Design Decisions Report

Overview of the Program:

Idea Selected:

The idea that has been selected for development is that of a vehicle simulation in an attempt to mock up a simple version of real-world vehicle behavior. In this exercise, vehicles are simulated to move, i.e., run or stop, while interfacing with a traffic light system governing their movement.

Overview of Implementation:

1. **Vehicle Representation**: Every vehicle in the simulation will be represented as an object of type `Vehicle`. The object will encapsulate attributes like the vehicle name, current position, speed, and the strategy the vehicle uses for movement.

2. **Strategy of vehicle movement**: A system will define different strategies for the movement of a vehicle, which defines how a vehicle should react in the simulation. Such strategies shall include choices to go ahead or stop the vehicle. Strategies should be implemented as concrete classes following some common interface and, hence, can be added and swapped easily.

3. **Traffic Light System**: The simulation should regulate the flow of traffic using a traffic light system and allow ways for controlling the state of the traffic light (red, yellow, green) and ways for vehicles to respond properly regarding the state.

4. **Main Simulation Execution**: Running the simulation will be the responsibility of this class. It initializes vehicles, prepares the traffic light system, and runs the loop of the simulation at each iteration over a specified duration.

Design/Implementation Decisions:

1. Factory Method Pattern:

Location: `Vehicle` class constructor.

The `Vehicle` class uses the factory method pattern to construct instances of movement strategies: `MoveForwardStrategy` and `StopStrategy`. This allows further simplification during the extension of the simulation with more strategies of movement without changing the `Vehicle` itself.

2. Singleton Pattern:

Location: `TrafficLightSystem` class.

Explanation: The `TrafficLightSystem` class is implemented with an instance to enforce that there should be one and only one instance of the traffic light system throughout the simulation. This will permit one to have different parts of the simulation that must always have consistency in the management of the traffic light state.

3. Strategy Pattern:

Locations: `MovementStrategy`, `MoveForwardStrategy`, `StopStrategy`.

Reasoning: The strategy pattern helps encapsulate and abstract away the different algorithms for vehicle movement. Using the definition of a common interface (`MovementStrategy`) and several implementations (`MoveForwardStrategy`, `StopStrategy`)), the simulation could easily switch other behaviors with moving without modification for class `Vehicle`.

4. Observer Pattern:

Location: Not to be executed directly in the given code, but could apply to, for example, informing vehicles about changes in traffic lights.

Reasoning: In this state, the observer pattern would notify the vehicles about any change of state in the traffic light. Vehicles will register themselves as observers of the traffic light system, hence notifying them in any case of light change and adapting their behavior accordingly.

5. State Pattern:

Location: Not implemented directly in the provided code, but could be used for managing the state of the traffic light.

This is the pattern that would have sufficed to account for the different states of the traffic light (e.g., red, yellow, green) and transitions of behavior within these states.

That would make the traffic light system much more modular and easily extendable with further states being added or complex state transitions. Logic and Data Structures

The simulation relies on simple data structures, so, to represent vehicle positions, speeds, and names.

The state of the traffic light should be represented by an enumerated type (`LightColor`) or a class (`LightColor`) of final static fields.

The use of lists and arrays is minimal in this implementation due to the simplicity of the simulation.

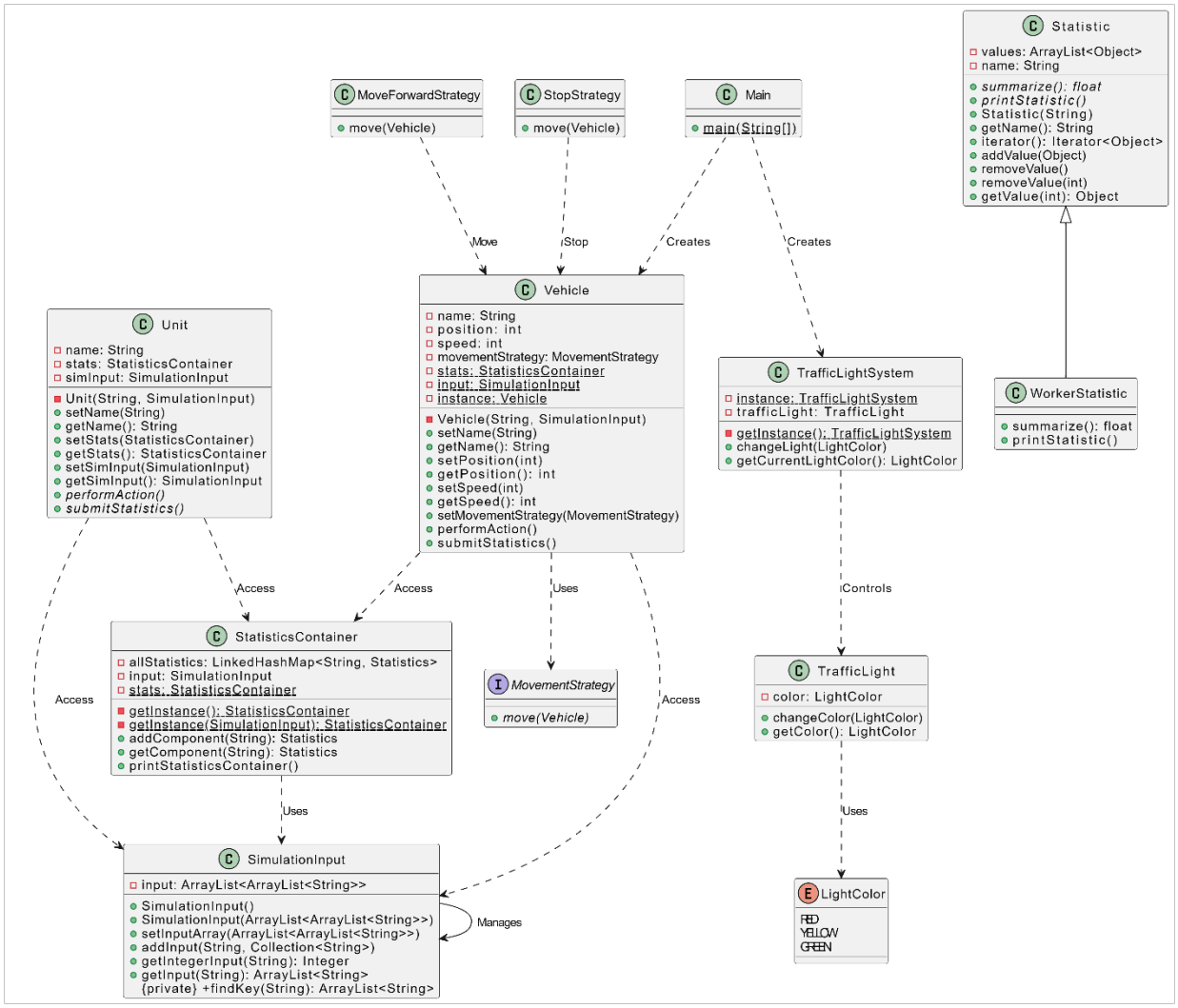
Usage of DRY, SOLID:

It observes the DRY (Don't Repeat Yourself) principle since common behaviors are encapsulated into reusable components, such as movement strategies.

Partially follows the SOLID principles: Single Responsibility Principle (each class with a single responsibility) and Open/Closed Principle (classes are open for extension but not modification). Synchronization Choices: This is not the case with the given code in its simulation, and therefore, synchronization is not explicitly implemented. However, when the corresponding simulation is generalized to concurrent execution or distributed systems, it may be added.

Visual Design of the Program

Sequence diagram



Conclusion:

In conclusion, flexibility, extensibility, and maintainability take the priority of the choices of the design decisions in the development of the vehicle simulation program. The design of the program encapsulates the movement behaviors through important design patterns, like the

strategy pattern, and the traffic light system management with the singleton pattern, probable use of the factory method pattern for object creation, hence modules are encouraged to be pluggable and replaceable. In addition, compliance with coding principles, such as DRY and SOLID, ensures that the codebase is kept short and reusable, hence resistant to change, to allow it to be easily maintained and changed with minimum compromise of existing functionality in the future.

Most importantly, potential measures of synchronization for multithreading scenarios and the considered selection of logic and data structures considerably reinforce the solidness and efficiency of the simulation, which paves a robust basis to represent every kind of traffic scenario, vehicle behavior diversity, and respective exploration.

References

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

2. Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

3. Freeman, E., & Robson, E. (2020). *Head First Design Patterns*. O'Reilly Media.

4. Fowler, M. (2018). *Refactoring: improving the design of existing code*. AddisonWesley Professional.

5. Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. AddisonWesley Professional.

6. Bloch, J. (2008). *Effective Java (the Java series)*. Prentice Hall PTR.

7. Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.