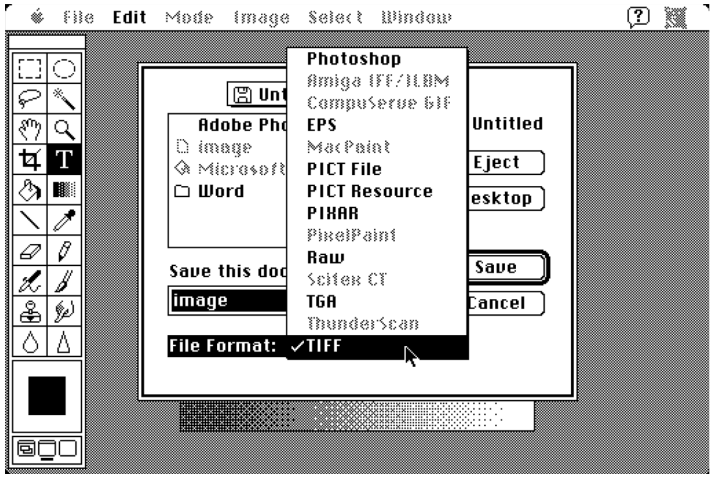
Building Desktop Applications

CS 346 Application Development

History: WIMP interfaces

- Paradigm designed at Xerox PARC in 1973.
  - See also desktop metaphor in computing.
- Popularized with Apple Macintosh in 1984.
- **W**indows, **I**cons, **M**enus, **P**ointer
  - Each program runs in a self-contained and isolated **Window**.
  - **Icons** represent actions e.g., printer, trash can.
  - **Menus** represent commands that can be issued by the user.
  - **Pointer** refers to the mouse-pointer.
- Advantages: Discoverable, Simple, Familiar.
- Disadvantages: Resources, Accessibility.



Macintosh user interface from 1984.

What differs from console?

1. Graphical user interfaces (GUI)
- Applications constrained to “windows”.
- Output via high-resolution graphics, animation.

2. Navigation
- Maneuver through windows/screens.

3. Keyboard + mouse interaction
- Keyboard shortcuts.
- Menus e.g., File, Edit, View, Window.
- Features: undo/redo, copy/paste, drag/drop.

Toolkits

- **Kotlin Multiplatform**: Compiler technologies that allow you to target multiple native platforms and share code between them.
  - e.g., sharing between Android and JVM.
- **Jetpack Compose**: Google's UI toolkit that was originally designed for Android development.
- **Compose Multiplatform**: A port of Jetpack Compose to other platforms, including desktop and iOS.

- We're *mostly* going to talk about Compose Multiplatform.
  - Standard functionality we've discussed + desktop specific additions.

Getting started

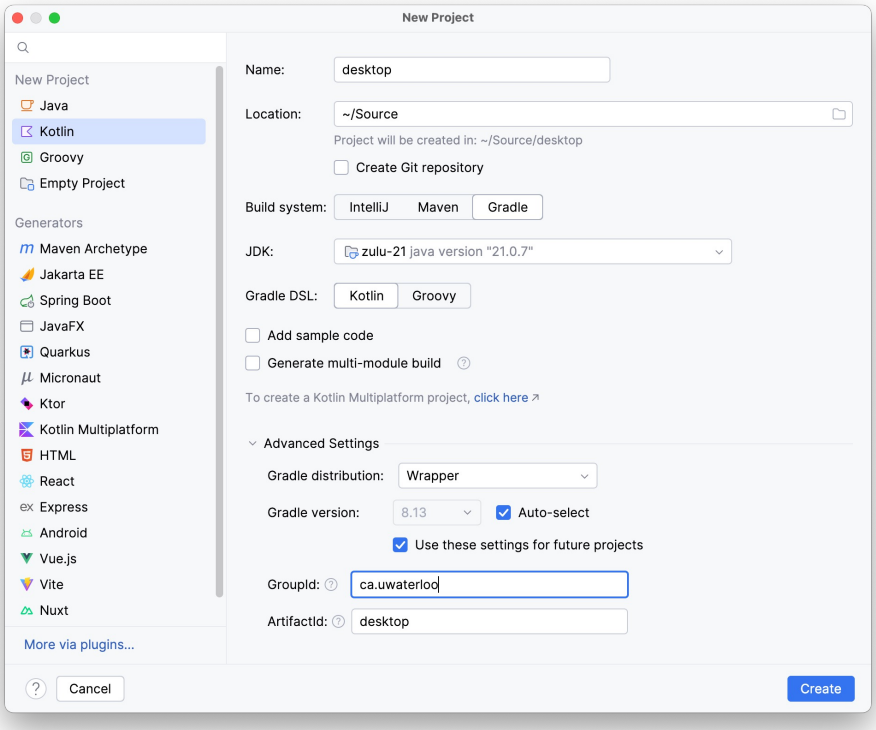
How to create a desktop project.

Step 1: Create a Project

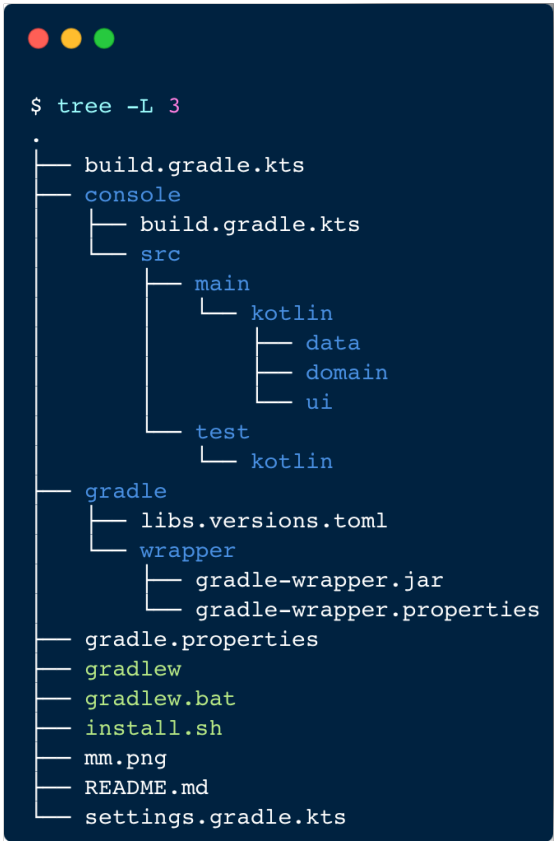
A desktop project is simply a regular Kotlin project with the Compose Dependencies added.

See course website:

Reference > Getting Started > Gradle project



Step 2: Modify the directory structure



build.gradle.kts – the main build configuration file

src/main/kotlin – source code goes here
src/test/kotlin – unit tests go here

libs.versions.toml– version catalog that lists dependencies

gradlew – script to run gradle tasks

Step 3: Add dependencies

You will need to update the **version catalog**:

libs.versions.toml

```
[versions]
kotlin-ver = "2.0.20"
compose-plugin = "1.6.11"

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

Step 3: Add dependencies

You will need to update the **build.gradle.kts**:

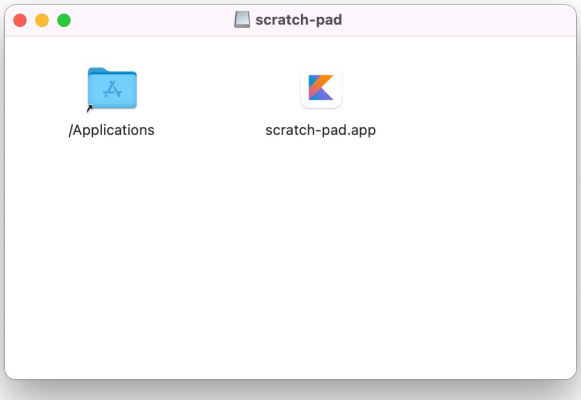
build.gradle.kts

```
plugins {
    alias(libs.plugins.kotlin.jvm)
    alias(libs.plugins.jetbrains.compose)
    alias(libs.plugins.compose.compiler)
}


dependencies {
    implementation (compose.desktop.currentOs)
}

compose.desktop {
    application {
        mainClass = “MainKt"
    }
}
```

Desktop Gradle Tasks

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!



Desktop Composables

Components specific to Compose Multiplatform and desktop applications.

Window

- One or more windows are required aka top-level composable.
- “Regular” window behavior is handled by the OS/toolkit.
  - e.g., resizing, dragging, minimize, maximize.
- Can replace window contents dynamically as-needed.

- Parameters
  - `Title` - Window title
  - `onCloseRequest` - lambda or function name to call on close
  - `State` - WindowState, including dimensions, position
  - `contents` - Window contents (also pass as trailing lambda)

Window

Independent, each has its own scene-graph.

```
fun main() {
    application {
        MaterialTheme {
            Window(
                title = "Window 1",
                onCloseRequest = ::exitApplication
            ) {
                Text("This is a window")
            }

            Window(
                title = "Window 2",
                onCloseRequest = ::exitApplication
            ) {
                Text("This is also a window")
            }
        }
    }
}
```

samples/desktop/desktop-compose -> run **MultipleWindows** main method

(WindowState)

Controls position, size.

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

samples/desktop/desktop-compose -> run **WindowState** main method

Dialog Box

Foreground modal window

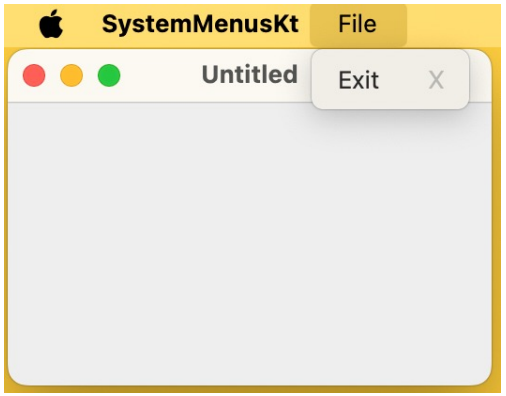
```
Window(
    title = "Main Window",
    onCloseRequest = ::exitApplication,
    state = WindowState(position = WindowPosition(Alignment.Center))
) {

    var isDialogOpen by remember { mutableStateOf(false) }
    Button(onClick = { isDialogOpen = true }) {
        Text(text = "Open dialog")
    }

    if (isDialogOpen) {
        DialogWindow(
            title = "Dialog Window",
            onCloseRequest = { isDialogOpen = false },
            state = rememberDialogState(position = WindowPosition(Alignment.Center))
        ) {
            Text("Dialog text goes here")
        }
    }
}
```

samples/desktop/desktop-compose -> run **Dialogs** main method

System Menu

OS determines position

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

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

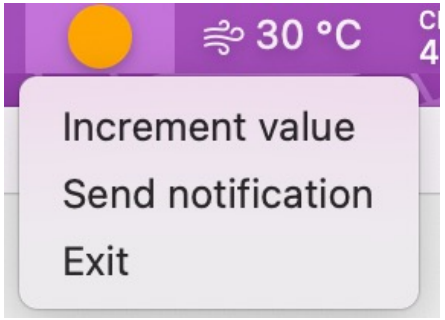


samples/desktop/desktop-compose -> run **SystemMenus** main method

System Tray

Taskbar or system tray icon

```
val trayState = rememberTrayState()
val notification = rememberNotification(
    "Notification", "Message from MyApp!"
)

Tray(
    state = trayState,
    icon = TrayIcon,
    menu = {
        Item("Increment value", onClick = { count++})
        Item("Send notification", onClick = {
            trayState.sendNotification(notification)
        })
        Item("Exit", onClick = { isOpen = false })
)
```

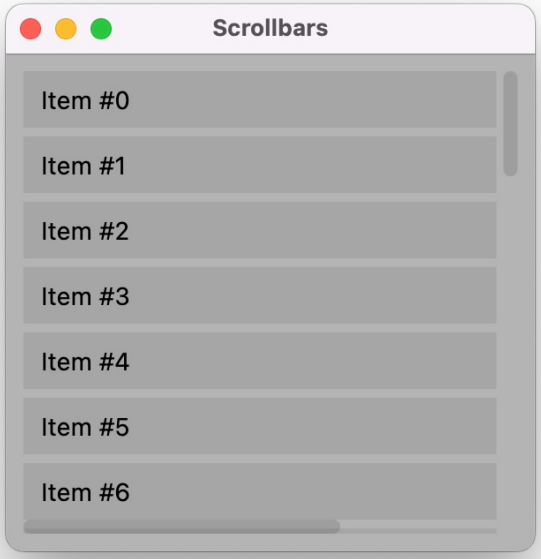


samples/desktop/desktop-compose -> run **SystemTray** main method

Scrollbar

Explicit, work with mouse/kb

```
Box(
    modifier = Modifier
        .fillMaxSize()
        .verticalScroll(stateVertical)
        .padding(end = 12.dp, bottom = 12.dp)
        .horizontalScroll(stateHorizontal)
) {
    Column {
        for (item in 0..30) {
            TextBox("Item #$item")
        }
    }
}

VerticalScrollbar(
    modifier = Modifier.align(Alignment.CenterEnd)
        .fillMaxHeight(),
    adapter = rememberScrollbarAdapter(stateVertical)
)

HorizontalScrollbar(
    modifier = Modifier.align(Alignment.BottomStart).fillMaxWidth,
    adapter = rememberScrollbarAdapter(stateHorizontal)
)
```



Navigation

How to move between screens?

Jetpack Navigation Concepts

The standard navigation library for Android is Jetpack Navigation.

It's been ported to desktop as well!

Common terms:

- A **navigation graph** describes all of the possible screen destinations and connections between then.
- A **destination** is a node that you can navigate to. This can be a composable (screen), or a dialog, or a different navigation graph (for complex user interfaces).
- A **route** identifies a destination and defines how to navigate to it.

Jetpack Navigation Library

The Navigation library represents the user's path as a stack of destinations. You can use this to move forward/backwards through navigation history.

Core classes:

- **NavController**: provides APIs for core functionality.
- **NavHost** is a composable that displays the contents for the current destination (determined by the navigation graph).
- **NavGraph** describes all possible destinations and the connections between them.

desktop-specific

standard navigation code

```
fun main() {
    application {
        MaterialTheme {
            Window(
                title = "Navigation",
                onCloseRequest = ::exitApplication
            ) {
                val navController = rememberNavController()
                NavHost(
                    navController = navController,
                    startDestination = ScreenA
                ) {
                    composable<ScreenA> {
                        ScreenAView(navController)
                    }
                    composable<ScreenB> {
                        val args = it.toRoute<ScreenB>()
                        ScreenBView(navController, args)
                    }
                    composable<ScreenC> {
                        val args = it.toRoute<ScreenC>()
                        ScreenCView(navController, args)
                    }
                }
            }
        }
    }
}
```

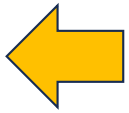
samples/desktop/desktop-compose -> run **Navigation** main method

Resources

How to load and use static content.

What are resources?

- Resources are static content e.g., images, sounds, fonts, strings that you might use in your application.
- These can be:
  - Bundled in your application e.g., image icon.
  - Loaded by your application at runtime.



Platform specific! e.g., resources could be on a HDD (desktop), SD card or cloud (mobile).

- Use cases:
  - **Interacting with content** e.g., the user directs your app to a specific resource like an image or video file to playback.
  - **Localization** e.g., replacing strings with locale-specific translations.
  - **Accessibility** e.g., replacing static images with high-contrast versions.

Resource Guidelines

- Almost all resources are read synchronously in the caller thread.
  - Assumes small/fast to load.
  - There are stream APIs for loading large files/resources (more on that later).
- Bundled resources
  - Place in the “resources/” folder in your source tree, and Kotlin can load them.
  - Will be packaged with your application by Gradle.
- Non-bundled (loaded at runtime)
  - Be careful: you will not have permission to access most locations.
  - Best practice: save files/resources in the user’s home directory

    e.g., `val homedir = System.getProperty("user.home")`

Interaction

Handling mouse and keyboard input on desktop.

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(it.key)
            }
        }
    ) {
        val text = remember { mutableStateOf("") }
        val textField = TextField(
            value = text.value,
            onValueChange = { text.value = it }
        )
    }
}
```

Window-level event handler

Widget-level event handler

samples/desktop/desktop-compose -> run **Interaction** main method

Mouse Clicks

```
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++}" }
        )
)
```

Multi-event handler necessary to handle all mouse inputs.

samples/desktop/desktop-compose -> run **Interaction** main method

Mouse Movement

```
var color by remember { mutableStateOf(Color(0, 0, 0)) }

Box(
    modifier = Modifier
        .background(Color.Magenta)
        .fillMaxWidth(0.9f)
        .fillMaxHeight(0.2f)
        .onPointerEvent(PointerEventType.Move) {
            val position = it.changes.first().position
            color = Color(
                position.x.toInt() % 256,
                position.y.toInt() % 256, 0
            )
        }
)
```

Drag handler. `it` contains a list of mouse movements.

samples/desktop/desktop-compose -> run **Interaction** main method

Testing

Considerations when writing unit tests for desktop applications.

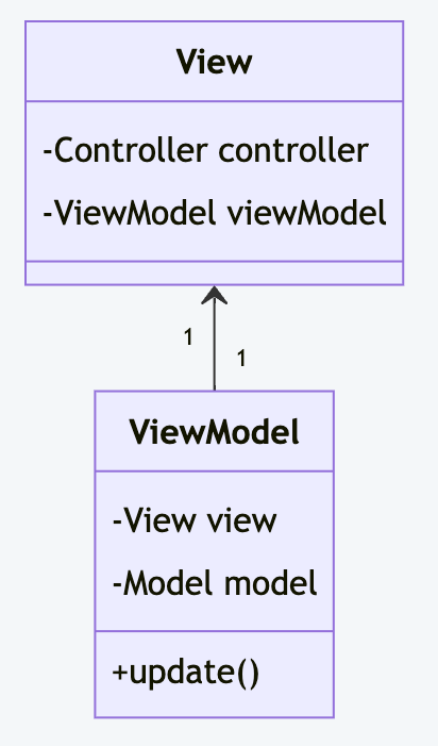
Adding GUI testing

Guidelines from earlier still apply.

Domain, Model, Service already covered.

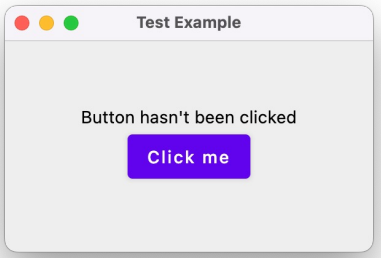
What do we need to change in our tests?

- Testing interaction & output (View)
- Test UI state (ViewModel)



```
fun main() {
    application {
        Window(title = "Test Example", onCloseRequest = ::exitApplication)
        {
            var text by remember { mutableStateOf("Button hasn't been clicked") }

            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.Center,
                modifier = Modifier.fillMaxSize().padding(10.dp)
            ) {
                Text(
                    text = text,
                )
                Button(
                    onClick = { text = "Clicked!" },
                ) {
                    Text("Click me")
                }
            }
        }
    }
}
```

This is the behaviour we want to test. i.e. click on the button and see how the UI changes.



samples/desktop-compose -> run **Example** main method

```kotlin
class ExampleTest {
    @get:Rule
    val rule = createComposeRule()

    @Test
    fun myTest(){
          rule.setContent {
            var text by remember { mutableStateOf("Button hasn't been clicked") }

            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.Center,
                modifier = Modifier.fillMaxSize().padding(10.dp).testTag("column") // tag
            )
            {
                Text(
                    text = text,
                    modifier = Modifier.testTag("text") // tag
                )
                Button(
                    onClick = { text = "Clicked!" },
                    modifier = Modifier.testTag("button") // tag
                ) {
                    Text("Click me")
                }
            }
        }

        // Tests the declared UI with assertions and actions of the JUnit-based testing API
        rule.onNodeWithTag("text").assertTextEquals("Button hasn't been clicked")
        rule.onNodeWithTag("button").performClick()
        rule.onNodeWithTag("text").assertTextEquals("Clicked!")
    }
}
```

Test actions performed in-order.

What actions can you test?

rule

- onNodeWithTag
- onNodeWithText
- onNode
- onAllNodes
- onRoot

- performClick()
- performKeyPress()
- performKeyInput ()
- performMouseInput ()
- performMouseMultiModal ()
- performScrollTo()
- performFirstLinkClick()

- assertExists
- assertDoesNotExist
- assertDeactivated
- assertTextEquals
- assertTextContains
- assertHasClickAction
- assertIsDisplayed
- assertIsEnabled
- assertHasFocus

Architecture

Components and structure for desktop applications.

Recall: MVVM

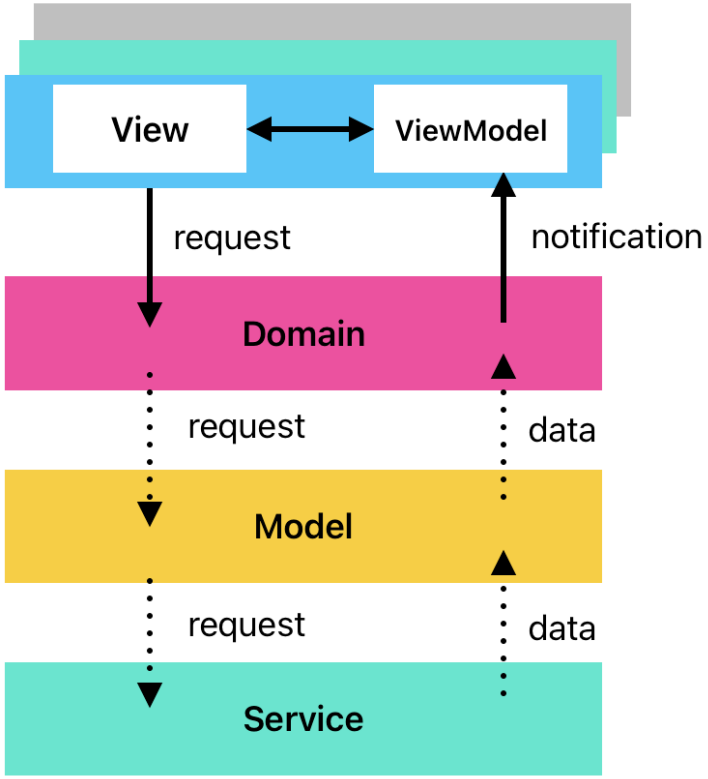
We will use Model-View-ViewModel, a layered architecture designed for graphical user interfaces.

Introduce Standard Layers

- Each layer has specific functionality.
- Requests flow down, and data flows up.
- *This also means that dependencies extend down*.

Expand the User Interface layer

- View: the interactive components
- ViewModel: backing state for these components



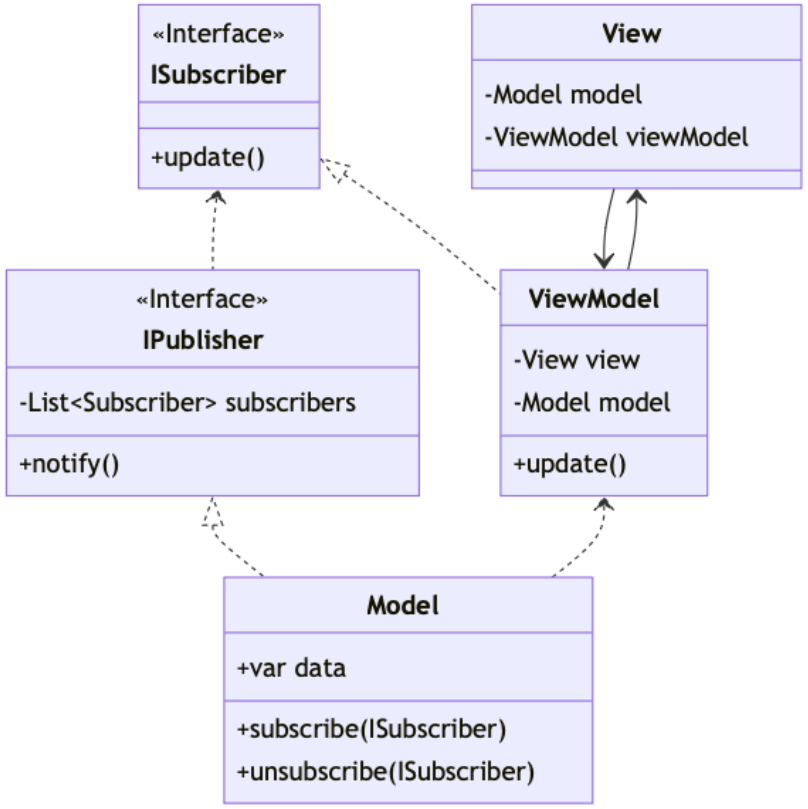
Recall: MVVM Implementation

Main classes

- **View**: input/output
- **ViewModel**: state for the view.
- **Model**: stores the application data.

There are often multiple views. Each View typically has one ViewModel associated with it.

MVVM uses the Observer pattern. The model typically notifies the ViewModel of state changes. The View and ViewModel are often tightly coupled so that updating the ViewModel data will refresh the View.



mm-desktop

Using Compose Multiplatform to port our application to desktop.

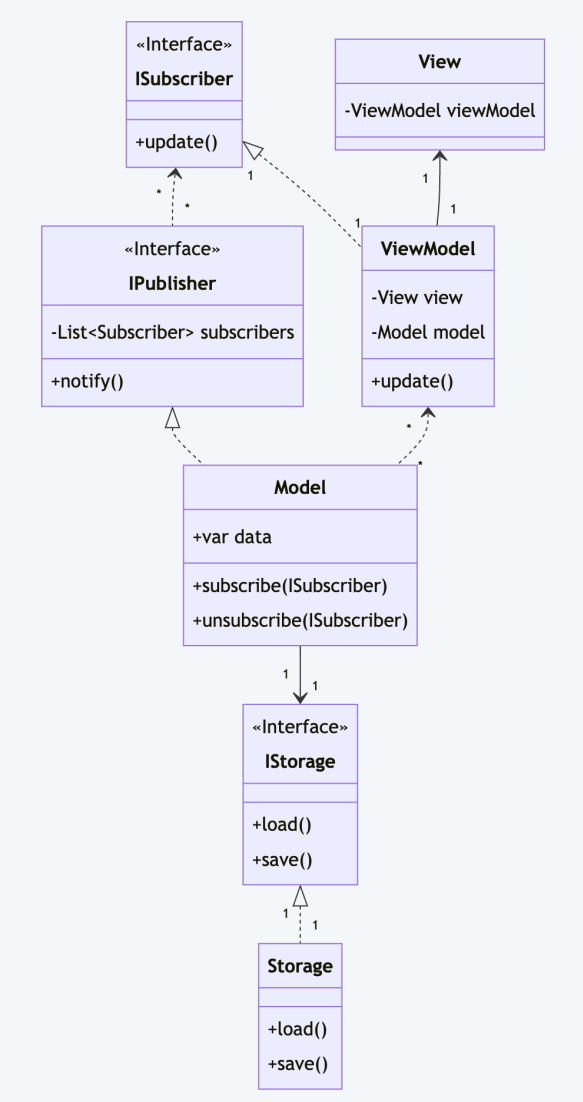
Layered architecture

Shown here

- **View & ViewModel**: the user interface, and its local state management.
- **Model**: stores the TODO items, and coordinates any changes to the list. Also handles save/load of data.
- **IStorage & Storage**: low-level interface and database / filestorage layer.

Not shown

- **Main** function, used as an entry point.



Main function

```
fun main() = application {
    MaterialTheme {
        Window(
            title = "Mastermind TODO",
            state = rememberWindowState(
                position = WindowPosition(Alignment.Center),
                size = DpSize(400.dp, 600.dp)
            ),
            onCloseRequest = ::exitApplication,
        ) {
            // wire dependencies together
            // storage <-- model <-- viewModel <-- view
            val storage = DBStorage(".mm.db")
            val model = Model(storage)
            val viewModel = ViewModel(model)

            // top-level composable
            View(viewModel)
        }
    }
}
```

Window setup

Dependencies

Root of scene graph

See GitHub: demos/mm-desktop

View

```
@Composable
fun View(viewModel: ViewModel) {
    val tasks = viewModel.tasks
    val scaffoldState = rememberScaffoldState()

    var showAddDialog by remember { mutableStateOf(false) }
    var showEditDialog by remember { mutableStateOf(false) }
    var selectedTask by remember { mutableStateOf<Task?>(null) }

    Scaffold(
        scaffoldState = scaffoldState,
        topBar = {
            TopAppBar(
                title = { Text("Task Manager") },
                actions = {
                    IconButton(onClick = { showAddDialog = true }) {
                        Icon(Icons.Default.Add,
                                    contentDescription = "Add Task")
                    }
        // ……
```

See GitHub: demos/mm-desktop

ViewModel

```
class ViewModel(private val model: Model) : Subscriber {                // intermediary between view/model

  var tasks by mutableStateOf(model.tasks.filterNotNull())
  init { model.add(this)}

  override fun update() {
    tasks = model.tasks.filterNotNull()                                 // change to model data -> update() called
  }

  fun addTask(title: String) { model.add(title)}                        // pass user requests to the model

  fun deleteTask(position: Int) { model.del(position)}

  fun updateTask(task: Task) {
    val existingTask = model.tasks.find { it?.id == task.id }
    if (existingTask != null) {
      existingTask.title = task.title
      existingTask.description = task.description
      existingTask.dueDate = task.dueDate
      existingTask.tags = task.tags
      model.notifySubscribers()
    }
  }
}
```

See GitHub: demos/mm-desktop

Model

```
class Model(private val storage: IStorage): Publisher() {
    var tasks = mutableListOf<Task?>()

    init {
        tasks = storage.readAll().toMutableList()
    }

    fun add(contents: String) {
        val task = Task(
            position = tasks.size + 1, title = contents,
            description = "", dueDate = "", tags = ""
        )
        storage.create(task)
        tasks.add(task)
    }

    fun del(position: Int) {
        val pos = position
        val task = tasks.find { it?.position == pos } ?: return
        storage.delete(task)
        tasks.remove(task)
        reposition()
    }
}
```

See GitHub: demos/mm-desktop

Storage interface

```
interface IStorage {
    // canonical operations
    fun create(task: Task): Int
    fun read(id: Int): Task?
    fun readAll(): List<Task?>
    fun update(task: Task)
    fun delete(task: Task)
    fun deleteAll()

    // extended operations
    fun upsert(task: Task)
}
```

See GitHub: samples/demos/mm-desktop

Reference

- Bolt UIX. 2025. Kotlin Multiplatform: What You Can Only Do in desktopMain
- JetBrains. 2025. Compose Multiplatform Documentation.
- JetBrains. 2025. Kotlin Multiplatform Documentation.