

OBJECT ORIENTED PROGRAMMING

Lab Assignment: Abstract Classes, Virtual Functions, Interfaces & Protocols, Composition over Inheritance, Delegation, Refused Bequest, and LSP

29 October 2026

Instructions: Time duration: **2 hours**. While C++ is the suggested reference language, all questions are language-agnostic. Students are free to use any object-oriented language of their choice (e.g., C++, Java, or Python), provided all specified access semantics, polymorphic behaviors, and contracts are strictly implemented.

1. Abstract Base Classes (ABC) and Dynamic Polymorphism

Design a publication catalog. Define an abstract base class `Publication` with protected data members `title` (string) and `price` (float), and two pure virtual functions `getData()` and `putData()`. Derive two concrete classes: `Book` (adding integer `pageCount`) and `AudioTape` (adding float `playingTime`), overriding both functions in each class. In `main()`, store both types of objects in a collection of base class pointers, take input from the user, and polymorphically display the complete catalog. Verify that directly creating an instance of `Publication` produces a compilation error.

2. Early Binding vs. Late Binding

Demonstrate compile-time binding versus runtime dynamic dispatch using a shape hierarchy. Create a base class `Shape` containing a **non-virtual** function `identify()` that prints "[Early Binding] Base Shape" and a **virtual** function `calculateArea()` returning 0.0. Derive `Circle` (attribute `radius`) and `Rectangle` (attributes `width`, `height`), overriding both functions in each derived class. Instantiate derived objects dynamically via base class pointers (`Shape*`), invoke both functions through the pointers, and explain why `identify()` executes the base version while `calculateArea()` invokes the derived version.

3. Virtual Destructors for Memory Cleanup

Create an abstract base class `MediaStream` declaring a pure virtual function `playStream()` and a virtual destructor. Derive a class `AudioBufferStream` that dynamically allocates an internal character array buffer in its constructor and frees it in its destructor with a confirmation message. In `main()`, allocate an `AudioBufferStream` object dynamically through a `MediaStream*` pointer and delete it through that base pointer, demonstrating that the virtual destructor guarantees the derived destructor executes and prevents memory leaks.

4. Pure Interfaces and Runtime Interoperability

Implement a decoupled notification system using a pure interface. Define `INotificationService` with no data members, declaring pure virtual methods `sendNotification(recipient, msg)` and `getChannelName()`. Implement two concrete services: `EmailNotificationService` and `SMSNotificationService`. Create a client class `AlertDispatcher` that receives an `INotificationService*` via constructor dependency injection and provides a method `dispatchAlert(user, text)`. In `main()`, demonstrate that `AlertDispatcher` can switch notification channels dynamically at runtime without modifying the dispatcher's code.

5. Composition over Inheritance

Implement a pluggable audit logging system using composition instead of inheritance. Define an interface `ILogger` with a pure virtual method `log(message)`, implemented by `ConsoleLogger` and `FileLogger`. Create a `BankingService` class that manages an account balance with `deposit(amount)` and `withdraw(amount)`, composing an `ILogger*` as a private member to audit all transactions. Provide a method `setLogger()` to swap the active logger dynamically at runtime, and demonstrate that switching logging targets requires no changes to the banking logic.

6. Method Delegation vs. Multiple Inheritance

A laboratory has two separate equipment classes: `Spectrometer` (method `analyze()`) and `ThermalRegulator` (method `setTemperature(celsius)`). Using delegation rather than multiple inheritance, build a composite control class `MaterialAnalyzer` that holds both devices as private members. Expose member functions `configureChamber(temp)`, `recordSample()`, and a workflow function `runFullExperiment(targetTemp)` that orchestrates heating and scanning by delegating calls to the internal devices.

7. Refactoring the “Refused Bequest” Code Smell

The *Refused Bequest* smell occurs when a subclass inherits methods from a parent that violate its core abstractions (such as a `Stack` inheriting arbitrary indexed insertions from a `List`). Implement a robust `SafeStack` class using composition rather than inheriting from a list: store a dynamic container privately and expose only legitimate stack operations (`push(val)`, `pop()`, `peek()`, and `isEmpty()`), ensuring invalid indexed modifications are impossible from client code.

8. Liskov Substitution Principle (LSP)

A banking system contains standard checking accounts and fixed deposit accounts where withdrawals are locked. Using interface segregation to ensure LSP compliance, define a base class `Account` containing universal operations (`deposit(amount)` and `getBalance()`), and a separate interface `IWithdrawable` declaring `withdraw(amount)`. Implement `StandardAccount` to inherit from `Account` and implement `IWithdrawable`, while `FixedDepositAccount` inherits only from `Account`. Write a client function `executeMonthlyDeduction(IWithdrawable& acc, double fee)` to demonstrate that non-withdrawable accounts are safely prevented from being passed at compile time.