Solid_Act 37-41

🧠 SOLID Principles – Activity Set

Fresh concepts, clearer instructions, and still open-ended to spark critical thinking.

---

37. S – Single Responsibility Principle (SRP)

📌 **Activity Title:** Student Grading System

🧠 **Scenario:**
You're building a system that stores student data, calculates averages, and prints a report. Right now, all of that is in **one class**.

📝 **Instructions:**

- Identify the responsibilities currently mixed in the class.
- Break it down into three:
  a. `Student` – holds name and grades
  b. `GradeCalculator` – computes average
  c. `GradeReportPrinter` – prints the result
- Make each class reusable on its own.

---

38. O – Open/Closed Principle (OCP)

📌 **Activity Title:** Notification Alert System

🧠 **Scenario:**
You're sending notifications for system events (e.g., success, error, warning). Right now, it uses a huge if-else block to decide what to display.

📝 **Instructions:**

- Replace the if-else block with a polymorphic solution
- Create a base class/interface: `Notification`
- Implement classes like `SuccessNotification`, `ErrorNotification`, `WarningNotification`
- The system should work with any new notification type **without editing the core logic**

39. L – Liskov Substitution Principle (LSP)

📌 **Activity Title:** Animal Feeding System

🧠 **Scenario:**
You have an `Animal` class with a `feed()` method. `Dog` and `Cat` work well. Then someone adds a `RobotPet` that doesn't eat — and it crashes the app when `feed()` is called.

📝 **Instructions:**

- Explain why `RobotPet` violates LSP
- Refactor the design: separate `EatableAnimal` from `MechanicalPet`
- Make sure only `EatableAnimal` subclasses have the `feed()` method

---

40. I – Interface Segregation Principle (ISP)

📌 **Activity Title:** Social Media Posting App

🧠 **Scenario:**
You created an interface with: `post_text()`, `post_image()`, `post_video()`. Now you're adding a service that only supports **text** — and you're forced to implement methods it doesn't need.

📝 **Instructions:**

- Identify the unnecessary method implementations
- Break the interface into:
  - `TextPoster`
  - `ImagePoster`
  - `VideoPoster`
- Let each class implement only the ones it supports
- Add a new platform (e.g., `TwitterPoster`) using your updated interfaces

---

41. D – Dependency Inversion Principle (DIP)

📌 **Activity Title:** Payment Gateway Integration

🧠 **Scenario:**
Your checkout system is directly connected to one payment method (`PayPal`). Now you need to support more like `GCash` and `Stripe`, but you're forced to change the core checkout class every time.

📝 **Instructions:**

- Spot the issue: Why is the checkout class tightly coupled to one service?
- Create a `PaymentProcessor` interface with a `process_payment()` method
- Implement separate classes for `PayPalProcessor`, `GCashProcessor`, `StripeProcessor`
- Refactor the checkout to accept any processor via constructor