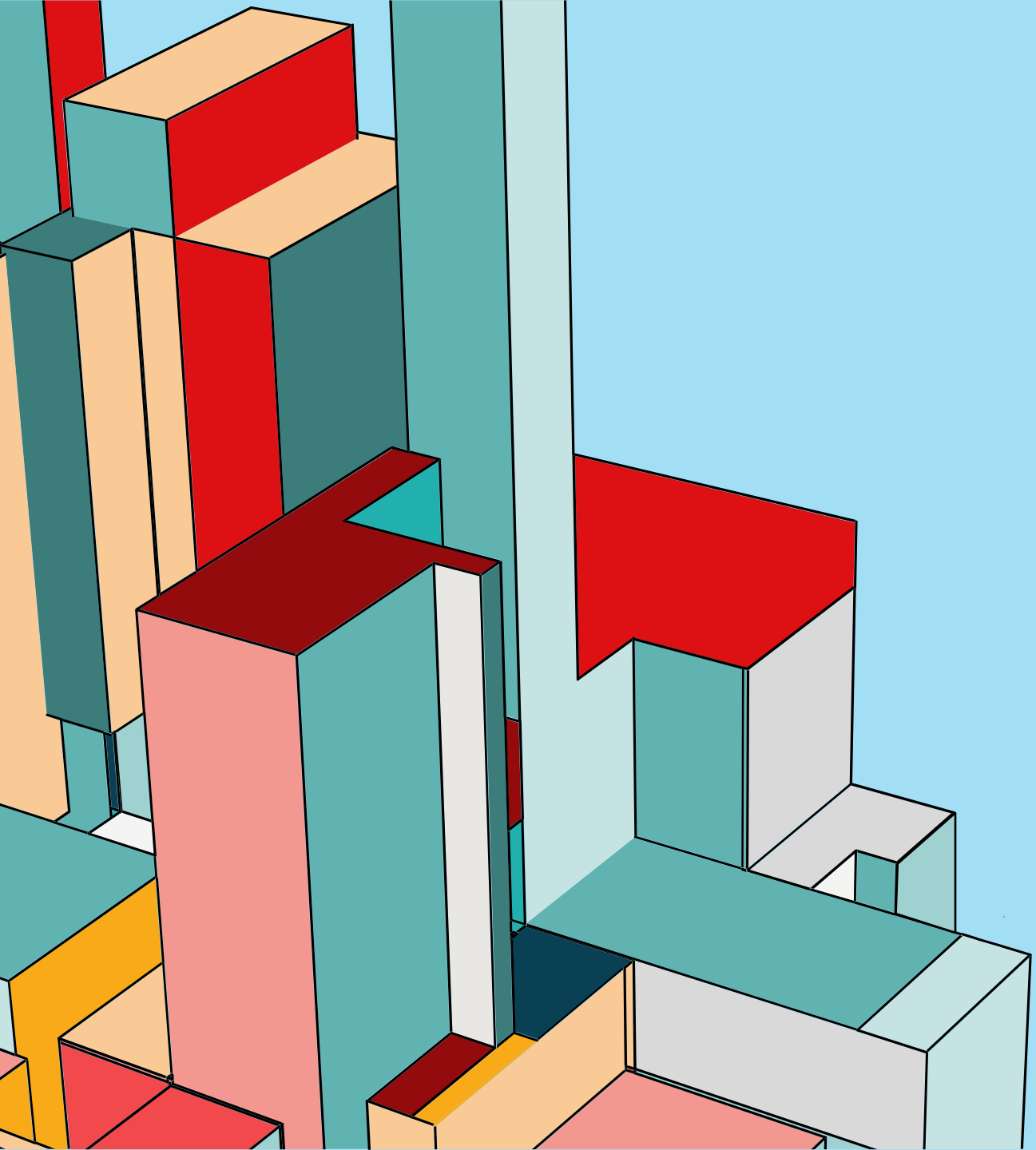
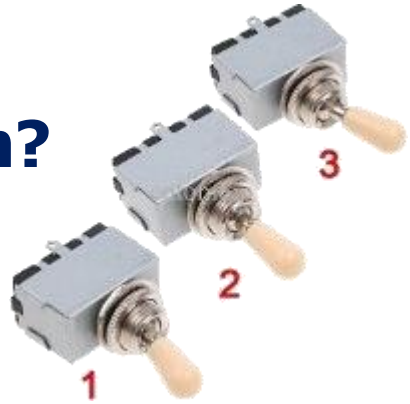


OBJECT- ORIENTED DESIGN PRINCIPLES

HOW MANY SCENARIOS (HOW MANY WAYS CAN THE MACHINE GO WRONG)?

2 switches, 2 settings each?

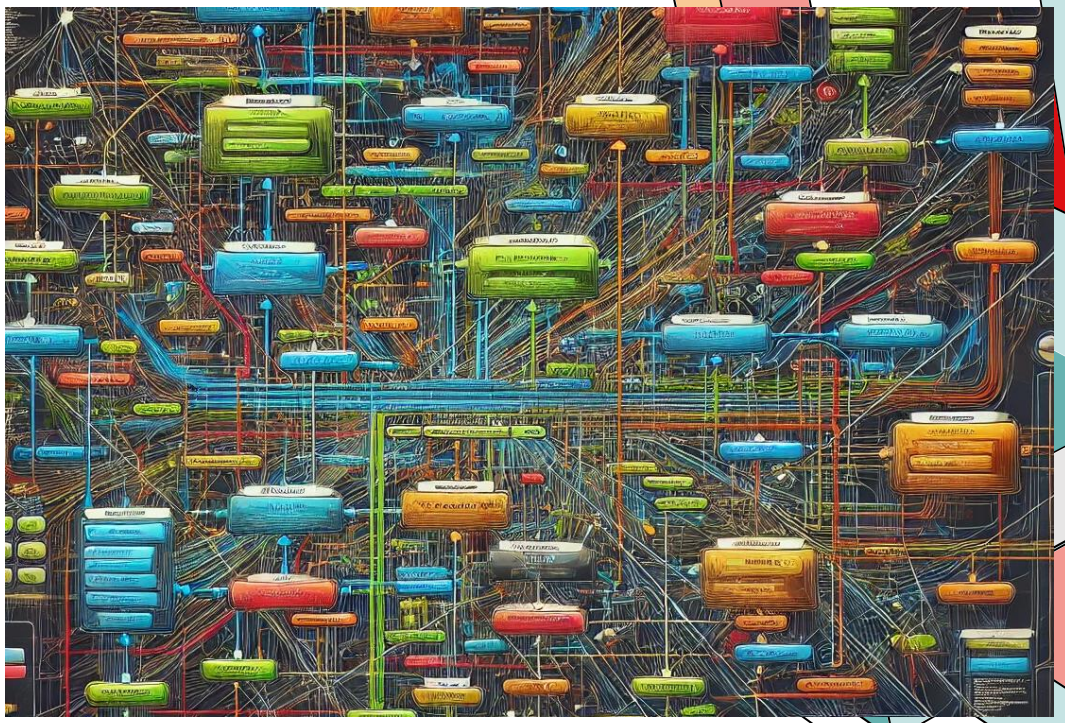
3 switches, 3 settings each?



5 switches, 5 settings each?

"Combinatorial Explosion"

Complexity grows exponentially with each additional element



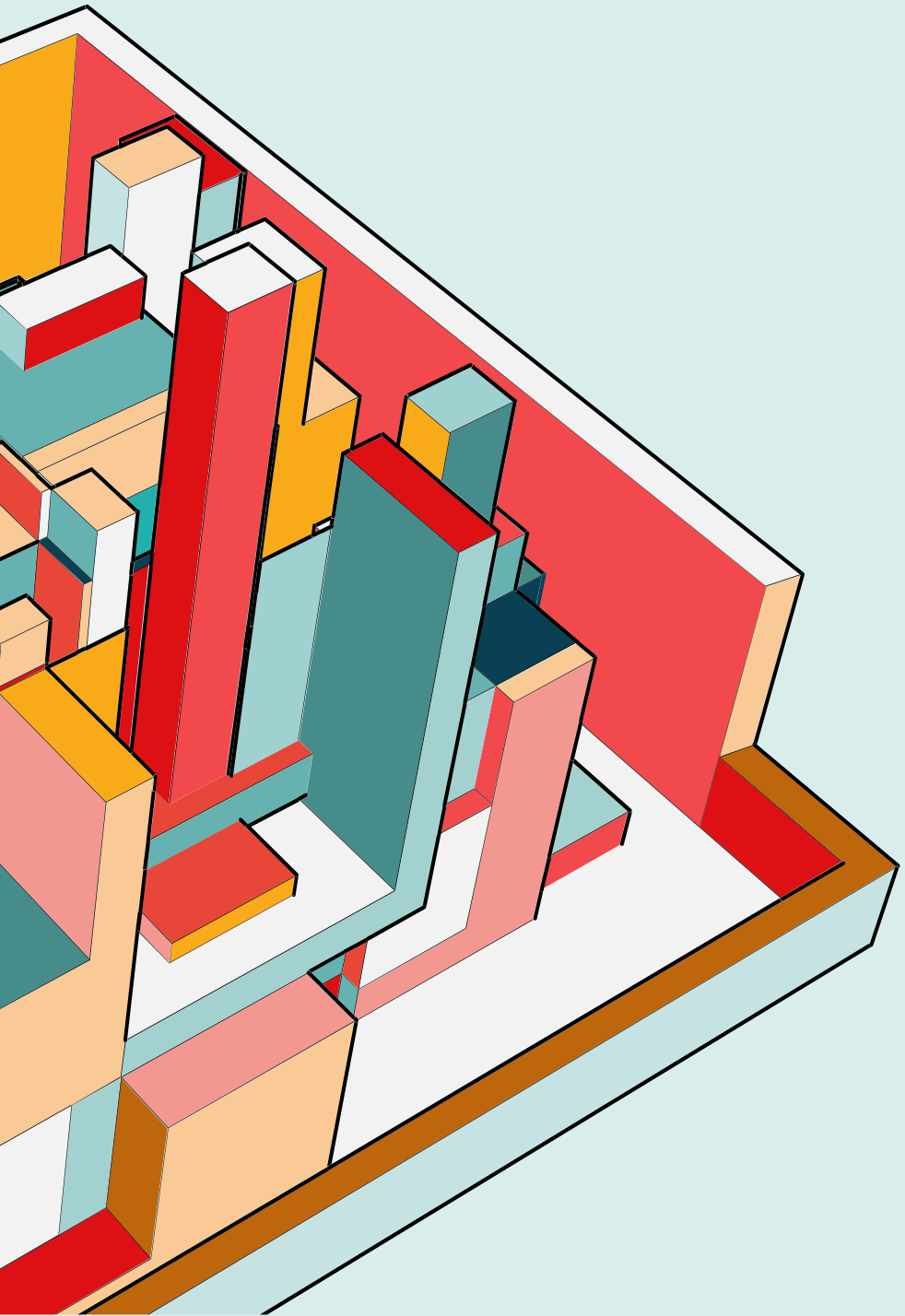
DESIGN WITH INDEPENDENT COMPONENTS

Avoid Combinatorial Explosion

Objects are mini-programs, each with its own data and operations.

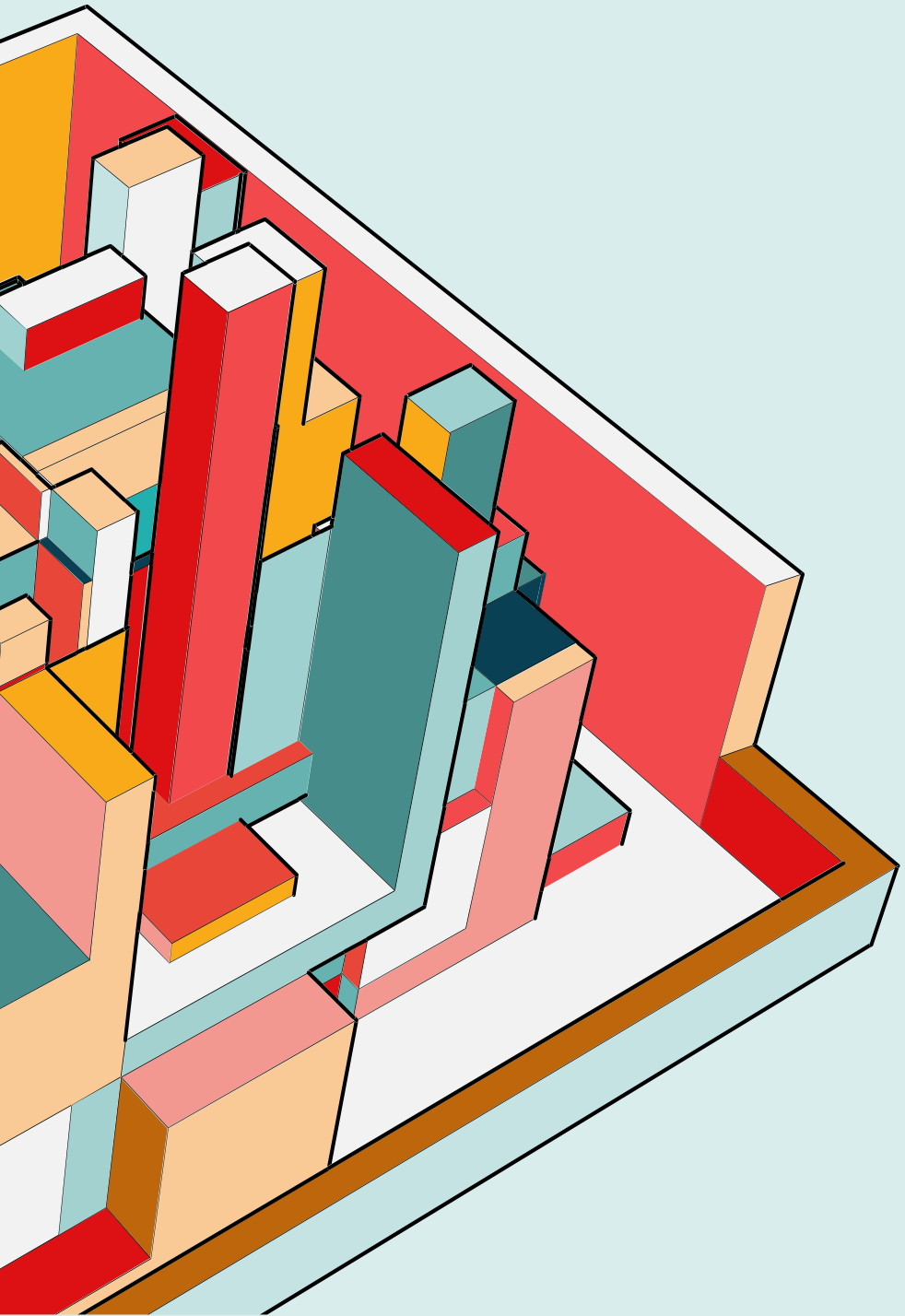
In Unix, we have a lot of little programs (grep, ls, cat...) that do just one thing. We can combine the little programs into bigger programs by having the output of one become the input of another.

OOP is the same. We make small, simple, specialized programs, and then make bigger programs by combining little programs ("Composition").



OBJECT-ORIENTED PROGRAMMING

is a programming paradigm where software is structured around "objects" that combine data (attributes) and behavior (methods). This approach mirrors real-world entities and promotes modular, reusable, and maintainable code.



OBJECT-ORIENTED PROGRAMMING

Key Benefits:

- Modularity
- Reusability
- Scalability
- Easier maintenance
- Enhanced security

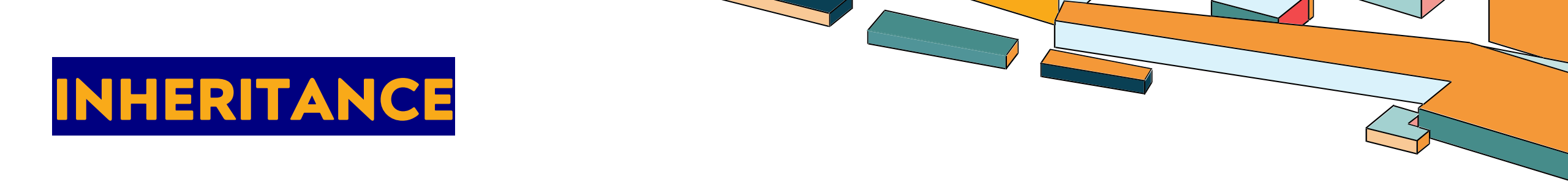
ENCAPSULATION

Encapsulation bundles data and methods into a single unit, restricting direct access to certain components and providing controlled interaction via methods.

Protects object integrity by hiding internal details.Provides a public interface for interaction (like "need-to-know" access).

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

INHERITANCE

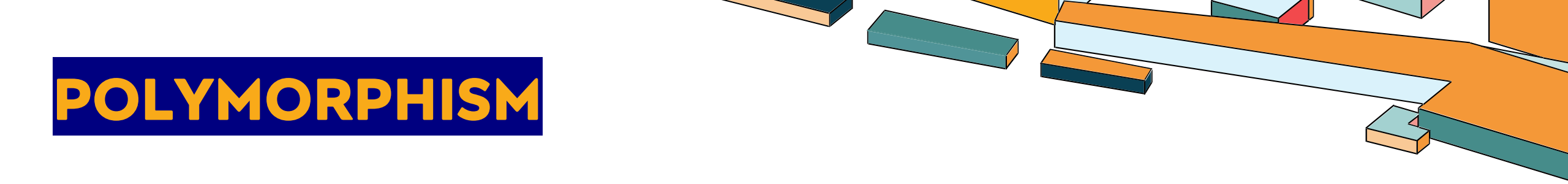


Inheritance allows a class (child) to acquire the properties and behavior of another class (parent).

It promotes code reuse and hierarchy creation.Enables extending or overriding parent class functionality.

```
class Animal:
    def eat(self):
        print("Eating")

class Dog(Animal):
    def bark(self):
        print("Barking")

dog = Dog()
dog.eat()  # Inherited from Animal
dog.bark()  # Defined in Dog
```

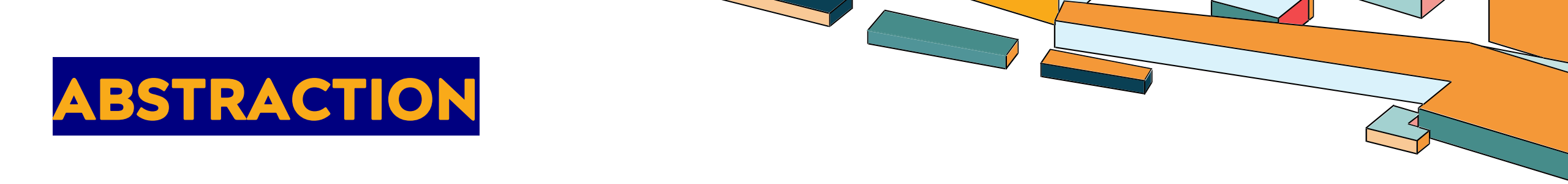
POLYMORPHISM



Polymorphism allows methods to be implemented differently depending on the object type.

It supports method overriding in derived classes.Provides a unified interface for diverse object types.

```
class Animal:
    def sound(self):
        pass

class Cat(Animal):
    def sound(self):
        return "Meow"

class Dog(Animal):
    def sound(self):
        return "Bark"

animals = [Cat(), Dog()]
for animal in animals:
    print(animal.sound())
```



ABSTRACTION

Abstraction hides complex implementation details while exposing only essential functionalities.

It focuses on "what" an object does, not "how."Achieved using abstract classes or interfaces.

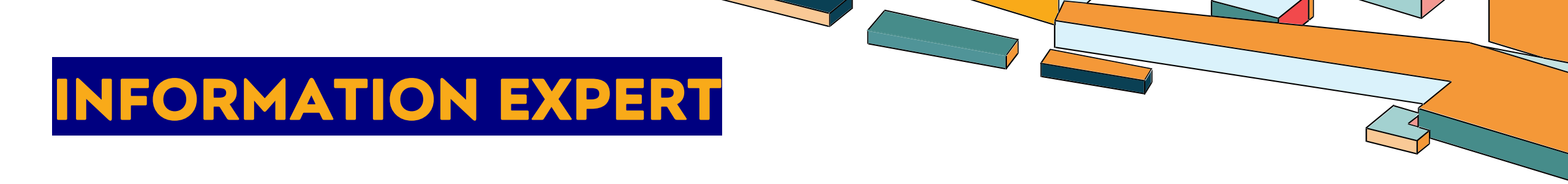
```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

class Car(Vehicle):
    def start(self):
        print("Car started")

car = Car()
car.start()  # Output: Car started
```

COMPOSITION AND DELEGATION

Composition involves building objects by combining simpler ones, emphasizing a "has-a" relationship. Delegation assigns tasks to helper objects rather than performing them directly.

It promotes modularity and flexibility.Avoids the complexities of deep inheritance hierarchies.

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def drive(self):
        self.engine.start()
        print("Car is moving")

car = Car()
car.drive()
```



INFORMATION EXPERT

The Information Expert principle assigns responsibility to the class that has the information needed to fulfill it.

It promotes logical distribution of responsibilities.Reduces unnecessary dependencies.

```python
class Invoice:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        return sum(item['price'] for item in self.items)

invoice = Invoice([{'price': 100}, {'price': 200}])
print(invoice.calculate_total())  # Output: 300
```

SEPARATION OF CONCERNS

Separation of Concerns divides a program into distinct sections, each addressing a separate concern.

It enhances modularity and maintainability.Reduces complexity by separating unrelated functionality.

```
class TaxCalculator:
    def calculate_tax(self, amount):
        return amount * 0.15

class InvoicePrinter:
    def print_invoice(self, details):
        print("Invoice:", details)
```

PACKAGE COHESION PRINCIPLES

Package cohesion focuses on organizing related classes and components into a logical grouping.

It ensures clarity and consistency within packages.Facilitates easier maintenance and scalability.

```python
class AnalyticsInterface:
    def track_sale(self, product, amount):
        raise NotImplementedError("This method should be implemented by subclasses")

# analytics/tracker.py
class SalesTracker(AnalyticsInterface):
    def track_sale(self, product, amount):
        print(f"Tracking sale: {product} - ${amount}")

# ecommerce/order.py
from ecommerce.interfaces import AnalyticsInterface

class Order:
    def __init__(self, order_id, products, analytics: AnalyticsInterface):
        self.order_id = order_id
        self.products = products
        self.analytics = analytics

    def finalize_order(self):
        total = sum(product.price for product in self.products)
        for product in self.products:
            self.analytics.track_sale(product.name, product.price)
        return total
```

PACKAGE COUPLING PRINCIPLES

Package coupling minimizes dependencies between packages, fostering independent development and maintenance.

It reduces risk of ripple effects from changes in one package.Encourages modular design.

```
class TaxCalculator:
    def calculate_tax(self, amount):
        return amount * 0.15

class InvoicePrinter:
    def print_invoice(self, details):
        print("Invoice:", details)
```

FAVOR COMPOSITION OVER INHERITANCE

Prefer composition ("has-a" relationship) to inheritance ("is-a" relationship) when designing systems.

It increases flexibility by assembling objects from components.Reduces dependency on rigid inheritance hierarchies.

```python
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def drive(self):
        self.engine.start()
        print("Car is moving")
```

SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle states that a class should have one and only one reason to change.

It ensures that a class has a focused purpose.Simplifies debugging and maintenance.Improves modularity and reduces code duplication.

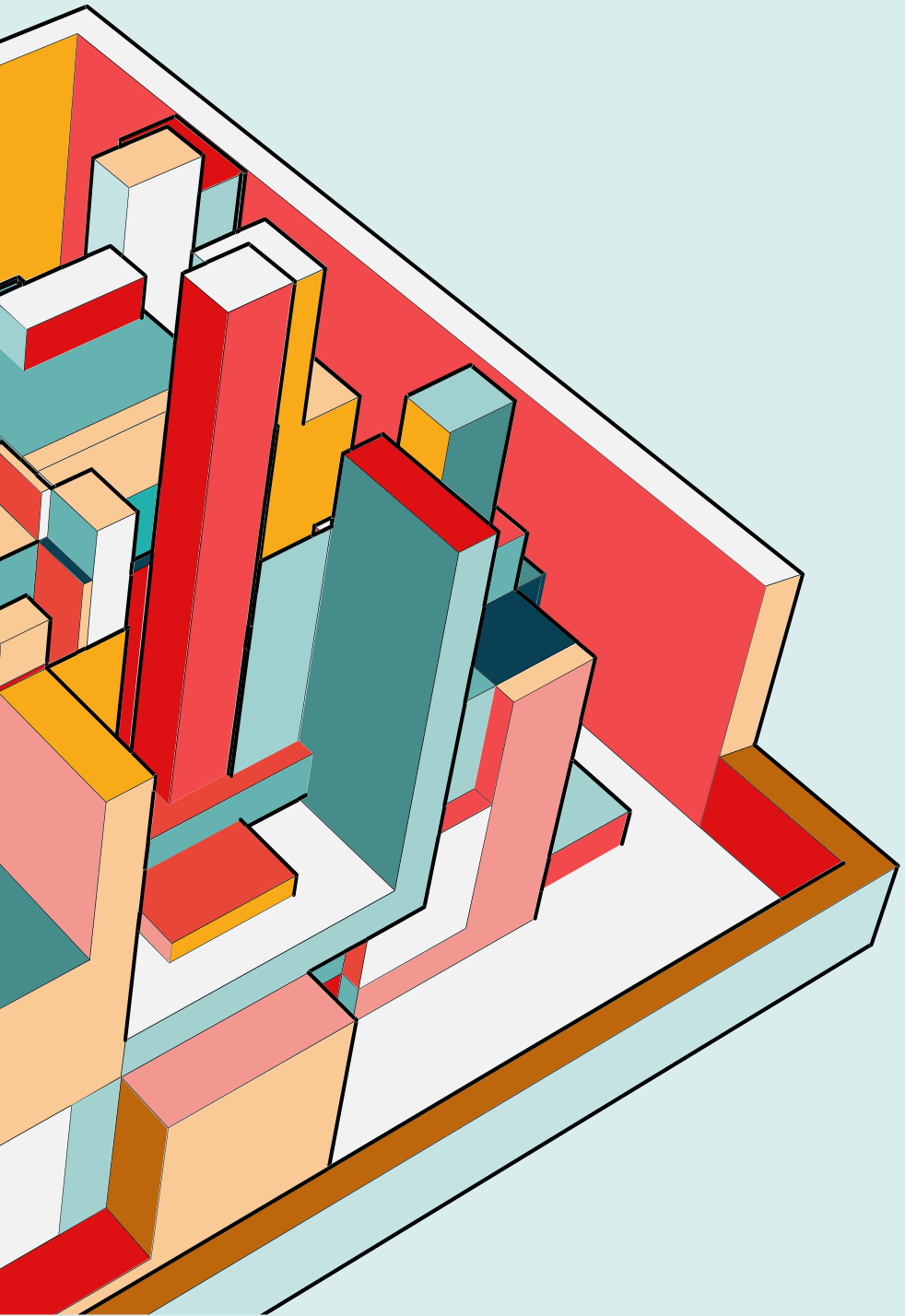
```python
class UserAuthenticator:
    def authenticate(self, username, password):
        # Logic to authenticate user
        return username == "admin" and password == "password"

class UserLogger:
    def log_access(self, username):
        print(f"User {username} logged in successfully.")
```

OPEN/CLOSED PRINCIPLE

The Open/Closed Principle states that software entities (classes, modules, functions) should be open for extension but closed for modification.

It promotes scalability by allowing behavior to be extended without altering existing code.Reduces the risk of introducing bugs into tested code.

```python
# Base class (closed for modification)
class DiscountStrategy:
    def apply_discount(self, total):
        return total

# Extended classes (open for extension)
class PercentageDiscount(DiscountStrategy):
    def __init__(self, percent):
        self.percent = percent

    def apply_discount(self, total):
        return total - (total * self.percent / 100)

class FlatDiscount(DiscountStrategy):
    def __init__(self, amount):
        self.amount = amount

    def apply_discount(self, total):
        return total - self.amount

# Usage
def calculate_total_with_discount(strategy: DiscountStrategy, total):
    return strategy.apply_discount(total)

# Example usage
total = 100
discount_strategy = PercentageDiscount(10)  # 10% discount
print(calculate_total_with_discount(discount_strategy, total))  # Output: 90
```



LISKOV SUBSTITUTION PRINCIPLE (LSP)

LISKOV SUBSTITUTION PRINCIPLE (LSP)

Derived classes must be substitutable for their base classes without altering program correctness.

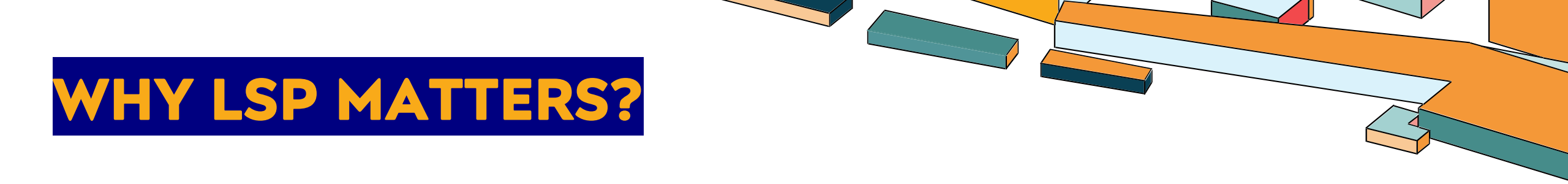
It ensures derived classes maintain the behavior of their base classes.Promotes robust, extensible designs.

```python
class Bird:
    def fly(self):
        print("Flying")

class Sparrow(Bird):
    pass

sparrow = Sparrow()
sparrow.fly()  # Output: Flying
```

LISKOV SUBSTITUTION PRINCIPLE (LSP)

```python
class Bird:
    def fly(self):
        print("Flying")

class Sparrow(Bird):
    pass

sparrow = Sparrow()
sparrow.fly()  # Output: Flying
```

```python
class Penguin(Bird):
    def fly(self):
        raise Exception("Penguins can't fly")

penguin = Penguin()
penguin.fly()  # Raises Exception: Penguins can't fly
```



WHY LSP MATTERS?

Adherence to LSP: Promotes code reuse and scalability, as developers can confidently extend functionality without altering existing code.

Violation of LSP: Results in brittle systems where adding new derived classes or modifying behavior can introduce unexpected side effects.

THANK YOU