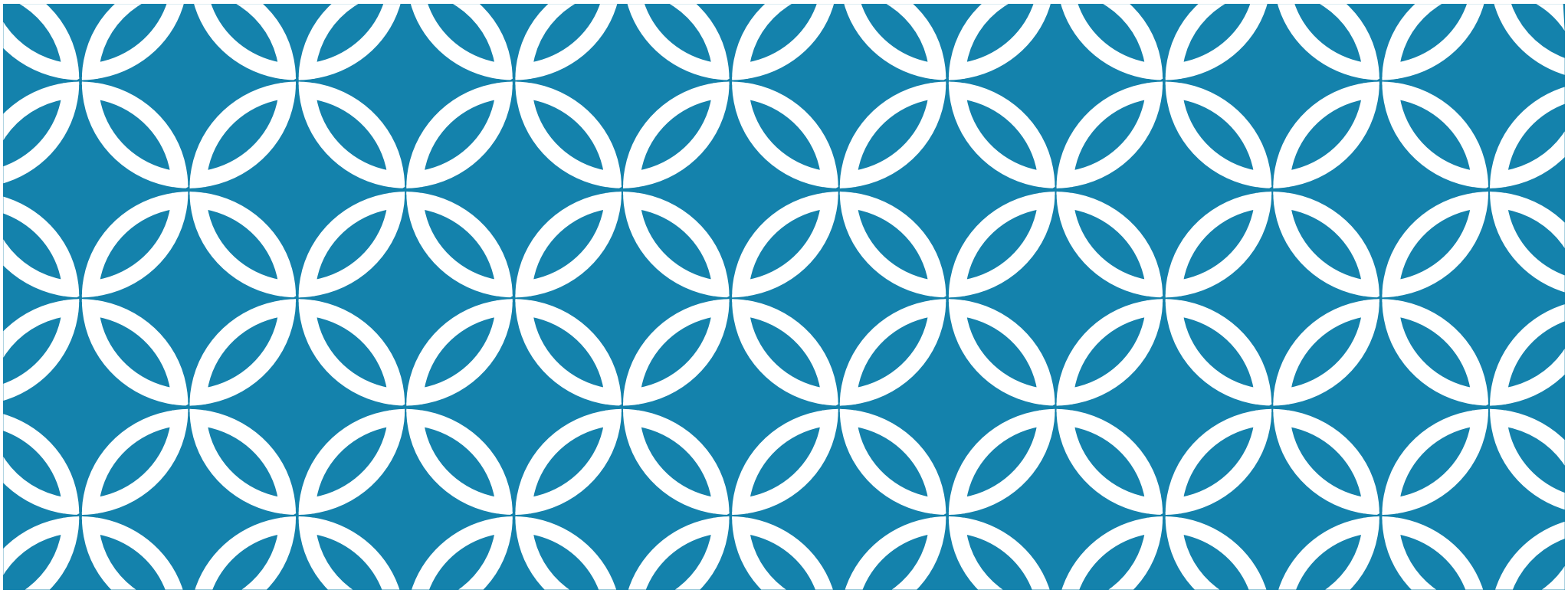
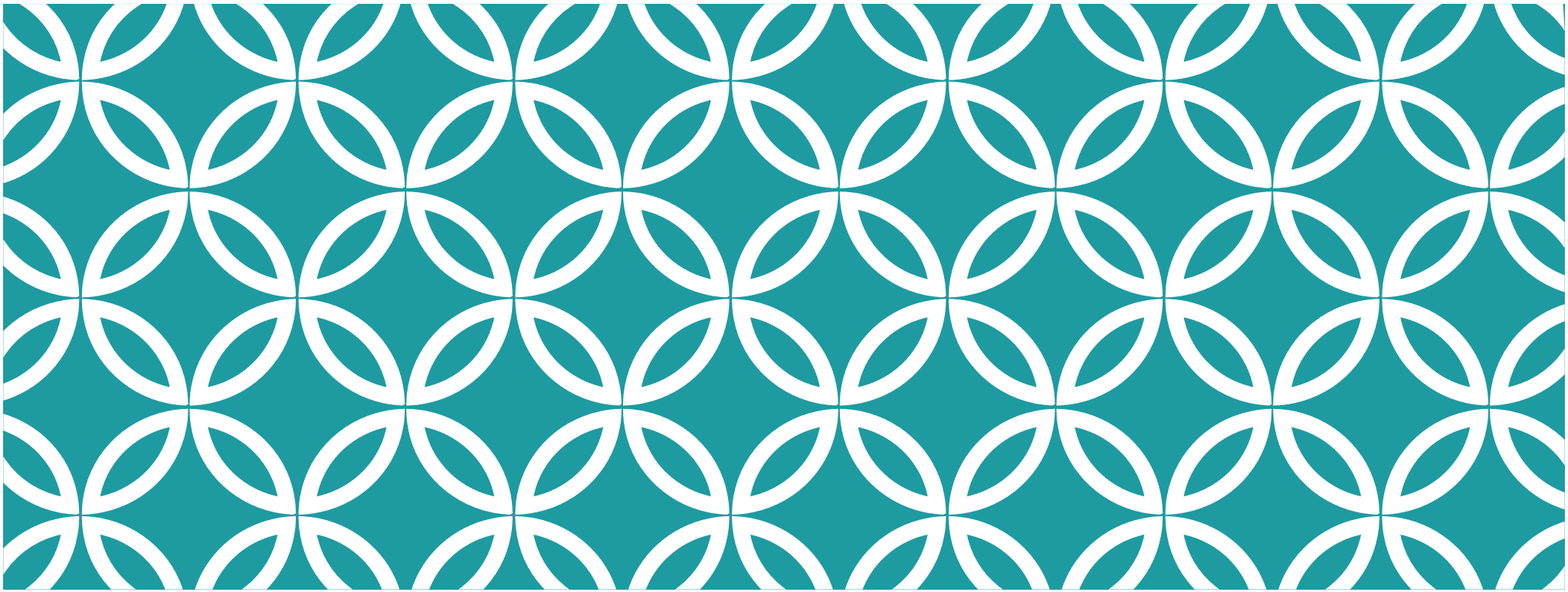


# DESKTOP APPLICATIONS

CS 346: Application Development



# FEATURES

CS 346: Application Development

# MODERN FEATURES

1. **Graphical User Interface.**

Interactive graphical interfaces are a major part of a modern application. We will spend considerable time discussing user interface toolkits that make it easy to build these interfaces.

Your application should support:

- Interaction using standard controls e.g., buttons, panels, images, scrollbars, and so on. We'll discuss how to use standard controls from a GUI toolkit.
- Standard navigation conventions for the platform. e.g., breadcrumb navigation for mobile, Window manipulation for desktop.
- Rich data, animations and other design elements that add to the aesthetics and appeal of the platform.

# MODERN FEATURES

## 2. **Standard interaction**

For mobile applications, support standard gestures to interact with the application:

- Tap to select elements on-screen.
- Drag to move or interact with elements.
- Swipe to scroll up/down.
- Pinch to zoom in/out.

For desktop applications, support keyboard and mouse:

- Keyboard shortcuts. Users should be able to use the keyboard for navigation & common tasks.
- Mouse support. The mouse should be used for most selection tasks, and for manipulating data (i.e., a right-click context menu, or dropdown menu).

In both cases, users should be able to navigate through multiple screens or sections of the application using a breadcrumb trail, or a series of links.

# MODERN FEATURES

3. **Rich Data Manipulation**

We expect to be able to manipulate data in a variety of standard ways:

- **Cut-Copy-Paste**. You should be able to use these commands in both desktop and mobile applications to manipulate text and image data.
- **Undo-Redo**. You should be able to undo and redo changes to your data. This is a common feature in desktop applications and is becoming more common in mobile applications (usually based on the features of the application you are building).
- **Drag-Drop.** When an application requires you to move data from one place to another, you should be able to drag and drop it. e.g., dragging an image from your file system into a dialog box.

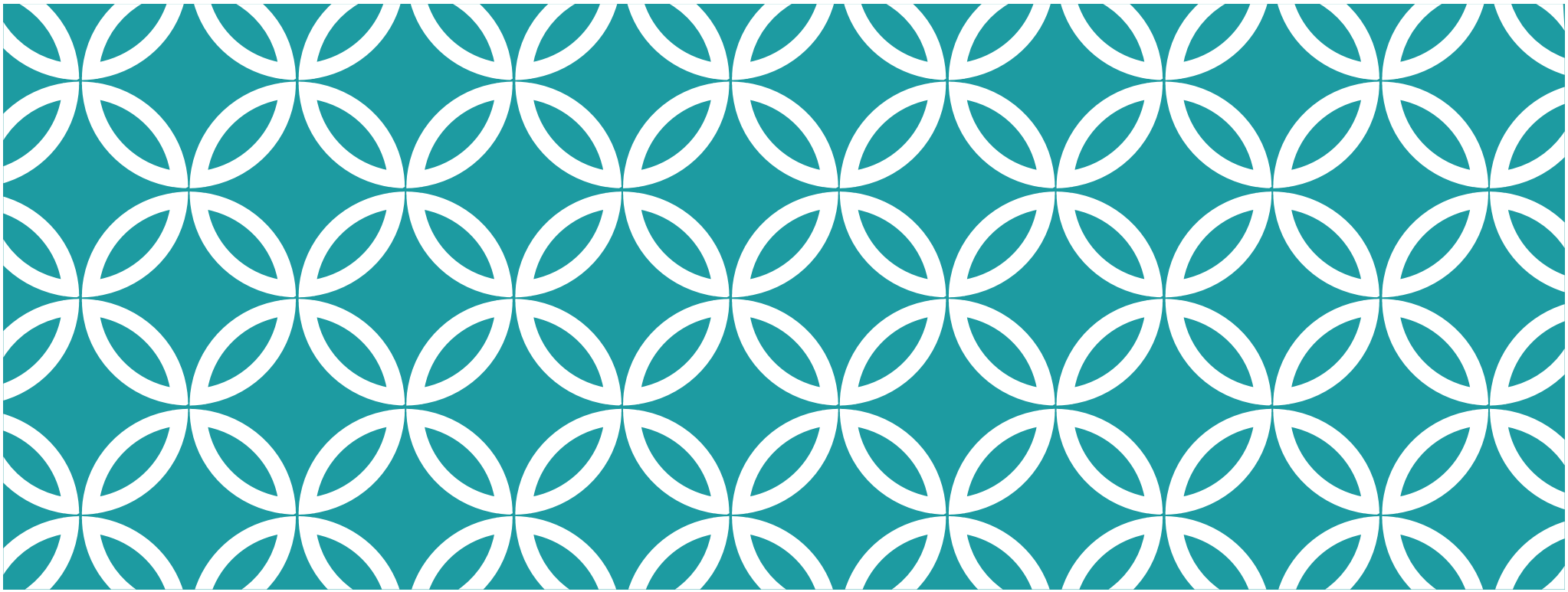
# MODERN FEATURES

4. **Database Support**

We should be able to store and retrieve data from a database. This can include a local database (like SQLite), or a shared online database (like MySQL or PostgreSQL). We should also be able to easily manipulate data into the correct storage format.

5. **Networking & Services**

Modern applications do not exist in a silo; they are expected to interact with other applications and online services to consume or share data. We will also discuss how to both design and consume services in a later section.



# COMPOSE > DESKTOP

CS 346: Application Development

# INSTALLING COMPOSE

**libs.versions.toml**

```
[plugins]
kotlin-jvm = {id = "org.jetbrains.kotlin.jvm", version.ref = “2.0.20"}
jetbrains-compose = {id = "org.jetbrains.compose", version.ref = “1.6.11"}
compose-compiler = {id = "org.jetbrains.kotlin.plugin.compose", version.ref = “2.0.20"}
```

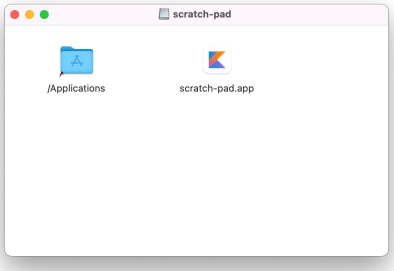
**build.gradle.kts**

```
plugins {
    alias(libs.plugins.jetbrains.compose)
    alias(libs.plugins.compose.compiler)
 }
dependencies {
   implementation(compose.desktop.currentOs)
}
```

# GRADLE TASKS FOR DESKTOP

Use the Gradle menu (`View` > `Tool Windows` > `Gradle`).

| Command | What does it do? |
| --- | --- |
| `Tasks` > `build` > `clean` | Removes temp files (deletes the `/build` directory) |
| `Tasks` > `build` > `build` | Compiles your application |
| `Tasks` > `compose desktop` > `run` | Executes your application (builds it first if necessary) |
| `Tasks` > `compose desktop` > `package` | Create an installer for your platform! |



# APPLICATION STRUCTURE

Your entry point for a desktop application is the main method. A desktop application needs to:

- use a **main method** as its entry point,
- declare a **top-level application scope,**
- declare one or more **windows** within that application scope.

```
fun main() {
    application {
        Window(
            onCloseRequest = ::exitApplication
        ) {
        }
    }
}
```

Minimum application definition (onCloseRequest must be specified).

# WINDOW POSITION/SIZE

Create a WindowState for the Window composable, and pass in the appropriate values.

```
fun main() {
    application {
        Window(
            title = "Window State Demo",
            state = WindowState(
                position = WindowPosition(0.dp, 0.dp),
                size = DpSize(300.dp, 200.dp)
            ),
            onCloseRequest = ::exitApplication
        ) {
            Text("This is a window")
        }
    }
}
```

# ADDING MENUS

```
fun main() = application {
    Window(onCloseRequest = ::exitApplication) {
        App(this, this@application)
    }
}

@Composable
fun App(
    windowScope: FrameWindowScope,
    applicationScope: ApplicationScope
) {
    windowScope.MenuBar {
        Menu("File", mnemonic = 'F') {
            val nextWindowState = rememberWindowState()
            Item(
                "Exit",
                onClick = { applicationScope.exitApplication() },
                shortcut = KeyShortcut(
                    Key.X, ctrl = false
                )
            )
        }
    }
}
```

# KEYBOARD INPUT

```
fun main() = application {
    Window(
        title = "Key Events",
        state = WindowState(width = 500.dp, height = 100.dp),
        onCloseRequest = ::exitApplication,
        onKeyEvent = {
            if (it.type == KeyEventType.KeyUp) {
                println("Window handler: " + it.key.toString())
            }
        }
    ) {
        MaterialTheme {
            Frame()
        }
    }
}
```



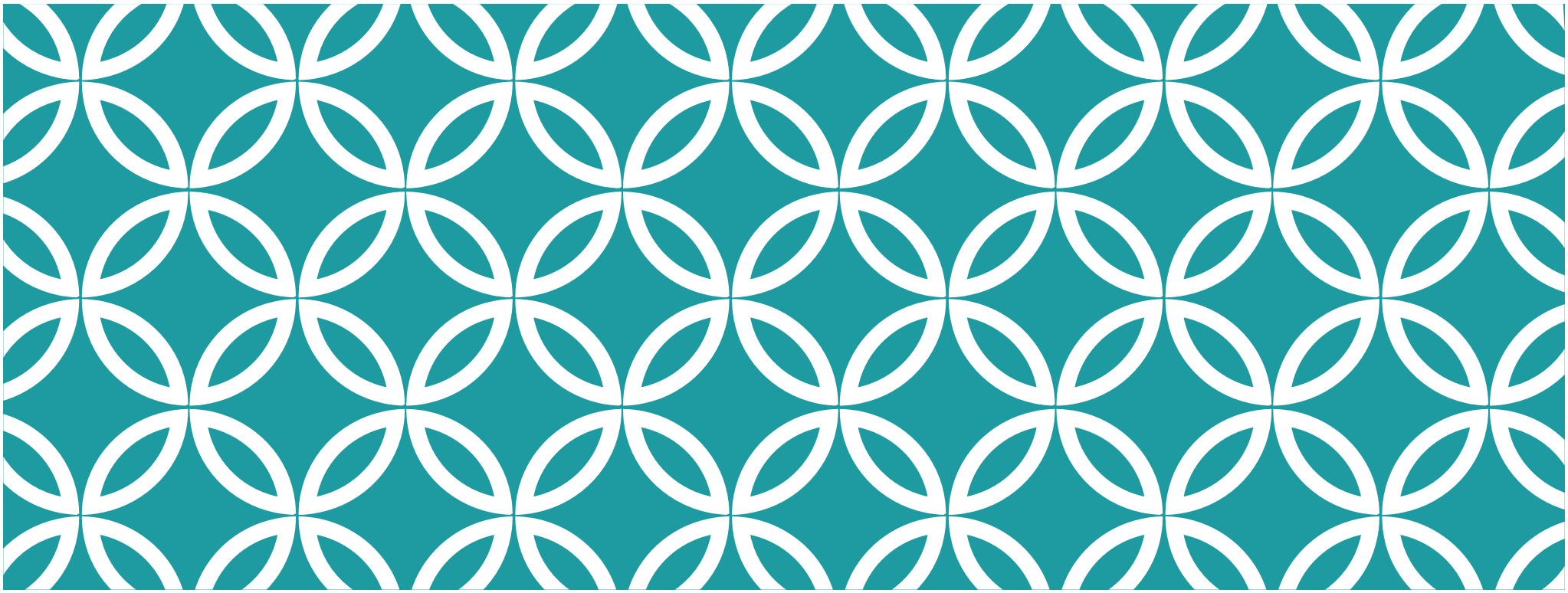
It's just a new event type

# MOUSE INPUT

```
Box(
    modifier = Modifier
        .background(Color.Magenta)
        .fillMaxWidth(0.9f)
        .fillMaxHeight(0.2f)
        .combinedClickable(
            onClick = { text = "Click! ${count++}" },
            onDoubleClick = { text = "Double click! ${count++}" },
            onLongClick = { text = "Long click! ${count++}" }
        )
)
```

# MOUSE MOVEMENT

```
var color by remember { mutableStateOf(Color(0, 0, 0)) }
Box(
    modifier = Modifier
        .wrapContentSize(Alignment.Center)
        .fillMaxSize()
        .background(color = color)
        .onPointerEvent(PointerEventType.Move) {
            val position = it.changes.first().position
            color = Color(position.x.toInt() % 256, position.y.toInt() % 256, 0)
        }
)
```



# ARCHITECTURE & DESIGN

CS 346: Application Development

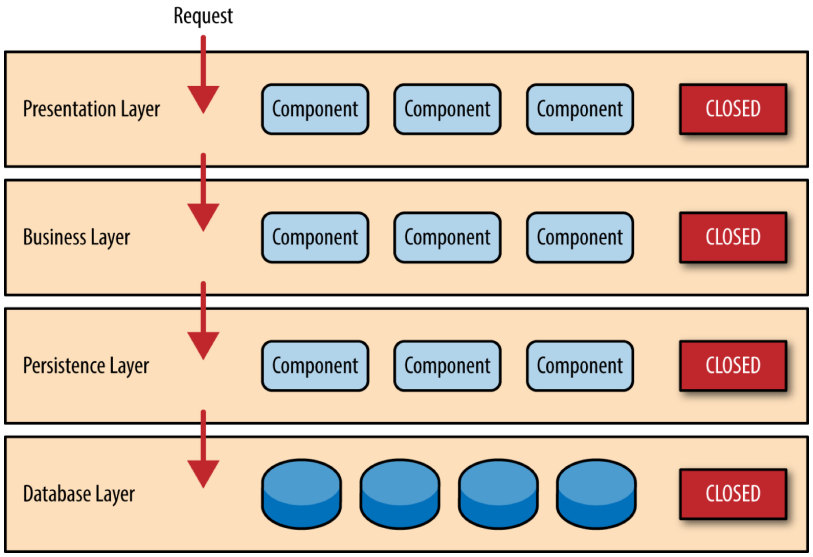
# MONOLITHIC: LAYERED

A **layered** or *n-tier* architecture is a very common architectural style that organizes software into horizontal layers, where each layer represents a ***logical*** division of functionality.

Each layer has specific functionality that is presents to the layer above (i.e. lower layers provide services up the stack).

*Dependencies extend down*: lower-levels provide functionality that is consumed by higher-levels.

e.g. presentation uses business logic, but business layer doesn't know anything about the UI.

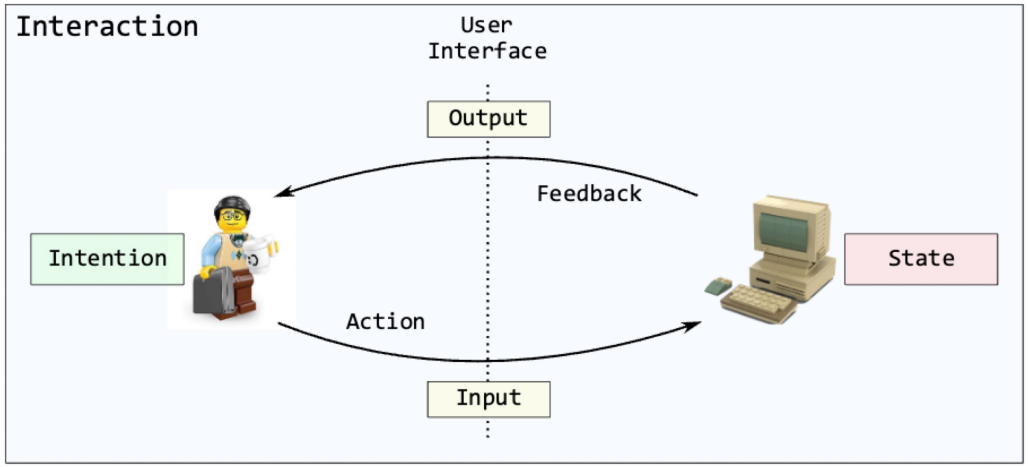



Advantages

- Layers isolated ("separation of concerns")
- High testability because components belong to specific layers in the architecture, and other layers can be mocked or stubbed.
- High ease of development.

# USER INTERACTION

People typically interact with technology to reach a goal. They determine what they need to do, and perform actions with the UI, which in turn provides feedback that lets them continue with their actions. It's an iterative cycle.

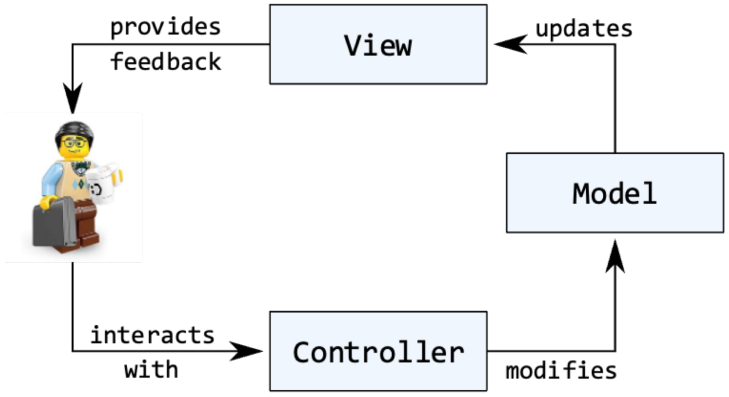



User interaction is a feedback loop. How do we model this?

# MODEL-VIEW-CONTROLLER (MVC) PATTERN

Developed at Xerox PARC for Smalltalk-80, and widely adopted.

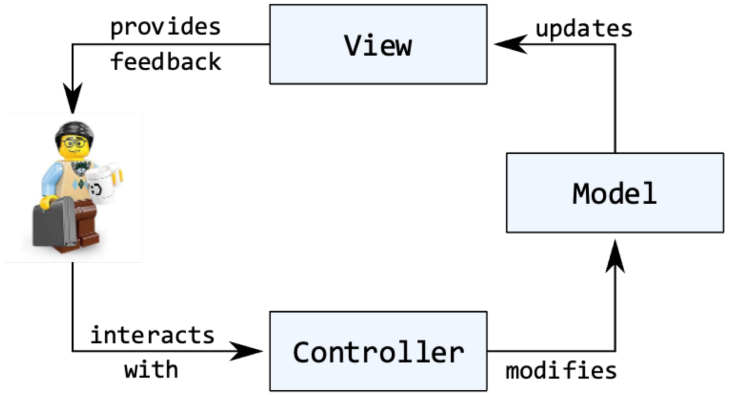
- **Model**: manages system state and its modification
- **View**: manages interface to provide feedback
- **Controller**: manages interaction to request system state modification




MVC models this interaction using the Observer pattern.

# MODEL-VIEW-CONTROLLER IMPLEMENTATION

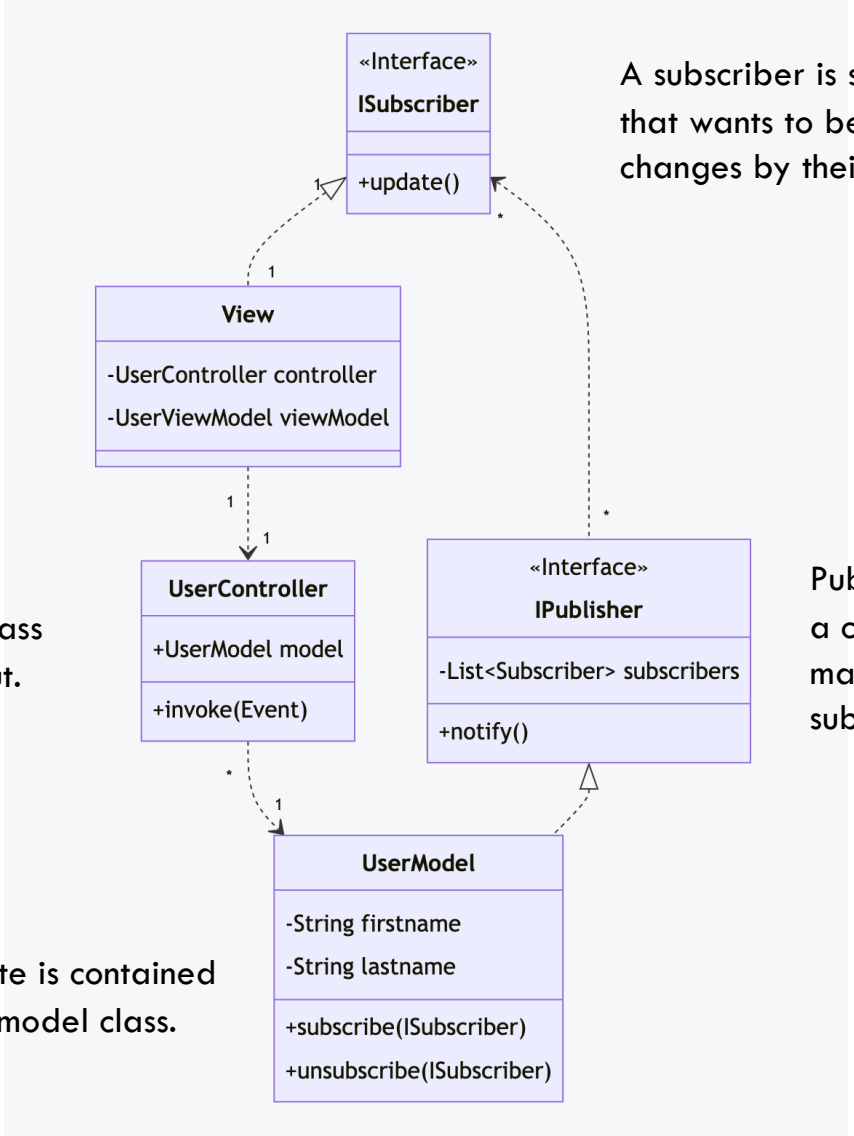
1. User performs an action on **Controller**. This can include keyboard or mouse input.
2. **Controller** asks **Model** to act upon input event. The controller interprets the intent of the request.
3. **Model** might change state and notify one or more **Views** that the state change occurred.
4. **View** retrieves updated state from **Model** and visualizes it for the User.




MVC models this feedback loop.

A subscriber is simply any class that wants to be notified of changes by their publisher.

We can have multiple views, each responsible for their own content.



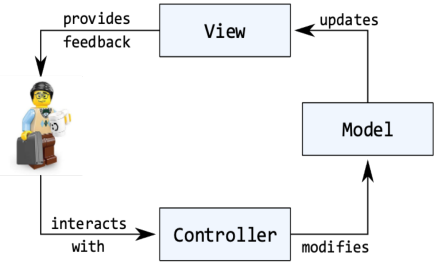

Controller class handles input.

Publishers commit to having a consistent method of managing lists of subscribers (views).

All state is contained in the model class.

# MVC BENEFITS?

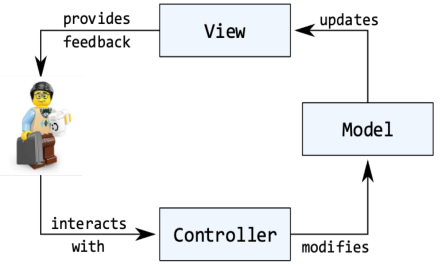
- **Separation of business logic and input/output**. Separation of concerns; ease of testing with discrete classes/responsibilities.
- **Support for multiple types of views**. e.g. simultaneously display data in both a chart and a table.
- The ability to **handle multiple types of inputs** e.g. joystick, mouse, trackpad, voice, retina-tracking, brain-computer interface…




MVC models this feedback loop.

# MVC CHALLENGES?

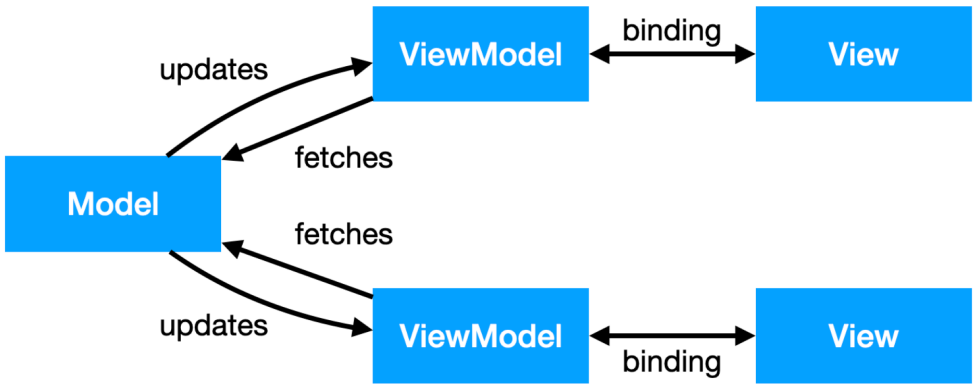
- It doesn't account for modern, complex interfaces
  - e.g., multiple views with their own data vs. hierarchical views with shared data
  - e.g., screen navigation, where screens are unloaded and swapped.
- Modern toolkits have also moved towards using binding mechanisms, where UI elements are automatically refreshed as application state changes.




MVC models this feedback loop.

# MODEL-VIEW-VIEWMODEL (MVVM)

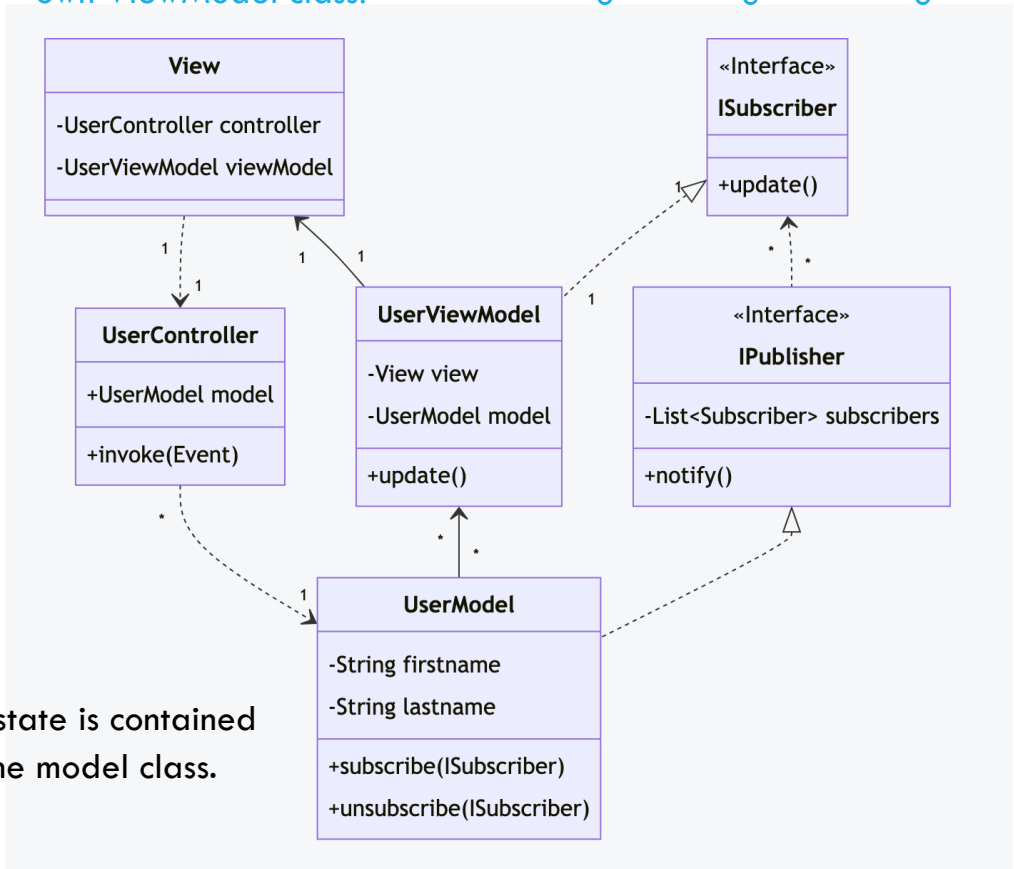
Model-View-ViewModel is an MVC variant that introduces another abstraction layer between the primary model and each view. This allows is to localize state for individual views/screens.



View state is stored in a ViewModel. Each View has its own ViewModel class!

The ViewModel subscribes (instead of the View). The View is notified of VM changed through a binding mechanism.

We can have multiple views, each responsible for their own content.



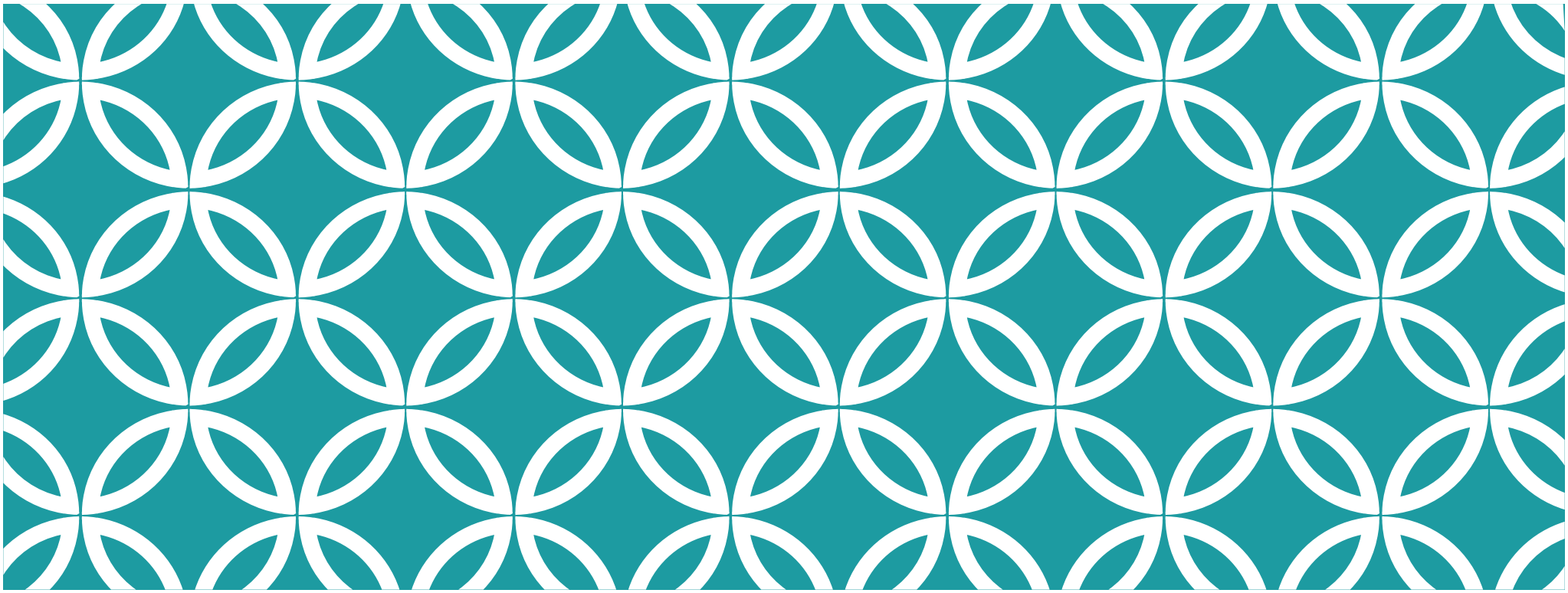

Controller class handles input.

Publishers commit to having a consistent method of managing lists of subscribers (views).

All state is contained in the model class.

```kotlin
fun main() = application {
    val userModel = UserModel()
    val userViewModel = UserViewModel(userModel)
    val userController = UserController(userModel)

    Window(
        title = "MVC Demo",
        state = WindowState(
            position = WindowPosition(Alignment.Center),
            size = DpSize(275.dp, 200.dp)
        ),
        resizable = false,
        onCloseRequest = ::exitApplication
    ) {
        UserView(userViewModel, userController)
    }
}
```



# CLEAN ARCHITECTURE

CS 346: Application Development

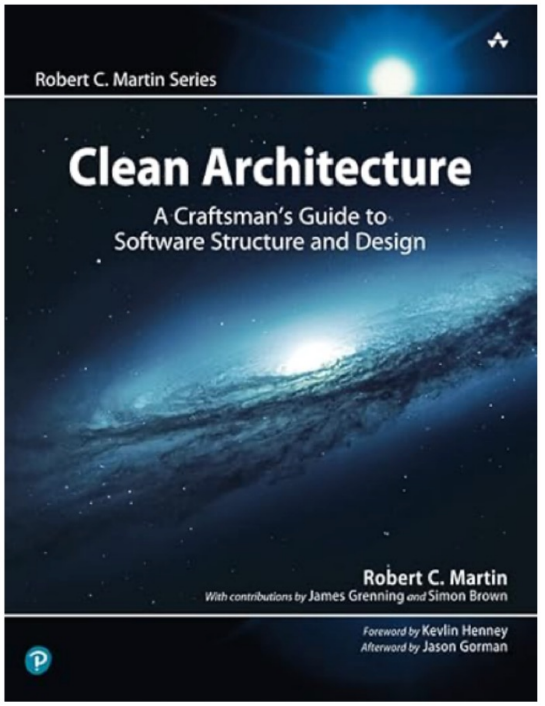
# CLEAN ARCHITECTURE

**Clean architecture** is an architectural style that is commonly used with modern mobile and desktop applications.

It was introduced by Robert C. Martin ("the SOLID guy") in 2012, and refined over the subsequent years (see his original blog post and the Clean Architecture book).

Clean architecture

- Generalized version of a Layered Architecture.
- Can handle multiple generalized services.
- Designed to solve cohesion and coupling issues.

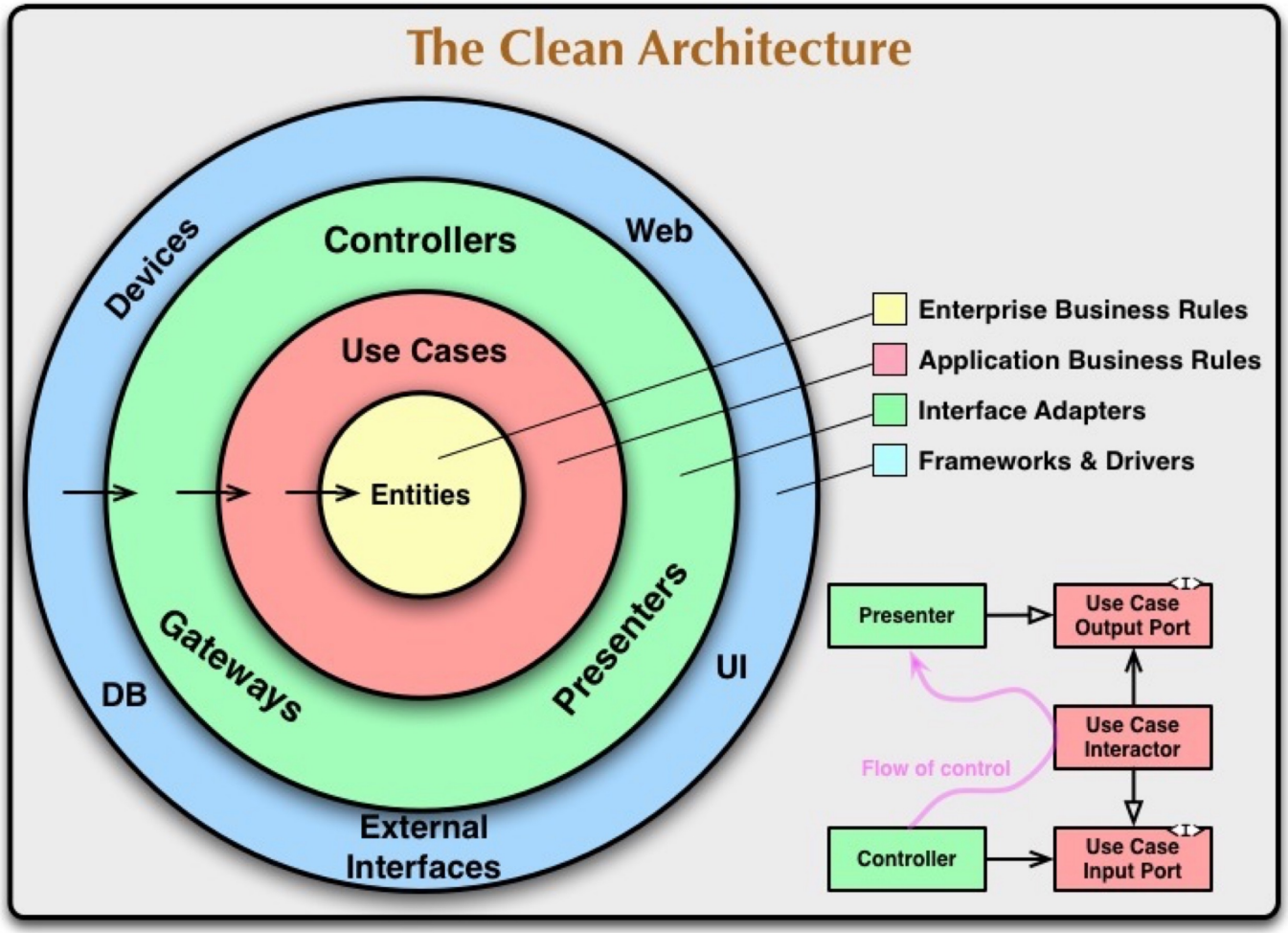



- Robert C. Martin. 2017. Clean Architecture.

# WHAT ARE THE BENEFITS?

Layering our architecture really helps to address our earlier goals (reducing coupling, setting the right level of abstraction). Additionally, it provides these other specific benefits:

- **Independent of frameworks**. The architecture does not depend on a particular set of libraries for its functionality. This allows you to use such frameworks as tools, rather than forcing you to cram your system into their limited constraints.
- **Testable**. The business rules can be tested without the UI, database, web server, or any other external element.
- **Independent of the UI**. The UI can change easily, without changing the rest of the system. A web UI could be replaced with a console UI, for example, without changing the business rules.
- **Independent of the database**. You can swap out Oracle or SQL Server for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.




- Robert C. Martin. 2017. Clean Architecture.

# LAYERS

**Entities** (Domain Models)

These are data classes that reflect your problem domain, e.g. classes like Customer, Invoice, Note. These are the core classes of your application. They don't have any external dependencies!

**Use Cases** (Application)

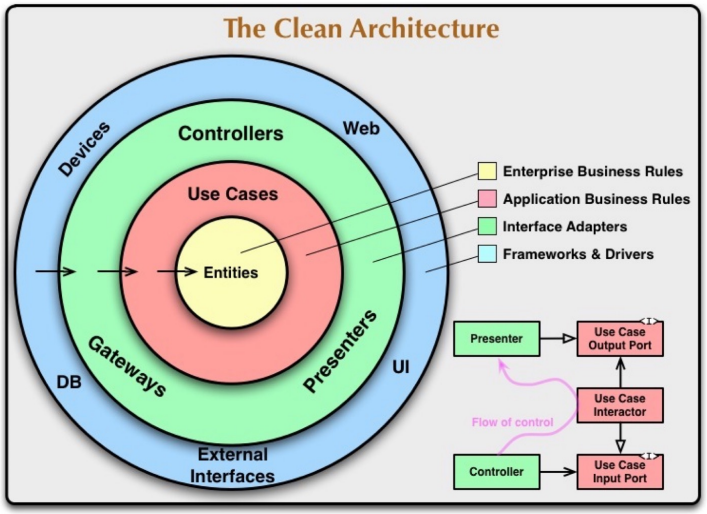
These are application-specific classes that build on the core entities. They facilitate data flow to and from the entities. This is also where you would implement core application logic.

**Controllers** (Infrastructure)

These are interface adapters that map data from the entities or use cases, to a format that is required by the external layers.

**Interfaces** (Frameworks)

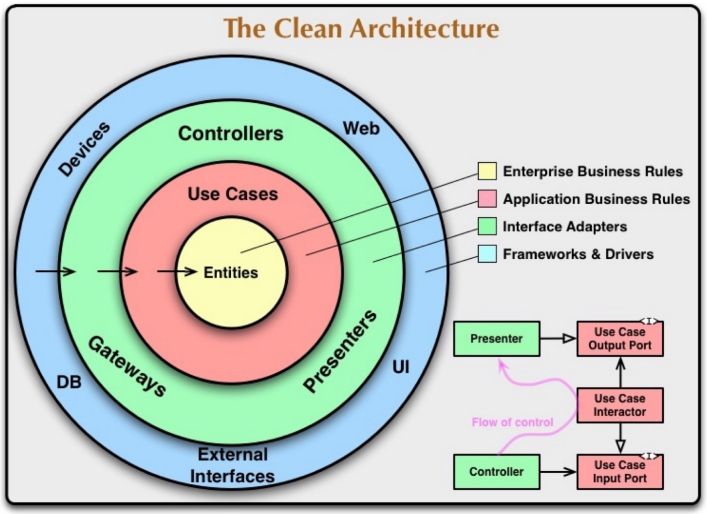
These are frameworks to access external services.



# DEPENDENCY RULE

The **dependency rule** describes relationship between layers: nothing in the inner circle can know about the outer circles. **Dependencies are only allowed to go from the outside to the inside;** outside classes can only refer to the same or a more inner layer.

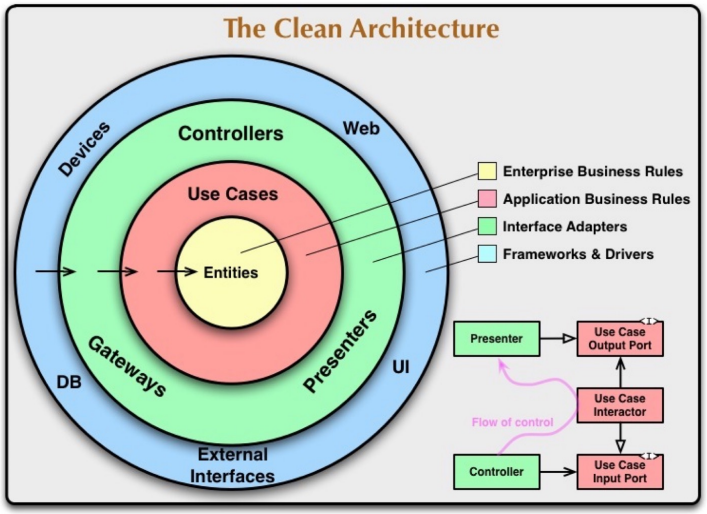
- This means that the innermost classes (entities) don't use any external frameworks, since that would introduce dependencies.
- External frameworks and libraries are pushed to the outside of the architecture.



# DEPENDENCY INVERSION

The diagram above shows flow-of-control: which classes call into which other classes. In this case, outside classes call into inner classes. This also represents the source code dependencies. We handle source code dependencies with dependency inversion.

- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.



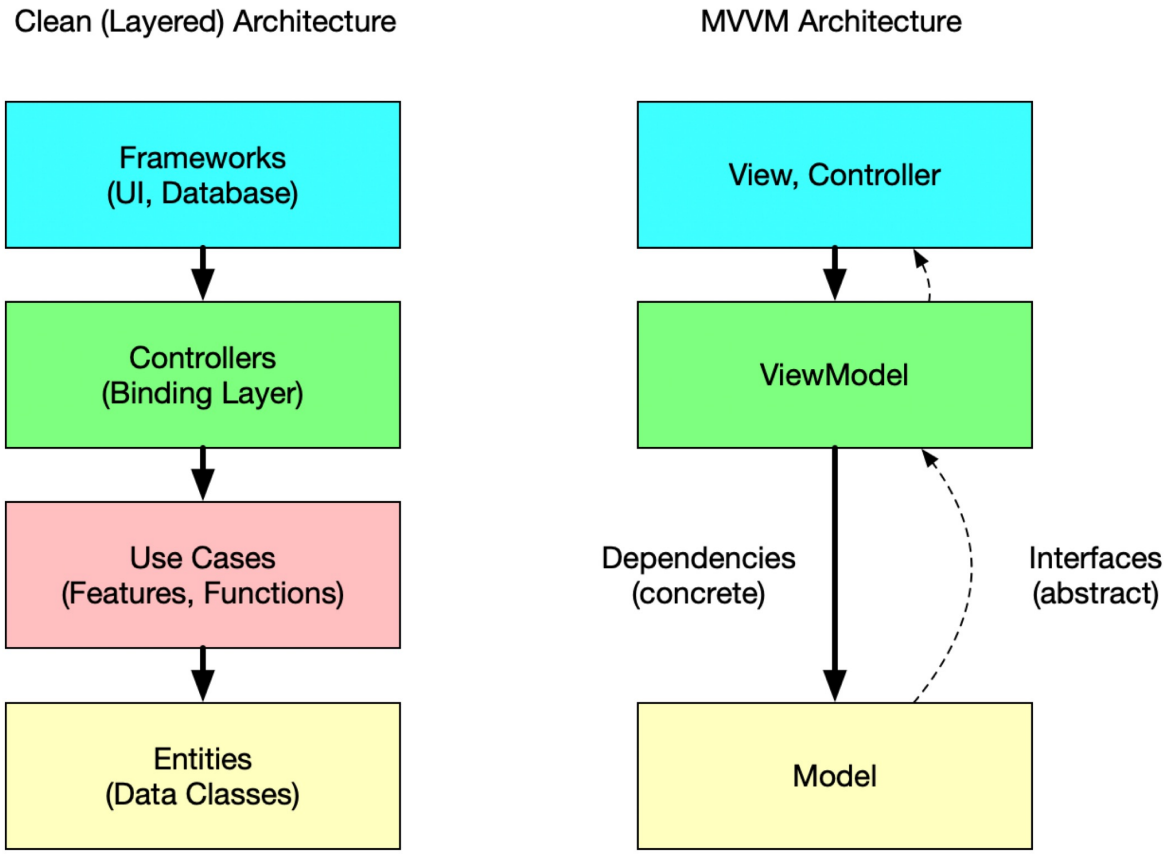
# SOURCE CODE: MODULARITY

How do we accomplish this in code?

- We set up packages to separate classes into modules.
- We use interfaces to describe the relationship between classes.

For example, we would set up a source code structure like this, where each subdirectory is a package.

```
project
├── application
├── database
├── entities
├── presentation
└── service
```




MVVM can be used to model our Clean Architecture as well.
The Model contains application state, which it can propagate to individual ViewModels.