Samuel Edet Ukim

OOP HW5

Part A: Conceptual Questions

Definition

- **Abstraction:** In my own words, abstraction in OOP is the process of simplifying complex reality by modeling classes based on the essential attributes and behaviors relevant to a particular context, while hiding the unnecessary implementation details. It focuses on what an object *does* rather than *how* it does it, providing a high-level view that is easier to understand and use.

- **Real-world analogy:** Consider a television remote control. The user interacts with buttons labeled "Power," "Volume Up," "Channel Down," etc. These buttons represent the abstract interface. The complex internal circuitry, signal processing, and communication protocols that happen when you press a button are hidden from the user. You don't need to know how the remote works internally to use it effectively.

Abstraction vs. Encapsulation

- **Brief comparison:** Abstraction focuses on hiding the complexity of *what* an object does, showing only the necessary features to the user. Encapsulation focuses on bundling data and the methods that operate on it, and controlling access to the data (the *how* of data protection and internal implementation).

- **Why someone might confuse the two:** Both concepts are related to information hiding and aim to simplify interaction with objects. Encapsulation achieves this by controlling access to internal data, while abstraction achieves it by hiding complex implementation details behind a simplified interface. They often work together; for example, a class might encapsulate its data and provide abstract methods to interact with that data.

Designing with Abstraction (Smart Thermostat)

- **Essential attributes:**
  1. `targetTemperature` (the desired temperature setting).
  2. `currentTemperature` (the actual measured temperature).
  3. `operatingMode` (e.g., "heating," "cooling," "off").
- **Essential methods:**
  1. `setTargetTemperature(float temperature)` (allows the user or system to set the desired temperature).
  2. `getOperatingStatus()` (returns the current operating mode and potentially the current temperature).
- **Why omit internal details:** Internal details like the circuit design, specific firmware routines for temperature sensing and control algorithms, or the communication protocol with the HVAC unit are irrelevant to how the user or the broader home automation

system interacts with the thermostat. Exposing these details would add unnecessary complexity to the interface, making it harder to understand and use, and would also make the system more susceptible to issues if those internal implementations change.

Benefits of Abstraction

- **Two benefits in large-scale projects:**
    1. **Reduced Complexity:** Abstraction breaks down a complex system into smaller, more manageable parts by providing simplified interfaces. This allows developers to focus on the high-level interactions between objects without being overwhelmed by the intricate details of each component.
    2. **Increased Modularity and Maintainability:** By hiding implementation details, abstraction reduces the dependencies between different parts of the system. Changes to the internal workings of one class are less likely to affect other classes that interact with it through its abstract interface, making the codebase easier to maintain, update, and extend.
- **How abstraction reduces code complexity:** Abstraction reduces code complexity by allowing developers to work with simplified models of objects, focusing on their essential functionalities rather than their intricate internal workings.

Part B: Minimal Class Example (C++)

C++

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

// Abstract Base Class
class BankAccount {
public:
    virtual void deposit(double amount) = 0;
    virtual bool withdraw(double amount) = 0;
    virtual double getBalance() const = 0;
    virtual ~BankAccount() {} // Virtual destructor for
proper cleanup in derived classes
};

// Derived Class
class SavingsAccount : public BankAccount {
private:
    double balance;
    std::string accountNumber;
```

```
    // Internal details (hidden from direct user
interaction)
    void logTransaction(const std::string& type, double
amount) {
        std::cout << "Logging transaction: " << type << " -
Amount: " << amount << std::endl;
        // In a real system, this would involve more
complex logging
    }

    bool checkSufficientFunds(double amount) const {
        return balance >= amount;
    }

public:
    SavingsAccount(std::string accNumber, double
initialBalance) : accountNumber(accNumber),
balance(initialBalance) {}

    void deposit(double amount) override {
        if (amount > 0) {
            balance += amount;
            logTransaction("Deposit", amount);
        } else {
            std::cout << "Invalid deposit amount." <<
std::endl;
        }
    }

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

```
    double getBalance() const override {
        return balance;
    }
};

int main() {
    SavingsAccount myAccount("12345", 1000.0);
    myAccount.deposit(500.0);
    myAccount.withdraw(200.0);
    std::cout << "Current Balance: " <<
myAccount.getBalance() << std::endl;
    return 0;
}
```

Part C: Reflection & Comparison

- **Distilling the Essentials in `SavingsAccount`:** In the `SavingsAccount` class, I would hide the `logTransaction()` method, the `checkSufficientFunds()` method, the `balance` data member, and the `accountNumber` data member from direct user calls. The public interface should primarily consist of `deposit()`, `withdraw()`, and `getBalance()`. This provides a clear and simplified way for the user to interact with their savings account without needing to know about the internal logging mechanisms, fund-checking procedures, or the underlying data storage.

- **Contrast with Polymorphism:** When a method like `deposit()` is called on a `SavingsAccount` object (which is a concrete implementation of the abstract `BankAccount` interface), it highlights polymorphism because the code is interacting with the object through the `BankAccount` interface (a base type). However, the actual `deposit()` method that gets executed is the specific implementation defined in the `SavingsAccount` class. This also showcases abstraction because the user of the `BankAccount` interface (or code interacting with it) doesn't need to know the specific steps involved in the `SavingsAccount`'s `deposit()` method (like how the balance is updated or how the transaction is logged); they only need to know that calling `deposit()` will increase the account's balance. The abstract `BankAccount` class defines *what* can be done, while the concrete `SavingsAccount` class defines *how* it's done, hiding the complexities.

- **Real-World Example:** In the domain of **gaming**, consider a game engine's API for handling different types of game objects (e.g., characters, items, scenery). The engine provides an abstract `GameObject` class with methods like `render()`, `update()`, and `handleCollision()`. Concrete game objects (like a `PlayerCharacter`, a `Potion`, or a `Tree`) inherit from `GameObject` and provide their specific implementations for these methods. The game logic can then interact with these diverse

objects through the unified `GameObject` interface, without needing to know the intricate details of how each object is rendered, updated, or handles collisions. This abstraction simplifies the game development process significantly.