**Samuel Edet Ukim**

**OOP HW2**

## Part A: Conceptual Questions

### Definition

- **Encapsulation:** In my own words, encapsulation is the bundling of data (attributes) and the methods (functions) that operate on that data within a single unit, [1] often a class. It also involves controlling the access to this internal data, typically by making it private and providing controlled public interfaces (methods) to interact with it.

- **Example of preventing unintended changes:** Imagine a `Car` class with a private `fuelLevel` data member. Without encapsulation, any part of the program could directly change `fuelLevel` to an invalid value (e.g., -50 or 1000). With encapsulation, the `Car` class would provide a public method like `refuel(int amount)`. This method can contain logic to ensure that the `amount` is valid (e.g., non-negative and doesn't exceed the tank capacity) before updating the private `fuelLevel`.

### Visibility Modifiers

- **Public:**
  - **Benefit:** High code flexibility, as any part of the program can directly access public members.
  - **Potential Drawback:** Low data safety, as internal data can be unintentionally or maliciously modified, leading to unpredictable behavior and making debugging harder.
- **Private:**
  - **Benefit:** High data safety, as internal data can only be accessed and modified through the class's own methods, allowing for controlled changes and maintaining data integrity.
  - **Potential Drawback:** Reduced code flexibility, as direct access to internal details is restricted, potentially requiring more methods to expose necessary functionalities.
- **Protected:**
  - **Benefit:** Allows derived classes to directly access and modify inherited members, providing flexibility for specialized behavior in subclasses.

  - **Potential Drawback:** Lower data safety compared to private, as derived classes can potentially introduce unintended changes, and changes to protected members can impact all derived classes, potentially reducing maintainability.
- **Scenario for intentionally using protected members:** You might intentionally use `protected` members in a base class when designing a framework or a set of related classes where derived classes are expected to extend or customize the base class's behavior by directly manipulating certain internal data, but direct access from unrelated parts of the program should still be restricted. For example, in a GUI framework, a base `Widget` class might have `protected` members for its internal geometry, allowing derived widgets like `Button` or `TextBox` to adjust their layout directly.

**Impact on Maintenance**

- Encapsulation can significantly reduce debugging complexity in large codebases because it isolates the impact of changes. If the internal implementation of a class (its private members and the logic within its methods) needs to be modified, as long as the public interface (the signatures and behavior of its public methods) remains the same, the rest of the codebase that uses this class should not break. This localized impact makes it easier to pinpoint the source of bugs during maintenance, as you primarily need to examine the class that was changed.

- **Brief example of code breaking if internal data is made public:** Consider a `ShoppingCart` class with a public `items` vector. If another part of the program directly adds a non-`Product` object to this vector, or modifies the quantity of an item without going through the `ShoppingCart`'s methods (which might enforce business rules like checking inventory), the internal state of the `ShoppingCart` could become inconsistent, leading to errors in calculations or unexpected behavior later on.

**Real-World Analogy**

- **Real-Life Object/System:** A television remote control.
- **Public Interface:** The buttons on the remote (power, volume up/down, channel up/down, mute, etc.). These are the ways the user interacts with the TV. The user doesn't need to know how the signals are generated or transmitted.
- **Private Implementation:** The internal circuitry, the battery, the infrared or Bluetooth transmitter, the specific encoding schemes for each button press. This is hidden from the user.
- **Why is it helpful to keep the private side hidden?** It simplifies the user's interaction – they only need to understand the function of the buttons. It also allows the manufacturer to change the internal implementation (e.g., use a different type of transmitter or encoding) without affecting how the user operates the remote, as long as the public interface (the buttons and their functions) remains the same. This makes the system more robust to internal changes and easier to upgrade or maintain.

**Part B: Small-Class Design (Minimal Coding)**

C++

```
#include <iostream>
#include <string>

class BankAccount {
private:
    double balance;
    std::string accountNumber;

public:
    // Constructor
    BankAccount(std::string accNumber, double
initialBalance);

    // Public Methods
    void deposit(double amount);
    bool withdraw(double amount);

    // Documentation (Conceptual)
    // Note: Do not directly manipulate the 'balance'
member. Use the provided public methods.
};

// (Minimal implementation for demonstration)
BankAccount::BankAccount(std::string accNumber, double
initialBalance) : accountNumber(accNumber),
balance(initialBalance) {}

void BankAccount::deposit(double amount) {
    if (amount > 0) {
        balance += amount;
        std::cout << "Deposit of " << amount << "
successful. New balance: " << balance << std::endl;
    } else {
        std::cout << "Invalid deposit amount." <<
std::endl;
    }
}

bool BankAccount::withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        std::cout << "Withdrawal of " << amount << "
successful. New balance: " << balance << std::endl;
        return true;
    } else {
        std::cout << "Insufficient funds or invalid
withdrawal amount." << std::endl;
        return false;
    }
}
```

**Encapsulation Justification:**

- **`balance` (private):** The `balance` should be private to ensure that the account's funds are only modified through controlled operations like deposits and withdrawals. Direct external manipulation could lead to inconsistencies (e.g., setting a negative balance without a withdrawal) and bypass any business rules or logging mechanisms the `BankAccount` class might implement.

- **`accountNumber` (private):** The `accountNumber` is a unique identifier for the account. It should be private to prevent accidental or malicious modification, which could lead to confusion, security breaches, or the inability to correctly identify the account. Once assigned, the account number should generally remain constant.

**Enforcing Constraints/Validations:**

- **`deposit(double amount)` (public):** This method enforces the constraint that only positive amounts can be deposited. It checks if `amount > 0` before updating the `balance`. This prevents unintended negative changes to the balance.

- **`withdraw(double amount)` (public):** This method enforces two constraints: the withdrawal amount must be positive (`amount > 0`), and there must be sufficient funds in the account (`amount <= balance`). It only modifies the `balance` and returns `true` if these conditions are met, ensuring the balance doesn't become negative due to withdrawals and preventing invalid withdrawal requests.

**Documentation:**

C++

```
/**
 * @brief Represents a bank account with a balance and
account number.
 *
 * The balance is a private member and should NOT be
directly manipulated.
```

```
 * Use the public methods (deposit, withdraw) to interact
with the balance,
 * as they enforce necessary constraints and validations.
 */
class BankAccount {
private:
    double balance;
    std::string accountNumber;

public:
    /**
     * @brief Constructs a BankAccount object.
     * @param accNumber The initial account number.
     * @param initialBalance The initial balance of the
account.
     */
    BankAccount(std::string accNumber, double
initialBalance);

    /**
     * @brief Deposits a positive amount into the account.
     * @param amount The amount to deposit. Must be greater
than zero.
     * @note Do not directly modify the private 'balance'
member.
     */
    void deposit(double amount);

    /**
     * @brief Withdraws a positive amount from the account
if sufficient funds are available.
     * @param amount The amount to withdraw. Must be
greater than zero and not exceed the current balance.
     * @return True if the withdrawal was successful, false
otherwise.
     * @note Do not directly modify the private 'balance'
member.
     */
    bool withdraw(double amount);
};
```
**Part C: Reflection & Short-Answer**

- **Pros and Cons of Hiding Internal Data:**
    - **Benefit 1 (Increased Data Integrity):** Hiding internal data behind methods allows the class to control how its data is modified, ensuring that invariants are maintained and the data remains in a valid state.
    - **Benefit 2 (Reduced Coupling):** Changes to the internal implementation of a class (its private members and method logic) are less likely to affect other parts of the program that use the class, as long as the public interface remains consistent. This reduces dependencies and makes the codebase easier to evolve.
    - **Potential Limitation/Overhead:** Introducing methods to access and modify data can sometimes add a small performance overhead compared to direct access (though compilers often optimize this). It can also lead to more code in the class interface if many attributes need to be indirectly accessed.
- **Encapsulation vs. Abstraction:** Encapsulation is about bundling data and the methods that operate on it, and controlling access to the data (how it's represented). Abstraction is about hiding complex implementation details and showing only the essential information to the user (what an object does). Encapsulation focuses on the "how" of data protection, while abstraction focuses on the "what" of functionality exposure.

- **Encapsulation and Abstraction as Information Hiding:** Both encapsulation and abstraction can be considered forms of "information hiding" because they aim to limit the amount of detail that is exposed to the outside world. Encapsulation hides the internal data representation and implementation details, while abstraction hides the complexity of the underlying operations, presenting a simplified view of the object's capabilities.

- **Testing Encapsulated Classes:** Even if data is private, we can still thoroughly unit test the class by focusing on its **public interface (the public methods)**. We can create test cases that call these public methods with various inputs (including boundary and edge cases) and assert that the object behaves as expected based on its observable state (e.g., by calling other public methods to check the resulting state or by observing any side effects). This approach tests the class's behavior and the correctness of its internal logic without needing to directly access the private data. For example, to test the `BankAccount`, we would call `deposit` and `withdraw` with different amounts and then check the balance using a hypothetical public `getBalance()` method (if needed for testing purposes, though not strictly required for the class's core functionality).