# Strategy Design Pattern: The Duck Analogy

***Core Concepts: Encapsulation & Abstract Polymorphism***

## 1. The Problem

In standard inheritance, if we define **fly()** in a base Duck class, every subclass inherits it. This creates a behavioral nightmare where RubberDucks and WoodenDecoys start flying in your code.

## 2. The Solution: Encapsulation

We apply the design principle: *Identify the aspects of your application that vary and separate them from what stays the same.*

**What stays the same:** A duck has feathers, it swims, and it looks like a duck.
**What varies:** How it flies and how it quacks.

We encapsulate these varying behaviors by moving them out of the Duck class and into their own behavior classes.

## 3. Python Implementation (The Code)

```python
from abc import ABC, abstractmethod

# --- ENCAPSULATED BEHAVIORS ---

class FlyBehavior(ABC):
    @abstractmethod
    def fly(self):
        pass

class FlyWithWings(FlyBehavior):
    def fly(self):
        print("Flying with real wings! ■")

class FlyNoWay(FlyBehavior):
    def fly(self):
        print("I can't fly. I just sit there.")

class QuackBehavior(ABC):
    @abstractmethod
    def quack(self):
        pass

class Squeak(QuackBehavior):
    def quack(self):
        print("Squeak! (Rubber duck style)")

# --- ABSTRACT POLYMORPHISM ---

class Duck(ABC):
    def __init__(self, fly_behavior: FlyBehavior, quack_behavior: QuackBehavior):
        self.fly_behavior = fly_behavior
        self.quack_behavior = quack_behavior

    def perform_fly(self):
        self.fly_behavior.fly()

    def perform_quack(self):
        self.quack_behavior.quack()

# --- CONCRETE SUBCLASSES ---

class MallardDuck(Duck):
    def __init__(self):
        super().__init__(FlyWithWings(), Squeak())

class RubberDuck(Duck):
    def __init__(self):
        super().__init__(FlyNoWay(), Squeak())

# --- RUNTIME EXECUTION ---
if __name__ == "__main__":
    my_duck = RubberDuck()
    print("Rubber Duck:")
    my_duck.perform_fly()

    print("\nGiving the rubber duck a jetpack...")
    my_duck.fly_behavior = FlyWithWings()
    my_duck.perform_fly()
```

## 4. Why This Works

**Abstract Polymorphism:** The Duck class relies on interfaces (FlyBehavior, QuackBehavior) rather than concrete implementations. This makes the system flexible and extensible.

**Composition over Inheritance:** Instead of being a flier, the Duck has a flying behavior. This allows behaviors to be swapped at runtime without modifying the Duck class.