Practical Exercise: Foundations of Object-Oriented Programming

Mobile App Domain Modeling & Implementation

Course: Mobile Application Development (Chapter 3)
Level: Technical University Students
Duration: 6–8 hours self-study + 1–2 hours online discussion
Format: Individual Assignment with Code & Documentation (2–3 pages)

Exercise Overview

In this exercise, you will design and implement core OOP concepts—**classes, objects, encapsulation, inheritance, and polymorphism**—applied to a real mobile application domain.

Key Learning Outcomes:

- Model real-world entities as classes with appropriate attributes and methods
- Apply encapsulation principles to protect data integrity
- Recognize inheritance hierarchies and design reusable base classes
- Implement polymorphism through method overriding and interfaces
- Write clean, maintainable code following OOP best practices

---

Part A: Domain Modeling

Task 1: Identify Entities & Design Classes (Self-Study: 2–3 hours)

Objective: Model a mobile application domain using OOP concepts.

Instructions:

Choose **one of the following domains** (or propose your own):

- **Ride-Hailing App** (Uber, Lyft, Yandex.Taxi)
- **E-Commerce App** (shopping cart, products, orders)
- **Social Media App** (users, posts, comments, likes)
- **Fitness Tracking App** (workouts, users, goals, achievements)
- **Banking App** (accounts, transactions, users, cards)

For your chosen domain:

1. **Identify at least 4–5 core entities** (real-world objects):
    - Example (Ride-Hailing): `User`, `Driver`, `Ride`, `Car`, `Location`, `Payment`
2. **For each entity, design a class with:**
    - **Class name** (singular, descriptive)
    - **3–5 attributes** (fields) with appropriate data types
    - **2–4 methods** (behaviors/actions the object can perform)
3. **Document in a table or sketch:**

Entity	Attributes	Methods	Purpose
Ride	`rideId` (String), `passengerId` (String), `driverId` (String), `pickupLocation` (Location), `dropoffLocation` (Location), `status` (String), `fare` (double)	`calculateFare()`, `updateStatus()`, `getRideDetails()`, `cancelRide()`	Represents a single ride request and its lifecycle
Driver	`driverId` (String), `name` (String), `rating` (double), `carLicense` (String), `phone` (String), `isAvailable` (boolean)	`acceptRide()`, `rejectRide()`, `completeRide()`, `getRating()`, `updateAvailability()`	Represents a driver offering rides
Location	`latitude` (double), `longitude` (double), `address` (String)	`getDistance()`, `getCoordinates()`, `getAddress()`	Represents a geographic point
Payment	`paymentId` (String), `amount` (double), `method` (String), `timestamp` (String), `status` (String)	`processPayment()`, `refund()`, `getReceipt()`	Handles payment transactions

Deliverable:

- Table or UML-style sketch of at least 4 classes with attributes and methods
- 1–2 sentences per class explaining its role in the domain

Part B: Encapsulation

Task 2: Access Control & Data Protection (Self-Study: 1–2 hours)

Objective: Implement encapsulation using access modifiers and getters/setters.

Instructions:

1. **Choose one class** from Task 1 (e.g., `Payment` or `Driver`).
2. **Identify which attributes should be:**
   - **Private** (only accessible within the class) — why?
   - **Public** (accessible from outside) — why?
   - **Protected** (accessible in subclasses) — why?
3. **For at least 3 private attributes, design getters/setters** that enforce business logic:

Example: `Payment class`

Attribute: amount (private)

- Why private? To prevent external code from arbitrarily changing payment amounts
- Getter: getAmount() → returns amount
- Setter: setAmount(double newAmount)
  - Business logic: Check if newAmount > 0 and ≤ max transaction limit
  - If invalid, throw exception or log error
  - If valid, update amount

4. **Write pseudo-code or actual code** for at least 2 setters with validation logic:

```
// Example in Java
private double amount;
private static final double MAX_AMOUNT = 100000.0;

public void setAmount(double newAmount) {
if (newAmount <= 0) {
throw new IllegalArgumentException("Amount must be positive");
}
if (newAmount > MAX_AMOUNT) {
throw new IllegalArgumentException("Amount exceeds maximum limit");
}
this.amount = newAmount;
}
```

```
public double getAmount() {
return this.amount;
}
```

5. **Explain in 3–4 sentences** how encapsulation protects this class's data integrity and prevents misuse.

Deliverable:

- Table mapping attributes to access levels (private/public/protected) with justification
- 2–3 setter methods with validation logic (pseudo-code or real code)
- Brief explanation of data protection benefits

Part C: Inheritance & Polymorphism

Task 3: Design an Inheritance Hierarchy (Self-Study: 1.5–2 hours)

Objective: Recognize common superclasses and create reusable base classes.

Instructions:

1. **Identify a base class** from your domain that multiple entities share traits:
   - Example: `User` as a superclass for `Passenger` and `Driver`
   - Or: `Vehicle` as a superclass for `Car`, `Motorcycle`, `Truck`
2. **Design the hierarchy:**

   Base Class: `User`

   - Attributes: `userId` (String), `name` (String), `email` (String), `phone` (String), `joinDate` (String), `rating` (double)
   - Methods: `getUserInfo()`, `updateProfile()`, `getContactInfo()`
3. **Subclass 1: `Passenger (extends User)`**
   - Extra attributes: `homeAddress` (String), `paymentMethods` (List), `rideHistory` (List)
   - Extra methods: `requestRide()`, `rateDriver()`, `viewRideHistory()`, `addPaymentMethod()`
4. **Subclass 2: `Driver (extends User)`**
   - Extra attributes: `carLicense` (String), `carDetails` (Car), `isAvailable` (boolean), `earnings` (double)
   - Extra methods: `acceptRide()`, `completeRide()`, `viewEarnings()`, `updateAvailability()`

5. **Create a simple diagram** showing:
   - `User` (parent)
   - Arrows pointing down to `Passenger` and `Driver` (children)
   - Labels showing inherited vs. new attributes/methods

Deliverable:

- Inheritance hierarchy diagram (text-based or sketched)
- List of inherited members (from `User`) for each subclass
- List of new/overridden members for each subclass

Task 4: Implement Polymorphism (Self-Study: 1–1.5 hours)

Objective: Demonstrate method overriding and interface-based polymorphism.

Instructions:

1. **Method Overriding Example:**

   In `User` class, define a method:
   public String getProfileInfo() {
   return "User: " + name + ", Email: " + email;
   }

   In `Passenger` subclass, override it:
   @Override
   public String getProfileInfo() {
   return "Passenger: " + name + ", Home: " + homeAddress + ", Rides: " + rideHistory.size();
   }

   In `Driver` subclass, override it:
   @Override
   public String getProfileInfo() {
   return "Driver: " + name + ", License: " + carLicense + ", Available: " + isAvailable;
   }

2. **Polymorphic Behavior Example:**

   Write pseudo-code showing how the same method call produces different results:

   // Polymorphism in action
   List<User> users = new ArrayList<>();
   users.add(new Passenger("Alice", "alice@mail.com"));
   users.add(new Driver("Bob", "bob@mail.com"));

   for (User user : users) {
   System.out.println(user.getProfileInfo());

// Alice's info printed as Passenger
// Bob's info printed as Driver
}

3. **Interface Example** (optional, advanced):

   Define a `Rateable` interface:
   interface Rateable {
   void submitRating(double rating, String comment);
   double getAverageRating();
   }

   Both `Passenger` and `Driver` implement `Rateable` (each with their own logic for handling ratings).

4. **Explain in 3–5 sentences:**

   - What is polymorphism?
   - How does method overriding enable it?
   - Why is polymorphism useful in your domain? (e.g., "We can call `getProfileInfo()` on any `User` without knowing if it's a Passenger or Driver")

Deliverable:

- Code examples (pseudo-code or real code) showing:
  - Method overriding in at least 2 subclasses
  - Polymorphic loop/list processing
  - (Optional) Interface implementation
- 3–5 sentence explanation of polymorphism benefits

Part D: Reflection & Analysis

Task 5: Design Decisions & Trade-offs (Online: 0.5–1 hour)

Objective: Develop strategic thinking about OOP design.

Instructions:

Answer **all three questions** (200–300 words total):

1. **Inheritance vs. Composition:**

   - Why did you choose to make `Passenger` and `Driver` inherit from `User` rather than compose them with a `User` object?
   - What would be the trade-offs of the alternative approach?

2. **Encapsulation Choices:**
    - Why did you choose to make certain attributes private instead of public?
    - What problems would arise if all attributes were public?
3. **Future Scalability:**
    - If your app grew to include `Admin`, `Support`, and `Partner` user types, how would your class hierarchy adapt?
    - Would you add more subclasses to `User`, or introduce interfaces? Why?

Deliverable: Answers to all 3 questions, 200–300 words total.

Evaluation Criteria

Criteria	Excellent (9–10)	Good (7–8)	Acceptable (5–6)	Needs Improvement (<5)
Domain Modeling	4–5 well-designed classes with clear roles; attributes & methods appropriate	3–4 classes; mostly appropriate design	2–3 classes; some design issues	Fewer than 2 classes; unclear purpose
Encapsulation	Thoughtful access control; validation logic prevents misuse	Correct use of private/public; basic validation	Access levels present; minimal validation	Inconsistent or missing access control
Inheritance Design	Clear hierarchy; shared traits properly elevated to base class	Reasonable hierarchy; some redundancy	Basic hierarchy; some confusion	Poorly designed or missing inheritance
Polymorphism	Method overriding clear & practical; polymorphic code demonstrated	Overriding present; explanation adequate	Overriding shown but underexplained	Minimal or missing polymorphism

Code Quality	Clean, readable, follows naming conventions	Generally clean; minor style issues	Readable but inconsistent style	Difficult to read or unclear
Strategic Thinking	Thoughtful reflection on design trade-offs; considers scalability	Good reflection; considers alternatives	Basic reflection; limited depth	Minimal or missing reflection

---

Deliverable Checklist

Submit as **PDF or DOCX (2–3 pages)** or **code file + documentation** containing:

- [ ] **Task 1:** Class design table/diagram (4–5 entities with attributes & methods)
- [ ] **Task 2:** Encapsulation table (access levels + justification) + 2–3 setter methods with validation
- [ ] **Task 3:** Inheritance hierarchy diagram + inherited vs. new members list
- [ ] **Task 4:** Polymorphism code examples (method overriding + polymorphic loop) + explanation
- [ ] **Task 5:** Strategic reflection (3 questions answered, 200–300 words total)
- [ ] **Code submission** (optional): Full source code for at least 3 classes (Java, Kotlin, C#, C++, or Python)

Resources & References

OOP Concepts:

- Gamma, Helm, Johnson, Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* (Chapters 1–2)
- Robert C. Martin. *Clean Code* (Chapter 6: Objects & Data Structures)
- Oracle Java Tutorials: Inheritance, Polymorphism

Language-Specific Guides:

- **Java:** Effective Java (Joshua Bloch)
- **Kotlin:** Kotlin in Action (Dmitry Jemerov & Svetlana Isakova)
- **C#:** C# Player's Guide (RB Whitaker)

Visualizing OOP:

- UML Class Diagrams: https://en.wikipedia.org/wiki/Class_diagram
- Tools: Draw.io, Lucidchart, Visual Studio (built-in UML tools), Miro

---

Timeline Guidance

Self-Study (6–8 hours):

- Hours 1–3: Task 1 (domain modeling & class design)
- Hours 3–5: Task 2 (encapsulation & validation)
- Hours 5–7: Task 3 (inheritance hierarchy design)
- Hours 7–8: Task 4 (polymorphism examples & code)

Online Sessions (1–2 hours):

- 0.5 hours: Task 1–2 discussion (design decisions, encapsulation patterns)
- 0.5 hours: Task 3–4 discussion (inheritance depth, polymorphic patterns in practice)
- 0.5 hours: Task 5 reflection & peer review
- 0.5 hour: Q&A and advanced OOP patterns (factory pattern, singleton, etc.)

Extension Tasks (Optional, Advanced)

1. **Implement an Abstract Class:**
    - Create an abstract `Transaction` class with abstract method `process()`
    - Implement concrete subclasses: `PaymentTransaction`, `RefundTransaction`, `BalanceTransfer`
2. **Interface Segregation:**
    - Design multiple interfaces (e.g., `Trackable`, `Rateable`, `Notifiable`) and have classes implement appropriate ones
3. **Dependency Injection:**
    - Refactor your classes to accept dependencies in constructors (e.g., `Ride` receives a `PaymentService` instance)
4. **Unit Testing:**
    - Write unit tests for your encapsulation logic (validate that setters reject invalid data correctly)
5. **Design Pattern Implementation:**
    - Apply a design pattern (Factory, Builder, Observer) to your domain

---

Notes for Instructors

This exercise bridges **theory (OOP principles) and practice (real-world application domains)**. Students develop:

- Object-oriented thinking and domain modeling skills
- Understanding of access control and data integrity
- Recognition of inheritance patterns and code reuse opportunities
- Practical polymorphism through method overriding
- Strategic design thinking

Discussion prompts for online session:

- "What shared traits did you identify that justified inheritance?"
- "Where does encapsulation protect your app from bugs or misuse?"
- "How would adding a new `Admin` user type change your class hierarchy?"
- "Can you show an example where polymorphism simplified your code?"
- "What's a design decision you'd change if you started over?"

Common pitfalls to address:

- Over-inheritance: making every variation a subclass rather than using composition or flags
- Weak encapsulation: public attributes that should be private
- Missing validation: setters that don't check constraints
- Unused polymorphism: hierarchies that don't exploit method overriding

---

Assessment Notes

Grading rubric integration:

- Emphasizes *design reasoning* over syntactic correctness
- Values clear documentation and explanation of choices
- Recognizes that multiple valid designs exist
- Encourages reflection on scalability and maintainability

Partial credit guidance:

- Code quality: 30% (correctness, readability, naming)
- OOP principles: 50% (encapsulation, inheritance, polymorphism)
- Strategic thinking: 20% (reflection, trade-off analysis)