**Assessment Questions** :

**Question 1: Extensibility in Multi-Type Content**

**Correct Answer:** **C**

The primary issue is that the current implementation uses conditional type checks (if (type == 'text') , else if (type == 'image')) to decide what widget to build. This makes the class not extensible because every time a new content type (e.g., video) is added, we must go back and modify the existing ContentItem class. This violates the Open-Closed Principle (OCP), which says classes should be open for extension but closed for modification.

**The fix :**

* Define an abstract base class ContentItem with an abstract build() method.
* Create subclasses like TextItem, ImageItem, and later VideoItem that each override build() with their specific logic.
* This way, adding a new content type requires only creating a new subclass, not modifying existing code.

**Why not the others?**

**A - Encapsulation violation**

* While it’s true that the fields type and data are public, this is not the *main* issue for extensibility.
* Even if you make them private with getters/setters, you’ll still need to edit the if , else logic when adding new types, so the extensibility problem remains.

**B - Single Responsibility Principle (SRP)**

* It’s somewhat true: ContentItem mixes **data** (type, data) and **UI rendering** (build).
* But the bigger issue in this context is not SRP it’s that adding new content types requires modifying the class, which directly violates **OCP**.
* Splitting classes would improve clarity but wouldn’t directly solve the extensibility problem.

**C - Open-Closed Principle (Correct)**

* This is the real issue: using conditionals for type checking makes the code **closed to extension**.
* The proper solution is polymorphism with an abstract class and subclasses, which makes it easy to add VideoItem, AudioItem, etc., without touching existing code.

**D - Liskov Substitution Principle (LSP)**

* LSP means subclasses should behave consistently with the base class.
* In this case, there’s no subclassing yet, only conditional checks.
* The problem isn’t about substitutability it’s about **how new types are added**. So LSP doesn’t apply here

**Question 2: User Model with Firestore**

**Correct Answer: C**

The main issues are:

1. Encapsulation violation => The fields name, age, email are public and can be directly modified from outside, which risks invalid states (like negative ages, empty emails, etc.).
2. Single Responsibility Principle violation => The UserModel class is handling two responsibilities: managing user data and saving it to Firestore.

**Fix:**

* Make fields private and expose them through validated getters and setters to preserve data integrity.
* Extract persistence logic (saveToFirestore()) into a separate service class (e.g., FirestoreService) so the model only manages data, not database operations.

**Why not the others?**

**A - Dependency Inversion Principle (DIP)**

* It’s true that saveToFirestore() tightly couples the class to Firestore, but the root issue here is mixing persistence inside the model itself.
* The correct first step is to extract persistence into a service — then if needed, apply DIP by making it interface-driven.

**B - Interface Segregation Principle (ISP)**

* ISP deals with interfaces that are too broad and force clients to depend on methods they don’t need.
* Here, we don’t even have an interface, just one class — so ISP isn’t the real violation.

**C - Encapsulation + SRP (Correct)**

* Fields are exposed without validation (Encapsulation issue).
* The class mixes model and persistence (SRP issue).
* This is the most direct and accurate answer.

**D - Open-Closed Principle (OCP)**

* Adding new fields requiring modification of updateUser() is not the central issue here.
* The main architectural flaws are Encapsulation and SRP, not extensibility.

**Question 3: Widget Safety in Navigation**

**Correct Answer: B**

The main issue is that SettingsScreen extends Screen but breaks the contract:

* All subclasses of Screen are expected to safely override navigate() in a consistent way.
* However, SettingsScreen.navigate() throws an exception, meaning it cannot be safely substituted for its base type.
* This is a direct violation of the Liskov Substitution Principle (LSP).

**Fix:**

* Introduce a Navigable interface and implement it only in screens that support navigation.
* Alternatively, create separate hierarchies: one for navigable screens and one for non-navigable screens.  
  This ensures behavioral consistency and prevents runtime errors.

**Why not the others?**

**A — Missing abstraction layer**

* The code already has an abstraction (Screen as a base class).
* The real problem is not a missing abstraction but that the subclass (SettingsScreen) violates the expected behavior.

**B — LSP violation (Correct)**

* This is the correct reasoning: subclasses must be replaceable for their base type without breaking client code.
* SettingsScreen breaks this, so LSP is violated.

**C — Polymorphism issue**

* While polymorphism is indeed misused, calling this just a “polymorphism issue” is too vague.
* The precise principle being broken here is LSP, not polymorphism in general.

**D — Single Responsibility Principle (SRP) violation in NavigationButton**

* NavigationButton is not really doing too much; it simply builds a button and triggers navigation.
* The real issue is with the screen hierarchy, not with the button itself.

**Question 4: Widget Controller Design**

### Correct Answer: C

The problem is that WidgetController **forces all implementers to handle methods they may not need**.

* Example: SimpleButtonController must implement handleAnimation() and handleNetwork(), even though a simple button doesn’t need them.
* This leads to unnecessary UnimplementedErrors and bloated, fragile code.

### Fix:

* Split WidgetController into smaller, role-specific interfaces:
  + LifecycleController (with initState() and dispose())
  + AnimationController (with handleAnimation())
  + NetworkController (with handleNetwork())
* Then, each controller class can implement only the interfaces it actually needs.

This adheres to the **Interface Segregation Principle**: “Clients should not be forced to depend on methods they do not use.”

### Why not the others?

### A — Open-Closed Principle (OCP)

* OCP is about being open for extension but closed for modification.
* The problem here isn’t about extending functionality—it’s about being forced to implement irrelevant methods.

### B — Encapsulation

* Encapsulation is about hiding internal state/behavior.
* There’s no exposure of internal state here, the issue is with oversized interfaces, not data hiding.

### C — Interface Segregation Principle (ISP)

* Correct. The interface is too broad, violating ISP.

### D — Dependency Inversion Principle (DIP)

* DIP is about high-level modules depending on abstractions instead of low-level details.
* This code doesn’t show any dependency injection or direct low-level coupling. So DIP isn’t the issue here.

### Question 5: Notification Service Design

### Correct Answer: C

**Problem:**

* AppNotifier **directly instantiates** LocalNotificationService (final LocalNotificationService service = LocalNotificationService();).
* This creates **tight coupling** between AppNotifier and one specific implementation.
* If later we want to send notifications via Firebase, Email, or Push, we’d have to modify AppNotifier. That breaks flexibility and testability.

### Fix:

* Define an **abstract interface** (e.g., NotificationService with a send(String message) method).
* Let LocalNotificationService, FirebaseNotificationService, etc., implement this interface.
* Inject the dependency into AppNotifier (constructor injection or via DI framework).
* This way, AppNotifier depends on **abstractions, not concretes**.

### Why not the others?

### A — Single Responsibility Principle (SRP)

* AppNotifier is only handling one responsibility: notifying users.
* Instantiating the service is not considered a second responsibility here—it’s more about dependency management.

### B — Liskov Substitution Principle (LSP)

* LSP issues appear when subclasses can’t replace their parent without breaking behavior.
* We don’t have inheritance or subclass misuse here, so LSP isn’t violated.

### C — Dependency Inversion Principle (DIP)

* Correct. The direct dependency on LocalNotificationService breaks DIP.

### D — Missing polymorphism

### True, there’s no polymorphism, but that’s not the core issue. The real architectural smell is tight coupling (DIP violation), not simply lack of inheritance.

### Question 6: User Story - Smart Ahwa Manager App

* Smart Ahwa Manager App :=> [Link](https://github.com/romisaa5/Smart-Ahwa-Manager-App)

**Post linkedIn :** => [Link](https://www.linkedin.com/posts/romisaafadel_solid-activity-7373336469844799489-xP5n?utm_source=share&utm_medium=member_desktop&rcm=ACoAAE6MjYsBsZQ8KxEBwldwSJ1MaD_ch9Xqoxs)