Design Patterns in Flutter

Flutter uses design patterns for its application development.

- **Decorator Pattern**
- **Composite Pattern**
- **Observer Pattern**

Decorator Pattern

- The **Decorator Pattern** adds new behavior to an object dynamically without altering its class or other objects.
- Example: `"hello"` becomes red with `red_color()` and then a title with `title()`.
- The original text remains unchanged — decorators (`red_color`, `title`) *wrap* it to add behaviors.

```
title(red_color(text("hello")))
```

Decorator Pattern in Flutter

- In Flutter, the decorator pattern is used to describe the structure of the widgets.
- To make a text in the center of an output screen, we use the decorator pattern as follows.

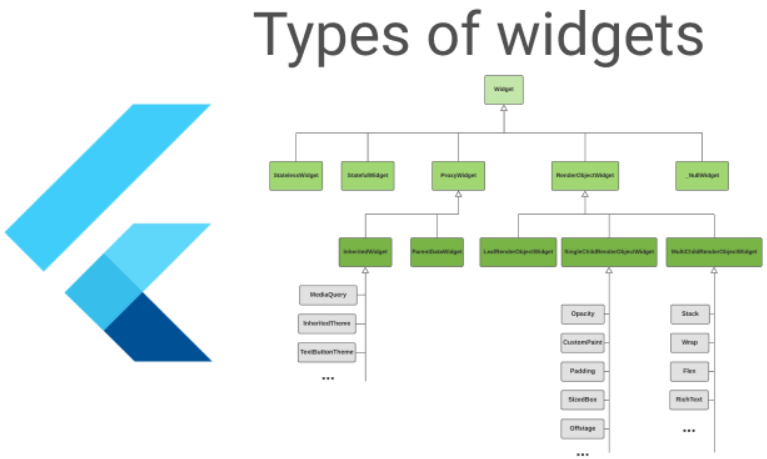
```
body: Center(
  child: Text(
    'Text One',
  ),
),
```

Composite Pattern

- The composite pattern describes a group of objects treated the same way as a single instance of the same type of object.
- The composite design pattern intends to "compose" objects into tree structures to represent part-whole hierarchies.
- The most frequently used composite pattern is the directory structure: the directory can recursively contain both directories and single files.

Composite Pattern in Flutter

- The **Composite Pattern** lets you treat a group of objects like a single object.
- It builds a **tree structure** to represent part–whole hierarchies.
- Example: a **directory** containing both files and subdirectories.



Using Decorator and Composite Together

- **Decorator Pattern**: Each Widget (like `MaterialApp`, `Text`) wraps another to add new behavior or style (e.g., theming, layout).
- **Composite Pattern**: Flutter builds a **Widget tree**, where parent and child Widgets form a hierarchical structure.

Example:

```
MaterialApp(home: Text('Hello, ASE 456 Students'));
```

Here, MaterialApp decorates and composes Text within a tree — showing both patterns in action.

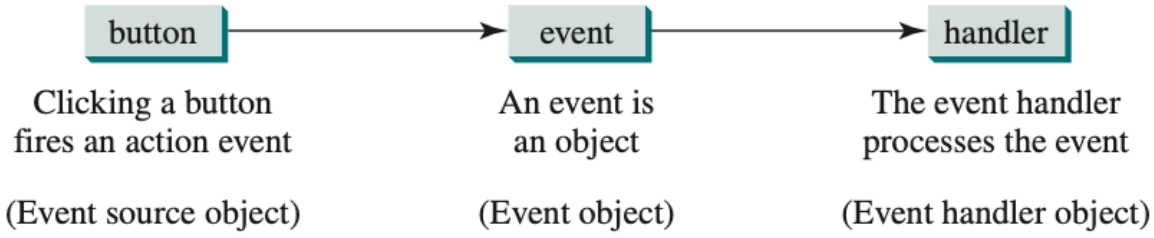
Creating Custom Widgets

- When a Widget becomes complex, we can define a **custom class** that implements the `build()` method returning another Widget.
- The `MyApp` class, which overrides `build()` to return a `Text` Widget.
- This works because Flutter applies the **Composite Pattern**, allowing Widgets to be built from other Widgets.

```
// MaterialApp(home: Text('Hello, ASE 456 Students'));
MaterialApp(home:MyApp()));
class MyApp extends StatelessWidget {
  @override Widget build(BuildContext context) {
    return Text('Hello, ASE 456 Students');
  }
}
```

Observer Pattern

- Observer is a design pattern that lets you define a subscription mechanism to notify objects about any events that happen to the object they're observing.



Observer Pattern in Flutter

- In this Flutter example, clicking the **Floating Action Button** triggers the `_incrementCounter` method.
- This method calls `setState()` with a lambda expression `() => { ... }` to update the state.
- Flutter then **automatically calls** the `build()` method in the background to refresh the UI.

```
FloatingActionButton(
  onPressed: _incrementCounter, // event
),

void _incrementCounter() {setState(() { ... });}

@override
Widget build(BuildContext context) { ... }
```