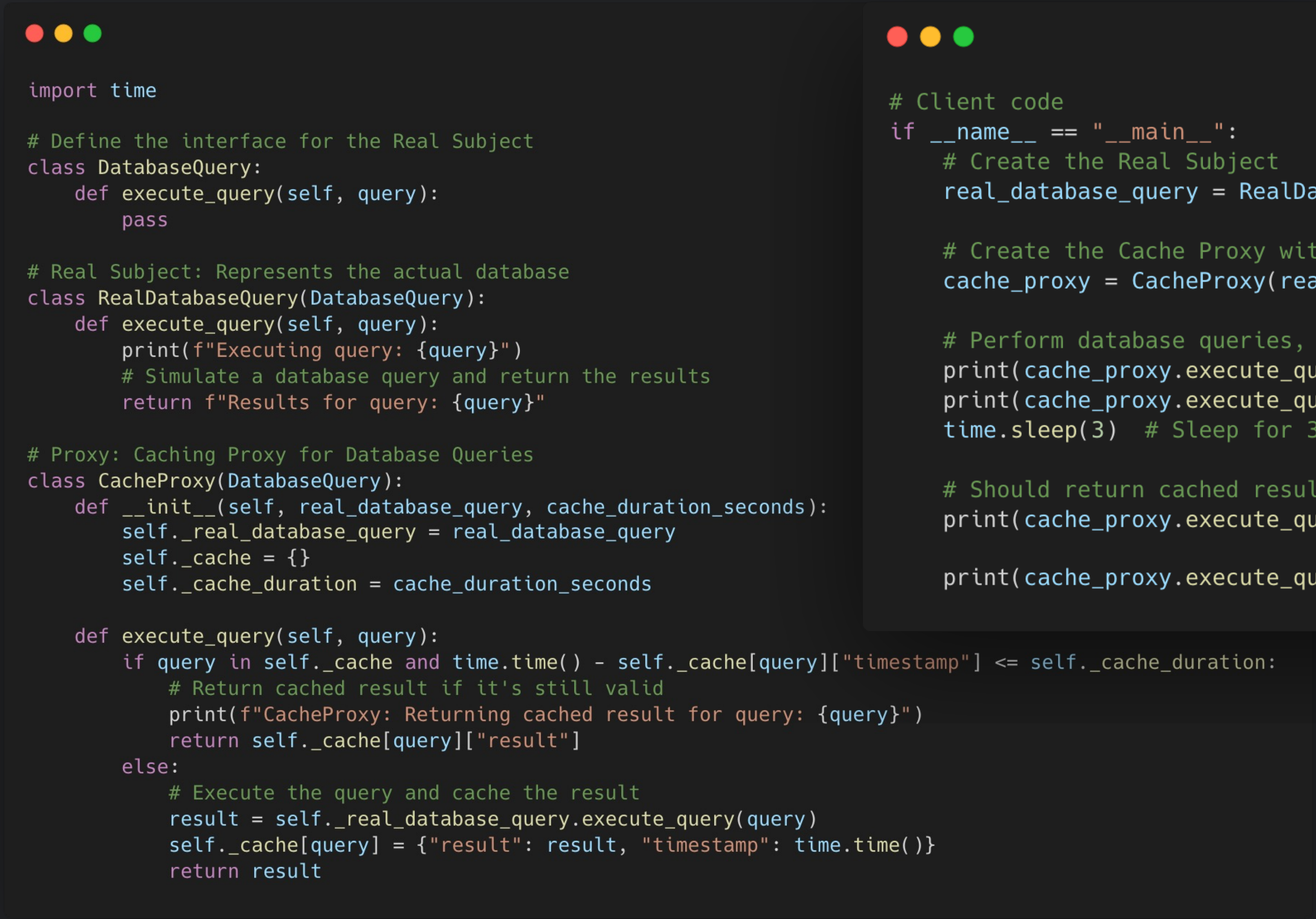
1 Proxy Interface

2 Real Subject

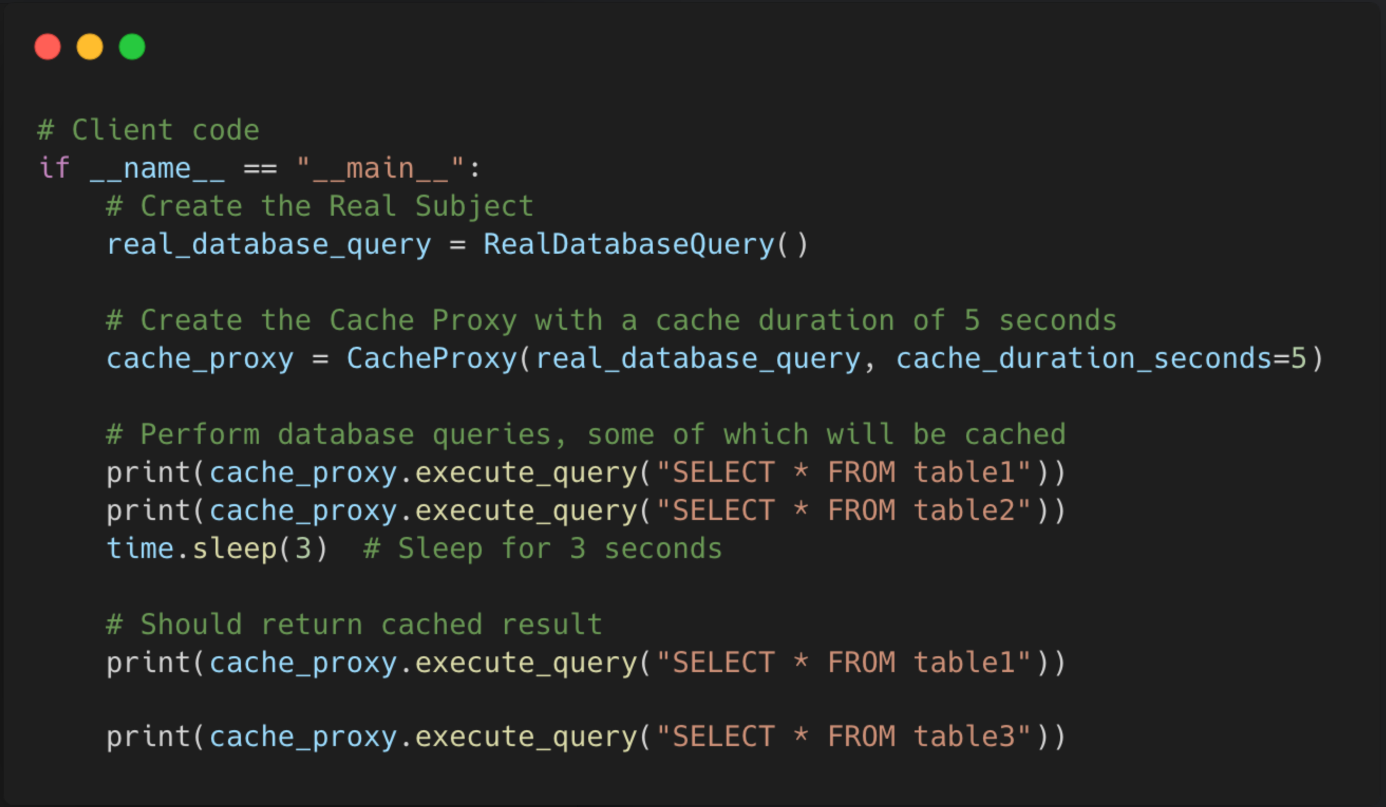
3 Proxy



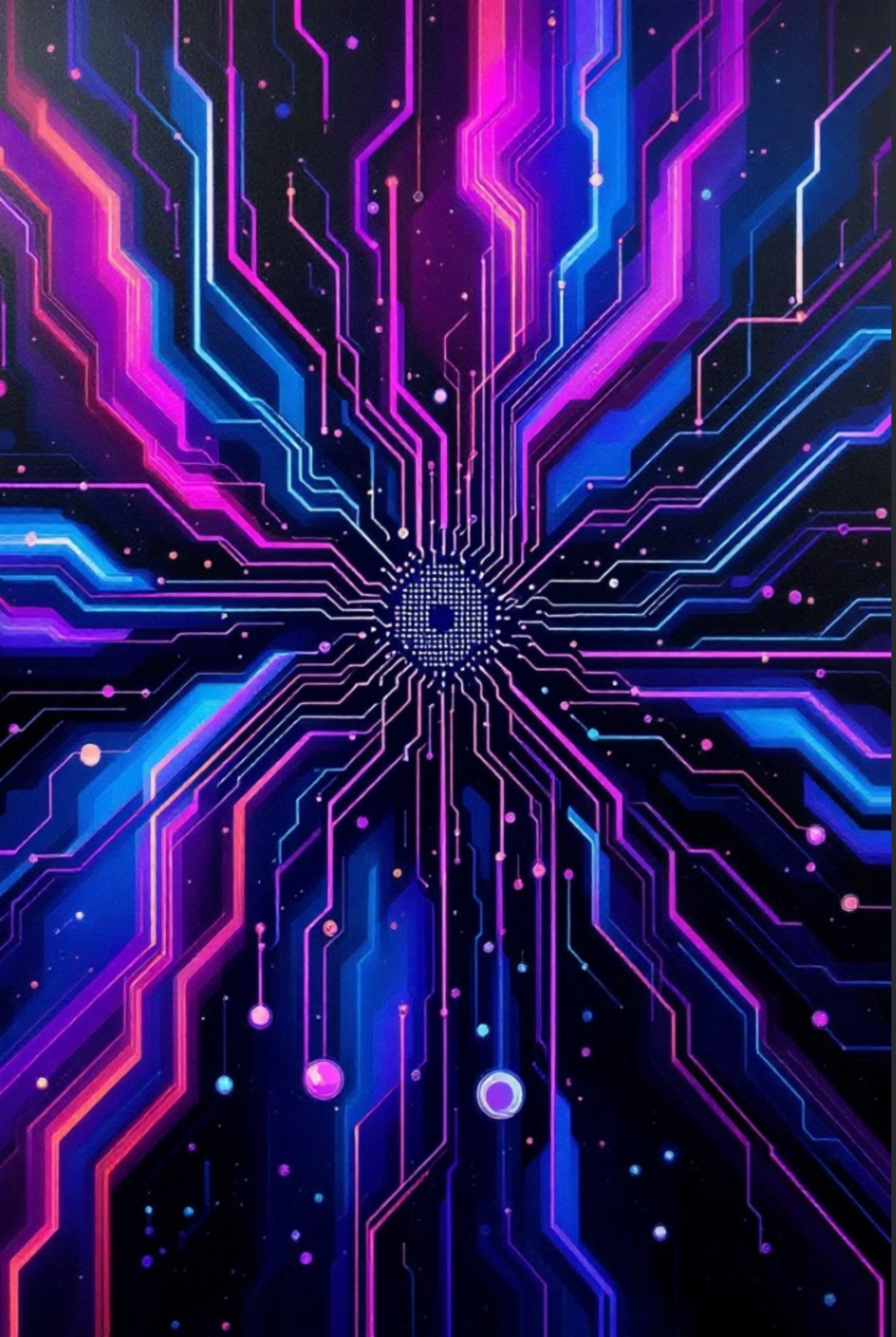
Caching Proxy for Database Queries



```python
import time

# Define the interface for the Real Subject
class DatabaseQuery:
    def execute_query(self, query):
        pass

# Real Subject: Represents the actual database
class RealDatabaseQuery(DatabaseQuery):
    def execute_query(self, query):
        print(f"Executing query: {query}")
        # Simulate a database query and return the results
        return f"Results for query: {query}"

# Proxy: Caching Proxy for Database Queries
class CacheProxy(DatabaseQuery):
    def __init__(self, real_database_query, cache_duration_seconds):
        self._real_database_query = real_database_query
        self._cache = {}
        self._cache_duration = cache_duration_seconds

    def execute_query(self, query):
        if query in self._cache and time.time() - self._cache[query]["timestamp"] <= self._cache_duration:
            # Return cached result if it's still valid
            print(f"CacheProxy: Returning cached result for query: {query}")
            return self._cache[query]["result"]
        else:
            # Execute the query and cache the result
            result = self._real_database_query.execute_query(query)
            self._cache[query] = {"result": result, "timestamp": time.time()}
            return result
```



```python
# Client code
if __name__ == "__main__":
    # Create the Real Subject
    real_database_query = RealDatabaseQuery()

    # Create the Cache Proxy with a cache duration of 5 seconds
    cache_proxy = CacheProxy(real_database_query, cache_duration_seconds=5)

    # Perform database queries, some of which will be cached
    print(cache_proxy.execute_query("SELECT * FROM table1"))
    print(cache_proxy.execute_query("SELECT * FROM table2"))
    time.sleep(3)  # Sleep for 3 seconds

    # Should return cached result
    print(cache_proxy.execute_query("SELECT * FROM table1"))

    print(cache_proxy.execute_query("SELECT * FROM table3"))
```



Behavioral Patterns in Software Engineering

Behavioral patterns focus on communication and interactions between objects in a system. These patterns provide robust solutions for managing complex object relationships and ensuring code reusability and flexibility.

JH **by Jozef Hrdý**

Introduction to Behavioral Patterns

1 Defining Behavior

Behavioral patterns address how objects interact and collaborate within a system.

2 Object Interactions

These patterns provide well-defined frameworks for managing complex relationships and interactions.

3 Flexibility and Reusability

Behavioral patterns promote code reuse and adaptability to changing requirements.

4 Common Patterns

Examples include Iterator, Mediator, Observer, Chain of Responsibility, and Command patterns.

The Iterator Pattern

1 **Traversal Logic**

Encapsulates the logic for iterating over a collection of objects.

2 **Consistent Interface**

Provides a unified way to access elements regardless of the underlying data structure.

3 **Client Independence**

Allows clients to access elements without knowing the internal structure of the collection.

Benefits of the Iterator Pattern

Enhanced Code Structure

Separates traversal logic from the collection itself, promoting clean and organized code.

Improved Reusability

Allows for flexible iteration over various data structures with a single iterator interface.

Reduced Complexity

Simplifies client code by hiding the intricacies of data structure traversal.

The Mediator Pattern

1 **Centralized Communication**

Defines a central mediator object that handles communication between other objects.

2 **Loose Coupling**

Objects don't directly interact with each other, reducing dependencies and complexity.

3 **Simplified Management**

Centralized communication management makes it easier to modify interactions and behavior.

Advantages of the Mediator Pattern

Reduced Coupling

Objects are independent of each other, allowing for easier modification and extension.

Enhanced Flexibility

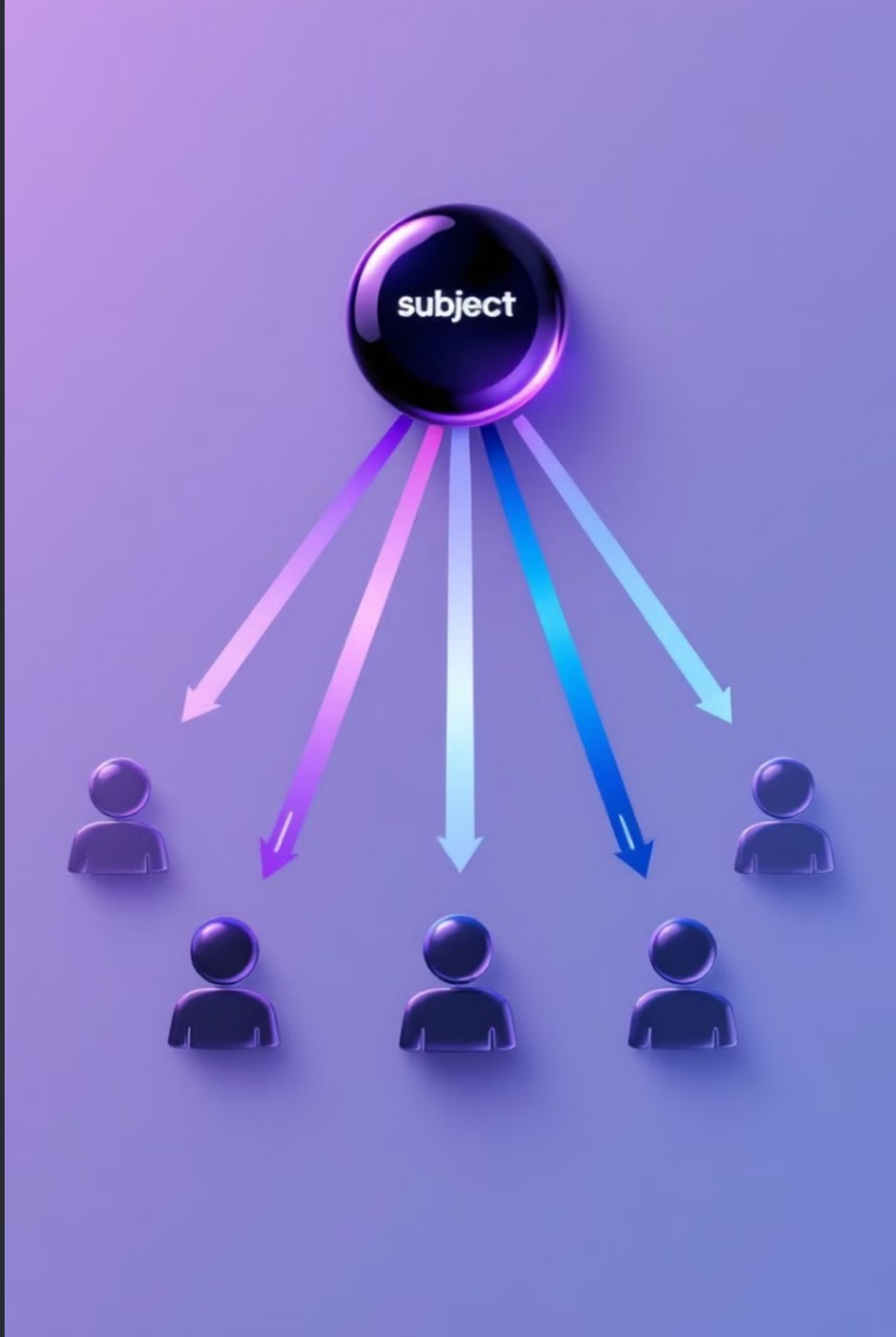
The mediator can adapt to changing communication requirements without affecting the interacting objects.

Simplified Maintenance

Changes in object interactions can be easily managed through the central mediator.

The Observer Pattern

Subject	Notifies observers of changes.
Observer	Receives notifications and updates from the subject.



Implementing the Observer Pattern

Observer Interface

Defines the methods that observers must implement to receive updates.

Attach/Detach Methods

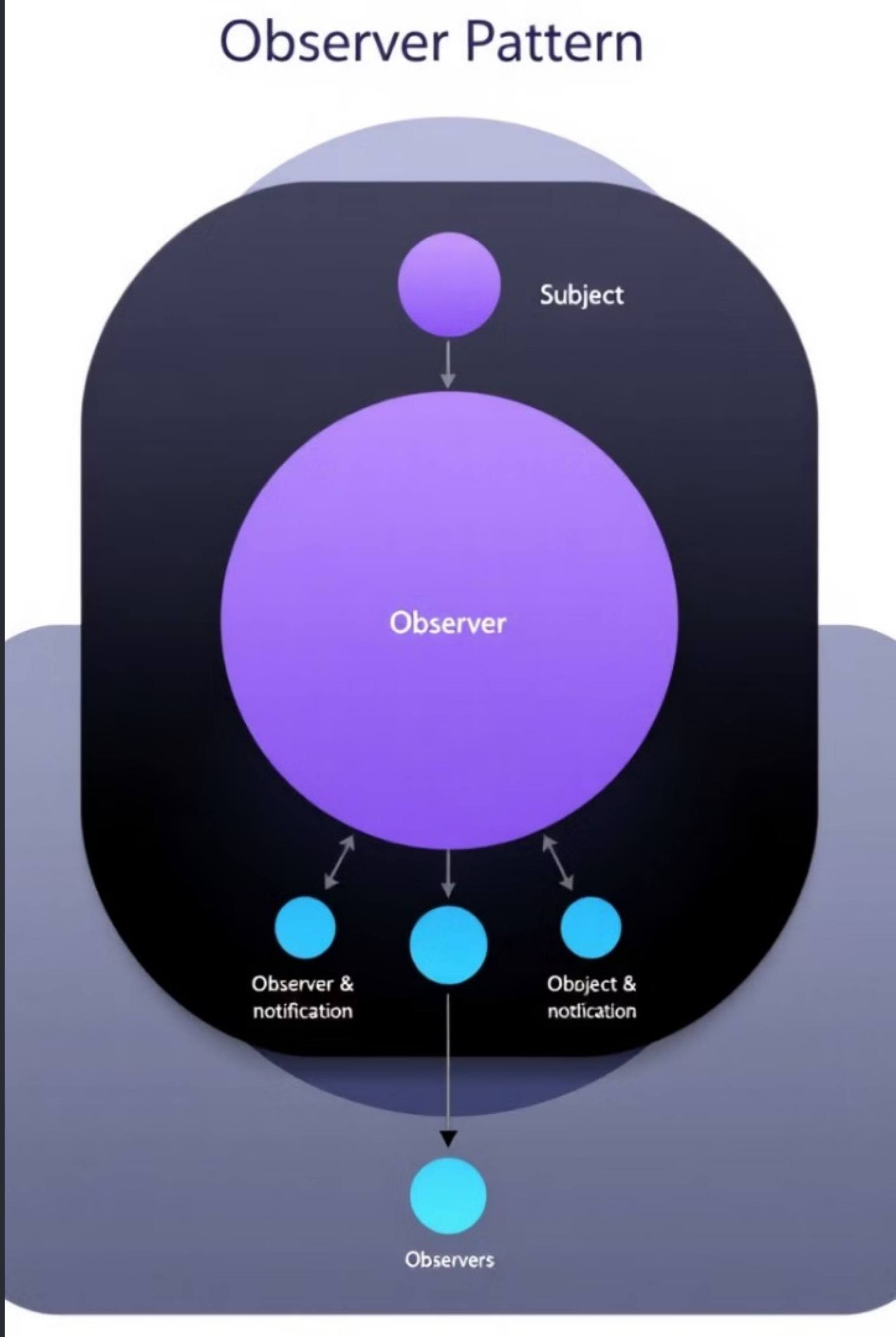
Subject provides methods for observers to subscribe and unsubscribe to updates.

Notify Method

Subject calls this method to notify observers when its state changes.

Update Method

Observers implement this method to handle updates from the subject.



Observer: Weather app

Imagine a weather tracking application that collects data from a weather station (temperature, humidity, pressure) and displays it on different types of devices, such as a mobile app, a web dashboard, or IoT devices.

Whenever any meteorological data changes, all connected devices must be automatically notified so they can display the updated information.

1

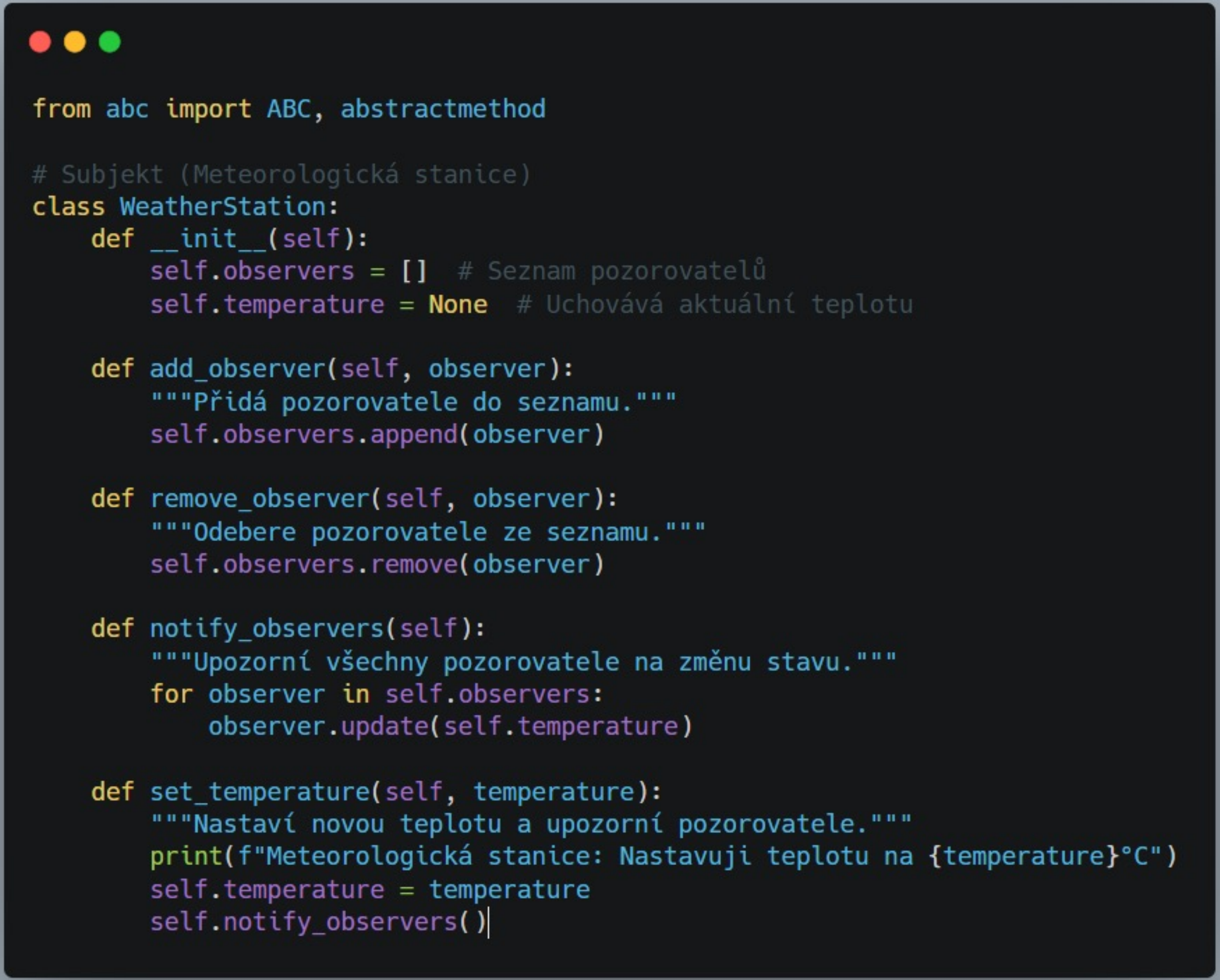
Instead of having each device poll the station for changes, we use the Observer pattern, where:
The weather station is the **subject**.

All the devices are **observers**.

Whenever there is a change in the meteorological data, the weather station (subject) notifies all registered devices (observers), allowing them to update and display the latest information.

Subject (WeatherStation): Maintains a list of observers and notifies them about state changes.

When the temperature changes (via the set_temperature method), it notifies all registered observers.



```python
from abc import ABC, abstractmethod

# Subjekt (Meteorologická stanice)
class WeatherStation:
    def __init__(self):
        self.observers = []  # Seznam pozorovatelů
        self.temperature = None  # Uchovává aktuální teplotu

    def add_observer(self, observer):
        """Přidá pozorovatele do seznamu."""
        self.observers.append(observer)

    def remove_observer(self, observer):
        """Odebere pozorovatele ze seznamu."""
        self.observers.remove(observer)

    def notify_observers(self):
        """Upozorní všechny pozorovatele na změnu stavu."""
        for observer in self.observers:
            observer.update(self.temperature)

    def set_temperature(self, temperature):
        """Nastaví novou teplotu a upozorní pozorovatele."""
        print(f"Meteorologická stanice: Nastavuji teplotu na {temperature}°C")
        self.temperature = temperature
        self.notify_observers()
```

Comparing the Patterns

1

Iterator

The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This is particularly useful when dealing with complex data structures where direct access to elements might be inefficient or difficult. The iterator encapsulates the traversal logic, allowing for flexible iteration and decoupling the collection from the traversal algorithm.

2

Mediator

The Mediator pattern defines an object that encapsulates how a set of objects interact. Instead of objects referring to each other explicitly, they communicate through the mediator. This reduces dependencies between objects, making the system more flexible and maintainable. Adding new objects or modifying existing interactions becomes simpler as all changes are managed through the central mediator.

3

Observer

The Observer pattern establishes a one-to-many dependency between objects. A subject maintains a list of its dependents (observers) and notifies them automatically of any state changes. This allows for loose coupling between the subject and observers, as the subject doesn't need to know the specifics of its observers. Observers only need to be aware of the update mechanism provided by the subject.

Conclusion and Key Takeaways

Behavioral patterns provide powerful solutions for handling complex object interactions. By understanding these patterns, developers can create more flexible, maintainable, and reusable code. Choose the appropriate pattern based on the specific communication and interaction needs of your system.