Architecture Components & MVVM

Applikationsudvikling

NIFR 2024 - https://github.com/nicklasdean/court_counter_text_mirror

Agenda

Architecture components & MVVM

- The observer pattern & recomposition
- Application state & state hoisting
  - Unidirectional Data Flow
  - Seperation of concerns (SoP)
- Model - view - ViewModel (MVVM)
- Application architecture

Patterns in software development

Design patterns

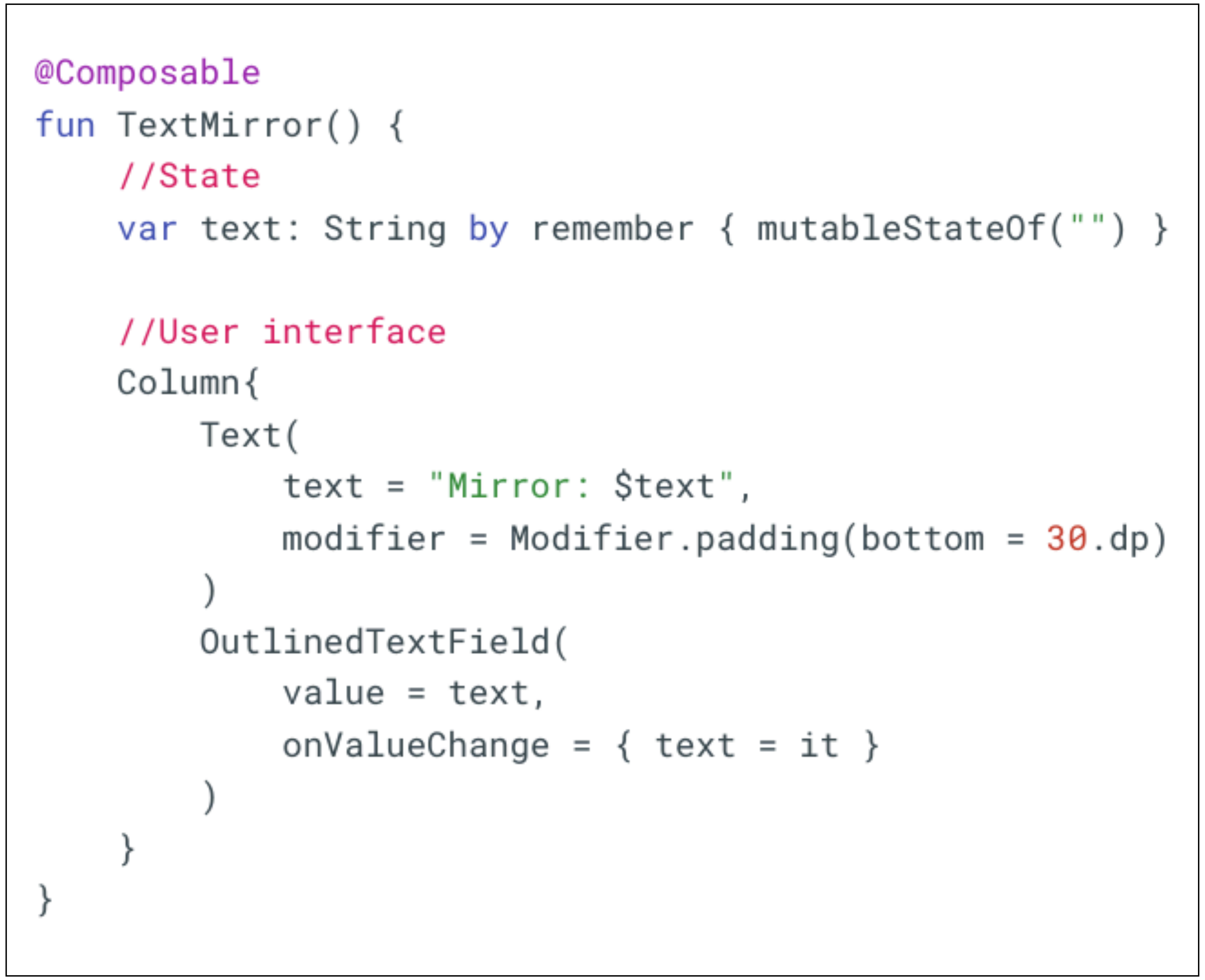
In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

https://en.wikipedia.org/wiki/Software_design_pattern

Highlights: Recomposition

From the following example

- When state changes the Text composable is called again
  - If statements and logic can change composition
- The best case scenario is to create loosely coupled components



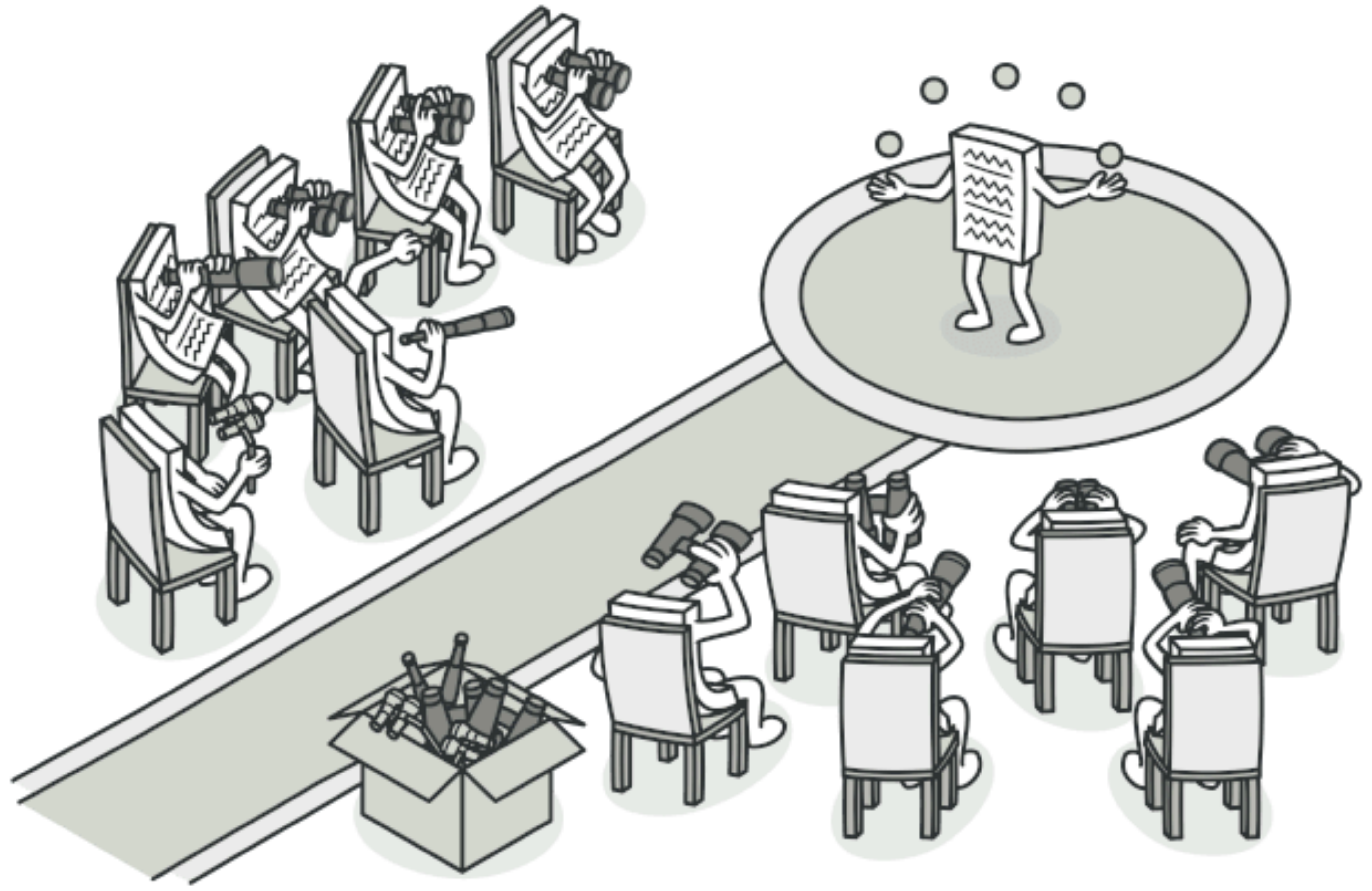
```
@Composable
fun TextMirror() {
    //State
    var text: String by remember { mutableStateOf("") }

    //User interface
    Column{
        Text(
            text = "Mirror: $text",
            modifier = Modifier.padding(bottom = 30.dp)
        )
        OutlinedTextField(
            value = text,
            onValueChange = { text = it }
        )
    }
}
```

The observer pattern

Android: Jetpack Compose

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

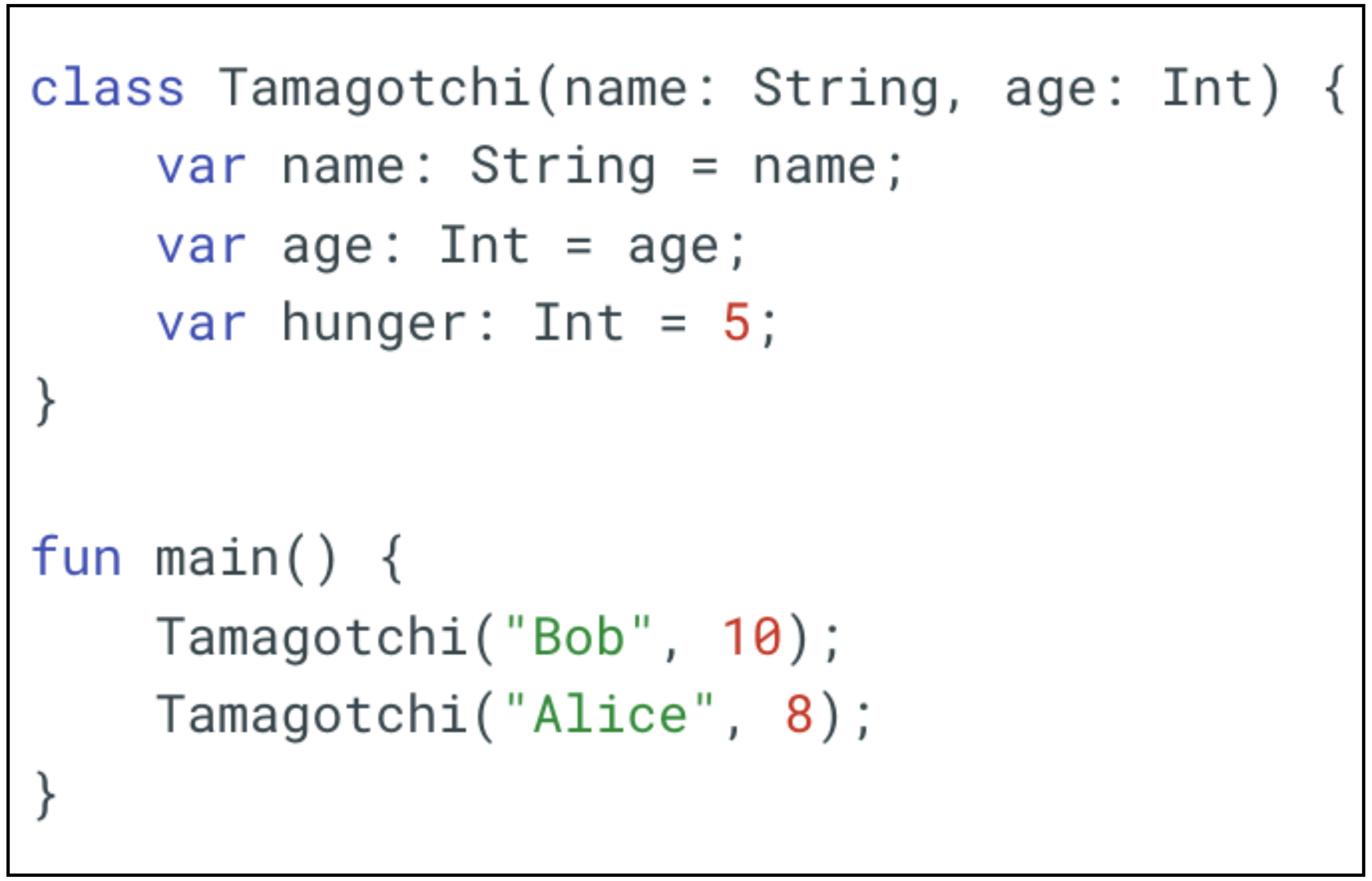


https://refactoring.guru/design-patterns/observer

State (again)

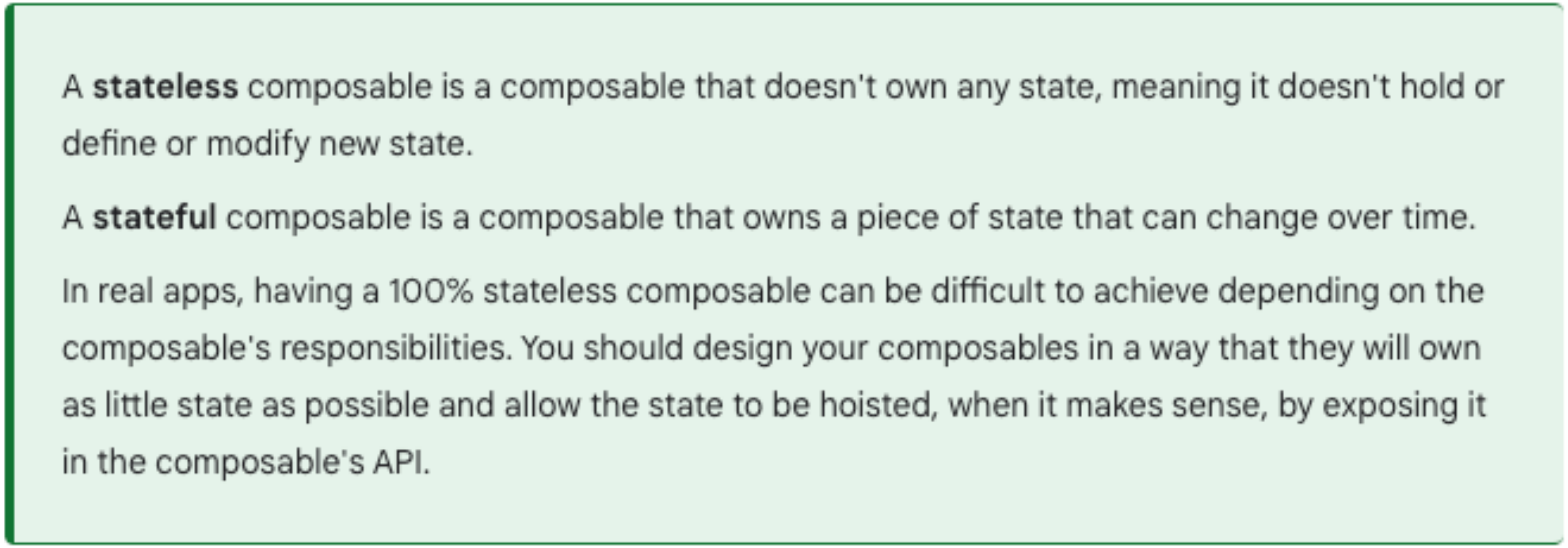
In an object

- Each object has various states
  - Notice that the object properties that represent values in the object
- The Tamagotchi class is designed in such a way that they are stateful
- A stateful objects keeps state encapsulated



```
class Tamagotchi(name: String, age: Int) {
    var name: String = name;
    var age: Int = age;
    var hunger: Int = 5;
}

fun main() {
    Tamagotchi("Bob", 10);
    Tamagotchi("Alice", 8);
}
```


A **stateless** composable is a composable that doesn't own any state, meaning it doesn't hold or define or modify new state.

A **stateful** composable is a composable that owns a piece of state that can change over time.

In real apps, having a 100% stateless composable can be difficult to achieve depending on the composable's responsibilities. You should design your composables in a way that they will own as little state as possible and allow the state to be hoisted, when it makes sense, by exposing it in the composable's API.

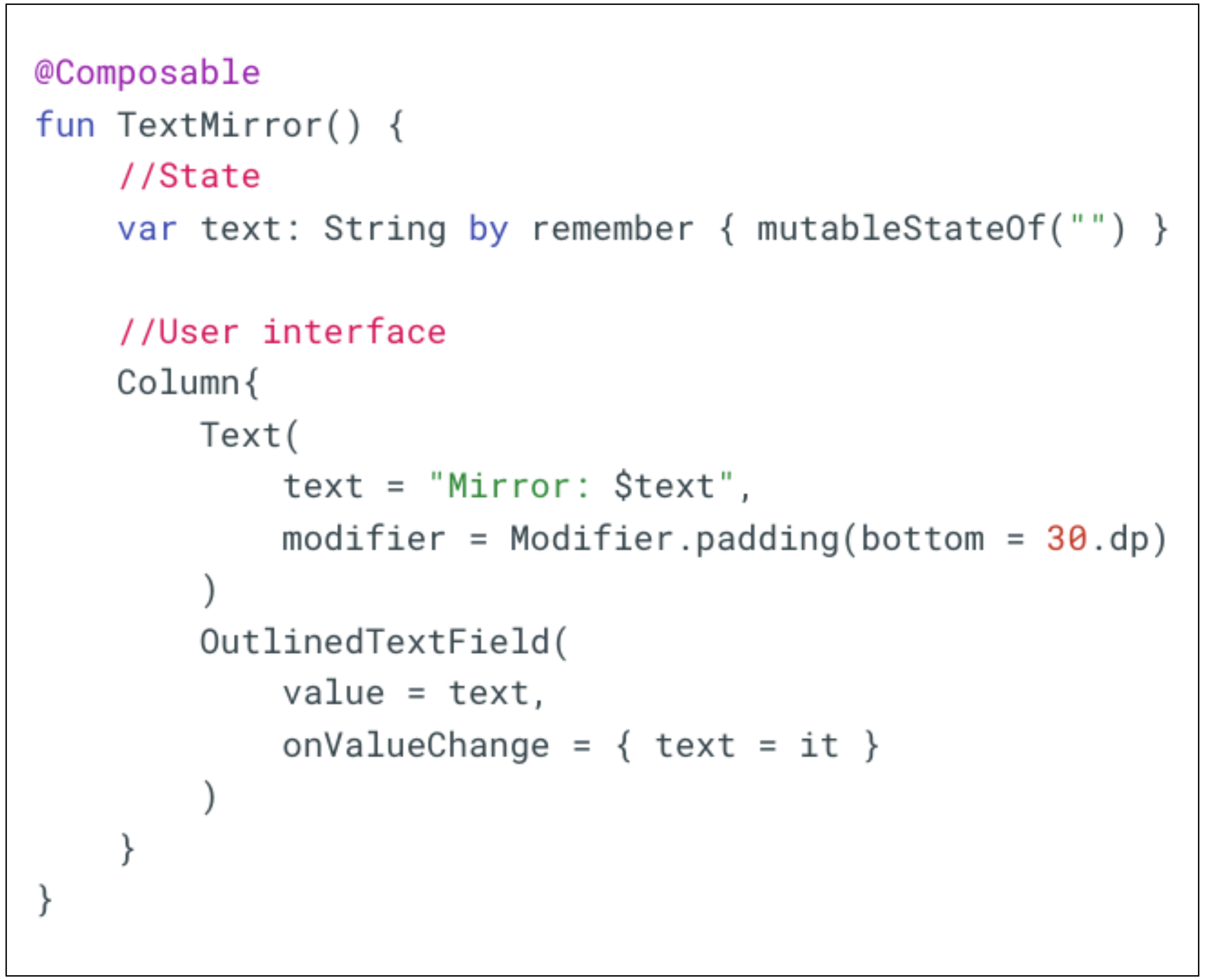
https://developer.android.com/codelabs/jetpack-compose-state#8

Example: Observers of state

Highlights

From the following example

- State (text variable) is tightly coupled to the user interface
    - As the text is inside the composable
- Tightly coupled state & UI makes code more difficult to test & reuse
- The best case scenario is to create loosely coupled components

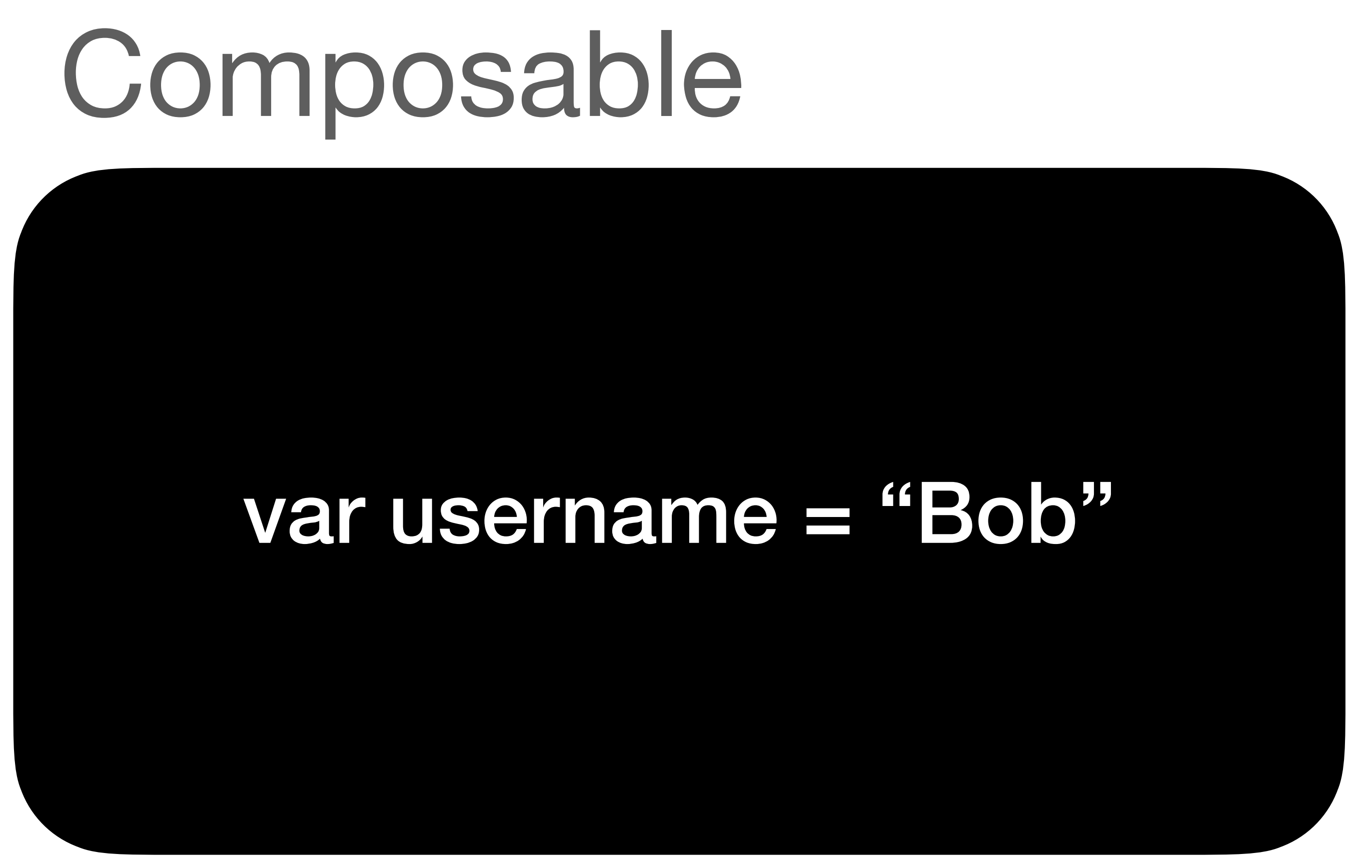


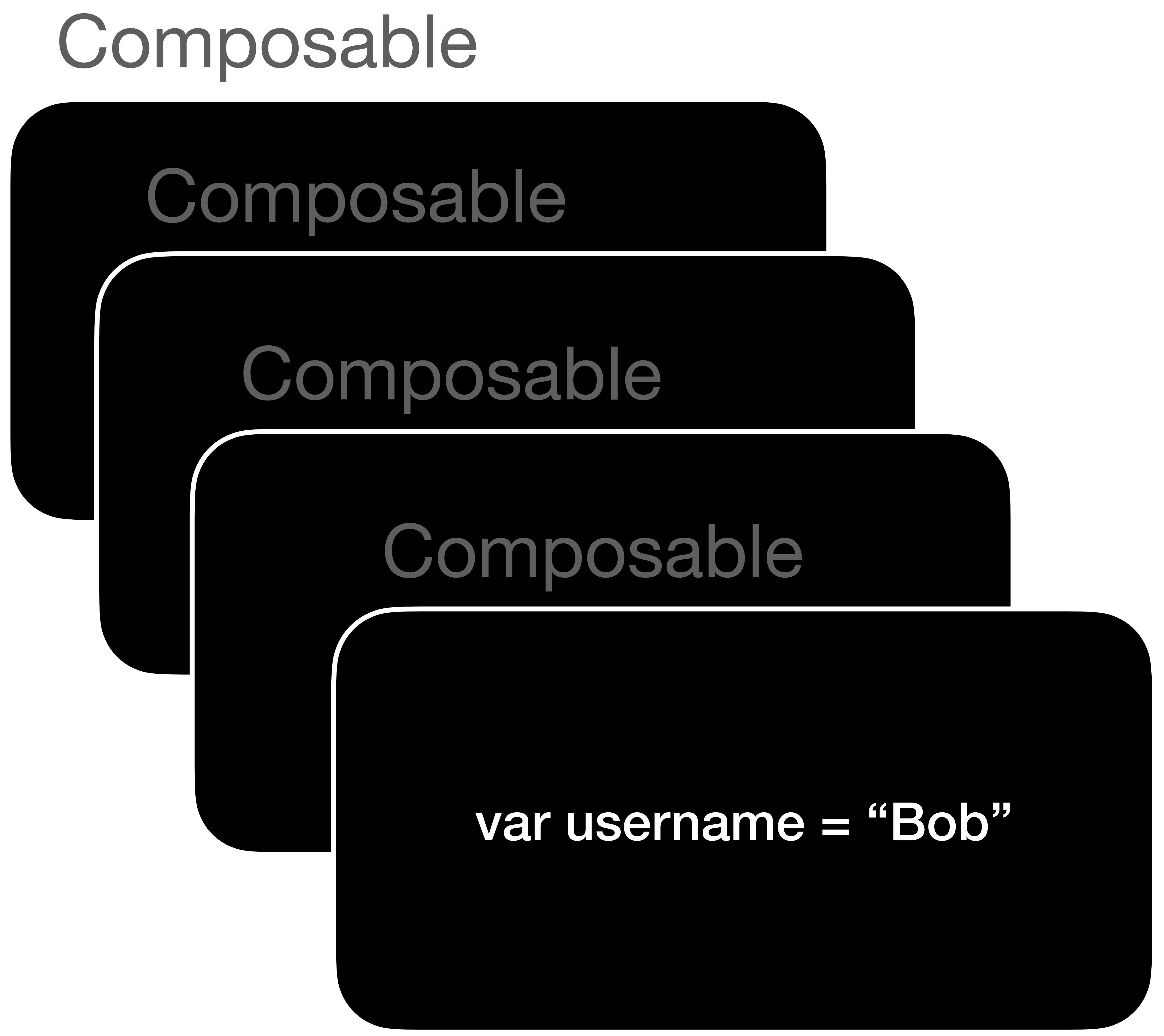
```kotlin
@Composable
fun TextMirror() {
    //State
    var text: String by remember { mutableStateOf("") }

    //User interface
    Column{
        Text(
            text = "Mirror: $text",
            modifier = Modifier.padding(bottom = 30.dp)
        )
        OutlinedTextField(
            value = text,
            onValueChange = { text = it }
        )
    }
}
```

Why?

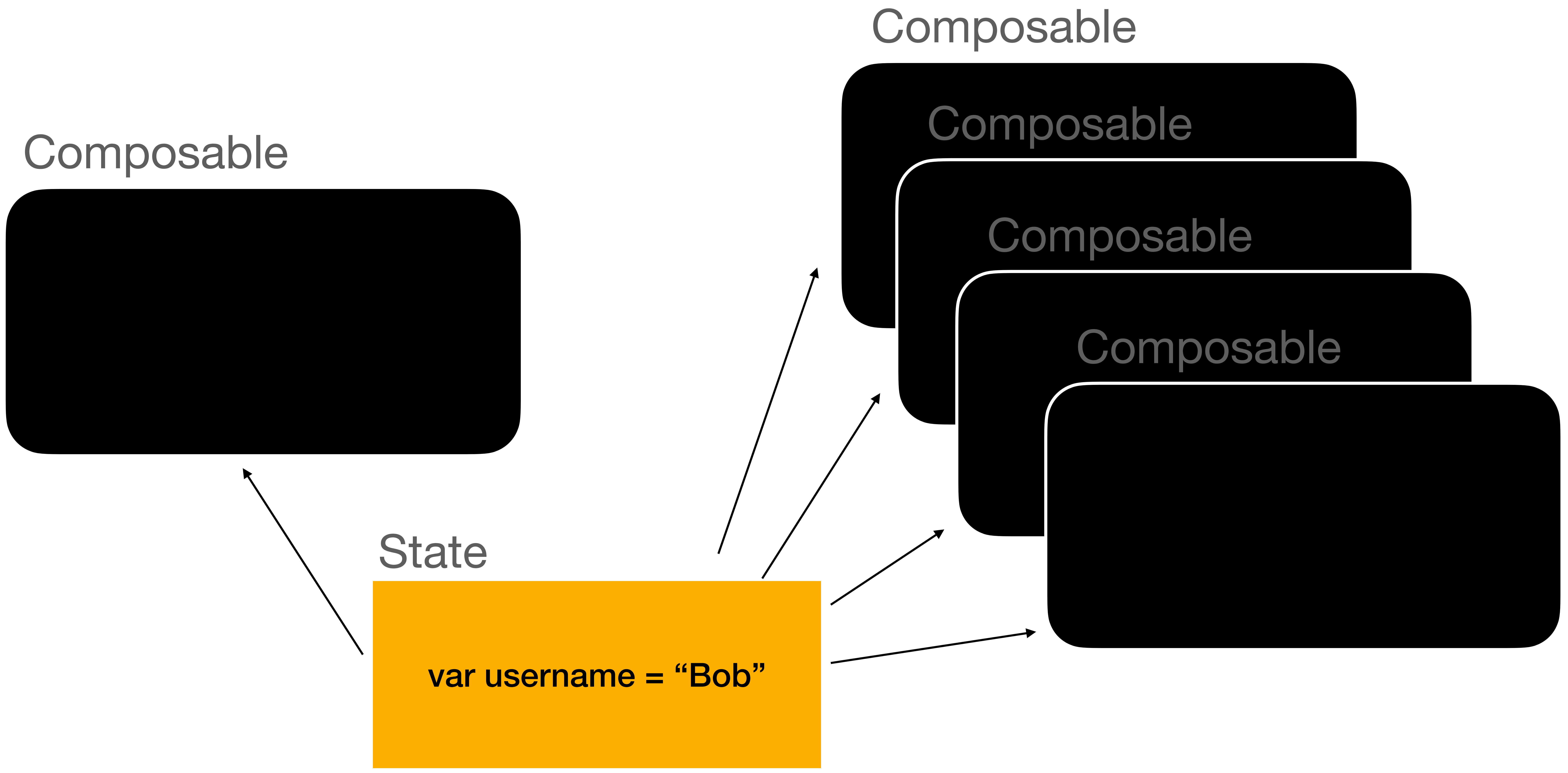
Harder to reuse





Resolve

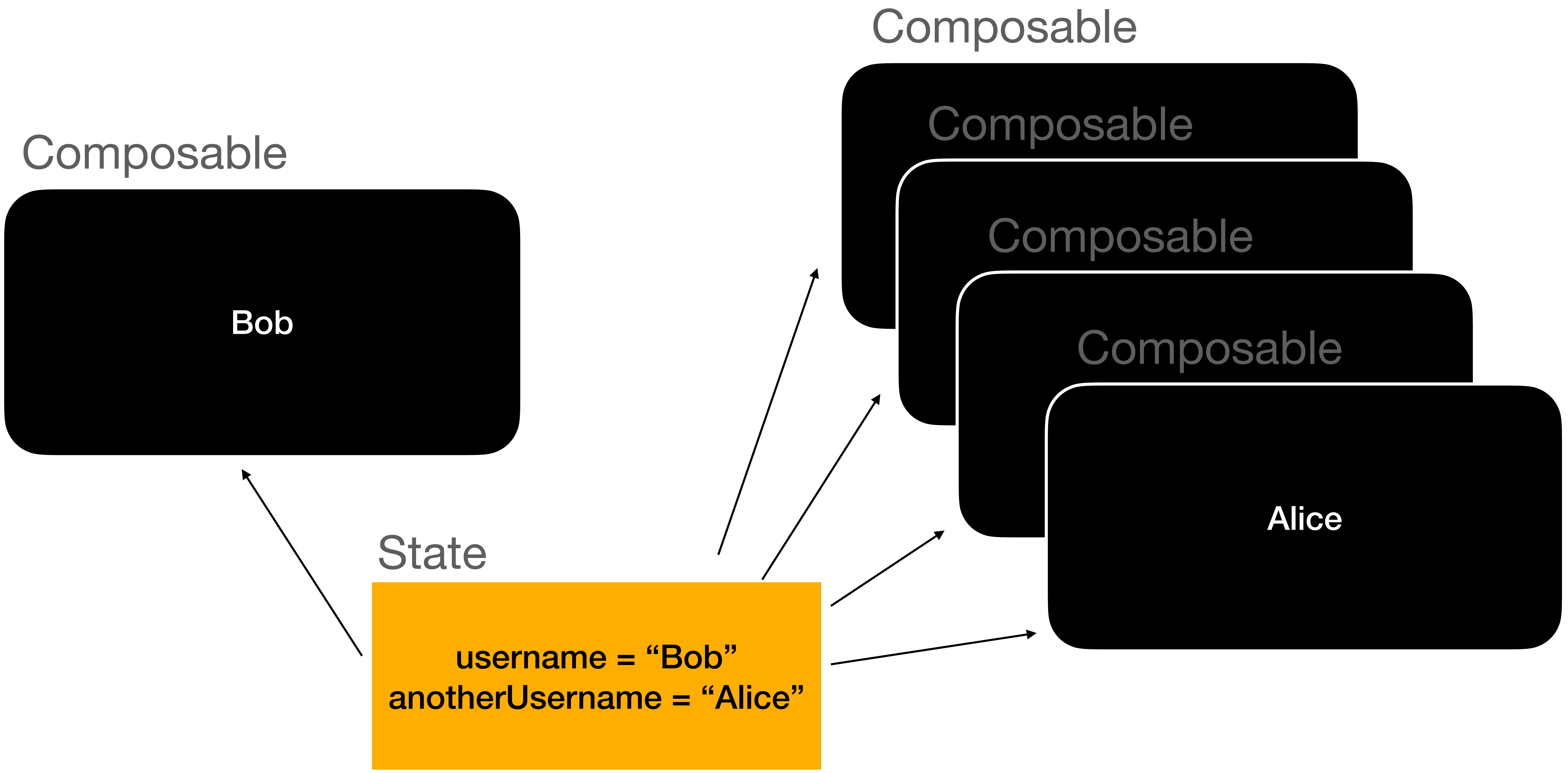
State hoisting



```
var username = "Bob"
```

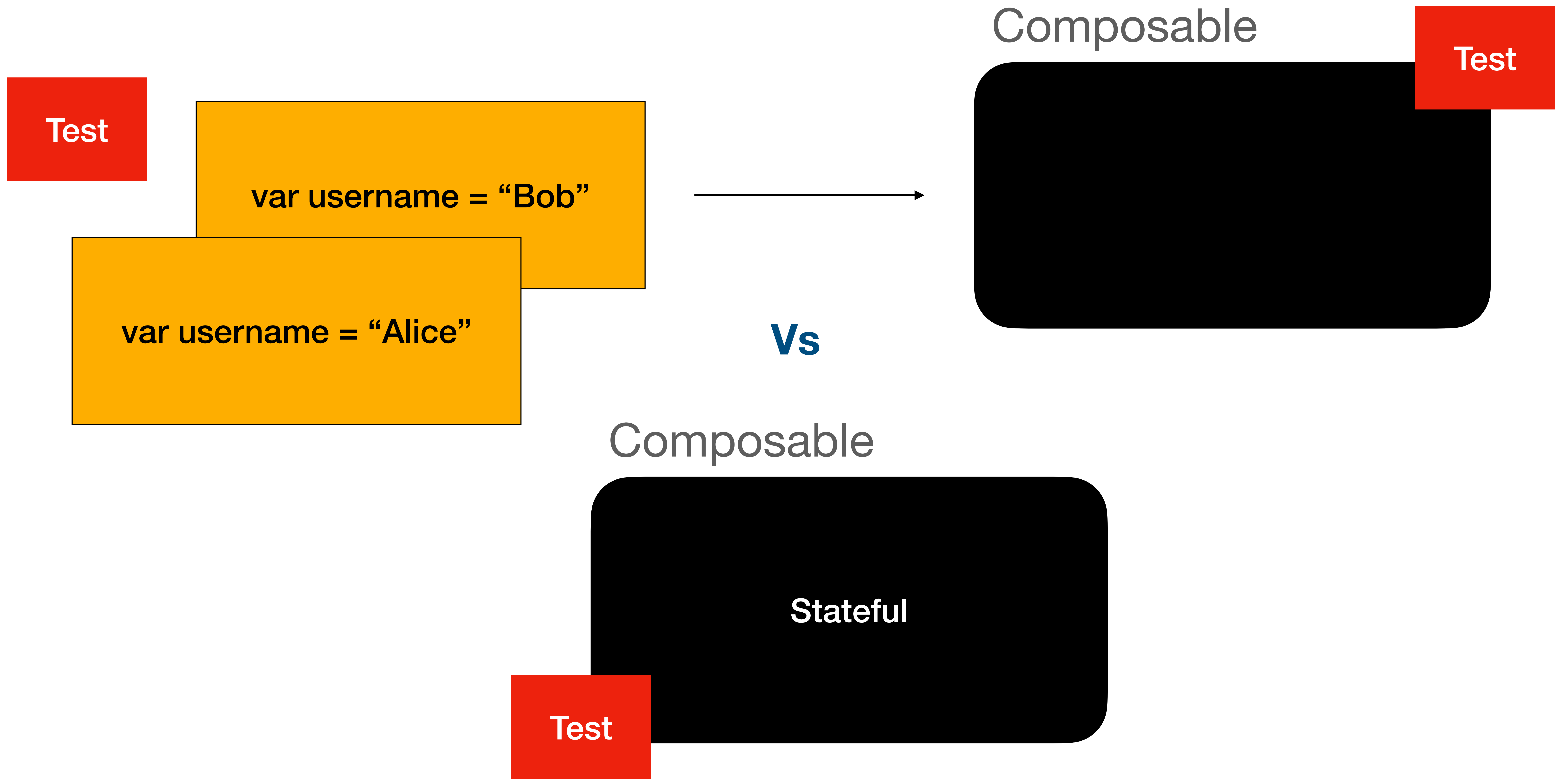
Resolve

State hoisting



Why?

A composable that is stateless is easier to test



Stateless composables

Summary

- **Single source of truth**: By moving state instead of duplicating it, we're ensuring there's only one source of truth. This helps avoid bugs
- **Shareable**: Hoisted state can be shared with multiple composables.
- **Interceptable**: Callers to the stateless composables can decide to ignore or modify events before changing the state.
- **Decoupled**: The state for a stateless composable function can be stored anywhere. For example, in a ViewModel.

https://developer.android.com/codelabs/jetpack-compose-state#8

How to hoist state conceptually?

Lifting state to a higher level in the component hierarchy

Making composable stateless

Identify state and remove

Stateful

```
@Composable
fun TextMirror(){
    var username: String by remember{ mutableStateOf("") };

    Column{
        Text(
            text = "Mirror: $username",
            modifier = Modifier.padding(bottom = 30.dp)
        )
        OutlinedTextField(
            value = username,
            onValueChange = { value -> username += value }
        )
    }
}
```

Stateless

```
@Composable
fun TextMirror(username: String){
    Column{
        Text(
            text = "Mirror: ${username}",
            modifier = Modifier.padding(bottom = 30.dp)
        )
        OutlinedTextField(
            value = state.username,
            onValueChange = { value -> state.username }
        )
    }
}
```

Conceptual State hoist

Danger: Code will not run
☠️ ☠️ ☠️ ☠️

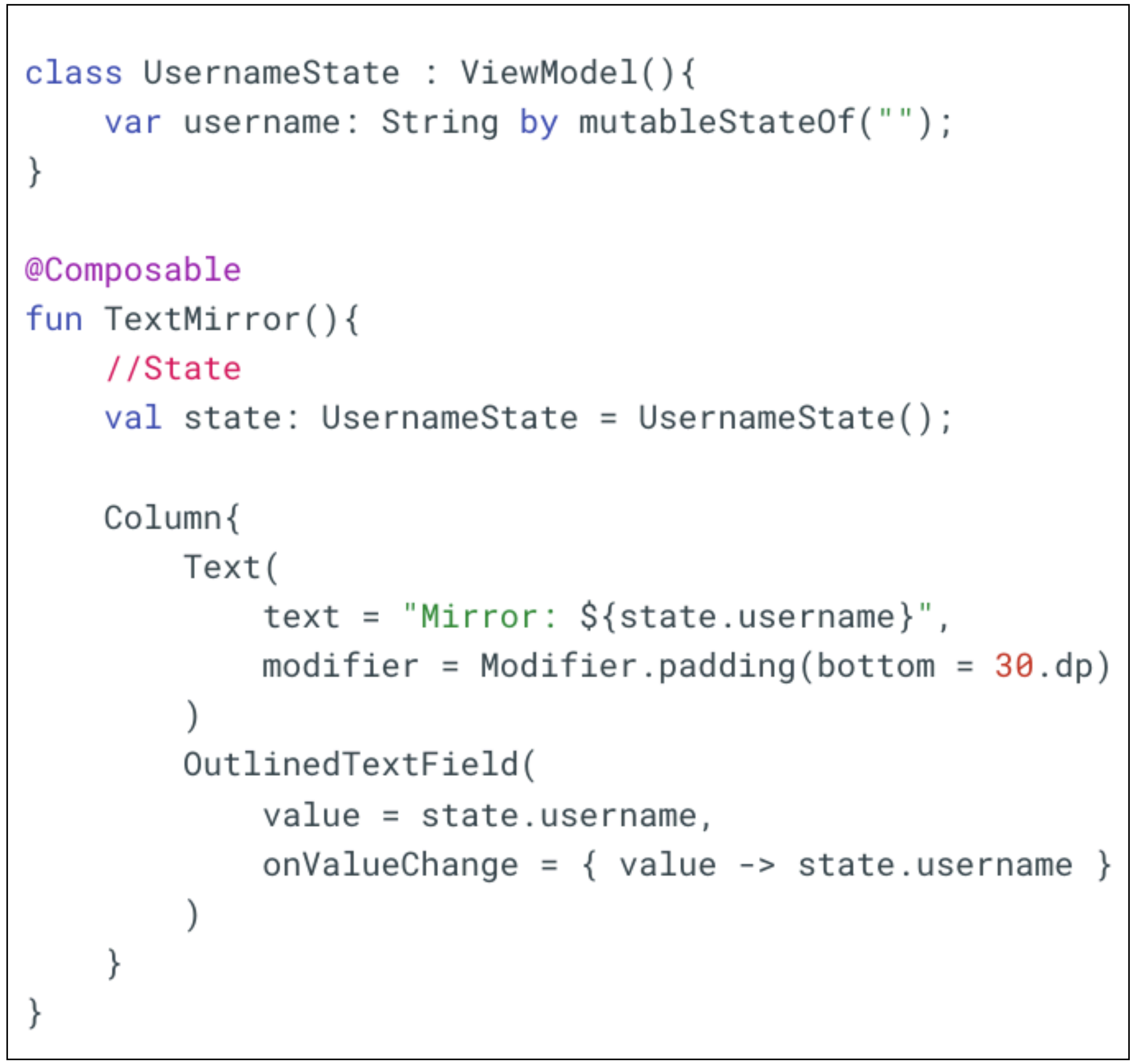
```
class UsernameState : ViewModel(){
    var username: String by mutableStateOf("");
}

@Composable
fun TextMirror(){
    //State
    val state: UsernameState = UsernameState();

    Column{
        Text(
            text = "Mirror: ${state.username}",
            modifier = Modifier.padding(bottom = 30.dp)
        )
        OutlinedTextField(
            value = state.username,
            onValueChange = { value -> state.username }
        )
    }
}
```

Resolve

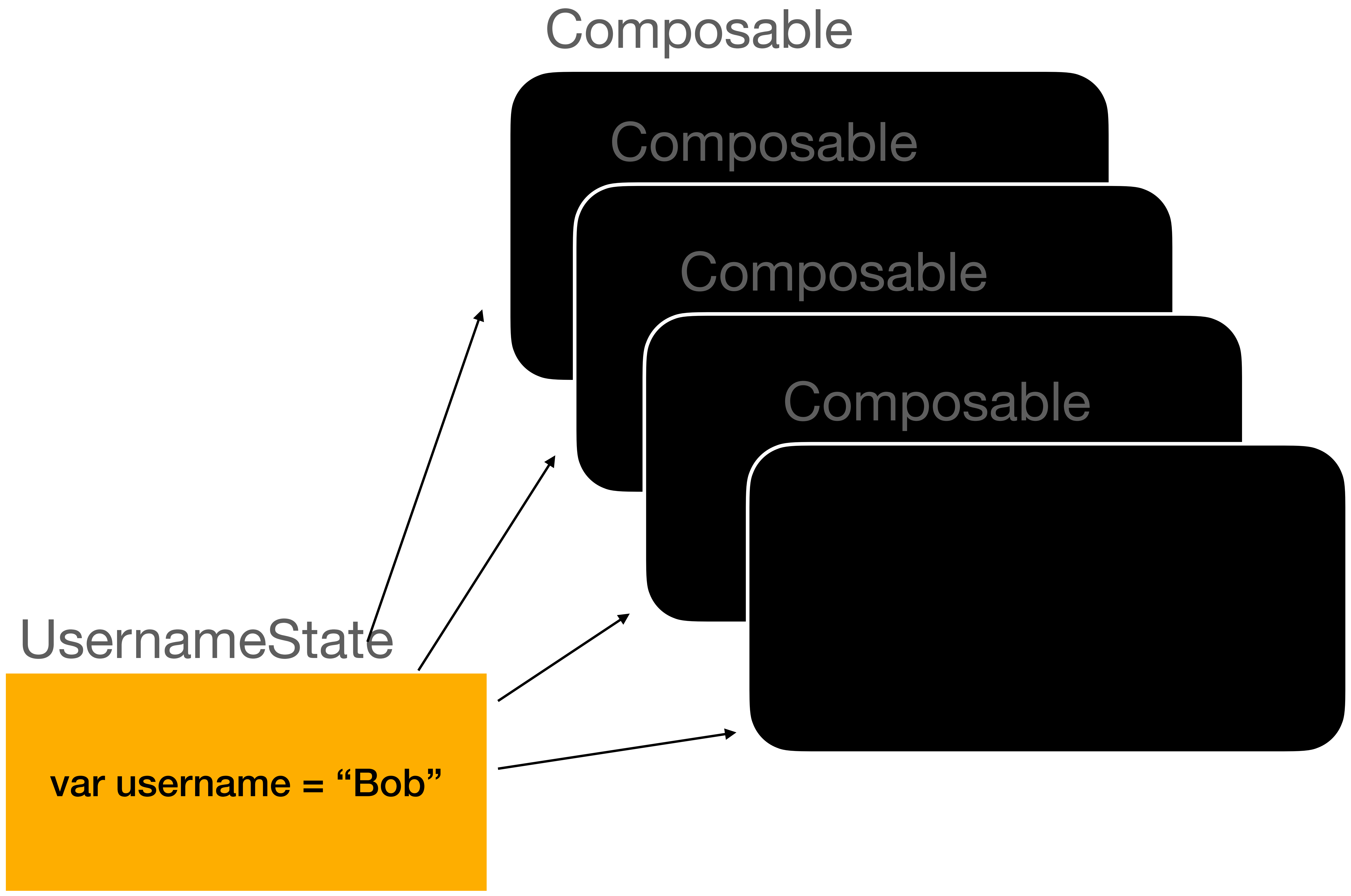
State hoisting



```
class UsernameState : ViewModel(){
    var username: String by mutableStateOf("");
}

@Composable
fun TextMirror(){
    //State
    val state: UsernameState = UsernameState();

    Column{
        Text(
            text = "Mirror: ${state.username}",
            modifier = Modifier.padding(bottom = 30.dp)
        )
        OutlinedTextField(
            value = state.username,
            onValueChange = { value -> state.username }
        )
    }
}
```



Unidirectional data flow

Two ways of visualising the data flow

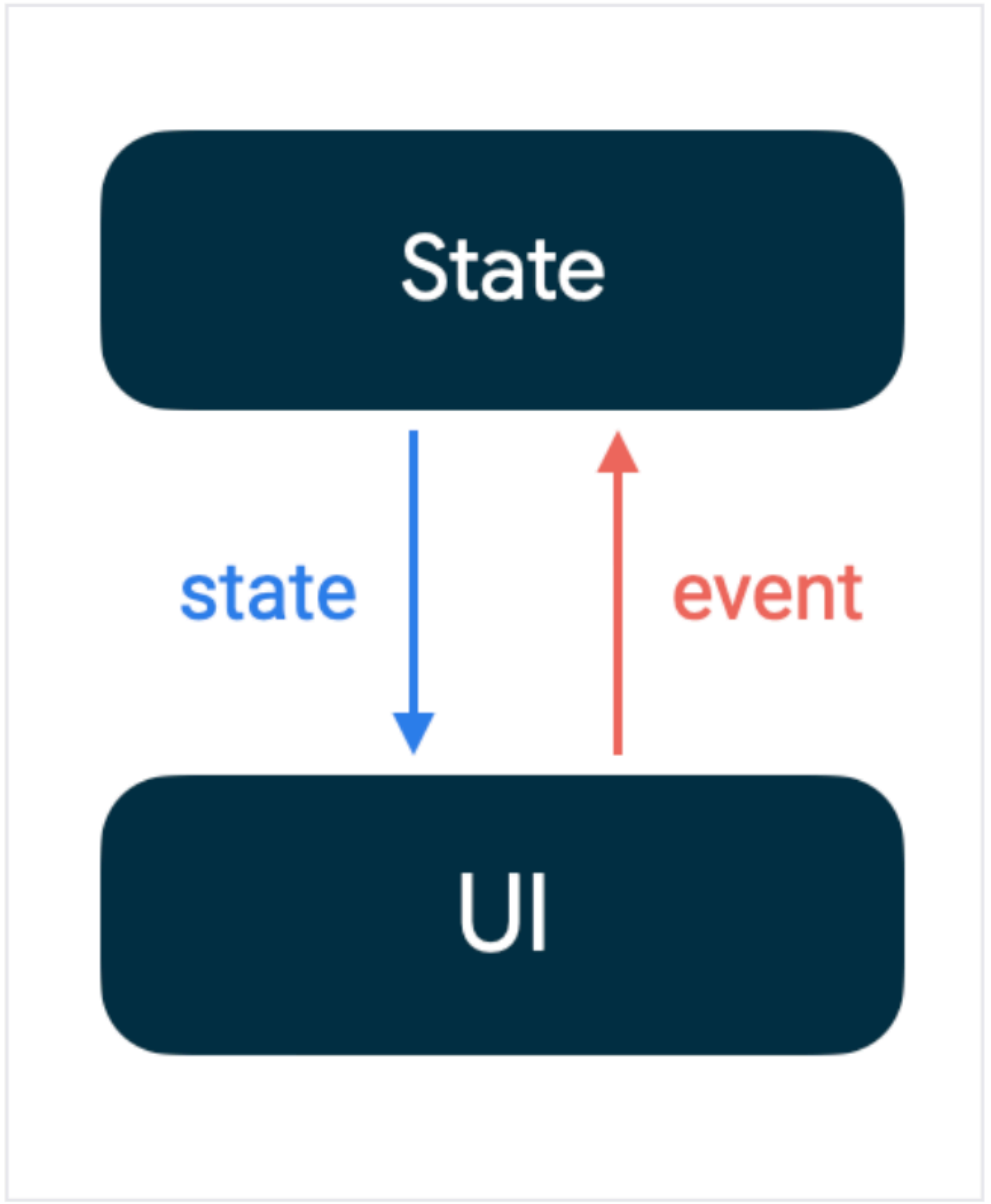
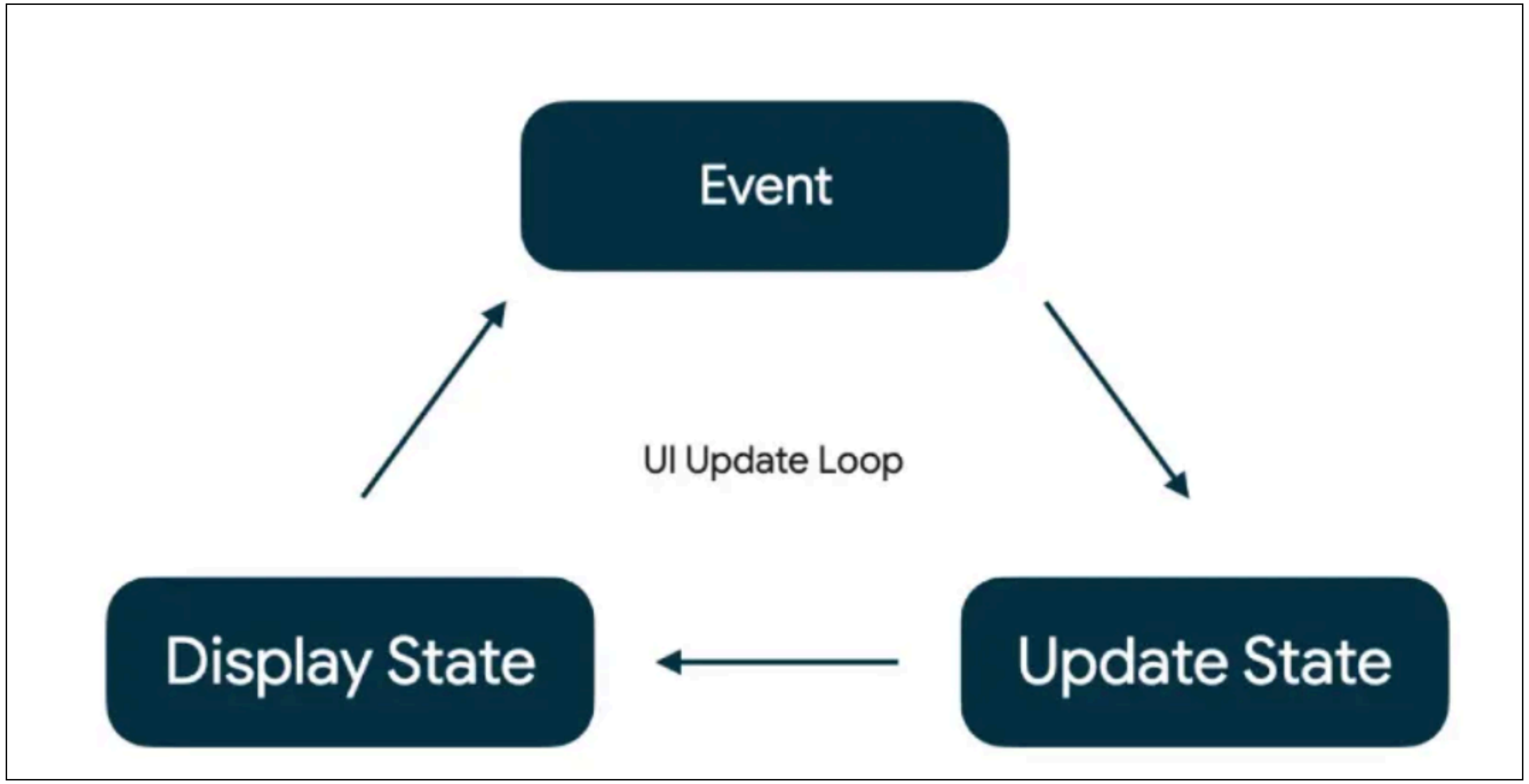


Figure 1. Unidirectional data flow.



Separation of concerns (SoP)

Design pattern

- In an application architecture where concerns are separated responsibilities are easy to understand, discuss and maintain
  - Everything related to the UI (rendering, look, events) happens in stateless composable functions
  - All state is hoisted to objects
  - The objects are called ViewModels

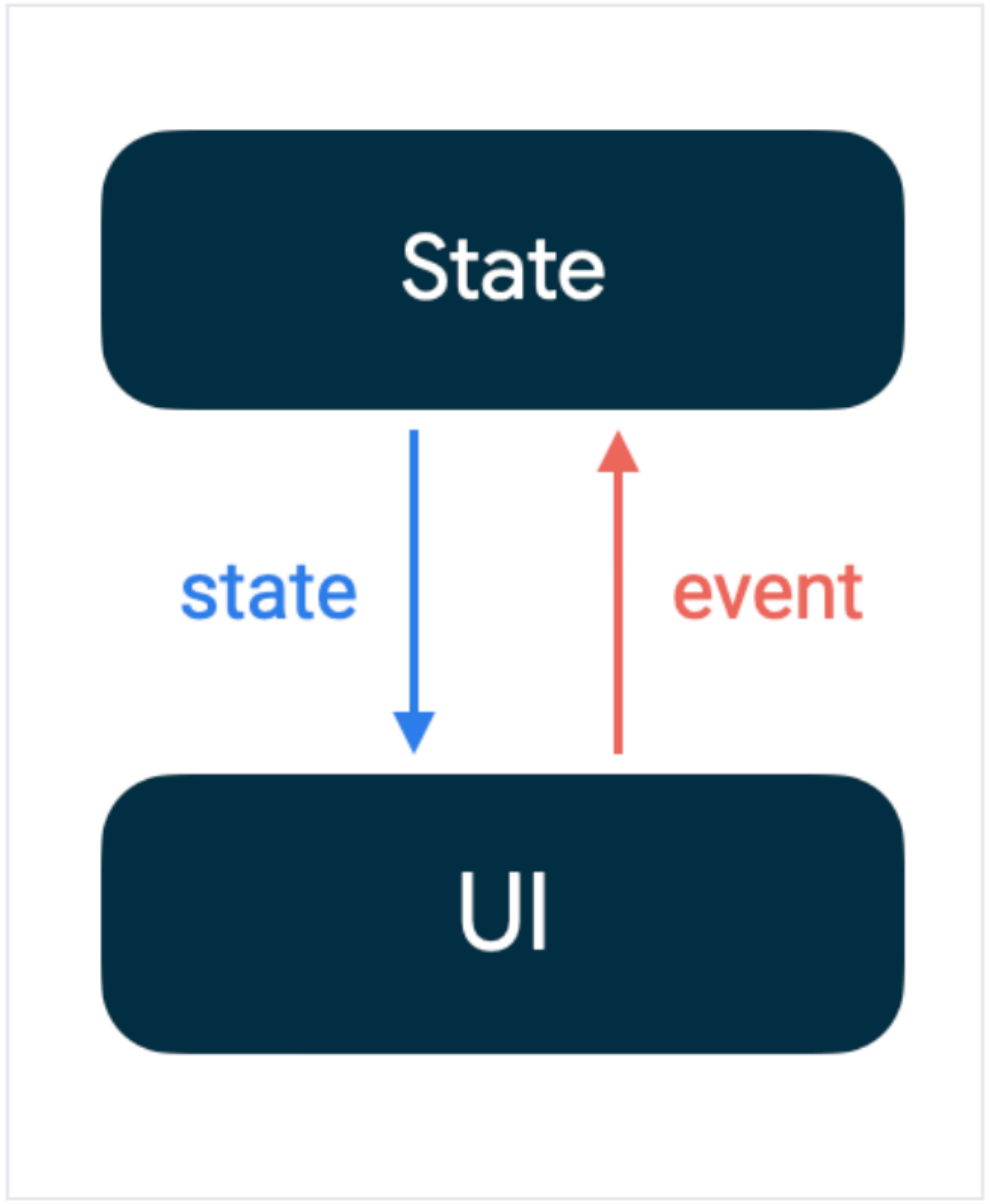


Figure 1. *Unidirectional data flow.*

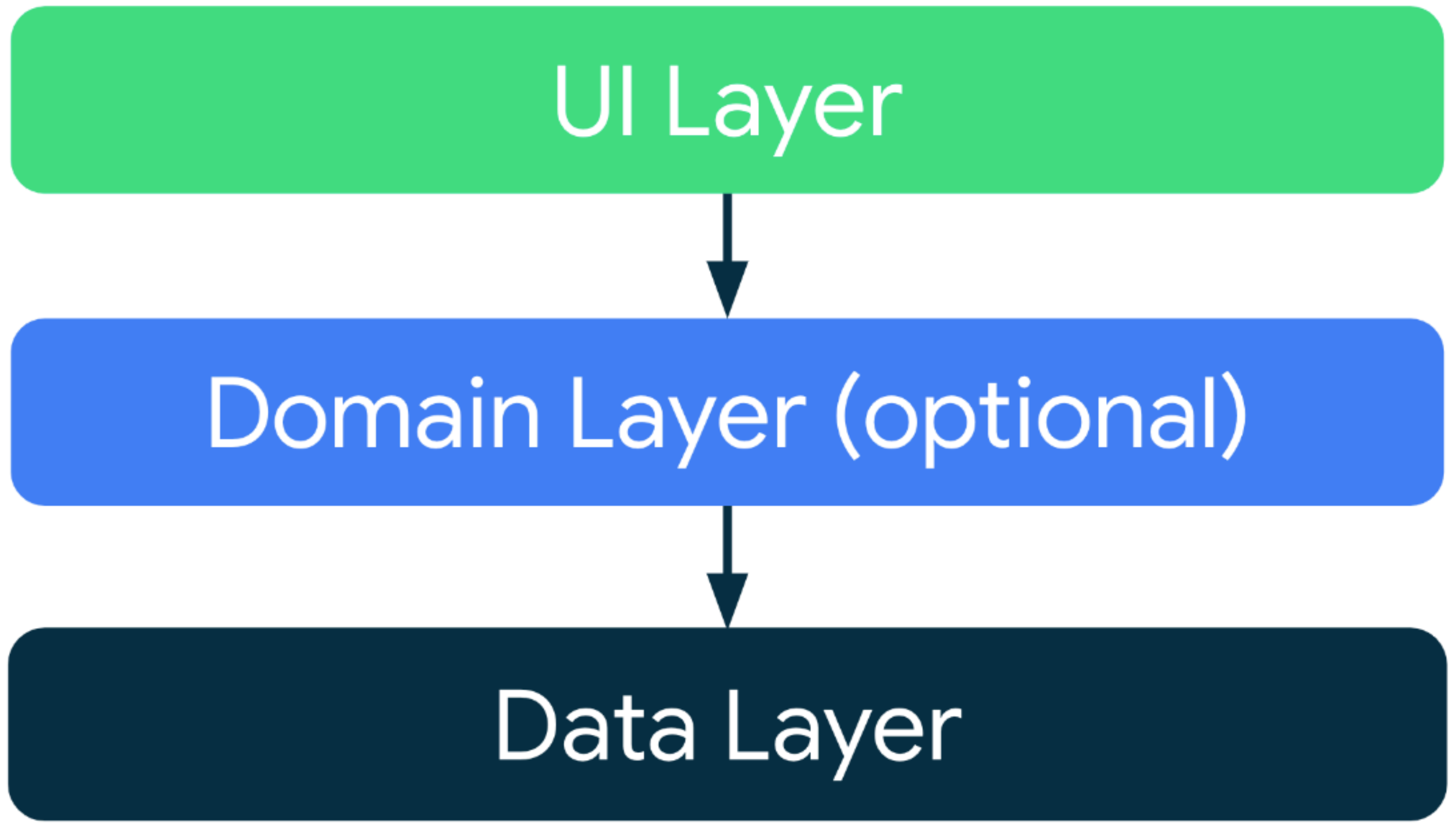
Separation of concerns

Application arhitecture

Composable functions

Business logic

Database/API connections

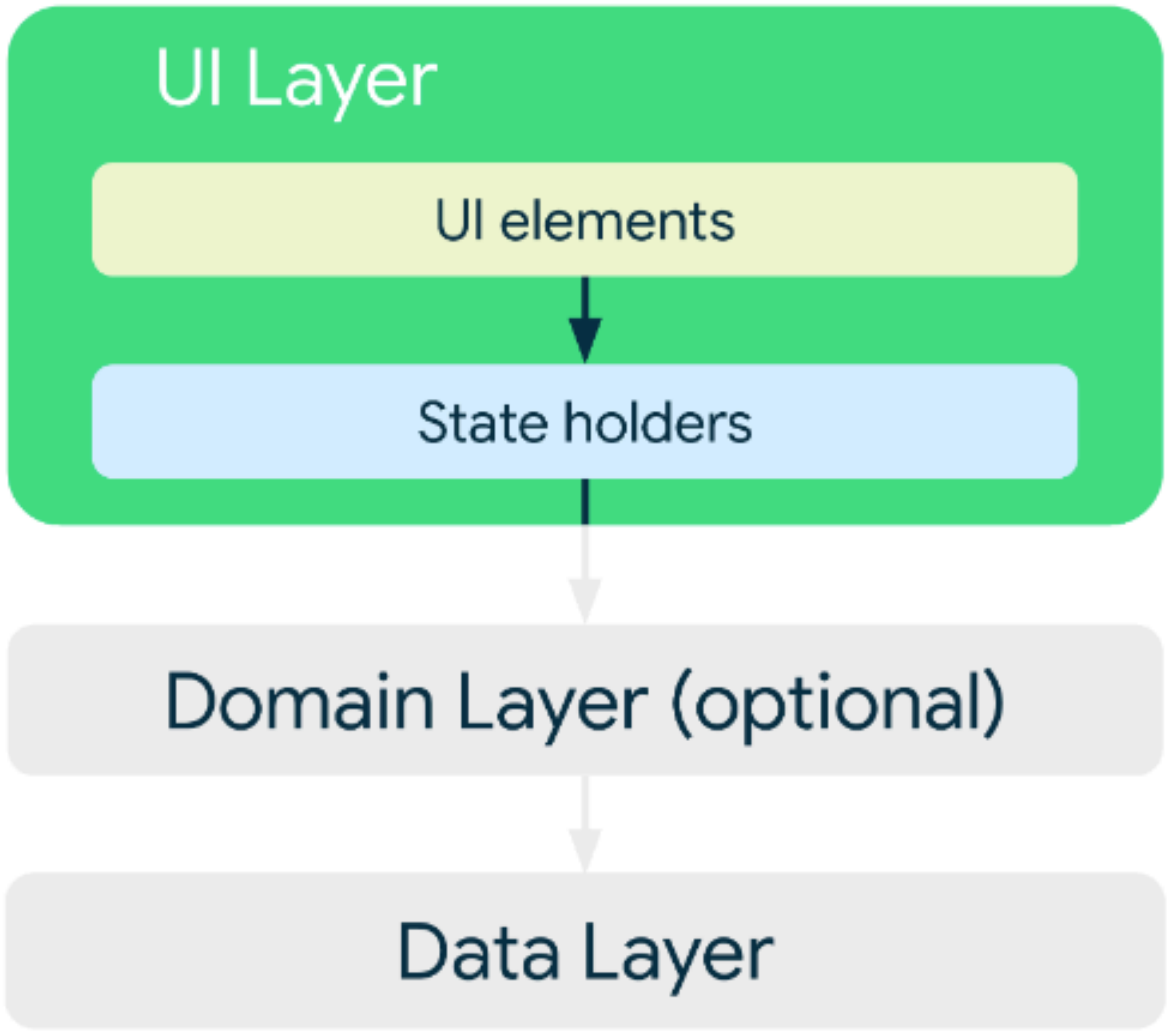


Separation of concerns

Application arhitecture

Composable functions

ViewModels



Summary & Example

State hoisting, SoP, unidirectional data flow

- Moving state from composable functions to objects ViewModels
- Separating the concerns such that UI elements are stateless and viewmodels are stateful
- Ensuring a unidirectional dataflow state is moving from state to UI and events are moving from UI to state
- Example: https://github.com/nicklasdean/court_counter_text_mirror

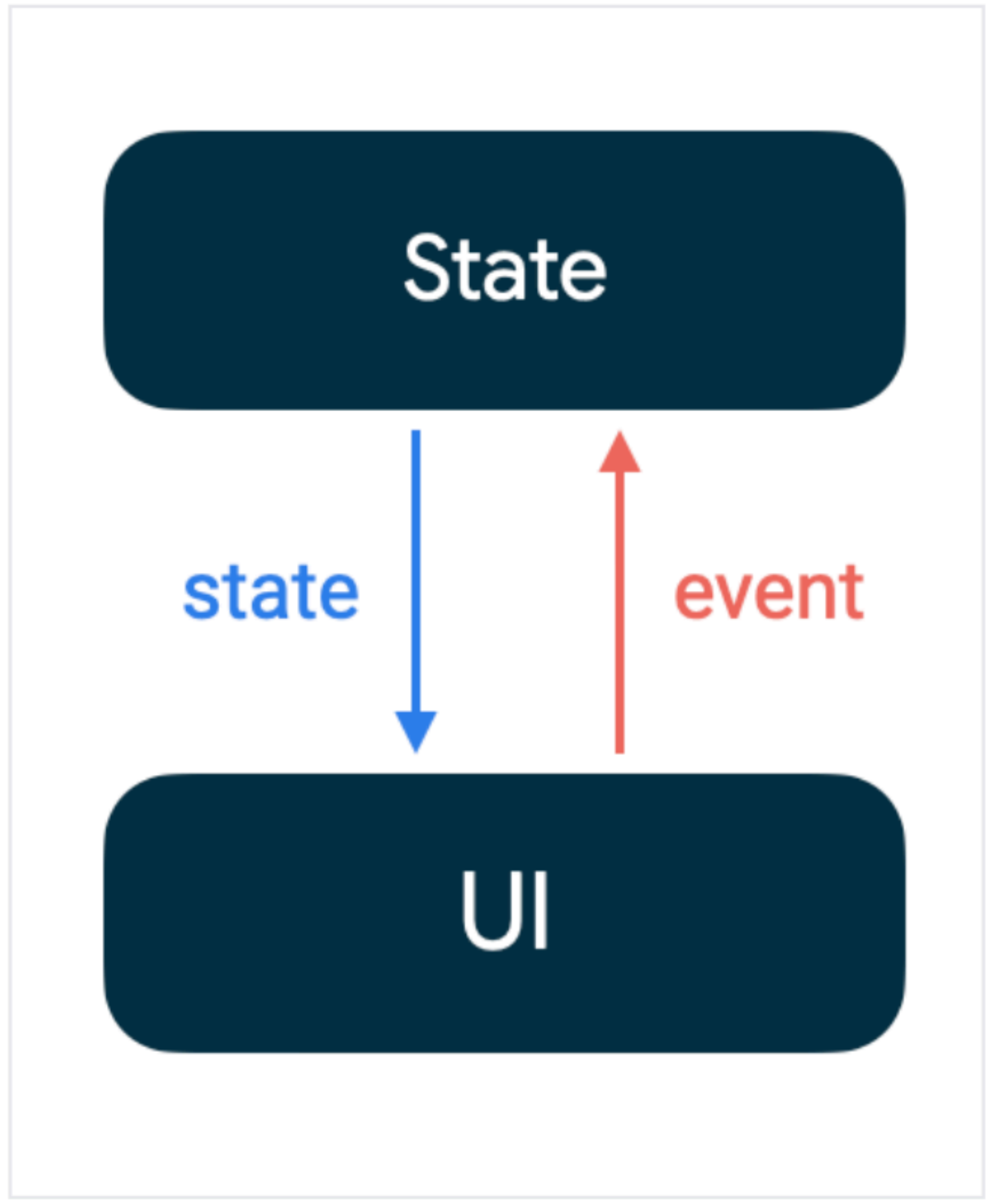
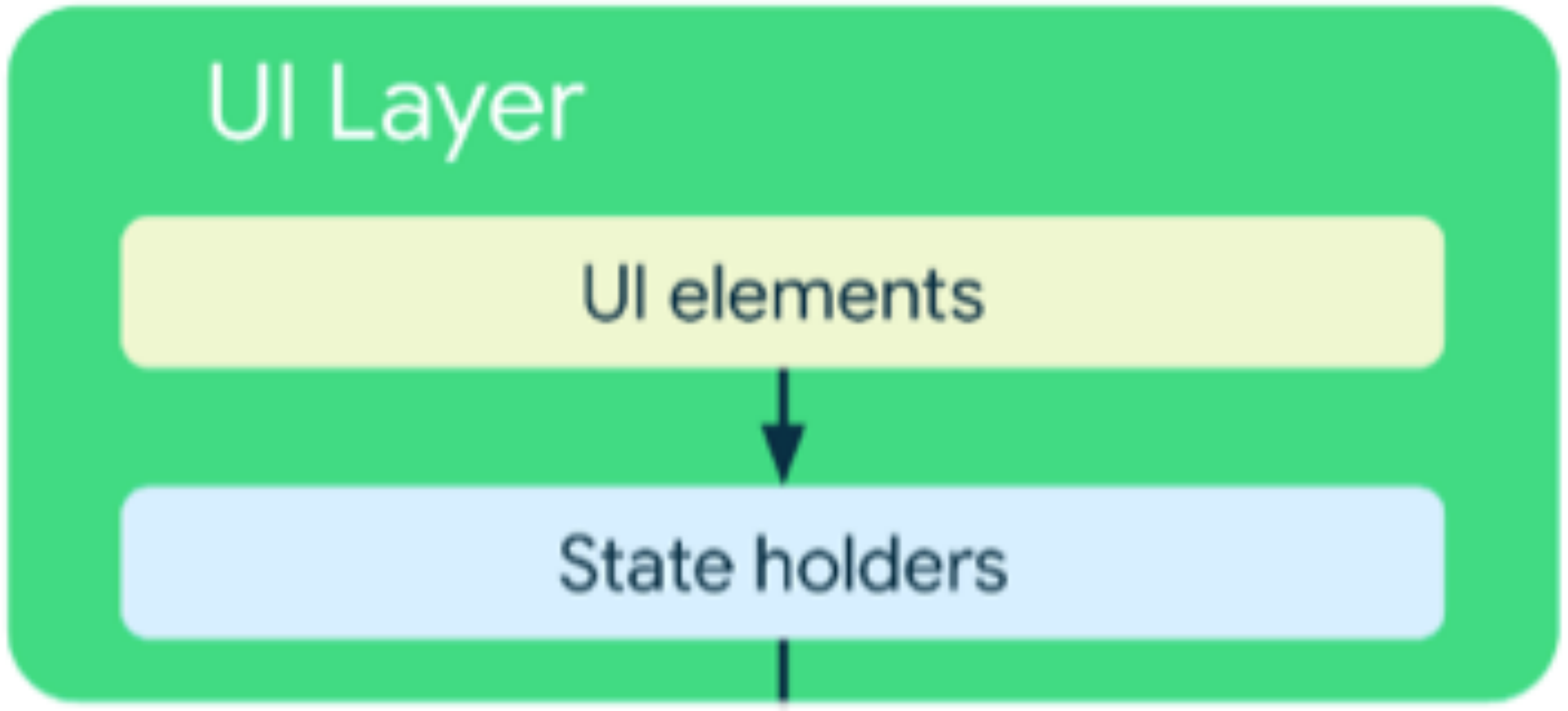


Figure 1. *Unidirectional data flow.*


UI Layer
UI elements
State holders