Technical Report on Architecture and Design Decisions

Universidad Distrital Francisco José de Caldas

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Autor:

Alejandro mauricio junco Oviedo 20231020129

Facultad de ingeniería de sistemas

Programación Avanzada

Docente: Carlos Andrés Sierra Virgüez

13/03/2024

Introduction:

The provided code represents a Python program designed to manage vehicles and their associated engines. The program utilizes object-oriented programming principles to create classes for engines and different types of vehicles. This report outlines the technical concerns and design decisions made in creating the architecture of the code.

Class Structure:

1. Engine Class:

- The `Engine` class represents the behavior of a vehicle engine.
- It encapsulates the properties of an engine, such as its name, type, potency, and weight.
- The class provides a constructor (`__init__`) to initialize these properties.

2. Vehicle Class:

- The `Vehicle` class is an abstract base class that defines the common attributes and behaviors of vehicles.
- It includes properties like the chassis, model, year, and an associated engine object.
- The class implements methods to retrieve information about the vehicle, such as the chassis, model, year, gas consumption, and engine details.
- It also includes a protected method (`_calculate_consumption`) to calculate the gas consumption based on the engine's potency, weight, and chassis type.
- The class enforces a constraint on the chassis type, allowing only 'A' or 'B' values.

3. Concrete Vehicle Classes:

- The code defines four concrete subclasses of `Vehicle`: `Car`, `Truck`, `Yatch`, and `Motorcycle`.
- Each subclass represents a specific type of vehicle and extends the base `Vehicle` class.
- The subclasses add additional attributes and behaviors specific to their respective vehicle types.
- For example, the `Car` class includes the number of passengers, the `Truck` class has the number of axles, the `Yatch` class has the length, and the `Motorcycle` class has the suspension type.

Design Decisions:

1. Inheritance: The code follows the principle of inheritance by defining a base `Vehicle` class and deriving concrete vehicle classes from it. This design decision promotes code reuse and

maintainability, as common attributes and behaviors are defined in the base class, while specific characteristics are added in the derived classes.

2. Encapsulation: The code encapsulates the state (attributes) and behavior (methods) of engines and vehicles within their respective classes. This encapsulation promotes data hiding and helps maintain the integrity of the objects by controlling access to their internal state.

3. Attribute Validation: The `Vehicle` class includes validation for the chassis type, ensuring that only valid values ('A' or 'B') are accepted. This helps maintain data integrity and prevents the creation of invalid objects.

4. Separation of Concerns: The code separates the concerns of engine management and vehicle management into distinct classes (`Engine` and `Vehicle`, respectively). This separation promotes code organization and makes it easier to maintain and extend the codebase.

5. User Input Handling: The code includes a menu-driven interface that allows users to create engines, create different types of vehicles, and display information about the created engines and vehicles. This design decision enhances the usability and interactivity of the program.

6. Global Data Structures: The code utilizes global data structures (`engines` and `vehicles`) to store the created engines and vehicles, respectively. While this approach may work for small-scale applications, it could potentially lead to issues in larger or more complex systems. An alternative design decision could be to encapsulate the management of these data structures within dedicated classes or modules.

Possible Improvements and Future Considerations:

1. Error Handling: The code could benefit from more robust error handling mechanisms, such as catching and handling specific exceptions, and providing more informative error messages to users.

2. Data Persistence: Currently, the created engines and vehicles are stored in memory and will be lost when the program terminates. To maintain persistent data, the code could be extended to incorporate file-based or database-based storage mechanisms.

3. Modularity and Separation of Concerns: While the code separates the concerns of engines and vehicles into separate classes, it could be further modularized by separating the user interface

logic, data management logic, and core business logic into separate modules or classes. This would enhance code organization, maintainability, and testability.

4. Inheritance Hierarchy: The current inheritance hierarchy for vehicles could be reviewed and potentially restructured to better align with real-world vehicle types and their relationships. This could involve introducing additional abstract classes or restructuring the existing hierarchy based on common attributes and behaviors.

5. Unit Testing: To ensure the correctness and robustness of the codebase, it is recommended to implement unit tests for the various classes and methods. This would help catch regressions and facilitate future code changes and refactoring.

6. Documentation and Code Comments: While the code includes some comments explaining the purpose of classes and methods, it could benefit from more comprehensive documentation and code comments. This would enhance the readability and maintainability of the codebase, especially for larger projects or collaborative development efforts.

Conclusion:

The provided code demonstrates a basic implementation of an object-oriented system for managing vehicles and their associated engines. The architecture follows common design principles such as inheritance, encapsulation, and separation of concerns. However, there is room for improvement in areas such as error handling, data persistence, modularity, unit testing, and documentation. By addressing these concerns, the codebase can be further enhanced and made more robust, maintainable, and extensible.