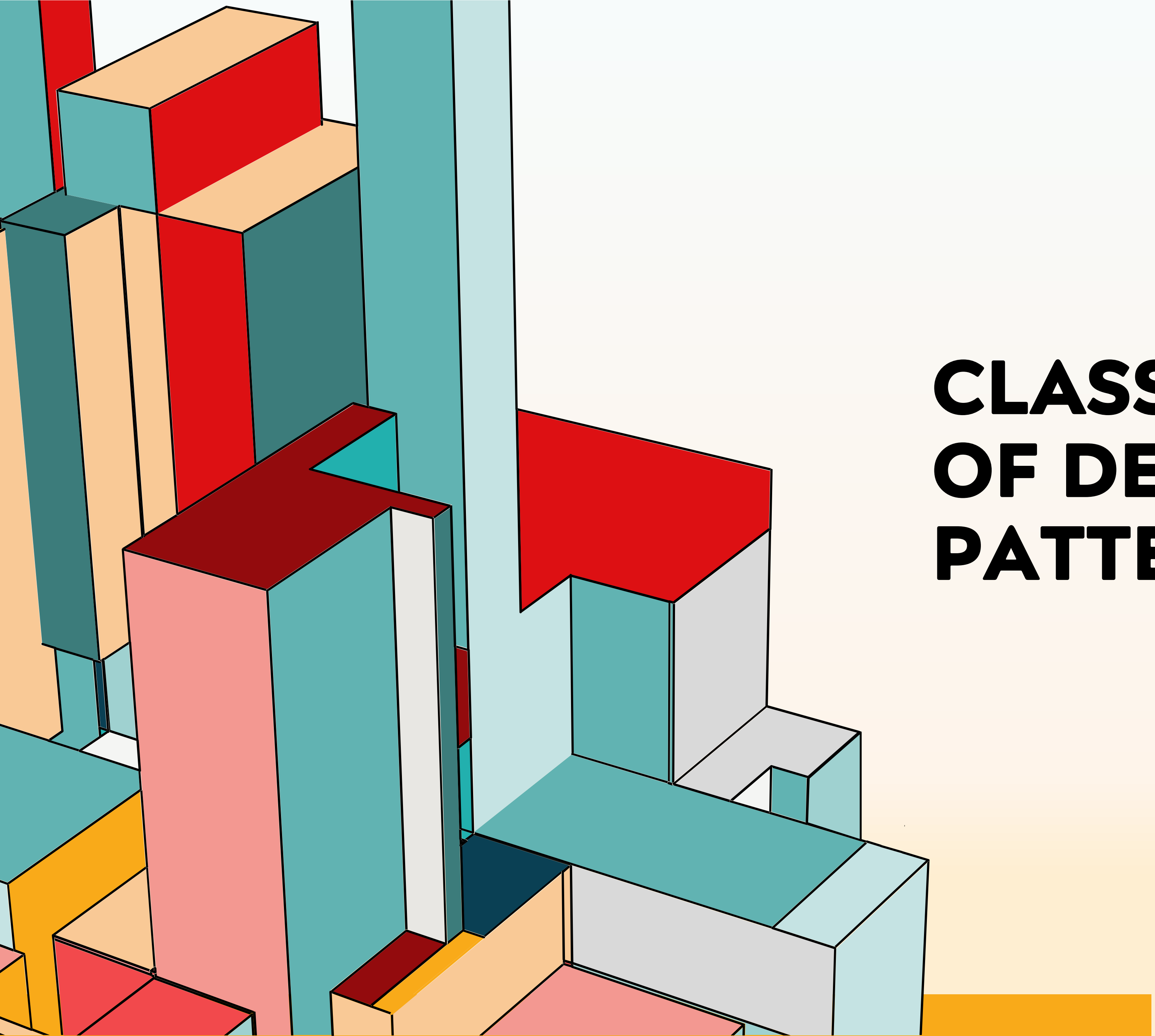
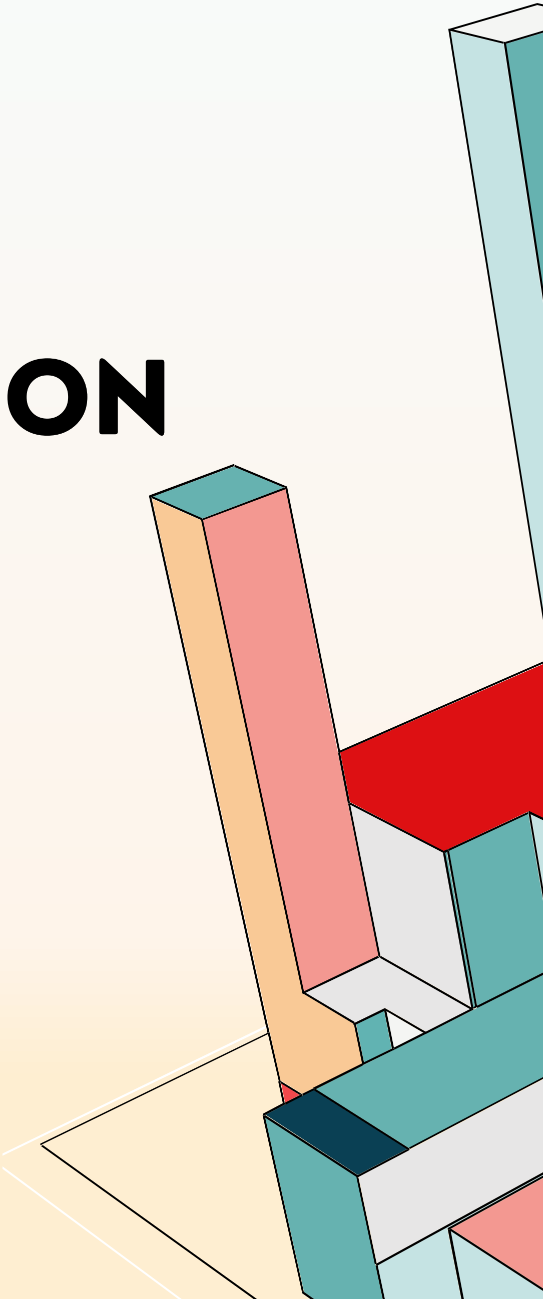


CLASSIFICATION OF DESIGN PATTERNS



INTRODUCTION TO DESIGN PATTERNS

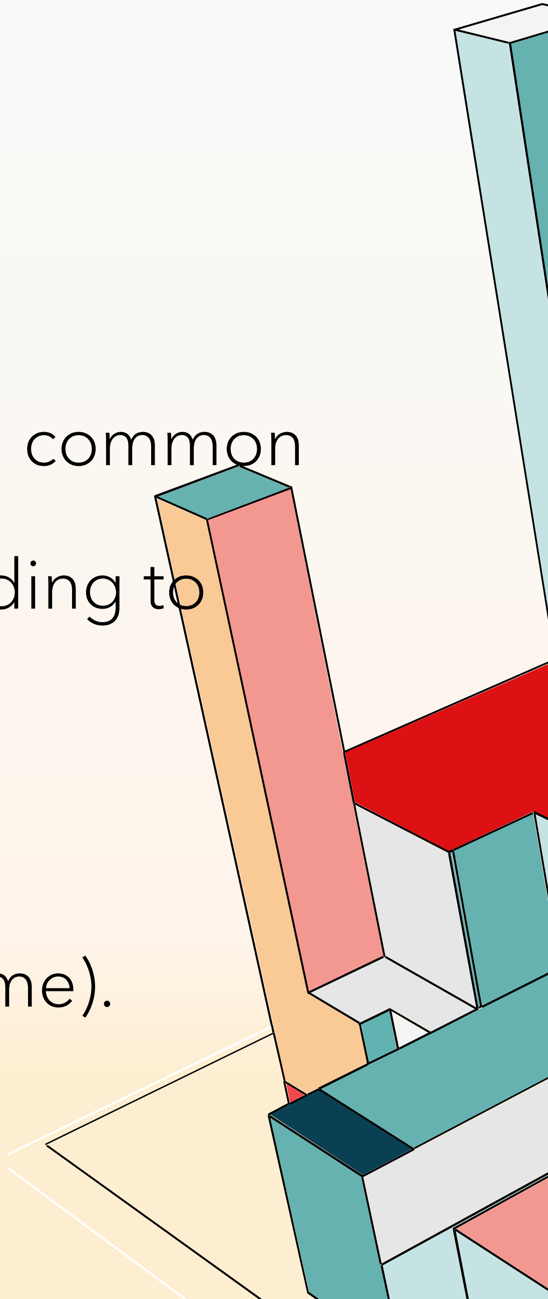
- Definition: Design patterns are general, reusable solutions to common design problems in software development.

- Importance:
- - Provide standardized solutions to design issues.
- - Enhance code readability, maintainability, and flexibility.
- - Facilitate better software architecture by promoting best practices.

WHY CLASSIFY DESIGN PATTERNS?

- Purpose of Classification:
- - Helps developers identify the right pattern quickly.
- - Supports faster learning by grouping patterns based on common goals.
- - Simplifies pattern selection by categorizing them according to their primary intent.

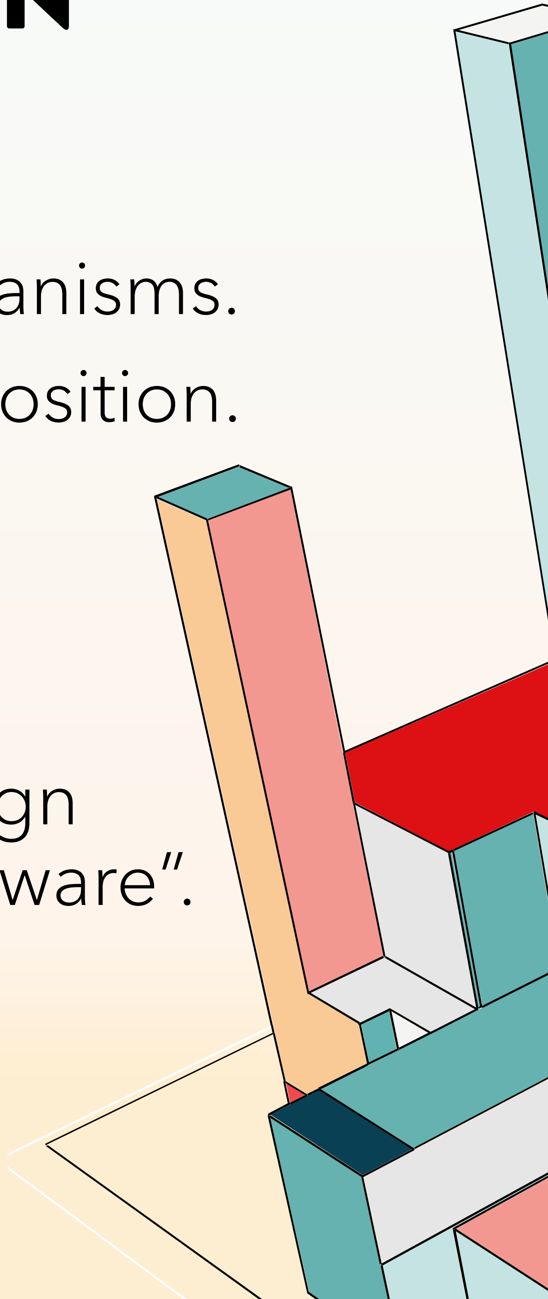
- Classification is based on:
- - Intent: What the pattern aims to achieve.
- - Scope: Class-level (compile-time) or Object-level (runtime).
- - Focus: Object creation, composition, or interaction.



THREE MAIN CATEGORIES OF DESIGN PATTERNS

- - Creational Patterns: Deal with object creation mechanisms.
- - Structural Patterns: Focus on class and object composition.
- - Behavioral Patterns: Handle object interaction and responsibilities.

- Originated from the Gang of Four (GoF) book: "Design Patterns: Elements of Reusable Object-Oriented Software".



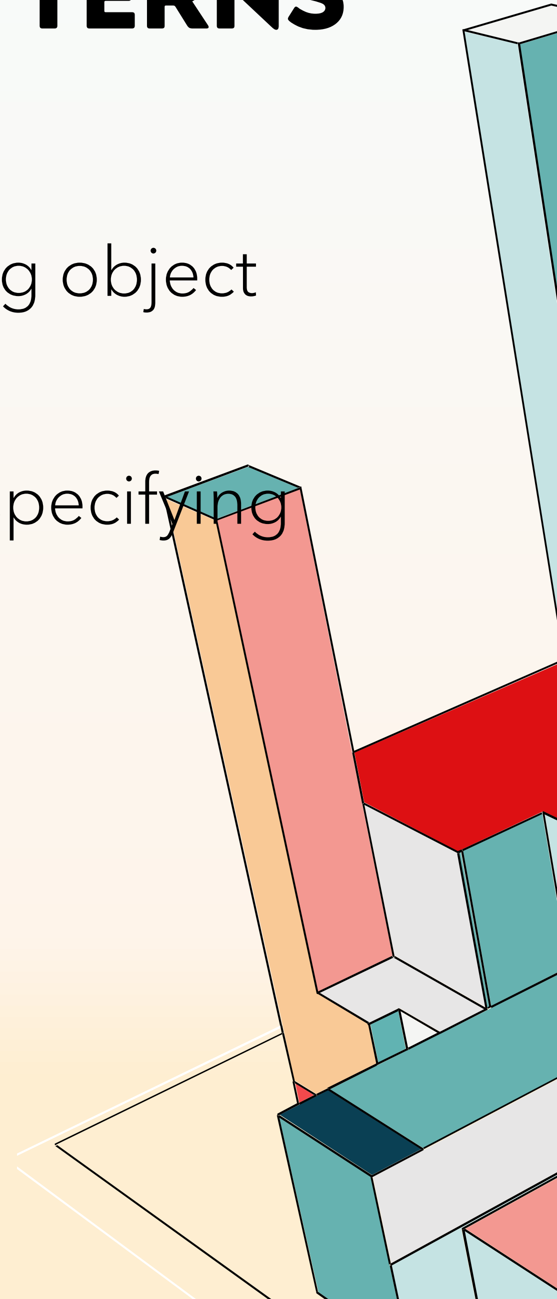
UNDERSTANDING CREATIONAL PATTERNS

- Objective: Abstracts the instantiation process, decoupling object creation from client code.

- Focus: Providing flexible ways to create objects without specifying exact classes.

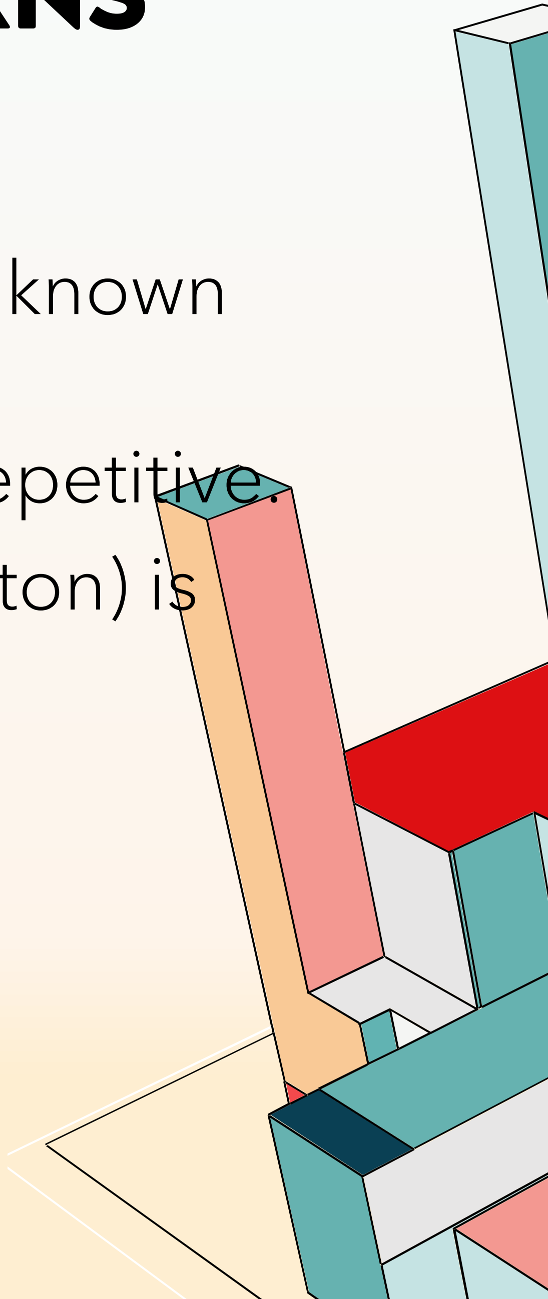
- Key Characteristics:
- - Encapsulates object creation logic.
- - Provides control over object creation.

- Scope: Primarily object-level (runtime).



WHEN TO USE CREATIONAL PATTERNS

- When the exact type or configuration of objects is unknown until runtime.
- When object creation logic becomes complex and repetitive.
- When ensuring a single instance of an object (Singleton) is necessary.



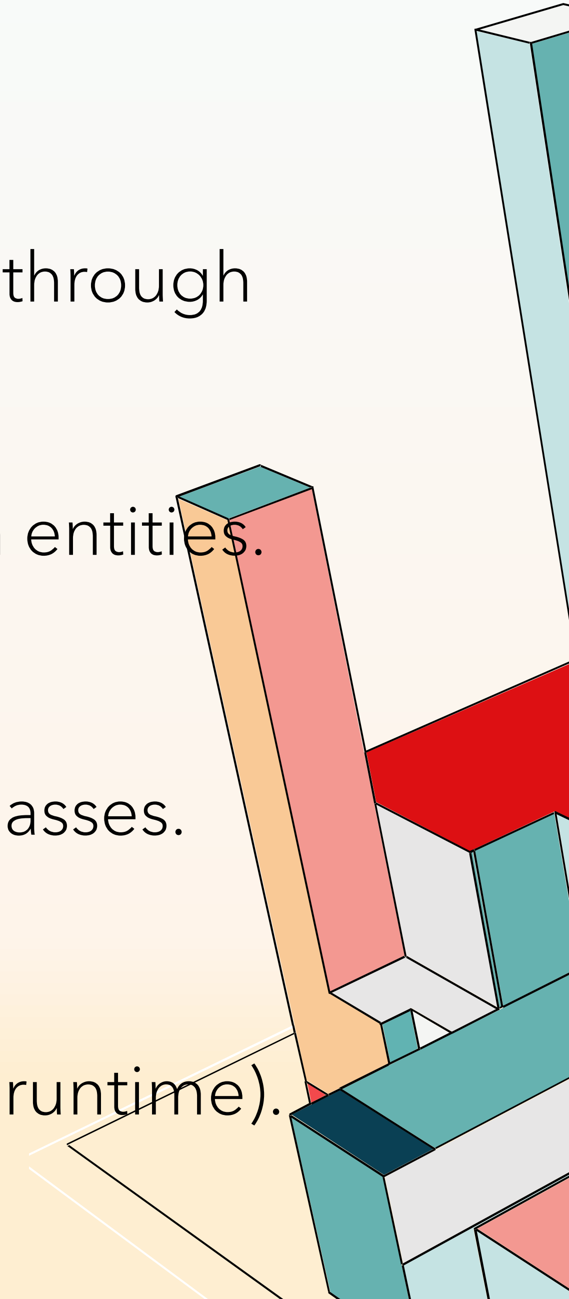
UNDERSTANDING STRUCTURAL PATTERNS

- Objective: Simplifying the structure of complex systems through flexible class and object composition.

- Focus: Identifying and simplifying relationships between entities.

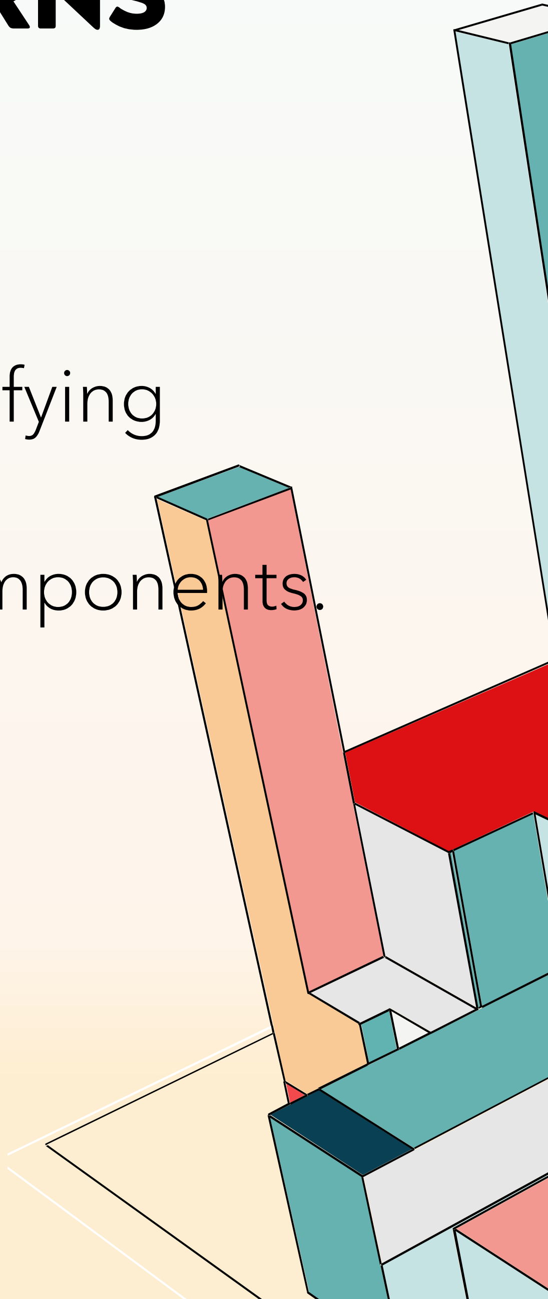
- Key Characteristics:
- - Improves system flexibility by organizing objects and classes.
- - Supports composition over inheritance.

- Scope: Both class-level (compile-time) and object-level (runtime).



WHEN TO USE STRUCTURAL PATTERNS

- When building complex hierarchies of objects.
- When you need to extend functionality without modifying existing code.
- When bridging incompatible interfaces between components.



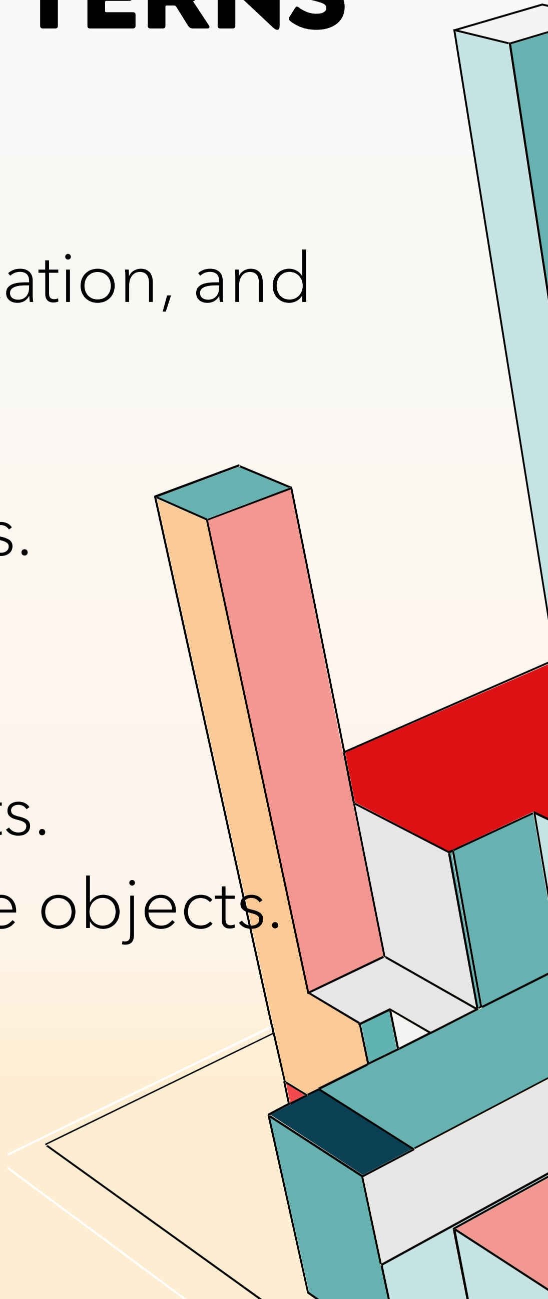
UNDERSTANDING BEHAVIORAL PATTERNS

- Objective: Concerned with object interaction, communication, and responsibility delegation.

- Focus: Ensuring loosely coupled and cooperative objects.

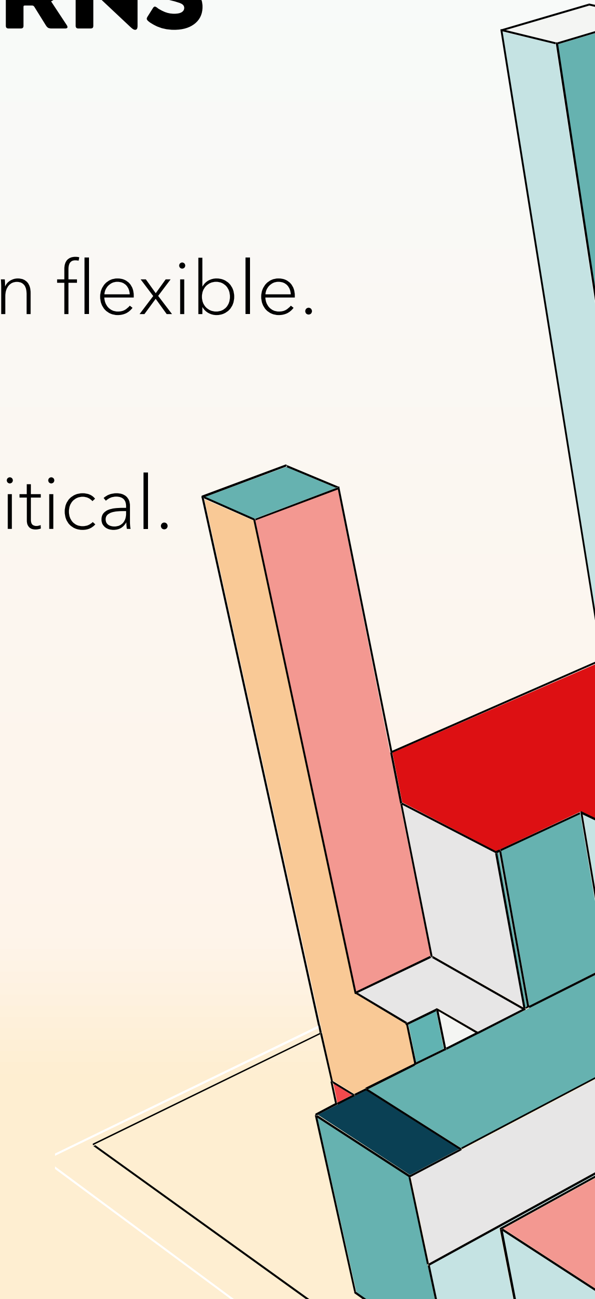
- Key Characteristics:
- - Defines clear communication protocols between objects.
- - Encapsulates behavior variations within interchangeable objects.

- Scope: Primarily object-level (runtime).



WHEN TO USE BEHAVIORAL PATTERNS

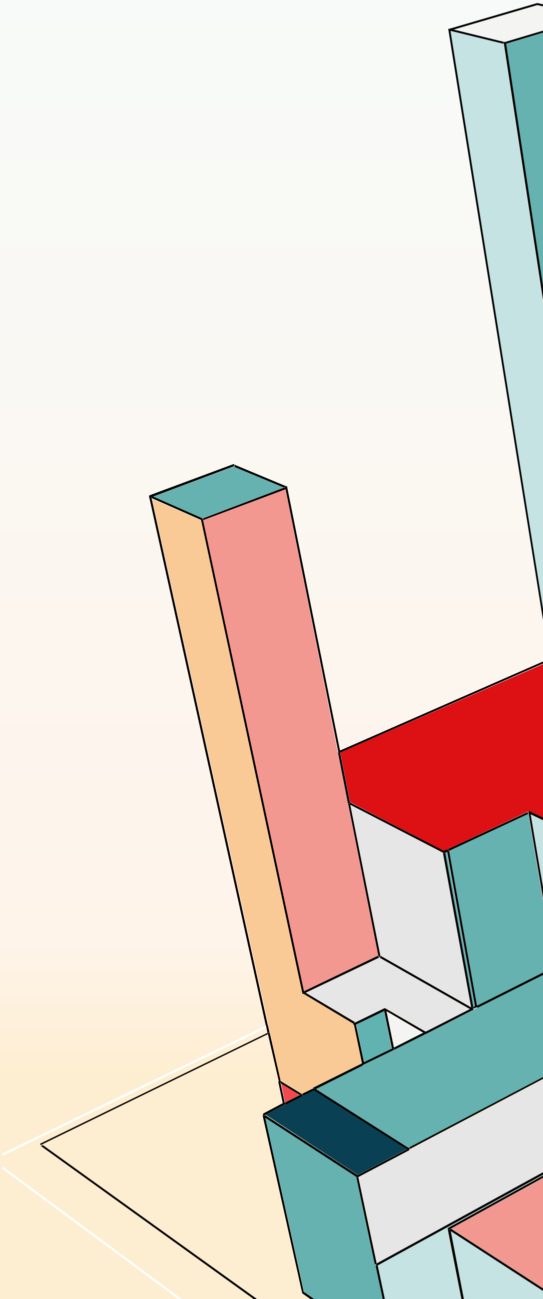
- When communication between objects must remain flexible.
- When different behaviors are needed at runtime.
- When avoiding tight coupling between classes is critical.



CREATIONAL VS. STRUCTURAL VS. BEHAVIORAL PATTERNS

- Primary Concern:
- - Creational: Object creation mechanisms
- - Structural: Object composition
- - Behavioral: Object interaction

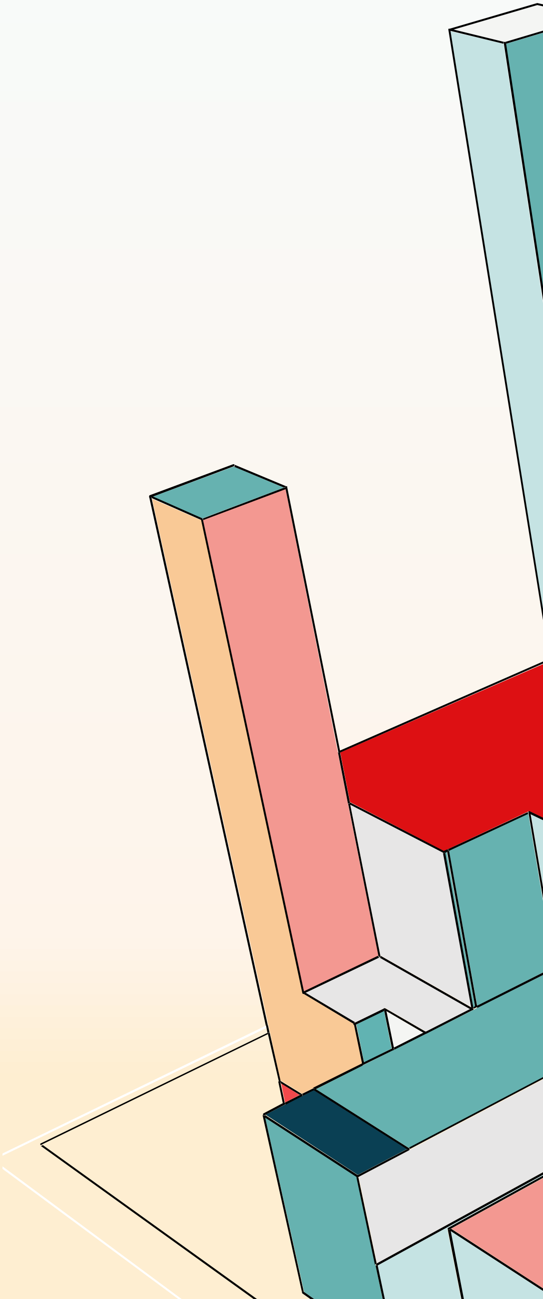
- Key Goal:
- - Creational: Flexibility in object creation
- - Structural: Simplifying structure
- - Behavioral: Decoupling communication



CREATIONAL VS. STRUCTURAL VS. BEHAVIORAL PATTERNS

- Focus on:
- - Creational: Instantiation logic
- - Structural: Relationships & interfaces
- - Behavioral: Object responsibilities

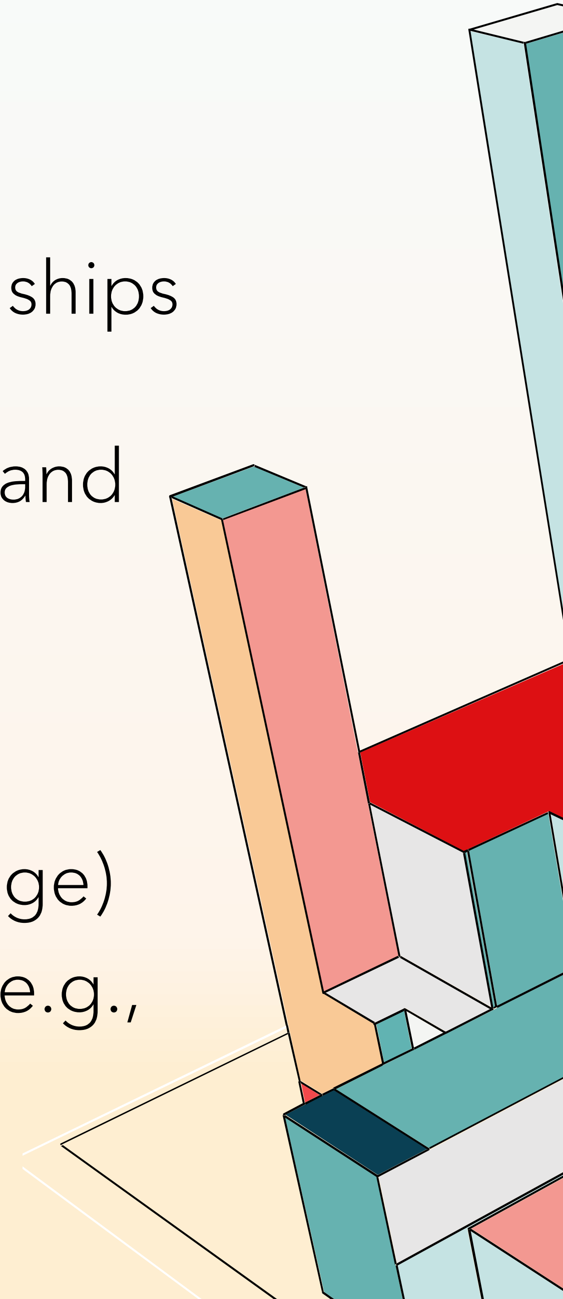
- Typical Outcome:
- - Creational: Abstracted creation process
- - Structural: Simplified object structure
- - Behavioral: Clear collaboration rules



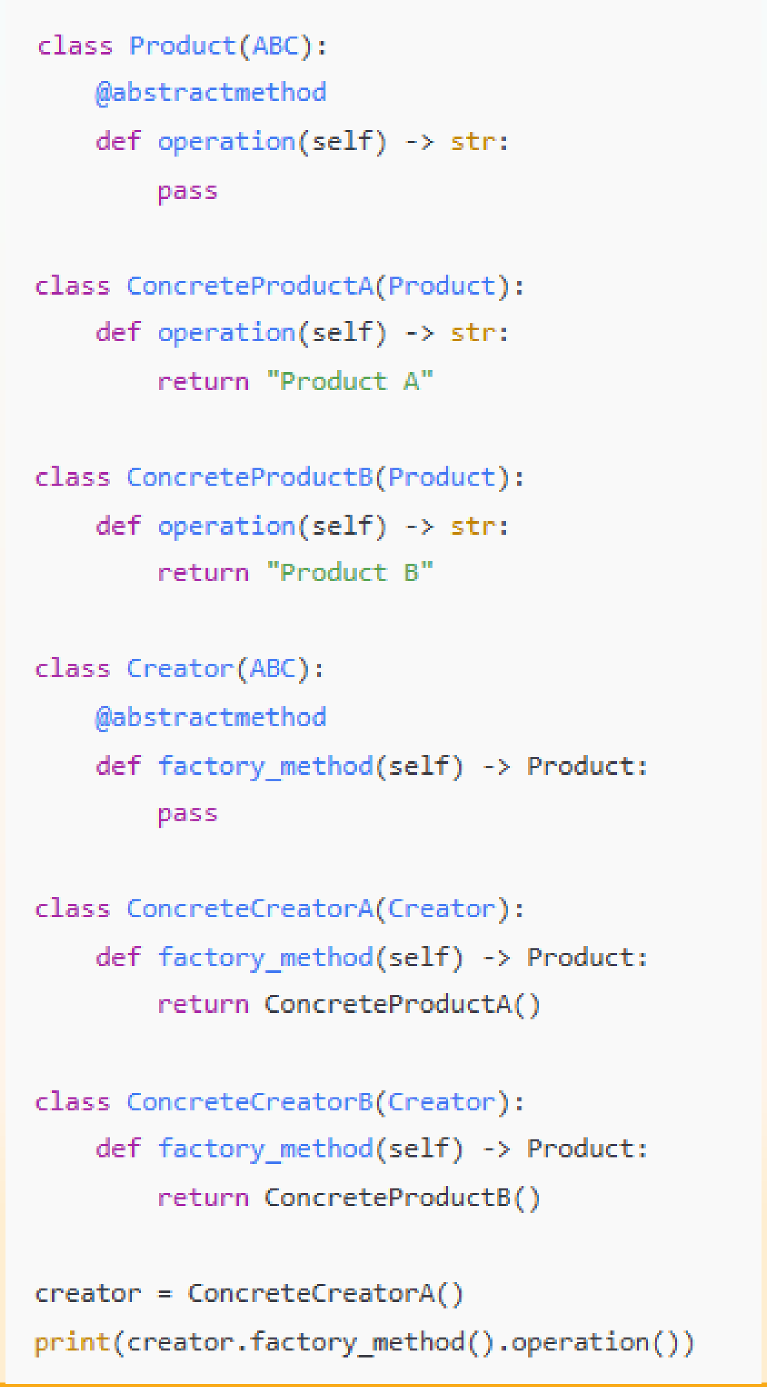
SCOPE OF PATTERNS

- Class Scope: Focus on inheritance and static relationships (compile-time).
- Object Scope: Focus on runtime object interactions and dynamic behavior.

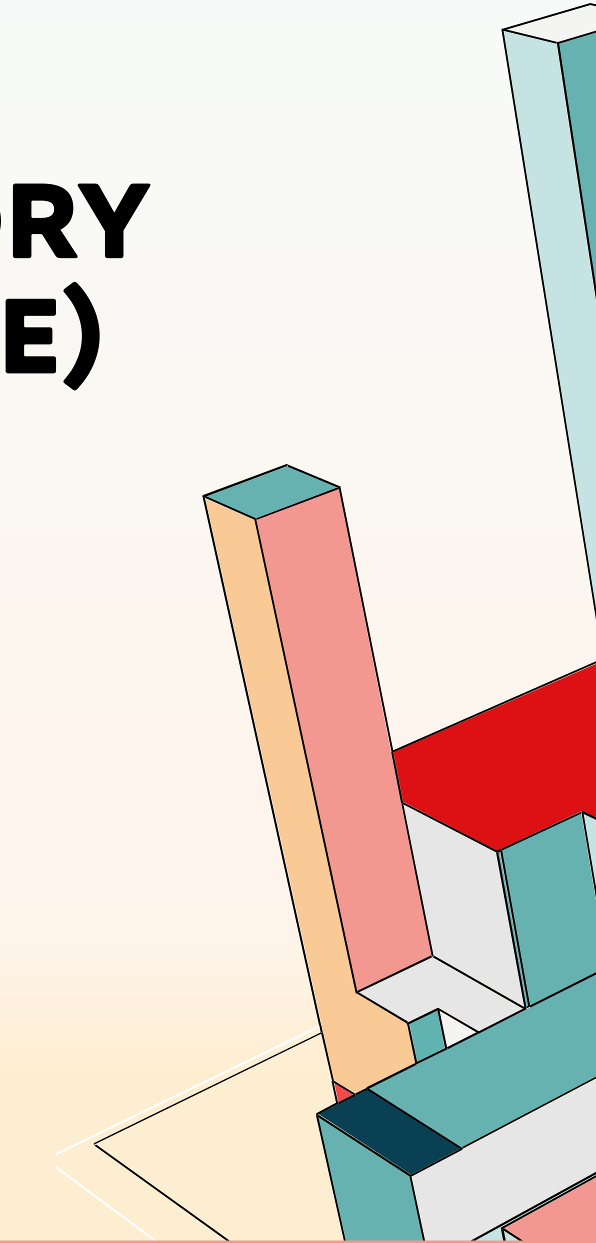
- Examples:
- - Class Scope: Structural patterns (e.g., Adapter, Bridge)
- - Object Scope: Creational and Behavioral patterns (e.g., Factory Method, Strategy)

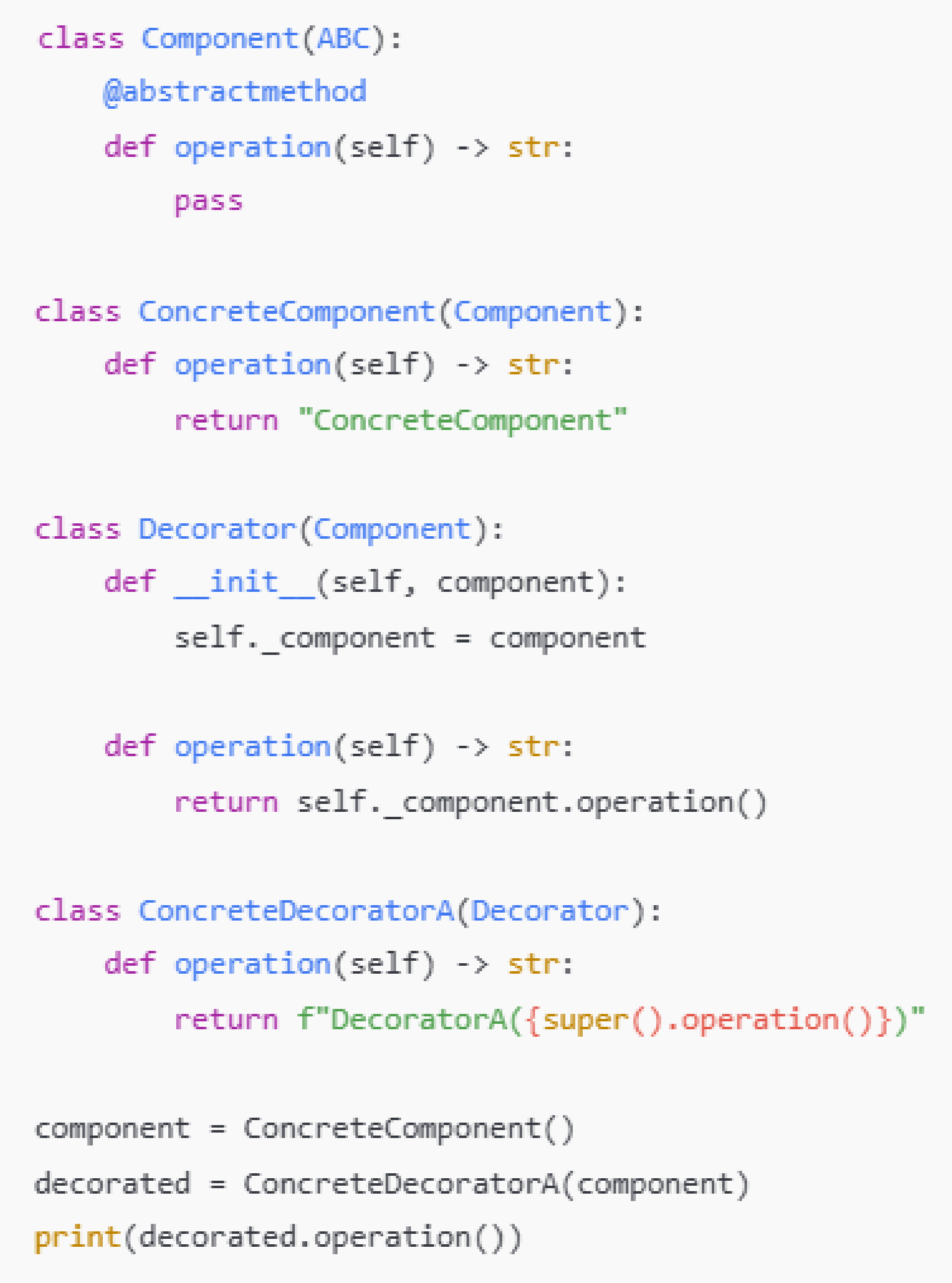


CREATIONAL PATTERN – FACTORY METHOD (EXAMPLE)



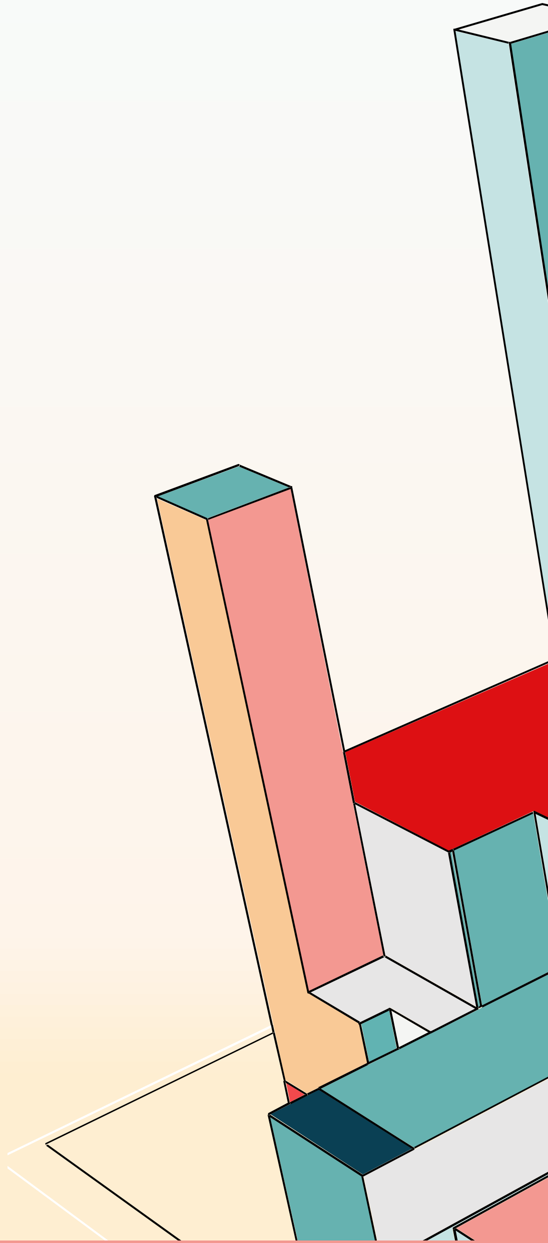
```python
class Product(ABC):
    @abstractmethod
    def operation(self) -> str:
        pass

class ConcreteProductA(Product):
    def operation(self) -> str:
        return "Product A"

class ConcreteProductB(Product):
    def operation(self) -> str:
        return "Product B"

class Creator(ABC):
    @abstractmethod
    def factory_method(self) -> Product:
        pass

class ConcreteCreatorA(Creator):
    def factory_method(self) -> Product:
        return ConcreteProductA()

class ConcreteCreatorB(Creator):
    def factory_method(self) -> Product:
        return ConcreteProductB()

creator = ConcreteCreatorA()
print(creator.factory_method().operation())
```



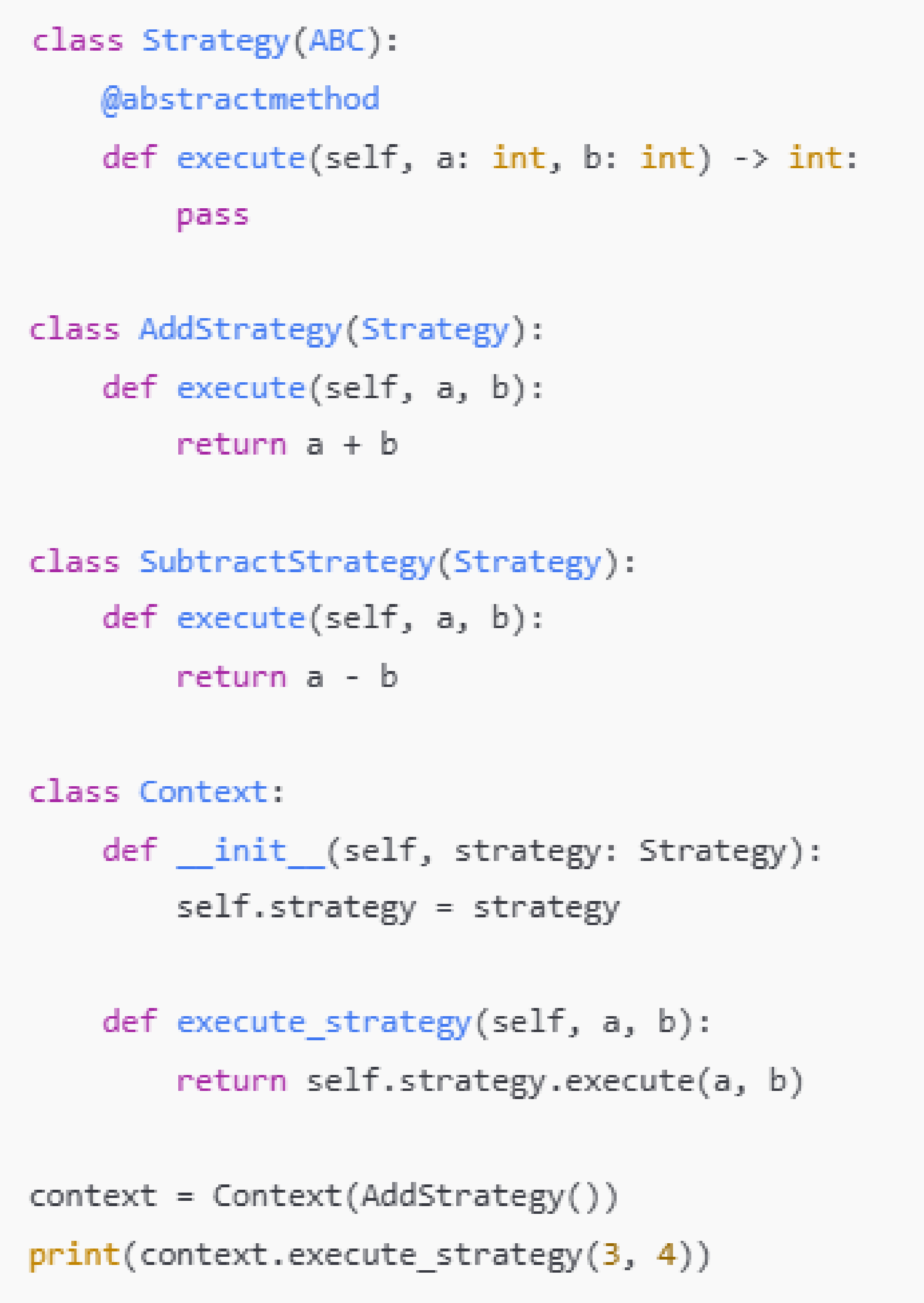


```python
class Component(ABC):
    @abstractmethod
    def operation(self) -> str:
        pass

class ConcreteComponent(Component):
    def operation(self) -> str:
        return "ConcreteComponent"

class Decorator(Component):
    def __init__(self, component):
        self._component = component

    def operation(self) -> str:
        return self._component.operation()

class ConcreteDecoratorA(Decorator):
    def operation(self) -> str:
        return f"DecoratorA({super().operation()})"

component = ConcreteComponent()
decorated = ConcreteDecoratorA(component)
print(decorated.operation())
```

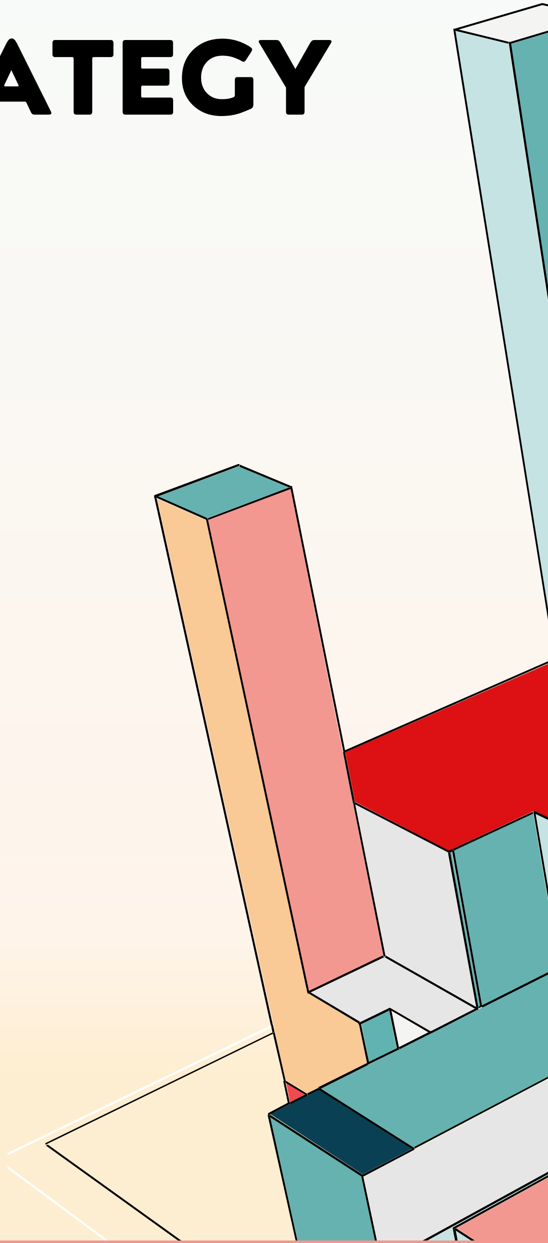
STRUCTURAL PATTERN – DECORATOR (EXAMPLE)



BEHAVIORAL PATTERN – STRATEGY (EXAMPLE)

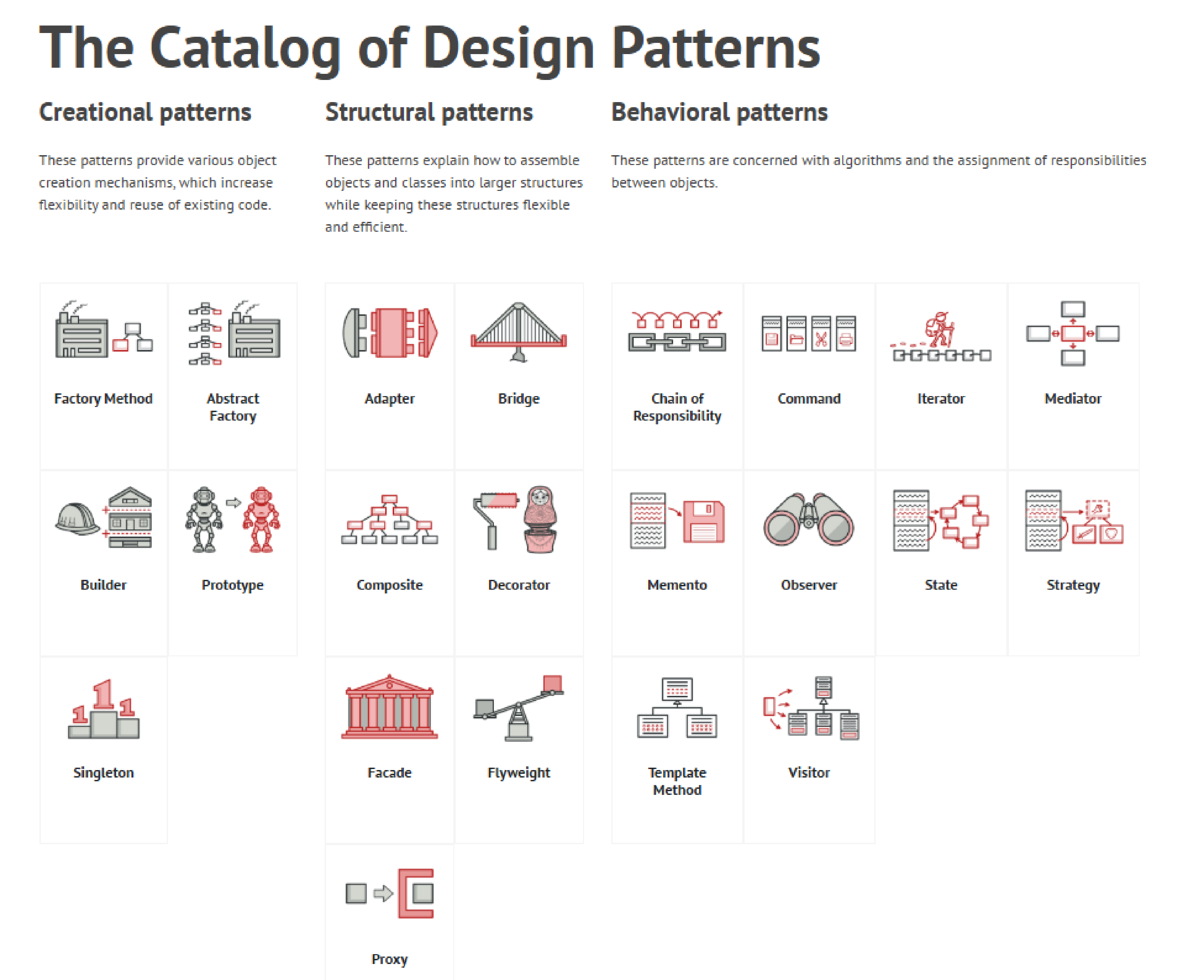


```python
class Strategy(ABC):
    @abstractmethod
    def execute(self, a: int, b: int) -> int:
        pass

class AddStrategy(Strategy):
    def execute(self, a, b):
        return a + b

class SubtractStrategy(Strategy):
    def execute(self, a, b):
        return a - b

class Context:
    def __init__(self, strategy: Strategy):
        self.strategy = strategy

    def execute_strategy(self, a, b):
        return self.strategy.execute(a, b)

context = Context(AddStrategy())
print(context.execute_strategy(3, 4))
```



KEY INSIGHTS FROM CODE EXAMPLES

- - Creational: Factory Method delegates object creation.
- - Structural: Decorator dynamically extends functionality.
- - Behavioral: Strategy allows runtime behavior changes.



The Catalog of Design Patterns

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural patterns

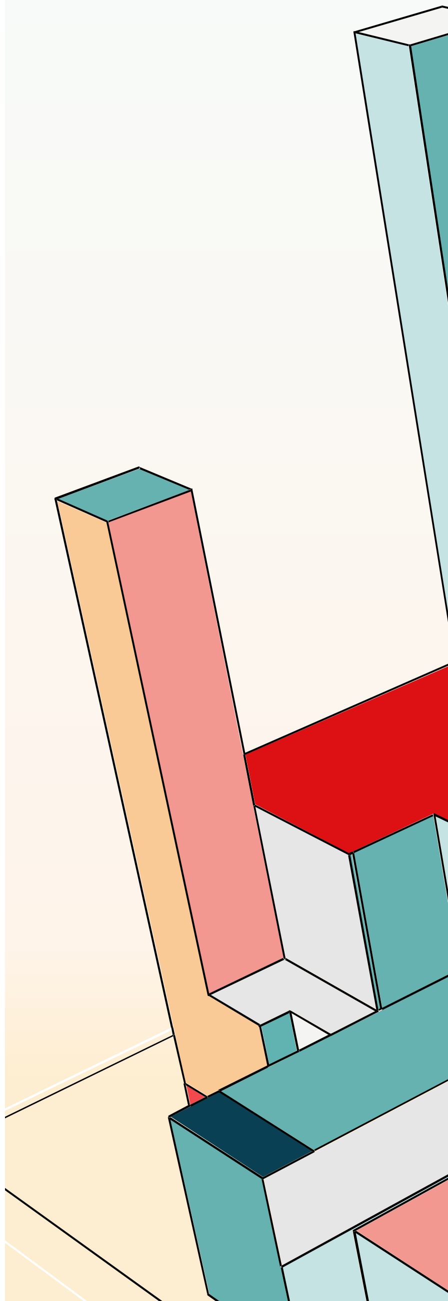
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

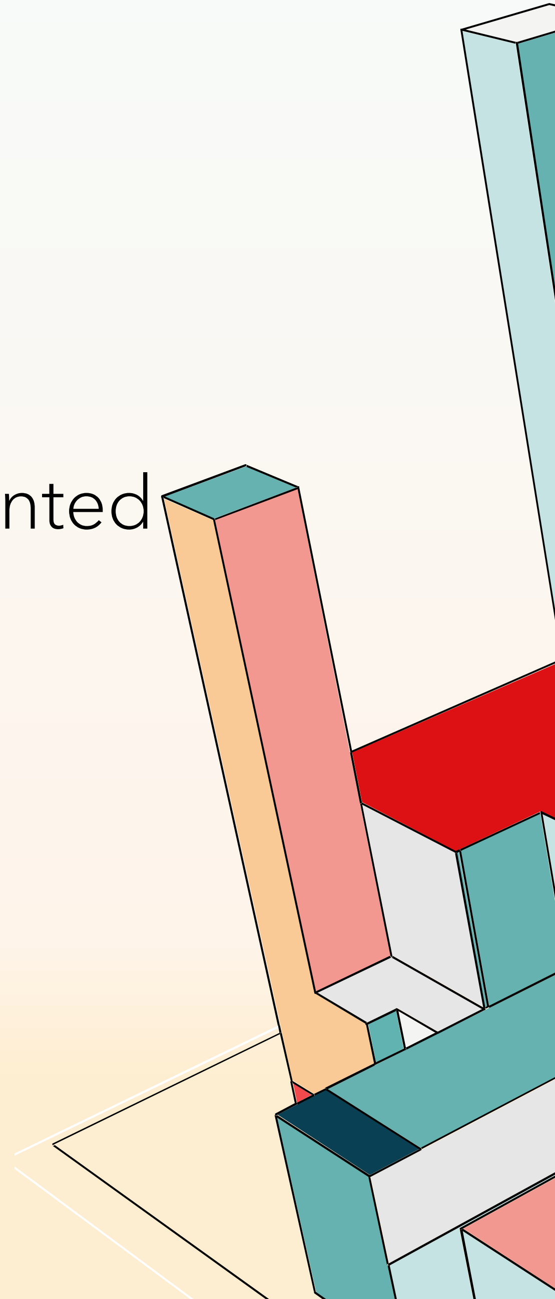


CONCLUSION

- Classification helps in pattern selection: Creation, Structure, or Behavior.
- Each pattern type solves different design challenges.
- Understanding the classification is crucial before learning individual patterns.

REFERENCES

- - Refactoring.Guru - Design Patterns Classification
- - Refactoring.Guru - Design Patterns Catalog
- - Design Patterns: Elements of Reusable Object-Oriented Software (Gang of Four)



THANK YOU