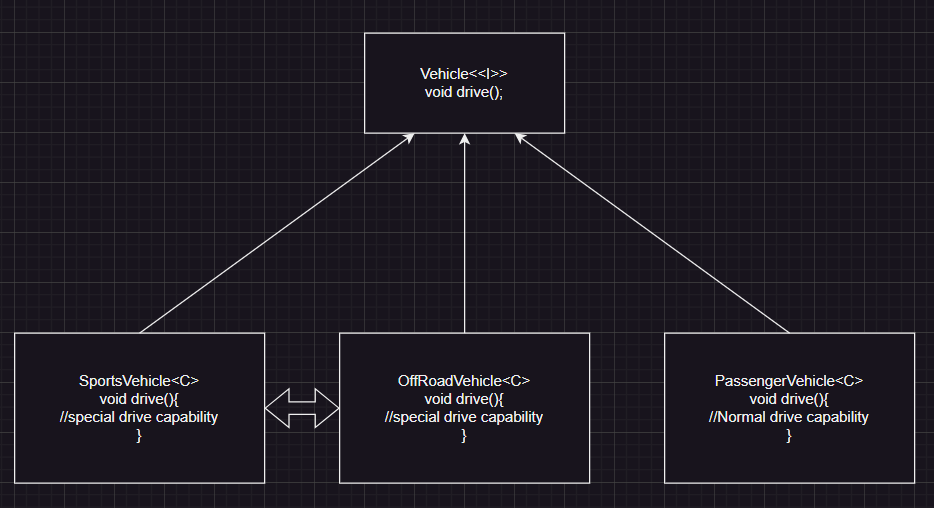
**Strategy Design Pattern:** Strategy pattern is also known as Policy Pattern. We define multiple algorithms and let client application pass the algorithm to be used as a parameter. One of the best example of strategy pattern is Collections.sort() method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.



In above diagram we could see different types of vehicles implementing the drive method. But we could see SportsVehicle & OffRoadVehicle have same implementation & behaviour is same, which results duplicate code & also if we want to change the drive behaviour we have to touch the subclasses in both classes.

To avoid this issue we will going to use strategy design pattern.

**Characteristics of the Strategy Design Pattern?**

The Strategy Design Pattern exhibits several key characteristics that make it distinctive and effective for managing algorithm variations in software systems:

* **It defines a family of algorithms:** The pattern allows you to encapsulate multiple algorithms or behaviors into separate classes, known as strategies.
* **It encapsulates behaviors:** Each strategy encapsulates a specific behavior or algorithm, providing a clean and modular way to manage different variations or implementations.
* **It enables dynamic behavior switching:** The pattern enables clients to switch between different strategies at runtime, allowing for flexible and dynamic behavior changes.
* **It promotes object collaboration:** The pattern encourages collaboration between a context object and strategy objects, where the context delegates the execution of a behavior to a strategy object.

Overall, the Strategy pattern is a useful design pattern that allows the behavior of an object to be selected dynamically at runtime, providing flexibility, modularity, and testability.

* If we not use strategy pattern we might face code duplication.
* If we not use strategy pattern it will become tight coupling & violating the Open/Closed principle.
* If we not use strategy pattern we might need to have hard coded logic.

**When to use the Strategy Design Pattern?**

Here are some situations where you should consider using the Strategy pattern:

* **Multiple Algorithms**:
  + When you have multiple algorithms that can be used interchangeably based on different contexts, such as sorting algorithms (bubble sort, merge sort, quick sort), searching algorithms, compression algorithms, etc.
* **Encapsulating Algorithms**:
  + When you want to encapsulate the implementation details of algorithms separately from the context that uses them, allowing for easier maintenance, testing, and modification of algorithms without affecting the client code.
* **Runtime Selection**:
  + When you need to dynamically select and switch between different algorithms at runtime based on user preferences, configuration settings, or system states.
* **Reducing Conditional Statements**:
  + When you have a class with multiple conditional statements that choose between different behaviors, using the Strategy pattern helps in eliminating the need for conditional statements and making the code more modular and maintainable.
* **Testing and Extensibility**:
  + When you want to facilitate easier unit testing by enabling the substitution of algorithms with mock objects or stubs. Additionally, the Strategy pattern makes it easier to extend the system with new algorithms without modifying existing code.

**When not to use the Strategy Design Pattern?**

Here are some situations where you should consider not using the Strategy pattern:

1. **Single Algorithm**:
   * If there is only one fixed algorithm that will be used throughout the lifetime of the application, and there is no need for dynamic selection or switching between algorithms, using the Strategy pattern might introduce unnecessary complexity.
2. **Overhead**:
   * If the overhead of implementing multiple strategies outweighs the benefits, especially in simple scenarios where direct implementation without the Strategy pattern is more straightforward and clear.
3. **Inflexible Context**:
   * If the context class tightly depends on a single algorithm and there is no need for flexibility or interchangeability, using the Strategy pattern may introduce unnecessary abstraction and complexity.

**Advantages of the Strategy Design Pattern**

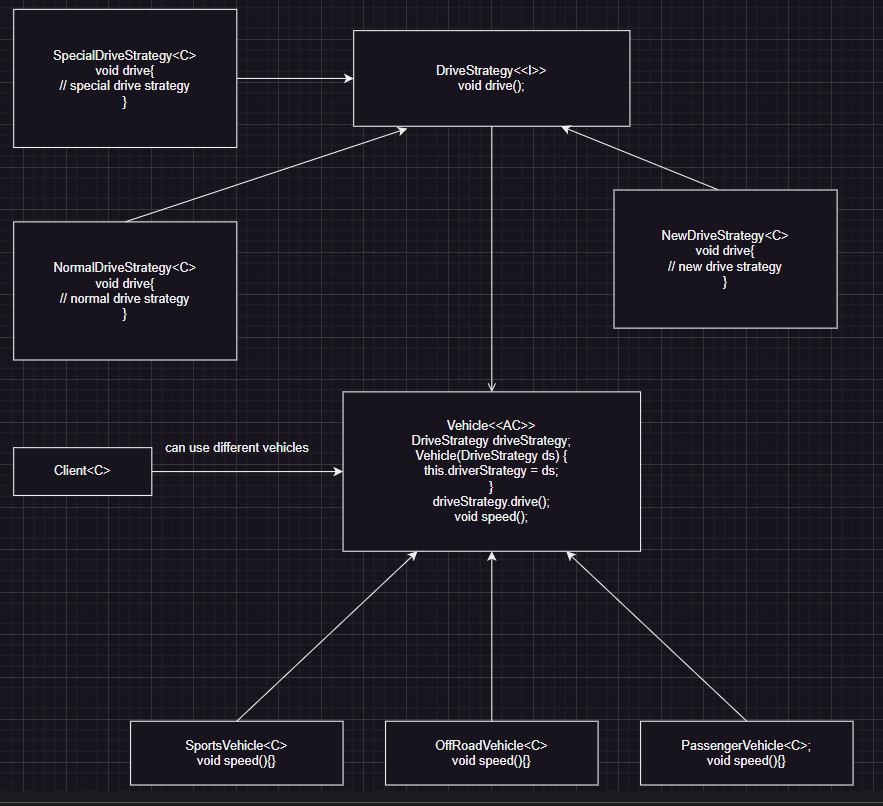
Below are the advantages of the strategy design pattern:

* A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
* By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
* The application can switch strategies at run-time.
* Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
* Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

**Disadvantages the Strategy Design Pattern**

Below are the disadvantages of the strategy design pattern:

* The application must be aware of all the strategies to select the right one for the right situation.
* Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
* In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.



* In above diagram we showed that take out drive capability into new strategy, so that different strategies can implement the strategy interface.
* Different vehicles can have has a relation with drive strategy so that client can decide which strategy can be used for different vehicles.
* While using strategy pattern we must check the behaviour not present in super class & sub classes it became duplicated.