TDDC73
Lecture 4

Outline

- Building components/api
- Mini-project
- Testing
- Grade 5

Advanced API Design & Component Architecture

Implementing Reusable Widgets for Modern UI SDKs

The SDK Mindset

Building for Developers

Abstraction

Hiding complexity behind clean interfaces.

Flexibility

The user (developer) needs to control content and style.

Safety

Preventing invalid states through type safety and constraints.

Principle 1: Clarity & Affordance

Your API is the “User Interface” for the developer. It should be intuitive and self-documenting.

Bad API

```
new Button("Click Me",
  true,
  0xFF0000
)
```

Good API

```
Button(label: "Click Me",
  enabled: true,
  color: Colors.Red
)
```

Naming: Use standard verbs/nouns. `onClick` is better than `doAction`. `isVisible` is better than `show` (boolean vs verb).

Defaults: Use 'Convention over Configuration'. A button should look like a button without 20 parameters.

Predictability: If a prop is called `color`, it should change the background, not the text.

Principle 2: Composition > Inheritance

The "Has-A" Relationship

In modern frameworks (React, Flutter, Compose), we avoid inheriting from a BaseButton. Instead, we compose smaller primitives.

- **Flexibility:** A Card isn't a special type of View; it's a Container that *has* a shadow and a border radius.
- **The Slot Pattern:** Don't limit your component. A Button shouldn't just take a String label. It should take a Child widget. This allows users to put Icons, Rows, or Images inside your button.

Composition Over Inheritance

A shift in perspective for building scalable, maintainable UI systems.

The Core Paradigm Shift

Inheritance ("Is-a")

Models relationships as a strict hierarchy. A specialized class **is a** type of a general class.

```
class RedButton extends Button {
    // Inherits all Button behavior
}
```

Composition ("Has-a")

Models relationships by combining smaller, independent parts. A class **has a** set of behaviors or components.

```
Button({ icon, label }) {
    return <Box>{icon} {label}Box>;
}
```

> **The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.**
>
> — Joe Armstrong, Creator of Erlang

The Composition Solution

Think of Composition like **LEGO blocks**.

You don't extend a "Brick" to make a "Wall". You *compose* multiple Bricks together.

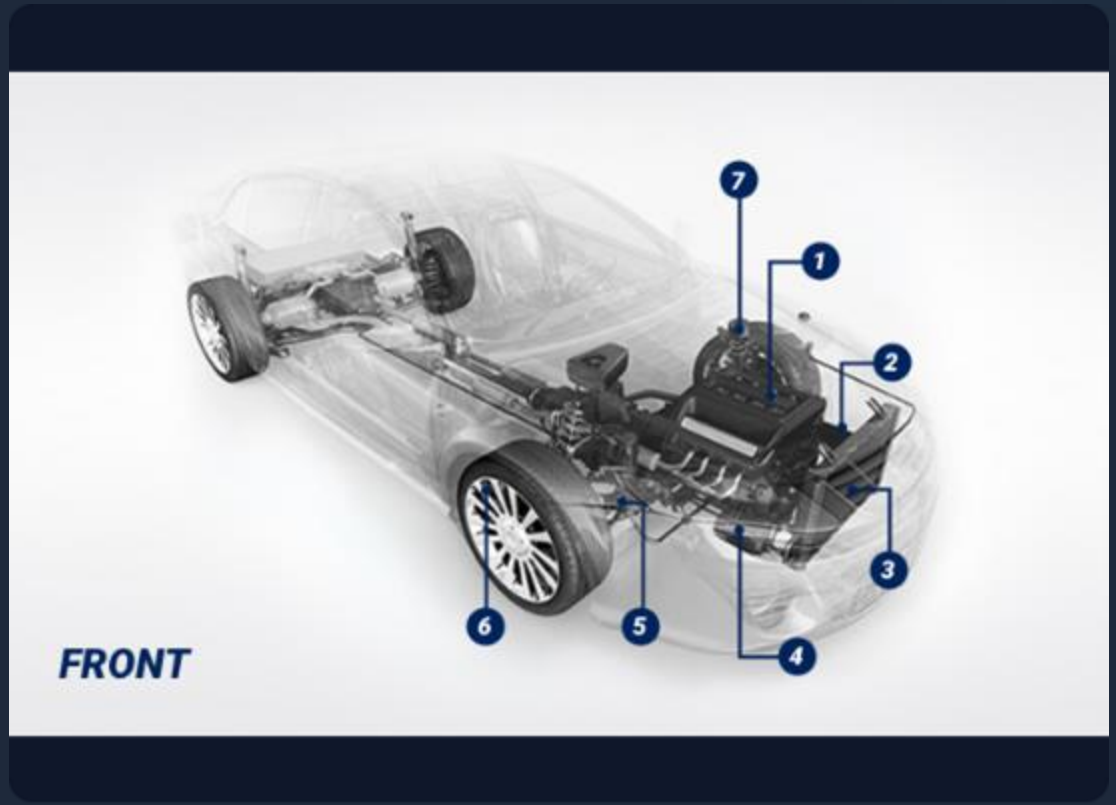
This allows you to build complex structures from simple, interchangeable, and reusable atomic units without worrying about a strict ancestry.

Analogy: The Car

How do you build a car?

- ✗ **Inheritance:** Car extends Vehicle. SportCar extends Car. RedSportCar extends SportCar.
- ✓ **Composition:** A Car **has** an Engine. A Car **has** Wheels. A Car **has** a PaintJob.

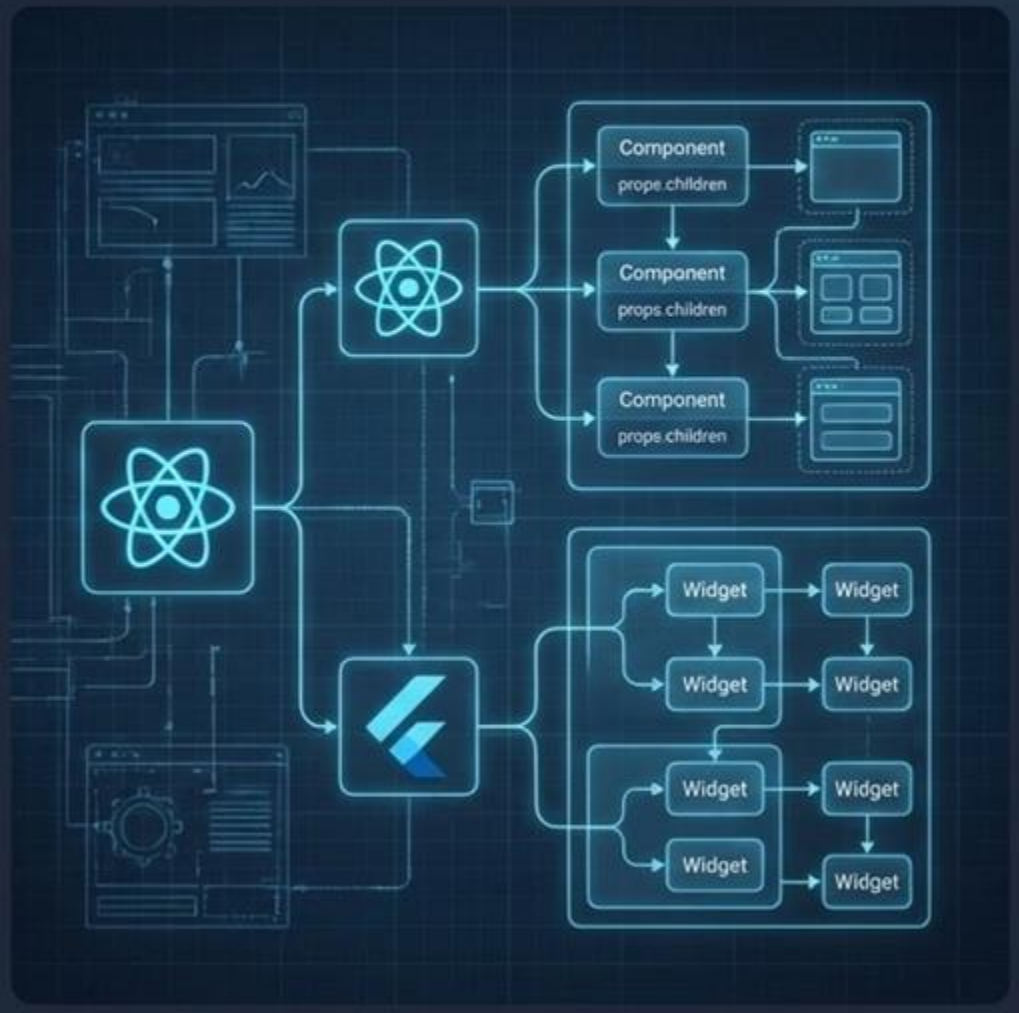
If you want to change the engine, you just swap the engine component. You don't need to redefine the entire taxonomy of the vehicle.



Adopted by Modern Frameworks

Modern libraries like **React**, **Vue**, and **Flutter** are built entirely on the principle of composition.

- **React:** "Components" compose other components via props.children.
- **Flutter:** "Widget composition" is the core architecture. Everything is a widget inside a widget.



Why It Wins for UI

Flexibility

Combine behaviors dynamically at runtime. You aren't locked into a structure defined at compile-time.

Reusability

Small, focused components (like a specialized Avatar) can be reused anywhere, inside any container.

Maintainability

Testing small, independent components is easier than testing a monolithic class with a massive inheritance chain.

Best Practices

- **Start Flattened:** Avoid extending classes unless absolutely necessary. Default to creating a new component that uses others.
- **Use Hooks/Mixins:** Share logic (state, effects) through composition primitives (like React Hooks) rather than base classes.
- **Keep Components Atomic:** A component should do one thing well. If it grows too large, break it down.
- **Inversion of Control:** Pass children or render props into components so the parent decides *what* to render, while the child decides *how* to wrap it.

The Component Contract

Props (Input)

Data passed down from the parent. These should be **immutable**. Changes in props trigger a re-render.

label, isEnabled, style

Events (Output)

How the component communicates up. Use **callbacks** or lambdas. Never modify parent state directly.

onClick(), onValueChange(v)

Slots (Children)

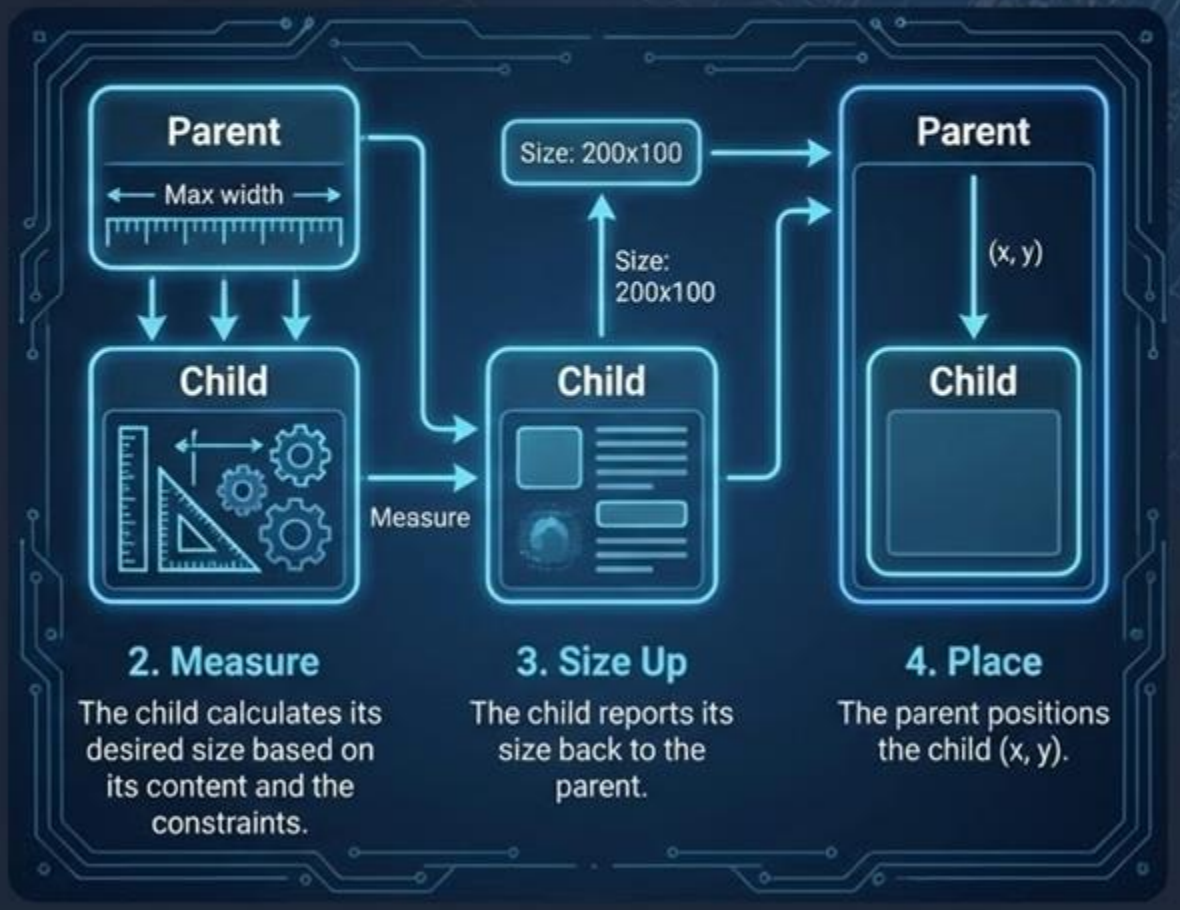
Content projection. The ability to wrap other components. This is critical for **generic** containers.

children, content, builder

Implementing Layouts: The Hard Part

For Grade 5, you must implement a layout (e.g., VStack). This requires understanding the **Layout Protocol**.

- **1. Constraints Down:** The parent tells the child: "You can be at most 300px wide."
- **2. Measure:** The child calculates its desired size based on its content and the constraints.
- **3. Size Up:** The child reports its size back to the parent.
- **4. Place:** The parent positions the child (x, y).

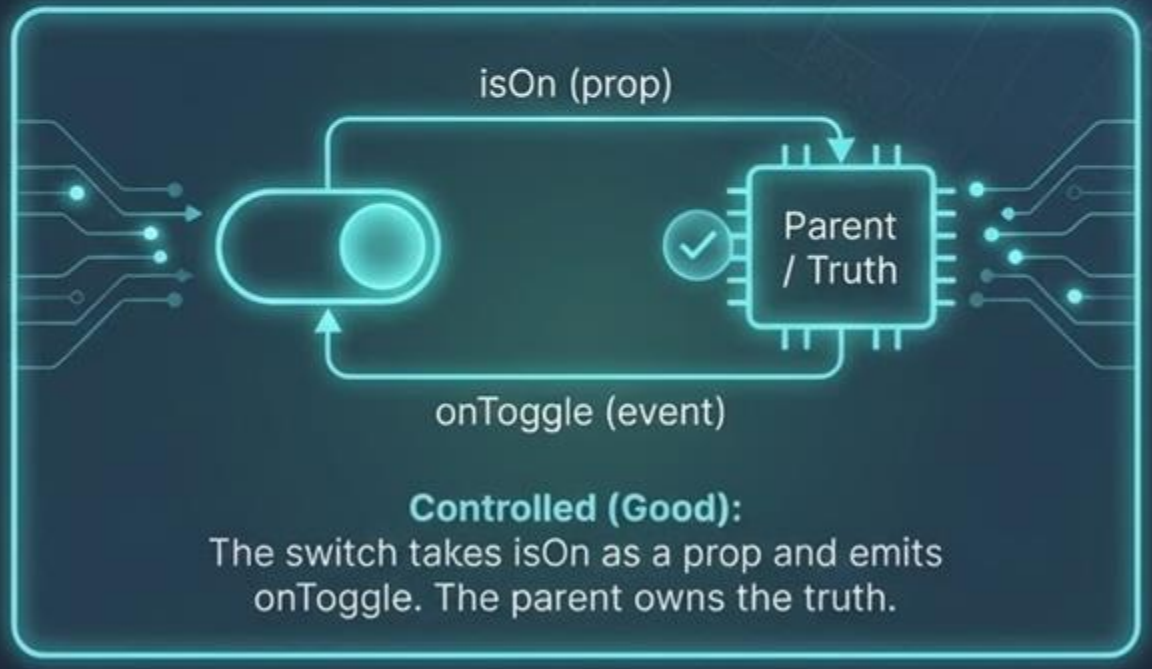


State: Controlled vs. Uncontrolled

State Hoisting

To make a component reusable, it should often be stateless (Controlled).

Uncontrolled (Bad for reuse): A toggle switch that manages its own isOn state. The parent can't force it to turn off.



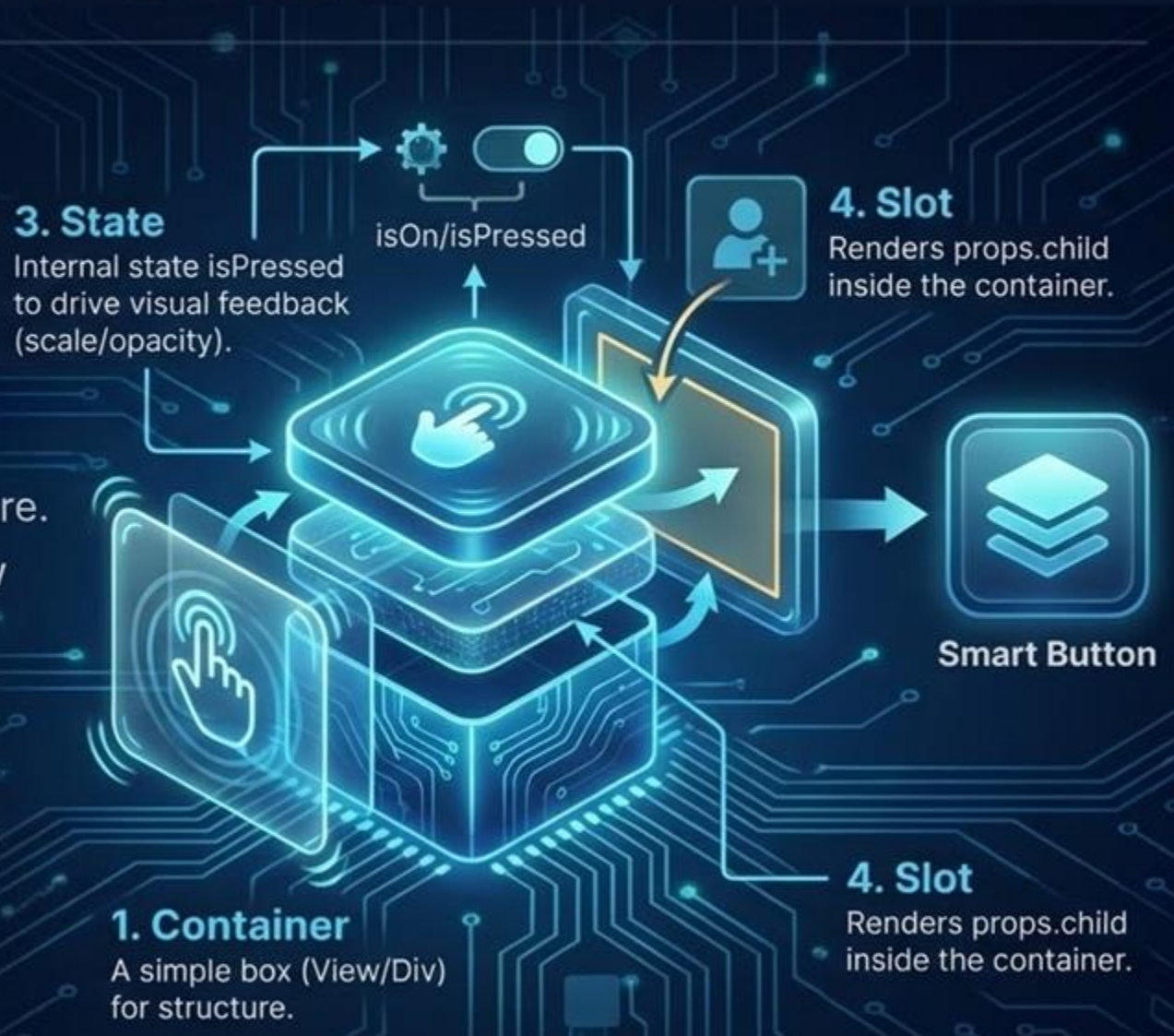
Controlled (Good):
The switch takes isOn as a prop and emits onToggle. The parent owns the truth.

Case Study: The "Smart" Button

Implementing the Logic

How do we build a button from scratch using low-level primitives?

1. **Container:** A simple box (View/Div) for structure.
2. **Gesture Detector:** Wraps the box to catch raw pointer events.
3. **State:** Internal state isPressed to drive visual feedback (scale/opacity).
4. **Slot:** Renders props.child inside the container.



Low-Level Drawing & Interaction

The Canvas

For custom shapes that aren't just rectangles, you use the Canvas API. This is "Immediate Mode" rendering.

drawCircle(), drawLine(), drawPath()

Gesture Detection

Raw input handling involves tracking the pointer lifecycle.

onDown -> onMove -> onUp/onCancel

Hit Testing

Determining which component receives the event. Usually handled by the framework's layout tree.

TDDC73 Mini Project

A Guide to Achieving Grade 3

Building Your Own Interaction SDK

What is the Project?

In the 2025/2026 curriculum,. It is no just about building an app; it is about architecture. You will create a **Reusable Mini SDK** and a corresponding **Example Application**.

Grade 3 Requirements

The Library

Create a standalone library containing at least **two** high-level interaction patterns.

The App

Build a functioning example application that imports and uses your library.

Functionality

The code must work, be bug-free, and clearly demonstrate the interaction patterns.

Architecture: Library vs. App

Separation of Concerns

The core challenge for Grade 3 is **architectural separation**.

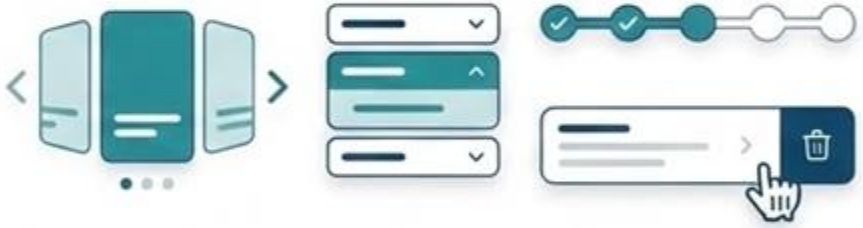
- **The Library:** Contains generic, reusable logic (e.g., "A Carousel"). It knows nothing about your specific data (e.g., "Sushi Menu").
- **The App:** Consumes the library and provides the specific context and data.

What is an ‘Interaction Pattern’?

Not Basic Widgets

A 'Button' or 'Input' is too simple. These are **low-level widgets** provided by the OS.

High-Level Patterns



Complex, stateful components that manage their own behavior. Think 'Carousel', 'Accordion', 'Wizard', or 'Swipe-to-Delete List'.

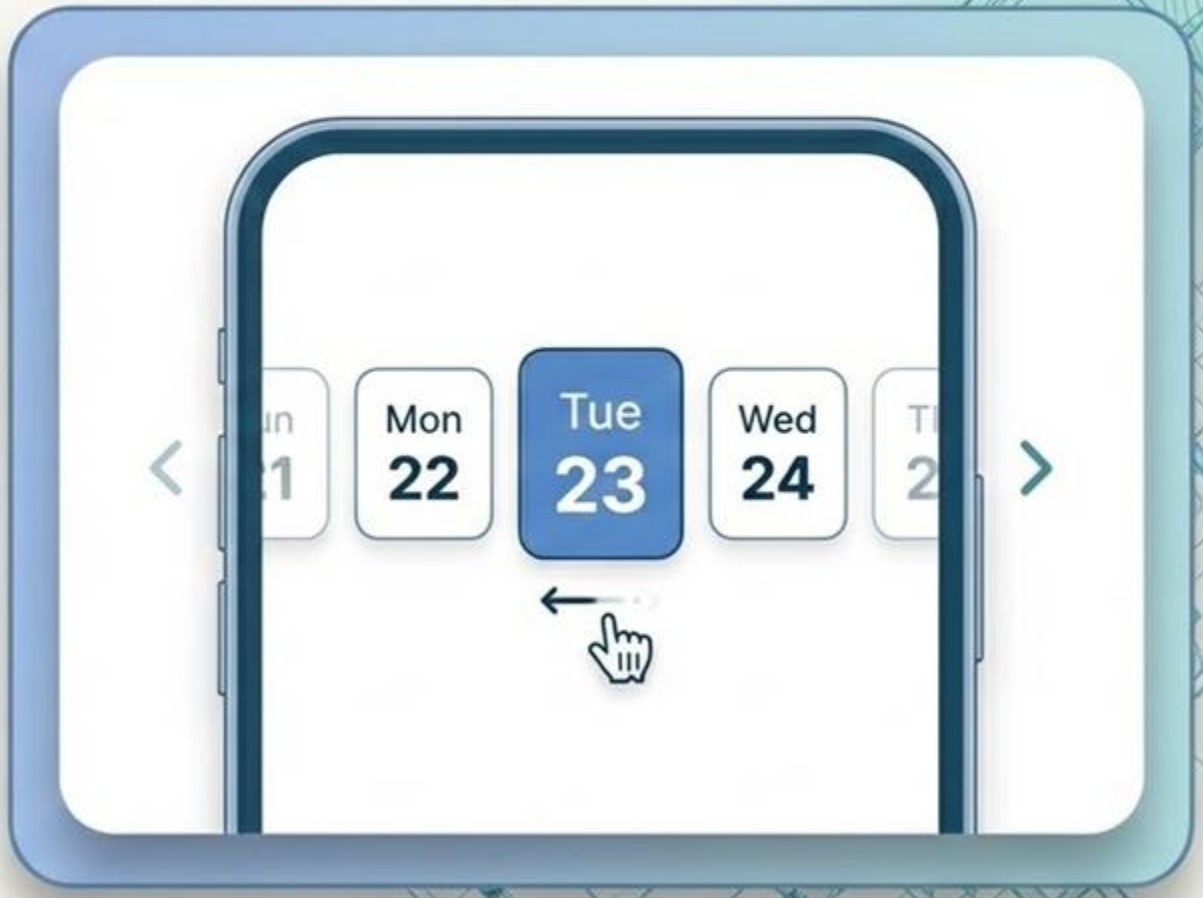
Example 1: The Carousel

A classic pattern

A classic pattern suitable for Grade 3.

- **Internal State:** Tracks the **currentIndex**.
- **Props:** Accepts an array of items (images/views).
- **Interaction:** User swipes or clicks arrows to change the visible item.

Key: The Carousel handles the transition logic; the App provides the images.



Example 2: The Accordion

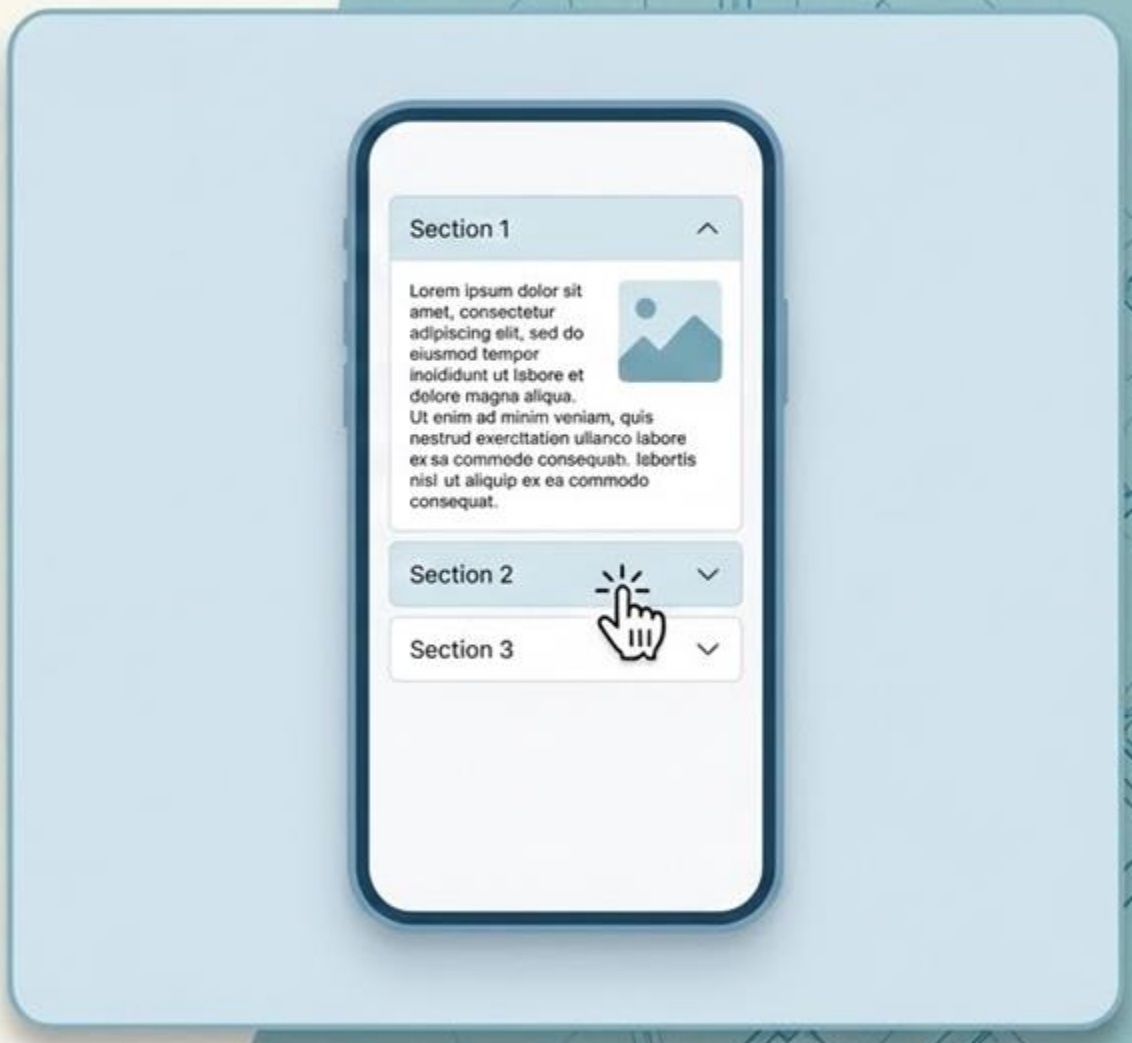
A classic pattern.

Behavior

- A list of headers where clicking one expands its content while optionally collapsing others.

Why it works for Grade 3

- It requires **managing state** (which ID is open?) and **animation** (expanding height), making it a perfect candidate for a **high-level pattern**.

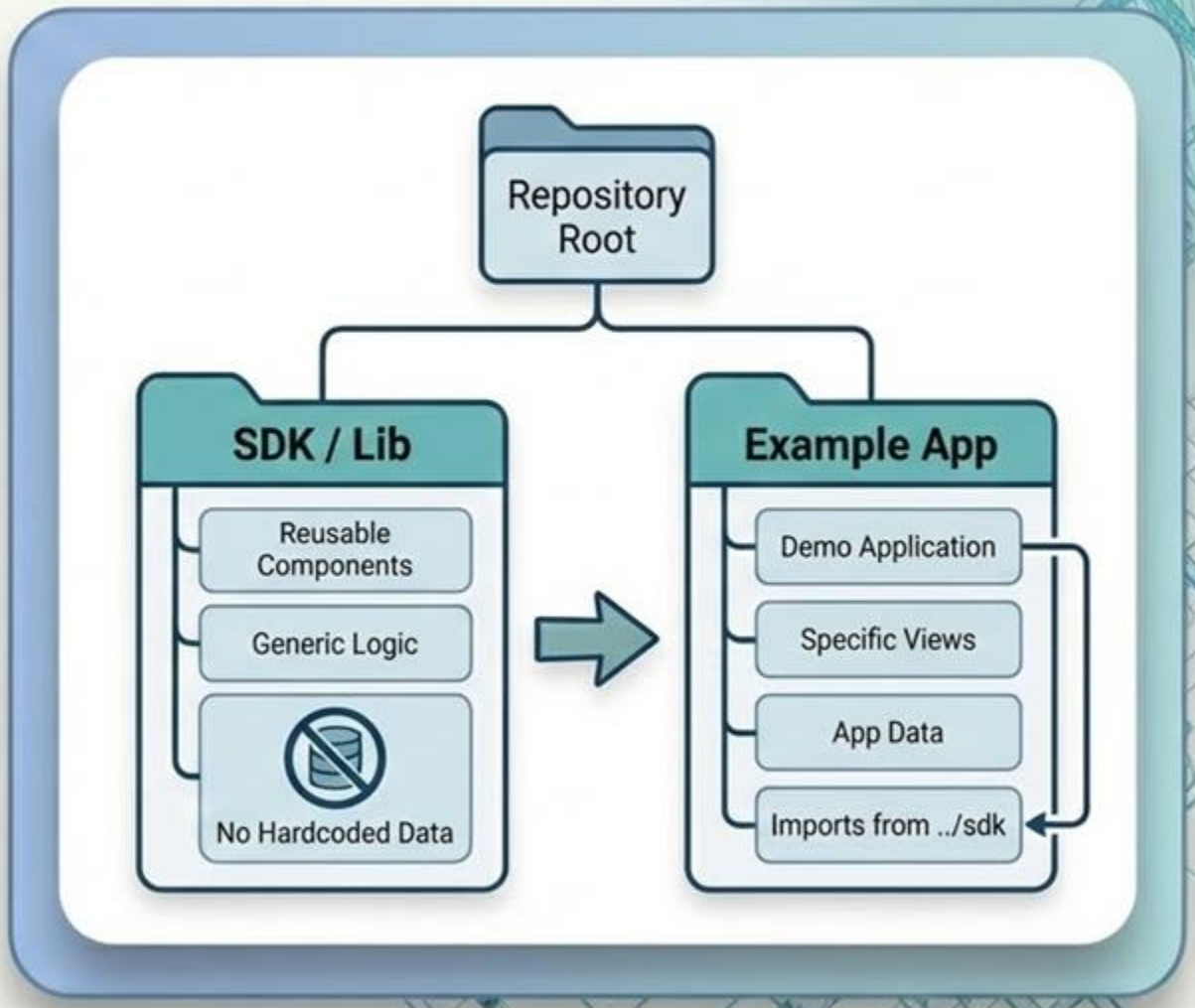


Project Structure

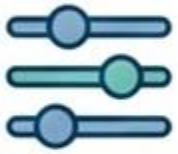
Organizing Your Code

Your repository should clearly distinguish between the SDK and the Example App. Do not mix them.

- **/sdk** or **/lib**: Your **reusable code** goes here. No hardcoded app data!
- **/example** or **/app**: The demo application that imports from ../sdk.



API Design: Props & Configuration



Flexibility

Don't hardcode colors or sizes. Allow the developer to pass them as **props**.

Data Agnostic

Your component should accept generic data. Use `**props.data**` or `**children**` instead of fetching specific URLs inside the library.

API Design: Events & Callbacks

Communicating Upwards

Your library components are "**dumb**" about the business logic. They notify the "**smart**" app when things happen.

Use callback functions (**events**) to signal user actions.

Example: Handling the Event


```
// Good Pattern: App handles the logic
<MyComponent onMove={(index) => {
  console.log("User moved to", index);
}} />
```

Recommended Workflow

1 Scaffold

Set up the folder structure with separate Library and App folders.

2 Draft

Build the component inside the App first to test functionality.

3 Extract

Move the logic to the Library folder. Replace hardcoded data with props.

4 Verify

Import it back into the App. Does it still work? Is it generic?

Documentation Matters

README.md: Even for Grade 3, you must explain how to use your library. Imagine you are handing this to another developer. Clear documentation is often the difference between a confusing project and a passing grade.

Installation

How to run the example.

Usage

Code snippets showing how to import and use your components.

Props List

Explain what configuration options are available.

Common Grade 3 Pitfalls

Tight Coupling

Tight Coupling

Mistake: Importing "App" files into your "Library". The Library must never depend on the App.

Hardcoded Data

Hardcoded Data

Mistake: The Carousel component fetches "sushi.json" internally. It should accept data via props.

Broken State

Broken State

Mistake: The component works once but fails if you navigate away and back. Test the lifecycle!

Submission Checklist

1

Clean Code
(Linted & Formatted)

2

Working Demo
(The Example App)

3

Repo Link
(GitLab/GitHub)

Ready to Build?

This project is your chance to practice real-world software architecture.

Focus on creating clean, reusable interfaces. A well-designed library is a powerful tool in any developer's portfolio.

Good luck!

Final Takeaways

- **Design APIs** that are declarative, predictable, and composable.
- **Separate Concerns:** Layouts handle position; Leaf nodes handle drawing.
- **Hoist State** to make components reusable and testable.
- **Test the Contract:** Your UI tests should verify that props correctly update the view and events are fired.

Quality Assurance:

UI Testing vs. Unit Testing

Strategies for Robust Interaction Programming

The Testing Spectrum

</>Unit Testing

Testing individual functions or classes in isolation. It verifies logic, algorithms, and data transformations.

"Does the calculateTotal() function return the right number?"

UI Testing

Testing the application as a black box through the user interface. It verifies workflows, rendering, and integration.

"Can the user log in and see the dashboard?"

Unit Testing: The Foundation

Characteristics

Fast:
Runs in milliseconds.

Isolated:
No database, no network, no screen.

Precise:
When it fails, you know exactly which line of code is broken.

UI Testing: The User Perspective

Characteristics

Integrated:
Tests the real app running on a device/simulator.

Slow:
Needs to boot the app, wait for animations, etc.

Broad:
Verifies that database, API, and UI all work together.

The Testing Pyramid

Structure your Strategy

Because UI tests are slow and expensive to maintain, they should be the tip of the pyramid, not the base.

10% UI/E2E Tests

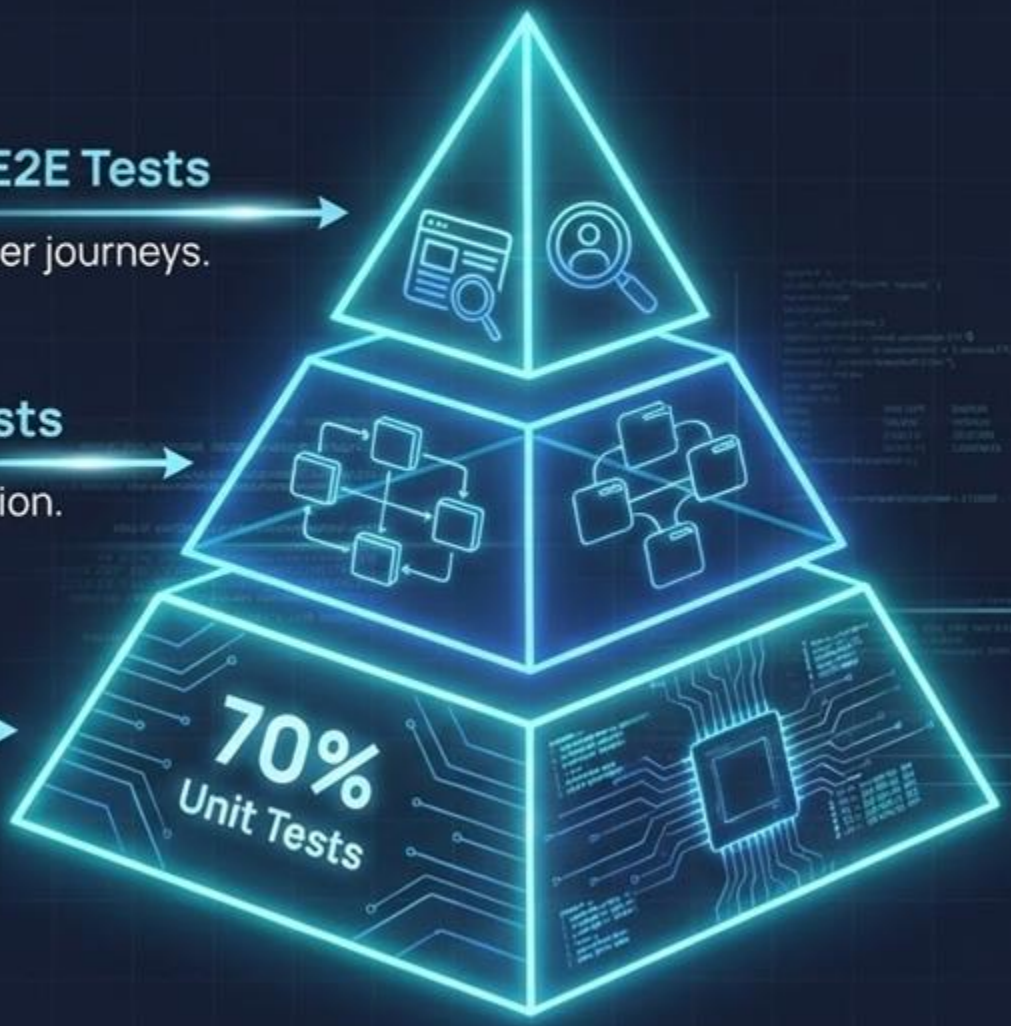
Critical user journeys.

20% Integration Tests

Component interaction.

70% Unit Tests

Fast feedback loop.

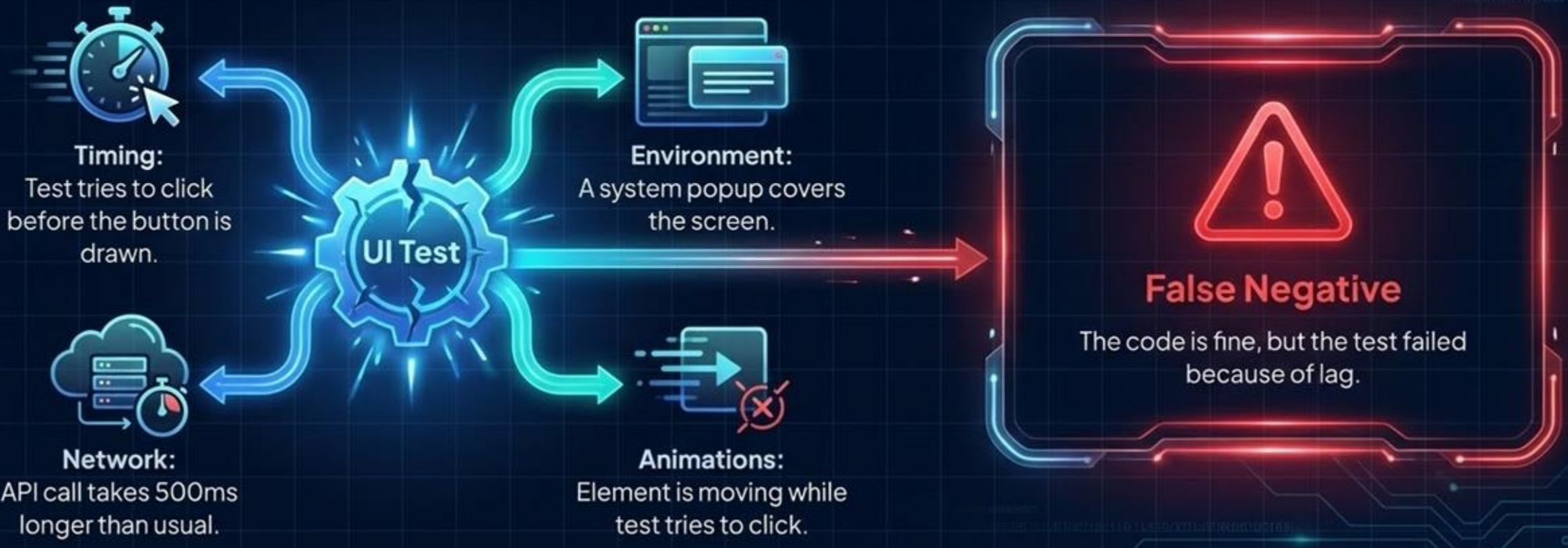


Deep Dive Comparison

Feature	Unit Testing	UI Testing
Scope	Single Function/Class	Full Application Flow
Speed	Milliseconds	Seconds to Minutes
Environment	Virtual / Node.js	Real Device / Browser
Fragility	Low (Stable)	High (Breaks on UI changes)
Cost	Cheap	Expensive

The Challenge: Flakiness

Why do UI Tests fail?



Anatomy of a UI Test

1. Find

Locate the element on the screen.

```
find.byText("Login")
```

2. Act

Simulate a user interaction.

```
tester.tap(button)
```

3. Assert

Verify the resulting state.

```
expect(home).toBeVisible()
```

Locators: Finding Elements

How you find elements determines how brittle your test is.

✕ Bad Locators

Relying on implementation details.

- xpath: /div[2]/div[4]/button
- css: .btn-primary-blue

Breaks if you change layout or styling.

✓ Good Locators

Relying on user semantics.

- getByText("Submit")
- getByRole("button", name: "Submit")
- testID: "submit-order-btn"

Resilient to refactoring.

The "Wait" Problem

Never Sleep

Using sleep(1000) is bad practice. It slows down tests and isn't reliable (what if it takes 1001ms?).

Use Polling/Await

Modern frameworks provide mechanisms to wait *until* a condition is met.

```
// Flutter await tester.pumpAndSettle(); // Playwright / JS
await expect(loc).toBeVisible(); // ^ Auto-retries for 5s
```

Synchronization

The test runner must synchronize with the UI thread to know when animations/requests are done.

Page Object Model (POM)

Design Pattern for Maintainability

- Don't scatter locators (selectors) across every test file.
- Create a class that represents a Page. The test interacts with the class methods, not the HTML/Widgets directly.
- **Benefit:** If the 'Login' button ID changes, you only fix it in one place (the Page Object), not in 50 tests.

Hermetic Testing

The Problem

If your UI test hits the real backend server, it will be slow and flaky (what if the server is down?).

The Solution: Mocking

Intercept network requests and return fake, predictable data.

```
intercept('GET', '/api/user', { body: { name: "Test User" } });
```

This makes UI tests **Deterministic**.

Summary Checklist

Unit Testing

- ✓ Test logic in isolation
- ✓ Mock everything else
- ✓ Run on every commit

UI Testing

- ✓ Test critical user journeys
- ✓ Use resilient locators
- ✓ Handle async/animations

Why UI Testing?

UI tests verify that the visual interface behaves as expected when a user interacts with it.

- **Catch Regressions:** Ensure new code doesn't break existing flows.
- **User Confidence:** Verify the critical paths (Login, Checkout) work.
- **Documentation:** Tests act as live documentation for feature behavior.

The Frameworks

Flutter

Uses flutter_test.

Tests run directly in the Dart VM.

Extremely fast and deterministic.

React Native

Uses RNTL & Jest.

Simulates the component tree.

Focuses on user-centric queries.

Jetpack Compose

Uses ui-test-junit4.

Interacts with the Semantics tree.

Runs on device or emulator.

Flutter Testing

Flutter tests interact with a virtual widget tree.

Finder: find.text(), find.byType()

Interaction: tester.tap(), tester.enterText()

Lifecycle: tester.pump() (advances one frame)

```dart
// The key concept is the "WidgetTester"
testWidgets('My Test', (tester) async {
  // 1. Build the widget
  await tester.pumpWidget(MyApp());

  // 2. Find widgets
  final titleFinder = find.text('Welcome');

  // 3. Verify
  expect(titleFinder, findsOneWidget);
});
```

Flutter: Interaction Example

```dart
testWidgets('Add item to list', (WidgetTester tester) async {
  // 1. Render UI
  await tester.pumpWidget(const TodoListApp());

  // 2. Simulate User Input
  await tester.enterText(find.byType(TextField), 'Buy Milk');
  await tester.tap(find.byIcon(Icons.add));

  // 3. Rebuild (Important in Flutter!)
  await tester.pump();

  // 4. Assertion
  expect(find.text('Buy Milk'), findsOneWidget);
});
```

Note the explicit await tester.pump() required to process the state change.

React Native Testing

Relies on **React Native Testing Library (RNTL)** running on Jest.

Render: Creates a virtual DOM for testing.

Screen: Global object to query elements.

FireEvent: Simulates press, changeText, scroll.

```
import { render, screen } from '@testing-
library/react-native';

test('renders correctly', () => {
  // 1. Render
  render(<WelcomeScreen />);

  // 2. Query & Assert
  const header = screen.getByText('Welcome');
  expect(header).toBeTruthy();
});
```

RNTL: Interaction Example

```
test('submits form data', () => {
  const onSubmit = jest.fn();
  const { getByText, getByPlaceholderText } = render(
    <Form onSubmit={onSubmit} />
  );

  // 1. Input Text
  fireEvent.changeText(getByPlaceholderText('Username'), 'JohnDoe');

  // 2. Press Button
  fireEvent.press(getByText('Submit'));

  // 3. Verify Function Call
  expect(onSubmit).toHaveBeenCalledWith({ username: 'JohnDoe' });
});
```

RNTL encourages testing inputs and outputs (user perspective) rather than internal state.

Jetpack Compose

Tests are strictly UI-based, interacting with the **Semantics Tree**.

Rule: createComposeRule() sets up the environment.

Finders: onNodeWithText, onNodeWithTag.

Actions: performClick, performTextInput.

```
@get:Rule
val rule = createComposeRule()

@Test
fun myTest() {
  // 1. Set Content
  rule.setContent { MyApp() }

  // 2. Assert
  rule.onNodeWithText("Welcome")
      .assertIsDisplayed()
}
```

Compose: Interaction Example

```
@Test
fun verifyCounterIncrement() {
    // 1. Load the UI
    rule.setContent { CounterScreen() }

    // 2. Verify Initial State
    rule.onNodeWithText("Count: 0").assertIsDisplayed()

    // 3. Perform Action
    rule.onNodeWithContentDescription("Increment Button")
        .performClick()

    // 4. Verify New State
    rule.onNodeWithText("Count: 1").assertIsDisplayed()
}
```

Compose tests sync automatically with the UI clock, reducing the need for manual waits.

Key Differences

Feature	Flutter	React Native	Jetpack Compose
Environment	Headless Dart VM	Node / Jest (JSDOM)	Android Device / Emulator
Sync/Async	Async (await pump)	Mostly Sync (some Async)	Sync (Auto-waiting)
Selection	Widget Finder	Queries (Text, Role)	Semantics Matchers
Primary Tool	flutter_test	@testing-library/react-native	ui-test-junit4

Best Practices

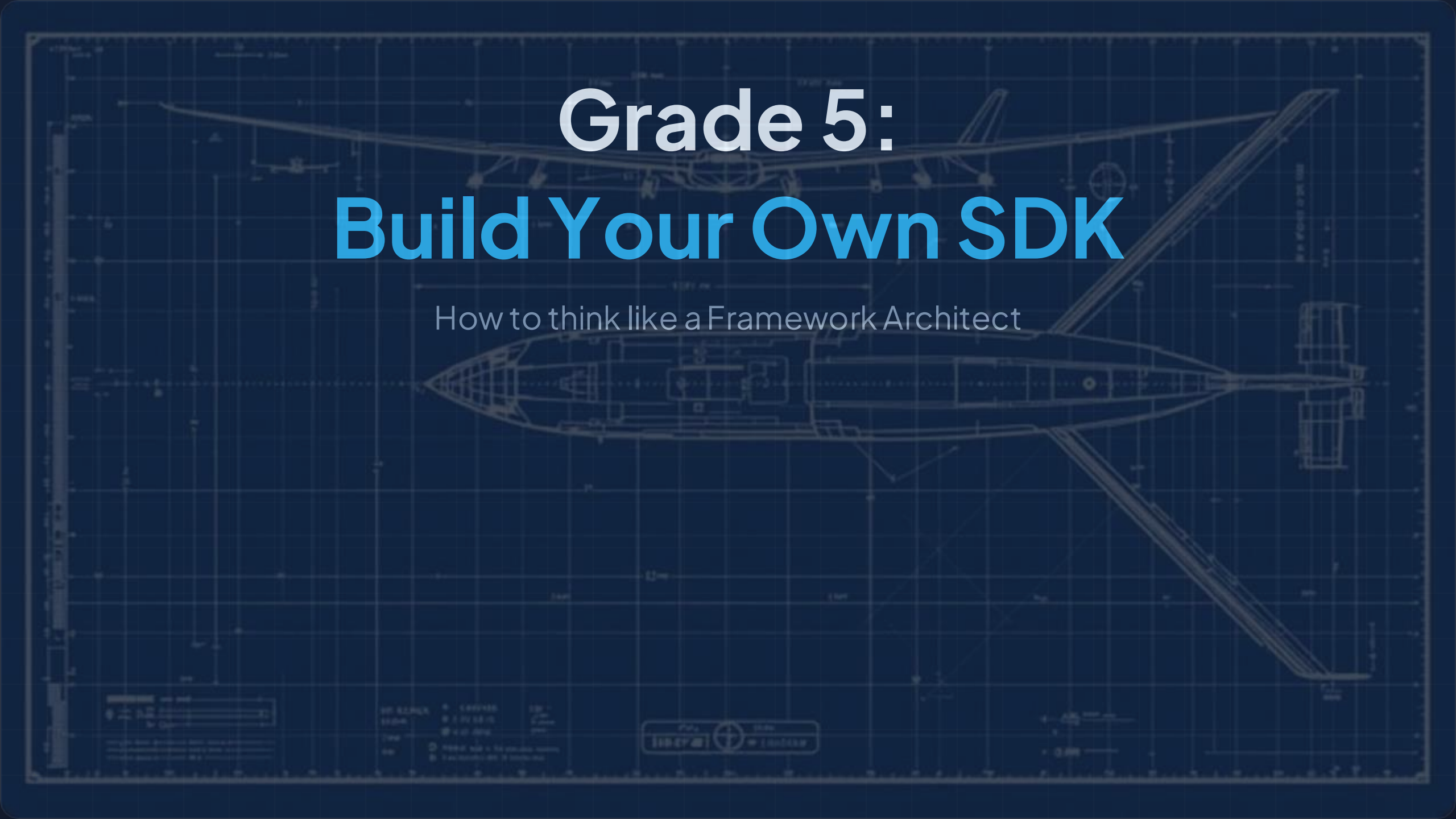
- **Test User Behavior:** Query by Text or Accessibility Label, not by internal implementation details.
- **Avoid Flakiness:** Don't use hardcoded sleep timers. Use the framework's synchronization (pump, findBy).
- **Page Object Model:** If your tests get complex, create "Robot" classes to abstract the interaction logic.

✅ **Good Finder**

onNodeWithText("Submit")

❌ **Bad Finder**

childAt(3).childAt(0)



Grade 5:
Build Your Own SDK

How to think like a Framework Architect

The Mental Shift

The Consumer Mindset

Normally, you think: *"I need a button."* You grab a pre-made component. You don't care how it draws itself or how it knows it was clicked.

The Architect Mindset

Now, you think: *"I need a distinct region of the screen that intercepts touch events and repaints itself when pressed."*

You stop using **Widgets** and start using **Primitives**.

Your Toolbox (The Primitives)

STRICTLY FORBIDDEN: High-level widgets (Scaffold, Column, Card, etc.) are **OFF-LIMITS**. You **MUST** use primitives.

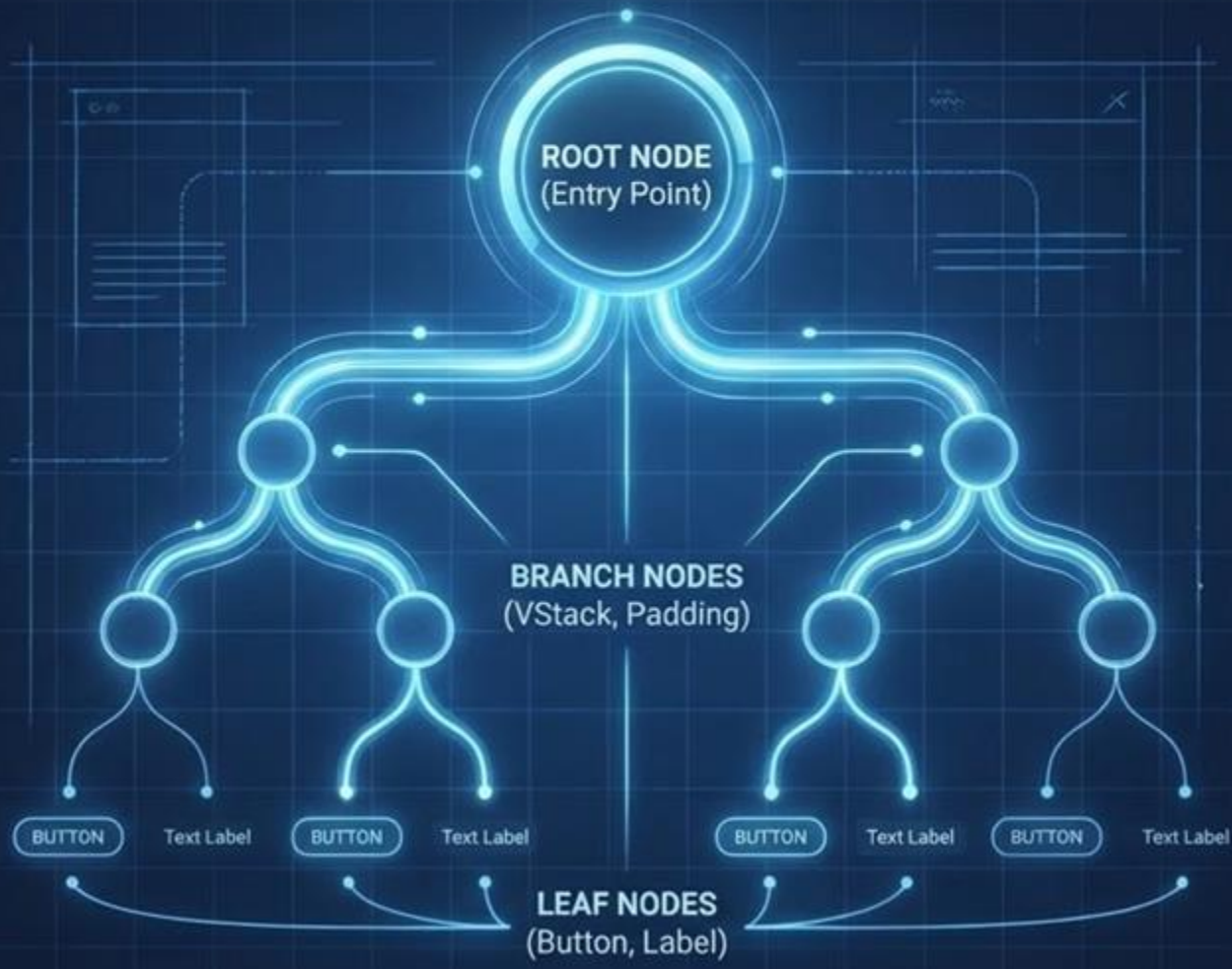
	FLUTTER	REACT NATIVE	COMPOSE
DRAW	`CustomPaint / Canvas`	`View (StyleSheet)`	`Canvas / DrawScope`
TOUCH	`GestureDetector`	`Pressable`	`Modifier.pointerInput`
LAYOUT	`CustomMultiChildLayout`	`View (Flexbox)`	`Layout Composable`

The Widget Tree

Everything is a Node

Your SDK is essentially a tree management system.

- **Leaf Nodes:** The UI components (Button, Label). They draw pixels.
- **Branch Nodes:** The Layouts (VStack, Padding). They don't draw; they calculate positions for their children.
- **Root Node:** The entry point of your app.



The Layout Protocol: Constraints

1. Constraints Down

Layout is a negotiation. The parent tells the child: *"You can be as small as 0, but no bigger than 300px."*

Never assume a size. Always respect the incoming constraints.

2. Sizes Up

The child calculates its needs based on the constraints. *"Okay, I have text that is 50px wide, so I need 50px."*

The child **reports** this size back to the parent.

3. Position Set

The parent receives the size, decides on an (X, Y) coordinate, and places the child.

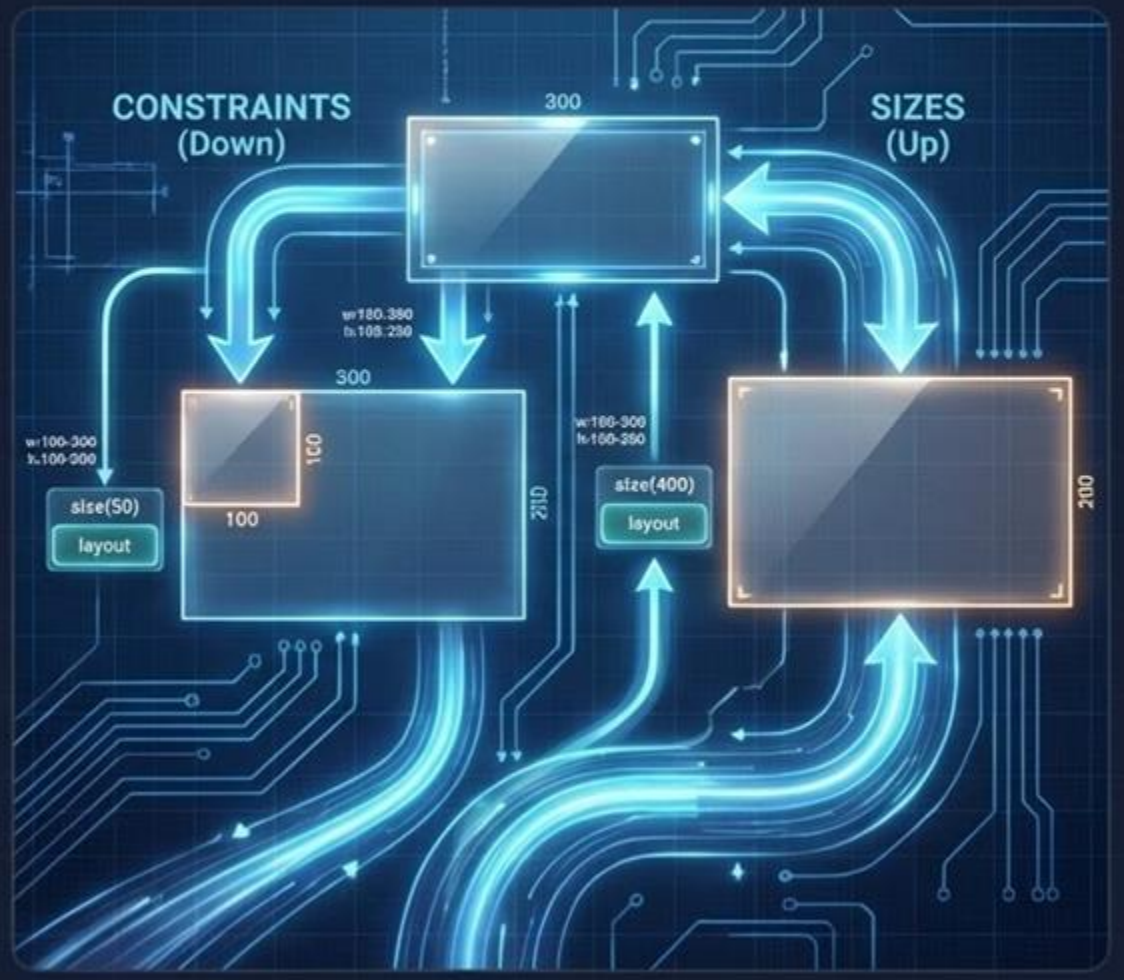
Visualizing the Pass

The "Single Pass" Rule

Efficient UI frameworks do layout in O(n). You traverse the tree down to pass constraints, and up to pass sizes.

$$\text{ParentWidth} = \max \text{ChildWidths}$$

$$\text{ParentHeight} = \sum \text{ChildHeights}$$



Deep Dive: Implementing VStack

```
VStack {
   Text("Always visible")

   if showDetails {
       Text("Details")
       Text("More information")
    }

    Text("Also always visible")
}
```

The Logic

A Vertical Stack is just a loop.

1. Keep track of a yOffset (cursor).
2. Tell each child it has infinite vertical space, but limited width.
3. After the child measures itself, place it at the current yOffset.
4. Increment yOffset by the child's height.

The "Child" Slot

Your layout components must be **generic**. Do not hardcode "Button" inside your Stack.

🚫Bad

Hardcoding children makes the component useless for anything else.

```
VStack(Button, Button)
```

✓Good

Accept an array of **any** widget type (Nodes).

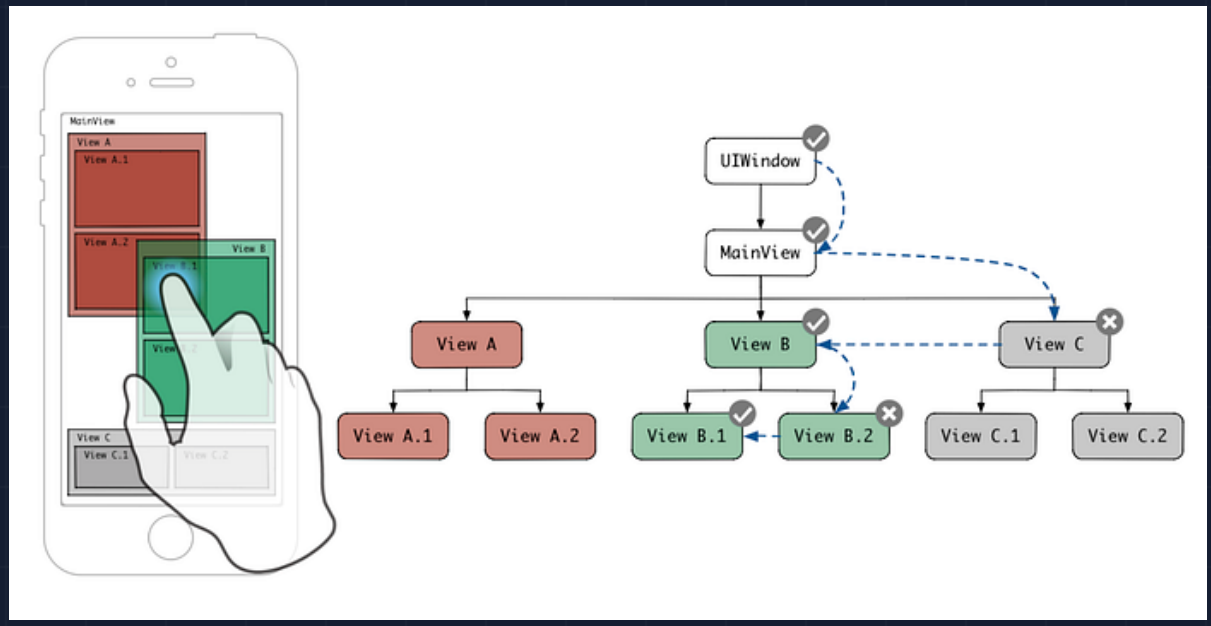
```
VStack(children: [Widget])
```

Handling Interaction

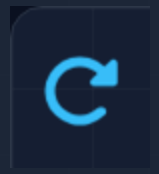
The Event Loop

How does a "Button" know it's clicked?

1. **Hit Test:** When the screen is touched, the framework checks which nodes overlap that coordinate (z-index matters!).
2. **Dispatch:** The event is sent to the top-most node.
3. **State Update:** The node sets isPressed = true.
4. **Re-render:** The node requests a repaint to show the "pressed" color.



Common Pitfalls



Infinite Layout Loops

Parent asks Child for size → Child asks Parent for size → Parent asks Child... Crash.

Fix: Ensure one direction of constraints (Down) and one direction of sizing (Up).

Hit Test Failures

Buttons not clicking? You might be drawing *outside* your bounds, or a transparent container is sitting on top of your button.

Coordinate confusion

Remember: (0,0) is usually relative to the **Parent**, not the Screen. Transform coordinates when moving down the tree.

Submission Checklist

- **2 UI Components:** (e.g., Button, Card). Must use Canvas/View primitives.
- **2 Layouts:** (e.g., VStack, HStack). Must implement the measure/place protocol.
- **Example App:** A working app that imports your library and builds a UI.
- **No High-Level Widgets:** `Scaffold` or `Column` found in your library code = Fail.

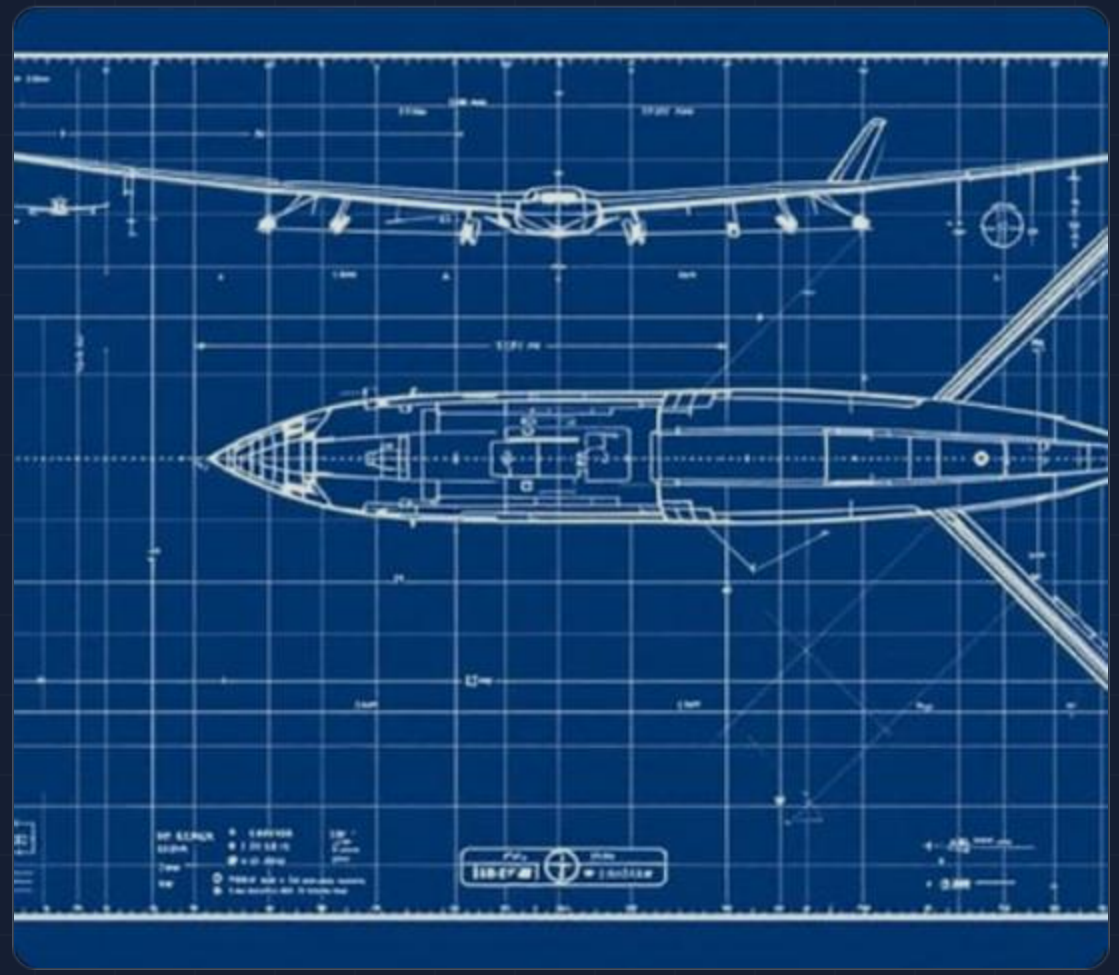


Image Sources

https://cdn.processon.io/admin/knowledge/article_content_img/67ecf9cf45b96a5aedca958c.png

Source: www.processon.io

https://solace.com/wp-content/uploads/2023/04/event-mesh-as-architectural-pattern_pic-05.png

Source: solace.com

https://media.geeksforgeeks.org/wp-content/uploads/20240618105149/User-Interface-Design-Infograph.webp

Source: www.geeksforgeeks.org

https://miro.medium.com/1*Hno_QgZ5umpSDdhkzV96Aw.png

Source: medium.com

https://cdn.prod.website-files.com/64e895a2f8733943c6d0ddef/65ae4850adc11b901b05fe95_Code%20Snippets%20UI%20Element%20-%20uinkits.svg

Source: www.uinkits.com