Problem Statements Using Abstraction

Problem Statement 1: **Bank Account System**

Imagine a **Bank Account System** where we need to model various types of accounts, such as **CheckingAccount** and **SavingsAccount**. Both types of accounts should have a method for depositing and withdrawing money, but the rules for calculating interest and handling transactions may differ between account types.

Abstraction Details:

- We create an abstract class called `BankAccount` that defines the basic methods for deposits, withdrawals, and getting the balance.
- Subclasses such as `CheckingAccount` and `SavingsAccount` will provide their own implementation of these methods.

Problem Statement:

1. Define an abstract class `BankAccount` with abstract methods `deposit()`, `withdraw()`, and `get_balance()`.
2. Create subclasses `CheckingAccount` and `SavingsAccount`, each with its own way of calculating interest or handling withdrawals (e.g., SavingsAccount may have interest accumulation).
3. Write a function `process_transaction()` that accepts any `BankAccount` object and performs deposit, withdrawal, and balance retrieval operations, abstracting the underlying logic.

Problem Statement 2: **Vehicle Simulation System**

In a **Vehicle Simulation System**, you need to model different types of vehicles such as **Car**, **Truck**, and **Motorcycle**. Each vehicle should have methods for starting the engine, stopping the engine, and refueling, but the exact implementation (like fuel consumption and engine start procedure) might differ across the vehicle types.

Abstraction Details:

- Define an abstract class `Vehicle` with abstract methods like `start_engine()`, `stop_engine()`, and `refuel()`.
- Subclasses such as `Car`, `Truck`, and `Motorcycle` should implement these methods with their own specifics (e.g., a `Truck` may have a larger fuel tank and different engine startup procedure).

Problem Statement:

1. Define an abstract class `Vehicle` with abstract methods `start_engine()`, `stop_engine()`, and `refuel()`.
2. Create subclasses `Car`, `Truck`, and `Motorcycle`, where each vehicle type implements the methods to suit its characteristics.
3. Write a function `vehicle_service()` that accepts any `Vehicle` object and performs operations like starting the engine, stopping the engine, and refueling, abstracting the differences between vehicle types.
