Heshawa Cooray

Part A: Conceptual Questions

Definition

- **Abstraction in OOP** is the process of simplifying a system by focusing on essential details and hiding the unnecessary complexities. It helps manage complex systems by defining clear interfaces and leaving out the inner workings that are not needed for users or other systems.
- **Real-world analogy**: A TV remote is an abstraction. You don't need to know how it works inside, just press the buttons to control the TV. The details like the circuit design and signal transmission are hidden from you.

Abstraction vs. Encapsulation

- Abstraction focuses on exposing only relevant information and hiding complexity. Encapsulation hides internal data and details, protecting the state of an object and only allowing access through methods.
- The confusion might arise because both hide details, but abstraction is more about what is shown (the interface), while encapsulation is about how the internal workings are protected.

Designing with Abstraction

For a **smart thermostat**:

- **Attributes**:
    1. Temperature setting
    2. Mode (e.g., heating or cooling)
    3. Current temperature
- **Methods**:
    1. setTemperature()
    2. toggleMode()
- I would omit details like how the temperature sensor works or the exact firmware logic because these aren't necessary for the user to interact with the thermostat effectively. The user just needs a simple interface to set the temperature and toggle modes.

Benefits of Abstraction

- **Two benefits**:
    1. Simplifies system maintenance by reducing complexity.
    2. Makes code more flexible and easier to extend.
- Abstraction reduces code complexity by focusing only on what's needed for the task at hand, leaving out unnecessary implementation details.

Part B: Minimal Class Example

```
// Base class
class BankAccount {
public:
  void deposit(double amount) {
    // Implementation for deposit
  }

  void withdraw(double amount) {
    // Implementation for withdraw
  }

  double getBalance() {
    return balance;
  }

private:
  double balance;
};

// Derived class
class SavingsAccount : public BankAccount {
private:
  double balance;
public:
  SavingsAccount() : balance(0.0) {}

  // Deposit method specific to SavingsAccount
  void deposit(double amount) {
    balance += amount;
  }

  // Withdraw method specific to SavingsAccount
  void withdraw(double amount) {
    if (amount <= balance) balance -= amount;
  }

  // Get balance method specific to SavingsAccount
  double getBalance() {
    return balance;
  }
};
```

- The BankAccount class is abstract, defining essential operations like deposit and withdraw.
- SavingsAccount implements the actual logic, while hiding internal details like the balance tracking mechanism or logging activities.

Part C: Reflection & Comparison

Distilling the Essentials

- In the SavingsAccount class, the balance and internal data like transaction history should be hidden. Only the deposit, withdraw, and getBalance methods should be accessible to the user, ensuring they only interact with the core functionality.

Contrast with Polymorphism

- Polymorphism is showcased when a method call like deposit(100) is called on a BankAccount reference, which could point to any derived class (like SavingsAccount). This shows how different types of accounts share the same interface but implement the details differently.

Real-World Example

- In gaming abstraction helps simplify APIs. For instance, a game might expose simple functions like moveCharacter() without exposing the complex physics engine that calculates the movement.

Optional: Additional Exploration

Interfaces vs. Abstract Classes

- An interface defines a contract of methods that must be implemented, but it can't contain any implementation. An abstract class can provide some method implementations while leaving others abstract.
- If you need multiple classes to have the same methods but with different behaviors, an interface might be better, as it forces all classes to implement those methods without imposing any inheritance hierarchy.

Testing Abstractions

- For unit testing abstract classes, you could create a concrete subclass or use mock objects to simulate the behavior of abstract methods and test how the rest of the system interacts with them.