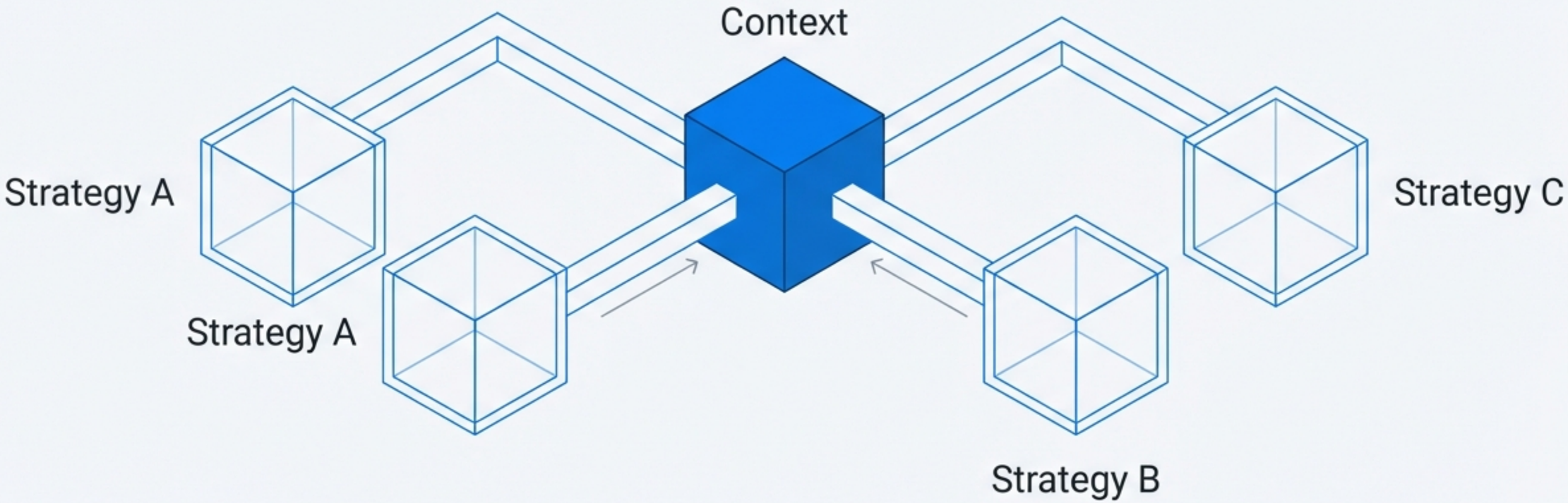
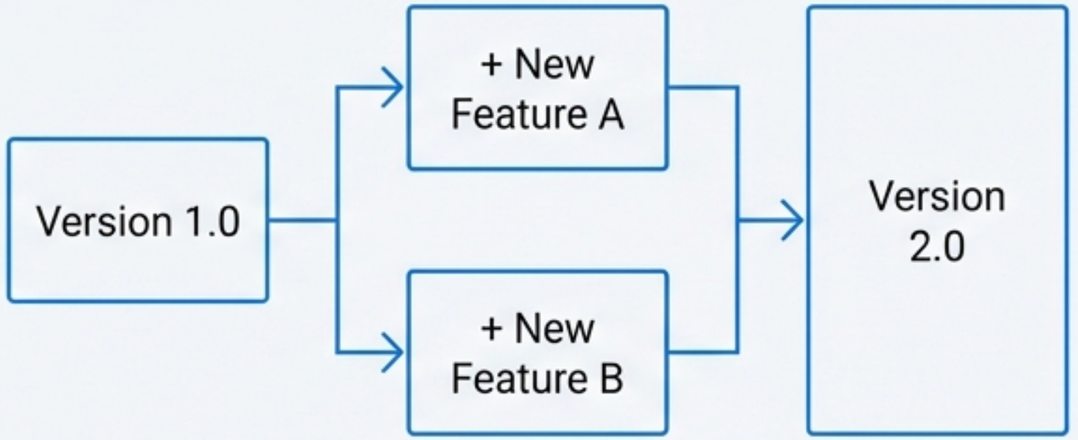
The Strategy Pattern

Mastering Dynamic Behaviour in Software Design



A Good Design Welcomes the Changes of Tomorrow

> **Change is the only constant.**
> Applications constantly evolve with new features and changing requirements. A flexible design is one where adding new functionality requires minimal changes to existing, stable code.

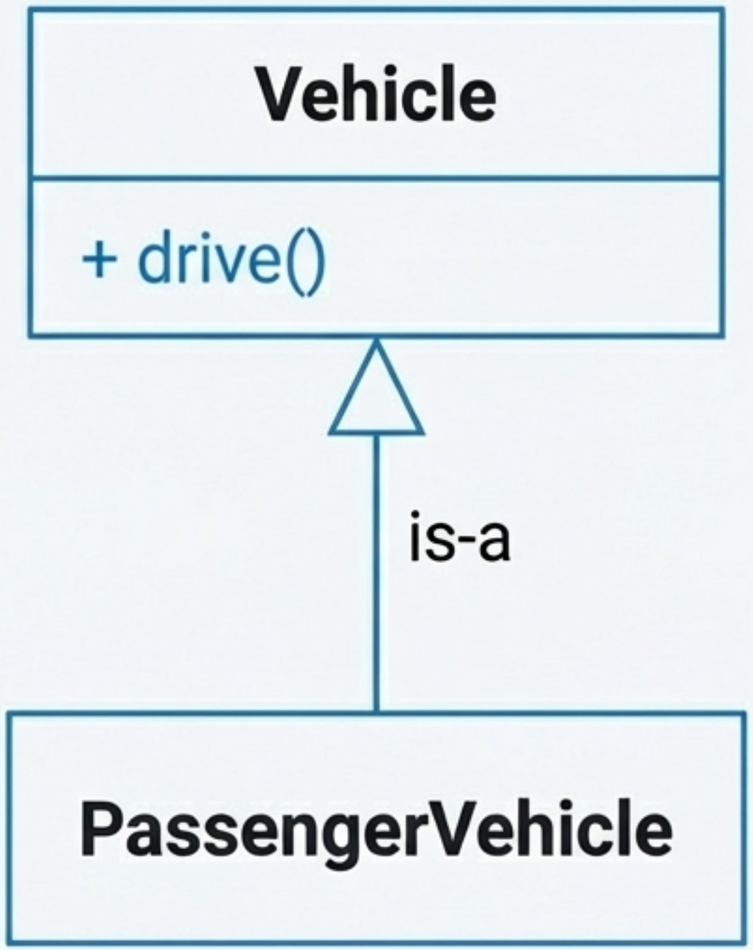


* Software is never static; new features and requirements are inevitable.
* A rigid design is brittle. It breaks easily, leading to bugs, duplicated code, and slower development cycles.
* Our goal is a flexible, evolving design that minimises friction when new features are integrated.

The Intuitive Start: Modelling Behaviour with Inheritance

Let's model a system of vehicles. An `is-a` relationship, or inheritance, seems like a natural fit.

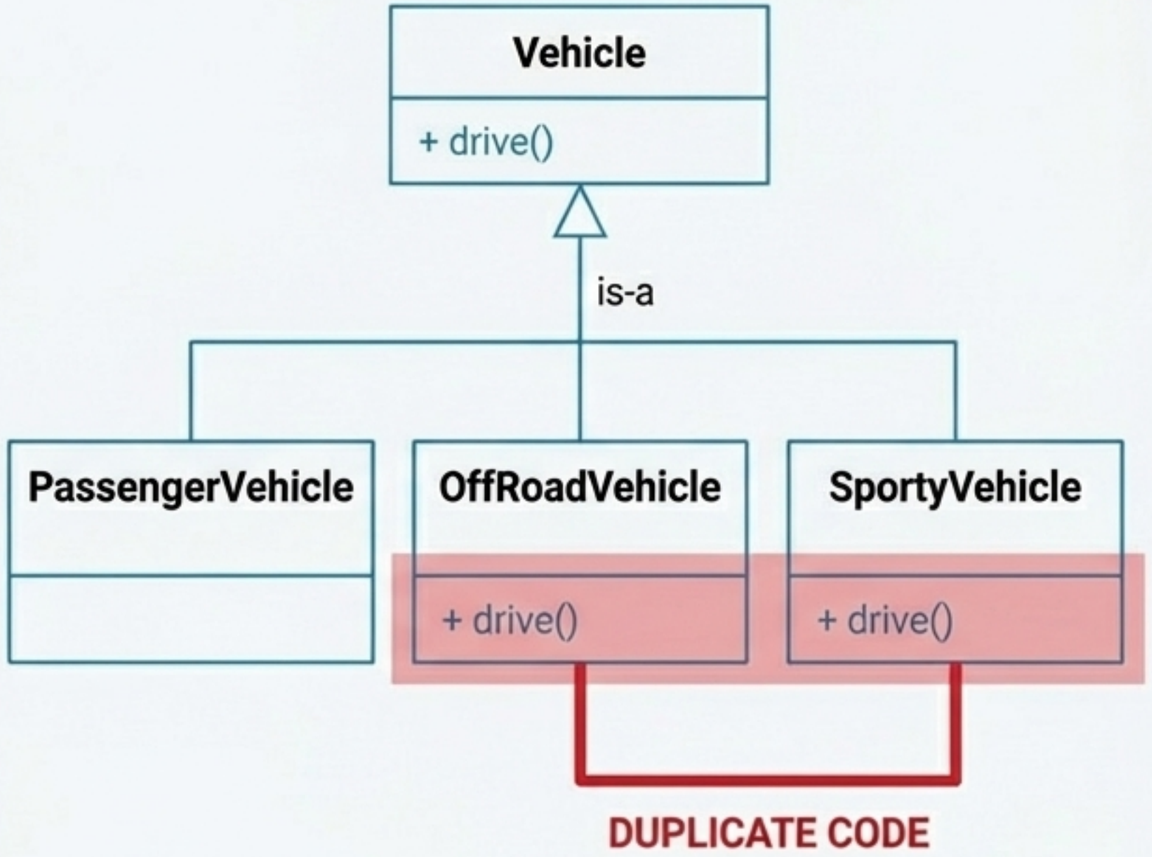
- We create a base `Vehicle` class.
- This base class contains a common `drive()` method with a default implementation, let's call it **"Normal Drive Capability"**.
- A subclass like `PassengerVehicle` can **inherit this method directly**, as its behaviour matches the parent's.



The First Crack Appears: When Behaviour Diverges

What happens when new types of vehicles have unique driving needs?

- Consider an `OffRoadVehicle` and a `SportyVehicle`. Both require a 'Special Drive Capability', not the normal one from the base class.
- **The Obvious Fix:** Each subclass overrides the `drive()` method with the special capability logic.
- **The Consequence:** This creates **code duplication**. The exact same logic for 'Special Drive Capability' is now copied and pasted into two separate child classes. We are not reusing code; we are duplicating it.

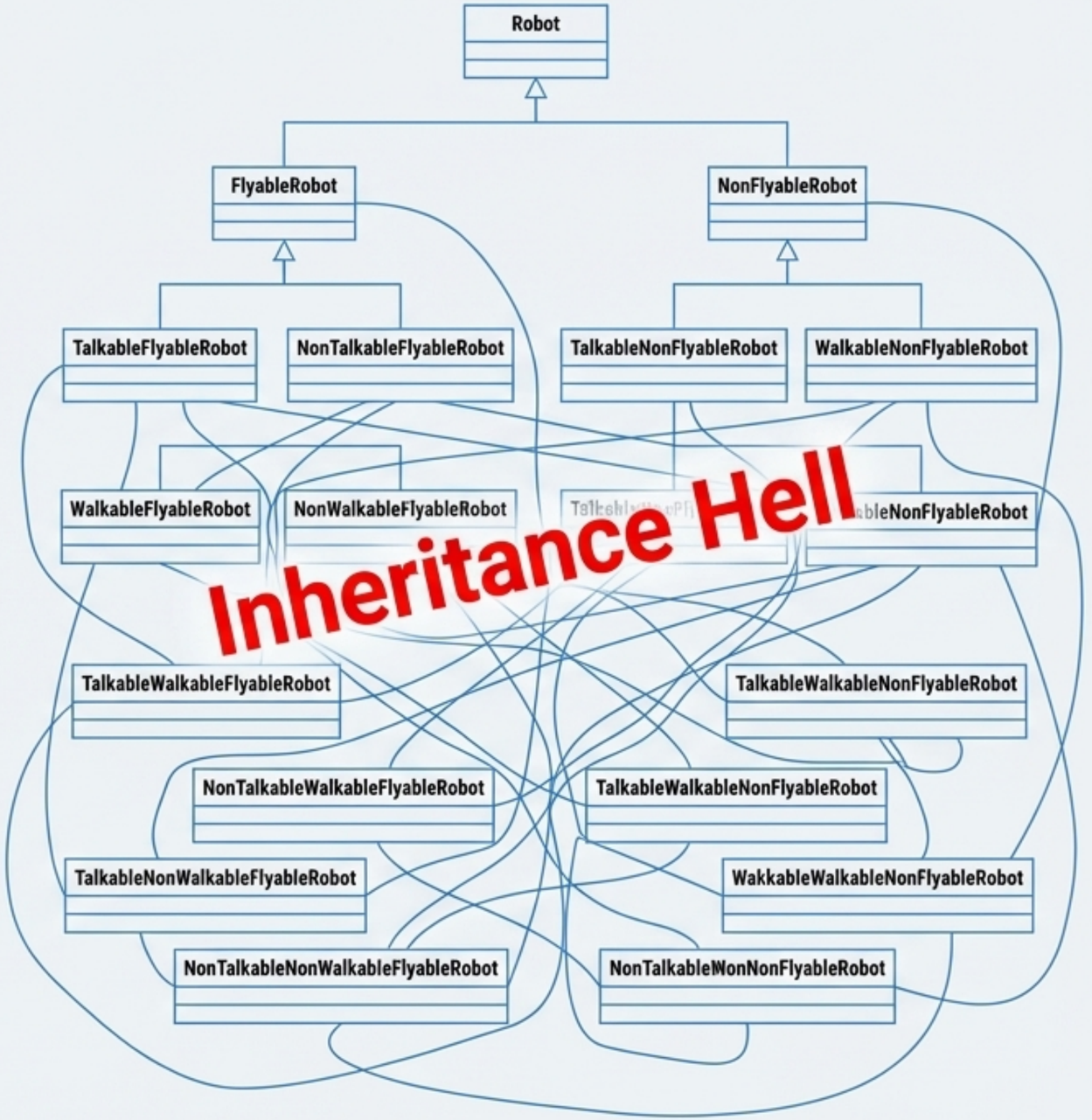


The Slippery Slope of an Exploding Hierarchy

With every new requirement, the problem compounds. How do we handle:

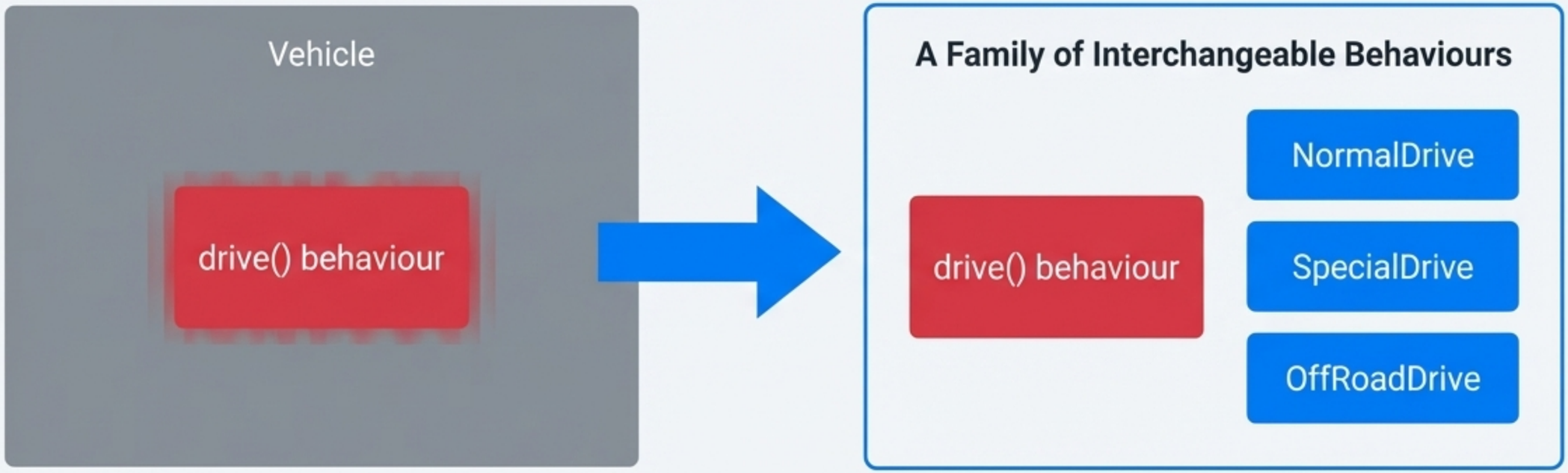
- Robots that can fly (`Flyable`) vs. those that can't (`NonFlyable`)?
- Robots that can talk (`Talkable`) vs. those that can't (`NonTalkable`)?
- Robots that can walk (`Walkable`) vs. those that can't (`NonWalkable`)?

To handle all permutations, the inheritance tree becomes a tangled, unmanageable web. This design is brittle, hard to maintain, and violates a core tenet of good design: the **Open-Closed Principle**.



The Guiding Principle: Encapsulate What Varies

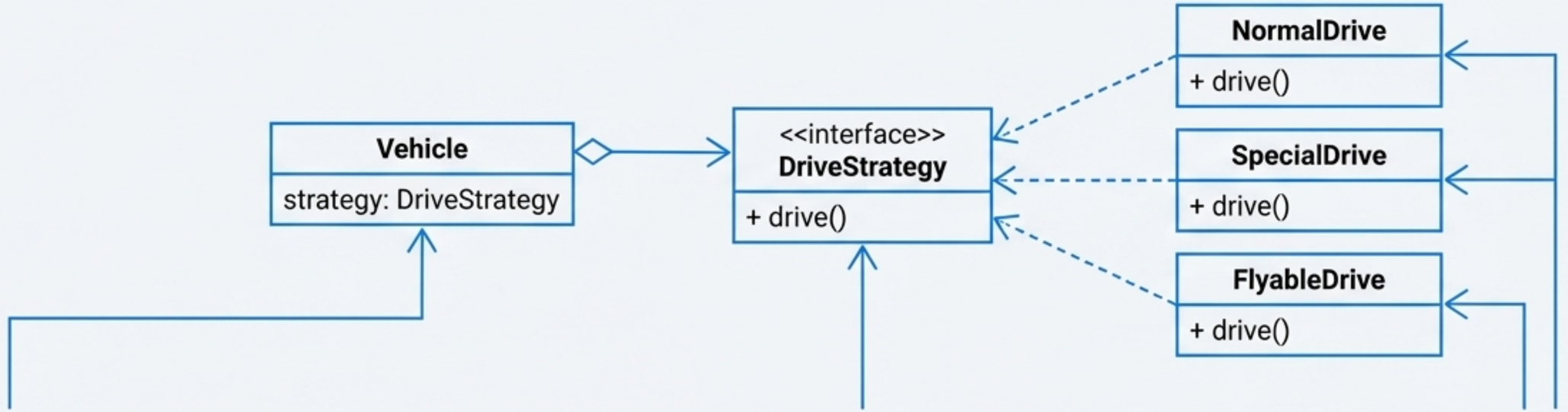
Identify the aspects of your application that vary and separate them from what stays the same.



- The root of our problem was not the `Vehicle` or `Robot` itself, but its **behaviours** (driving, flying, talking), which change from one subclass to another. These are the parts that vary.
- Instead of letting the base class manage all behavioural variations through inheritance, we will pull these behaviours out into their own distinct families of objects.

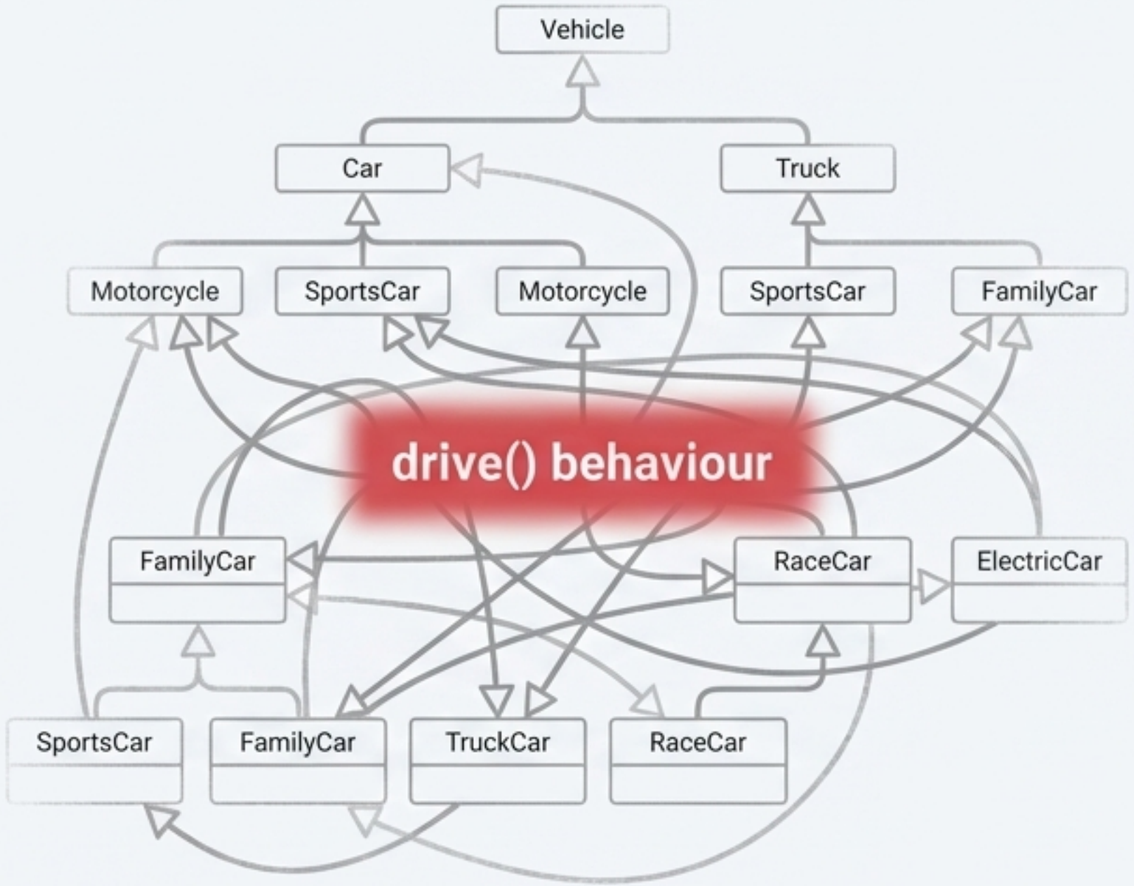
The Solution: The Strategy Pattern

> “The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”

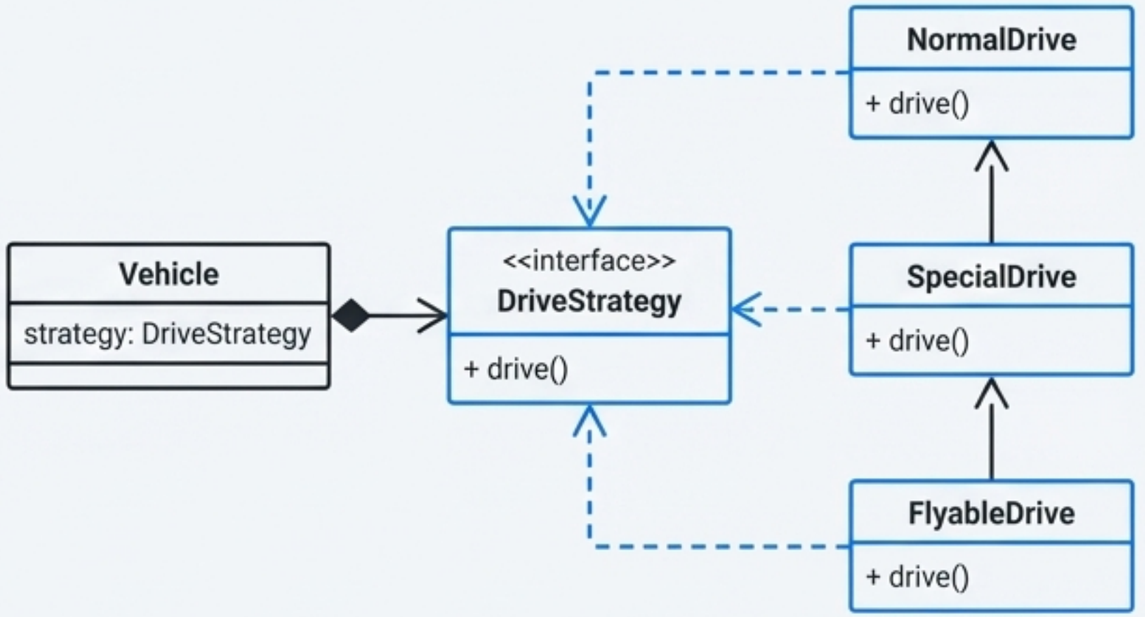


- **Context** (e.g., `Vehicle`): The class that needs a behaviour. It maintains a reference to a Strategy object but is decoupled from the concrete implementation.
- **Strategy** (e.g., `DriveStrategy` **Interface**): A common interface for all algorithms. The Context uses this interface to call the algorithm defined by a **Concrete Strategy.**
- **Concrete Strategies** (e.g., `NormalDrive`, `SpecialDrive`): Classes that implement the actual algorithms, providing different variations of the behaviour.

From a Rigid Hierarchy to a Flexible Composition

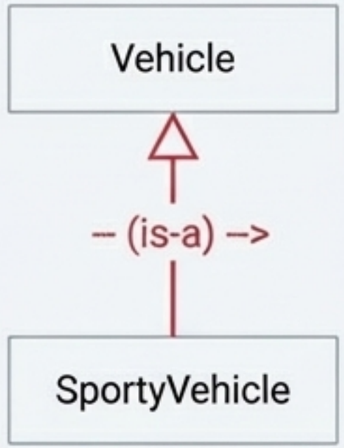


Rigid & Duplicative

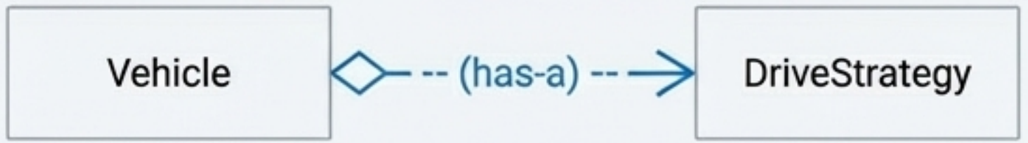


Flexible & Reusable

The Core Shift: Favour Composition Over Inheritance



- We are changing the fundamental relationship. Instead of inheriting behaviour, we delegate it.
- **Inheritance (is-a):** A `SportyVehicle` *_is a_* type of `Vehicle`. Its driving behaviour is locked in at compile time by its position in the class hierarchy.
- **Composition (has-a):** A `Vehicle` *_has a_* `DriveStrategy`. Its behaviour is delegated to a separate, pluggable strategy object. This can even be changed at runtime.
- The `Vehicle` class is now a **Context**. It doesn't perform the driving; it delegates that task to whichever strategy object it currently holds. It becomes a 'dumb' object that orchestrates behaviour.



Dynamic Behaviour Through Constructor Injection

The client now has full control over an object's behaviour at the moment of creation. We pass the desired behaviour (the strategy) into the constructor.

```
// 1. The Vehicle class receives the strategy via its constructor
public class Vehicle {
    DriveStrategy driveStrategy;

    public Vehicle(DriveStrategy driveStrategy) {
        this.driveStrategy = driveStrategy;
    }

    public void drive() {
        driveStrategy.drive();
    }
}
```

Constructor Injection

Delegate the call

```
// 2. The client creates objects with specific strategies
DriveStrategy special = new SpecialDriveStrategy();
Vehicle offRoader = new Vehicle(special); // Inject special driving
offRoader.drive(); // Executes: "Special driving capability."

DriveStrategy normal = new NormalDriveStrategy();
Vehicle goodsVehicle = new Vehicle(normal); // Inject normal driving
goodsVehicle.drive(); // Executes: "Normal driving capability."
```

The Strategy Pattern is Everywhere in Modern Software

This pattern is fundamental for building flexible systems. You will find it in:

Payment Processing

A `PaymentSystem` context can use different strategies.

- `PayByCreditCard`
- `PayByPayPal`
- `PayByUPI`

Navigation Routing

A `NavigationApp` context calculates routes based on a strategy.

- `RouteForCar`
- `RouteForBike`
- `RouteForPublicTransport`

Data Sorting

A data collection can be sorted using various algorithms.

- `QuickSort`
- `MergeSort`
- `InsertionSort`

The Tangible Benefits of a Strategy-Based Design

✅ **Upholds the Open-Closed Principle**: You can introduce new strategies (e.g. a new payment method) **without ever** modifying the client context (the `PaymentSystem`). The system is open for extension but closed for modification.

✅ **Eliminates Large Conditional Blocks**: Replaces fragile `` `if-else` `` or `` `switch` `` statements with clean, polymorphic objects.

✅ **Promotes Code Reusability & Decoupling**: Algorithms are encapsulated in their own classes. They are independent, easier to test in isolation, and can be reused across different contexts.

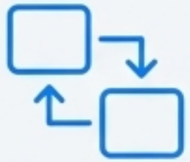
✅ **Enables Dynamic Flexibility**: Algorithms can be selected and swapped at runtime, allowing an application to adapt its behaviour on the fly.

When Should You Use the Strategy Pattern?

This should be your go-to pattern when you encounter these scenarios:

1

Multiple Behavior Implementations
When a class has a behaviour that can be implemented in several different ways, and you want to switch between these variations easily.

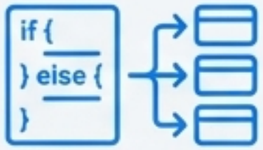


2

Dynamic Algorithm Selection
When you need to let a client choose from a family of algorithms and be able to swap them out dynamically at runtime.

3

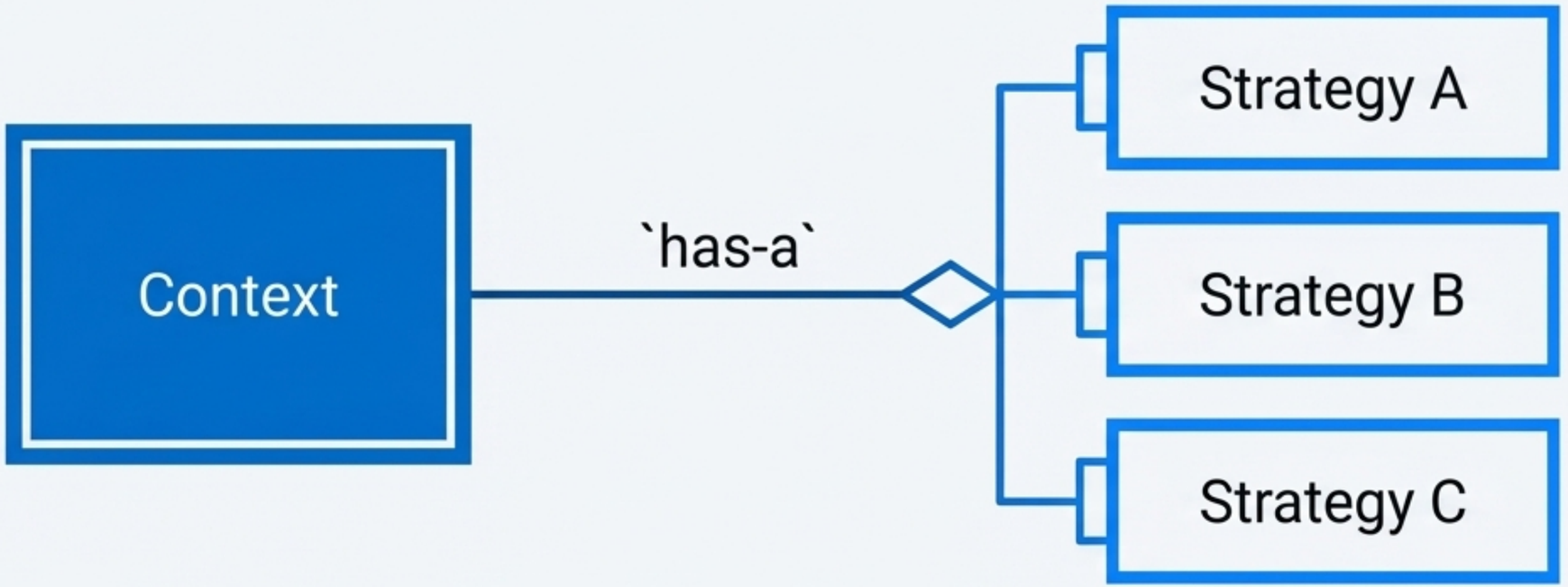
Isolating Business Logic
When you want to isolate the business logic of a class from the implementation details of its algorithms, especially if those algorithms are likely to change.



4

Replacing Large Conditionals
When you have a class with a large conditional statement that switches its behaviour based on some property. This is a strong indicator that the behaviours should be extracted into separate strategies.

Encapsulate What Varies



The Strategy Pattern is more than a technique; it is a design philosophy. To build robust, maintainable, and scalable software, you must identify what changes, pull it out, and let flexible composition replace rigid inheritance.