I see you've shared a tree structure image that outlines various concepts related to Flutter development. I'll break down each section in the image, provide explanations, and include code examples where applicable. This will cover the theories and practical code examples for each concept.

Level 1: Basic Concepts

1. Style

- **Material Design**:
  - Material Design is a design system developed by Google that aims to create a consistent and cohesive user experience across platforms. In Flutter, Material Design widgets are provided out of the box.
  - **Example**:

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Material Design Example')),
        body: Center(child: Text('Hello, Material Design!')),
      ),
    );
  }
}
```

- **Cupertino Design**:
  - Cupertino widgets are designed to mimic the iOS design language. They are used when you want your Flutter app to have an iOS look and feel.
  - **Example**:

```dart
import 'package:flutter/cupertino.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return CupertinoApp(
      home: CupertinoPageScaffold(
        navigationBar: CupertinoNavigationBar(
          middle: Text('Cupertino Design Example'),
        ),
        child: Center(child: Text('Hello, Cupertino Design!')),
      ),
    );
  }
}
```

2. Widgets

- **Stateless Widgets**:
  - Stateless widgets do not maintain any state of their own. They are immutable and only depend on their configuration for rendering.
  - **Example**:

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(child: MyStatelessWidget()),
      ),
    );
  }
}

class MyStatelessWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('I am a Stateless Widget');
  }
}
```

- **Stateful Widgets**:

- Stateful widgets maintain a mutable state. The `State` object associated with the `StatefulWidget` can be changed, and the UI is updated accordingly.
- **Example**:

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(child: MyStatefulWidget()),
      ),
    );
  }
}

class MyStatefulWidget extends StatefulWidget {
  @override
  _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  String _text = 'I am a Stateful Widget';

  void _changeText() {
    setState(() {
      _text = 'State has changed!';
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text(_text),
        ElevatedButton(
          onPressed: _changeText,
          child: Text('Change Text'),
        ),
      ],
    );
  }
}
```

- **Inherited Widgets**:
  - Inherited widgets are used to pass data down the widget tree efficiently. They allow child widgets to subscribe to data changes.
  - **Example**:

```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(child: MyWidget()),
      ),
    );
  }
}

class MyInheritedWidget extends InheritedWidget {
  final int counter;
  final Widget child;

  MyInheritedWidget({this.counter, this.child}) : super(child: child);

  @override
  bool updateShouldNotify(MyInheritedWidget oldWidget) {
    return counter != oldWidget.counter;
  }

  static MyInheritedWidget of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<MyInheritedWidget>
();
  }
}

class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MyInheritedWidget(
      counter: 10,
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text('Inherited Widget Example'),
          Text('Counter: ${MyInheritedWidget.of(context).counter}'),
        ],
      ),
    );
  }
}
```

3. Core Design Principles

- **KISS (Keep It Simple, Stupid)**:

    - This principle emphasizes simplicity in design and development, making code more maintainable and understandable.
- **SOLID**:
    - The SOLID principles are a set of five design principles that help developers build maintainable and scalable software:
        1. **Single Responsibility Principle (SRP)**
        2. **Open/Closed Principle (OCP)**
        3. **Liskov Substitution Principle (LSP)**
        4. **Interface Segregation Principle (ISP)**
        5. **Dependency Inversion Principle (DIP)**
- **DRY (Don't Repeat Yourself)**:
    - DRY is a principle aimed at reducing the repetition of code patterns. By applying DRY, you make your codebase more maintainable and reduce redundancy.