Turram Toussaint

2143 OOP

Prof Griffin

Part A: Conceptual Questions

Definition

* **What is abstraction in OOP?**Abstraction is the concept of exposing only essential features of an object while hiding the internal implementation details to simplify usage and focus on what the object does rather than how it does it.
* **Real-world analogy:** If a TV remote lets you change the channel and volume without needing to understand the electronics inside—it hides the internal wiring and logic while exposing simple buttons.

Abstraction vs. Encapsulation

* **Comparison:**
  + **Abstraction** hides complexity by showing only relevant data and behavior.
  + **Encapsulation** hides data by bundling it and restricting direct access through access modifiers.
* **Why they’re confused:** Both involve hiding information and improving security or clarity, but abstraction focuses on "what" is exposed, while encapsulation focuses on "how" data is protected.

Designing with Abstraction: Smart Thermostat

* **Attributes:**
  + currentTemperature
  + targetTemperature
  + mode (heat/cool/auto)
* **Methods:**
  + setTargetTemperature(int temp)
  + switchMode(string mode)
* **Omitted internals:** Circuit design or firmware routines are irrelevant to users or high-level systems and would only complicate the design unnecessarily.

Benefits of Abstraction

* **Two benefits:**
  + Easier to manage complexity by separating interface from implementation.
  + Promotes reusability and scalability across modules or teams.
* **Reducing complexity:** Abstraction hides low-level operations so developers can interact with clean, meaningful interfaces.

Part B: Minimal Class Example (C++ - Banking System)

// Abstract class

class BankAccount {

public:

virtual void deposit(double amount) = 0;

virtual void withdraw(double amount) = 0;

virtual ~BankAccount() {}

};

// Derived class

class SavingsAccount : public BankAccount {

public:

void deposit(double amount) override {

// Encrypt, log, and update balance

cout << "Depositing $" << amount << " to savings.\n";

}

void withdraw(double amount) override {

// Check balance, log, update ledger

cout << "Withdrawing $" << amount << " from savings.\n";

}

};

**Note:** Internals like logging, encryption, and ledger updates are hidden from the user interface.

Part C: Reflection & Comparison

Distilling the Essentials

* **Hidden from user:**
  + Internal transaction logs
  + Encryption/decryption logic
  + Ledger balancing or audit trail updates
* **Why:** Exposing these details would overwhelm the user and increase the risk of misuse; the interface should stay focused on core actions.

Contrast with Polymorphism

* A BankAccount\* pointer can hold a SavingsAccount object and call withdraw().
* This demonstrates **abstraction** (exposing only what’s necessary) and **polymorphism** (allowing the call to be resolved at runtime based on the actual object type).

Real-World Example

* **Domain:** Healthcare  
   A PatientRecord API abstracts away the medical database structure, exposing only methods like getVitals() or addDiagnosis() to simplify clinician use.

(Optional) Additional Exploration

Interfaces vs. Abstract Classes

* **Summary:** An interface defines a contract with no implementation, while an abstract class can provide partial implementation alongside abstract methods.  
   Interfaces support multiple inheritance; abstract classes do not (in some languages like Java or C#).
* **Suitable Scenario for Interface:** In a plugin system where different modules (e.g., payment processors) must conform to the same IPaymentProcessor interface regardless of internal structure.

Testing Abstractions

* **Strategy:** Use **mock classes** that inherit from the abstract class and implement the abstract methods with testable behaviors. Then unit test through those concrete implementations.

Let me know if you want this in a formatted PDF or would like to expand this into a small project!