COM4510/6510 Software Development for Mobile Devices

Lecture 4: Design Patterns

Temitope (Temi) Adeosun
The University of Sheffield
t.adeosun@sheffield.ac.uk

Lecture Overview

- Design pattern overview
- MVC
- MVP
- MVVM
- Conclusion

Design Patterns Overview

- In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem
  - It is a template for how to solve a problem that can be used in many different situations
  - It is a method rather than a finished design that can be transformed directly into code
- They can speed up the development process by providing tested, proven development paradigms
- Effective software design requires considering issues that may not become visible until later in the implementation
  - Reusing design patterns helps to prevent subtle issues that can cause major problems
  - They improve code readability for coders and architects familiar with the patterns.

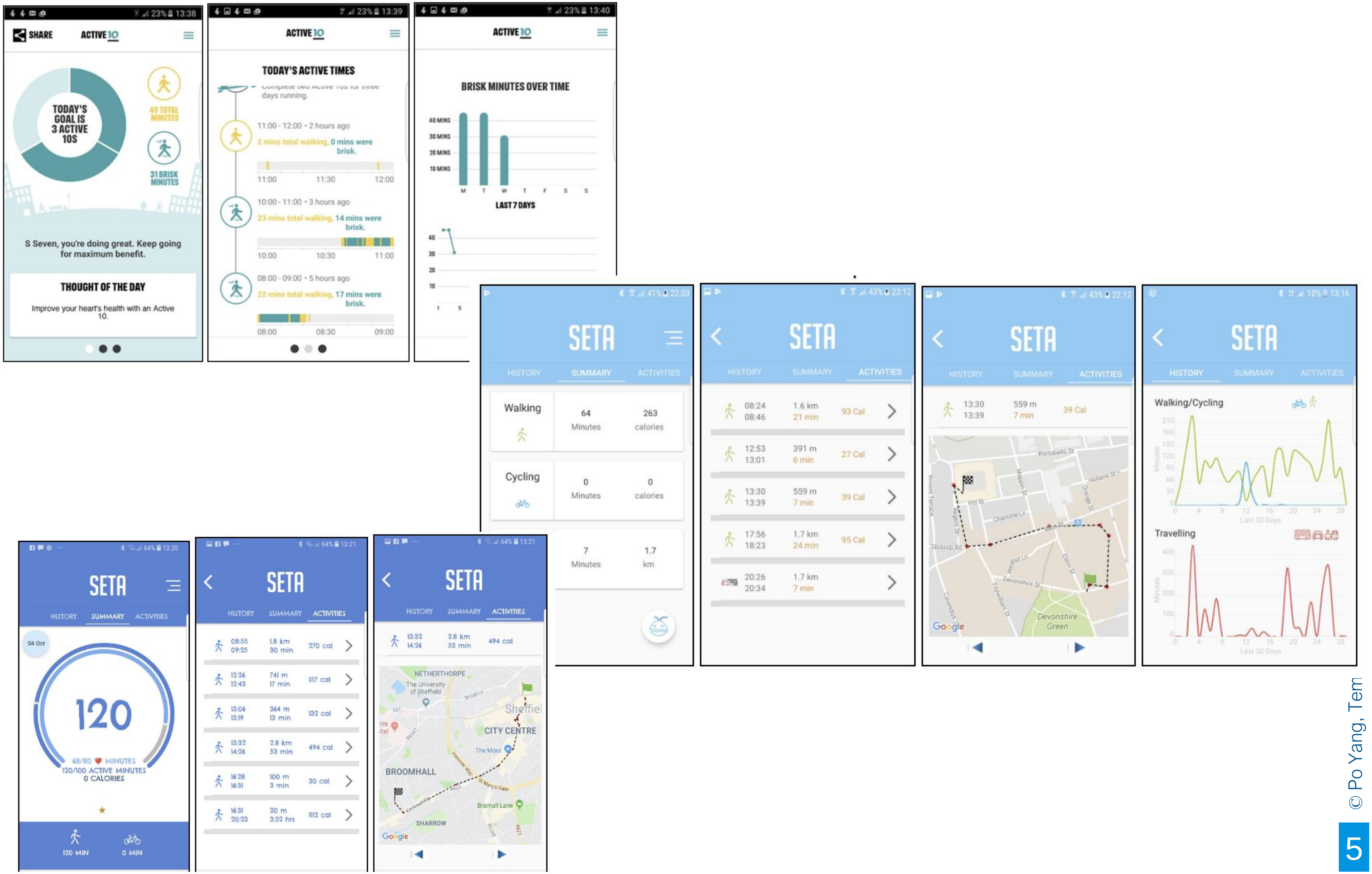
More on design patterns at: https://sourcemaking.com/design_patterns

Issues with Mobile Apps

- Generally an app is not valuable for its interface (GUI) only
  - An app is generally valuable for its business logic, i.e. the service it provides
- Issues that emerge in developing mobile apps:
  - The frequency of interface design changes
    - To keep the user engaged
    - To improve usability and use
    - Just because the owner wants so
  - The frequent request for adaptation to different contexts of the same business logic
    - To provide different user experiences in different contexts
      - e.g. for a version for teenager or children, or versions for special events

The University Of Sheffield.

Three apps, one logic



SHARE ACTIVE 10

TODAY'S GOAL IS 3 ACTIVE 10S

49 TOTAL MINUTES

31 BRISK MINUTES

S Seven, you're doing great. Keep going for maximum benefit.

THOUGHT OF THE DAY

Improve your heart's health with an Active 10.

ACTIVE 10

TODAY'S ACTIVE TIMES

Complete two Active 10s for three days running.

11:00 - 12:00 • 2 hours ago

2 mins total walking, 0 mins were brisk.

10:00 - 11:00 • 3 hours ago

23 mins total walking, 14 mins were brisk.

08:00 - 09:00 • 5 hours ago

22 mins total walking, 17 mins were brisk.

ACTIVE 10

BRISK MINUTES OVER TIME

LAST 7 DAYS

SETA

HISTORY SUMMARY ACTIVITIES

Walking	64 Minutes	263 calories
Cycling	0 Minutes	0 calories
	7 Minutes	1.7 km

SETA

HISTORY SUMMARY ACTIVITIES

08:24 08:46	1.6 km 21 min	93 Cal
12:53 13:01	391 m 6 min	27 Cal
13:30 13:39	559 m 7 min	39 Cal
17:56 18:23	1.7 km 24 min	95 Cal
20:26 20:34	1.7 km 7 min	

SETA

HISTORY SUMMARY ACTIVITIES

13:30 13:39	559 m 7 min	39 Cal

SETA

HISTORY SUMMARY ACTIVITIES

Walking/Cycling

Travelling

SETA

HISTORY SUMMARY ACTIVITIES

04 Oct

120

68/80 MINUTES

120/100 ACTIVE MINUTES

0 CALORIES

120 MIN 0 MIN

SETA

HISTORY SUMMARY ACTIVITIES

08:55 09:25	1.8 km 30 min	270 cal
12:26 12:43	741 m 17 min	157 cal
13:06 13:19	364 m 13 min	132 cal
13:32 14:26	2.8 km 53 min	494 cal
16:28 16:31	100 m 3 min	30 cal
16:31 20:23	20 m 3:52 hrs	1112 cal

SETA

HISTORY SUMMARY ACTIVITIES

13:32 14:26	2.8 km 53 min	494 cal

Issues with Mobile Apps (cont.)

- Junit Testing:
  - Testing is key in software engineering
  - How do you test something like an app that is tightly tied to its UI?
    - Automated systems pressing randomly across the screen
      - useful but to largely ineffective
    - ???

In comes separation of concerns

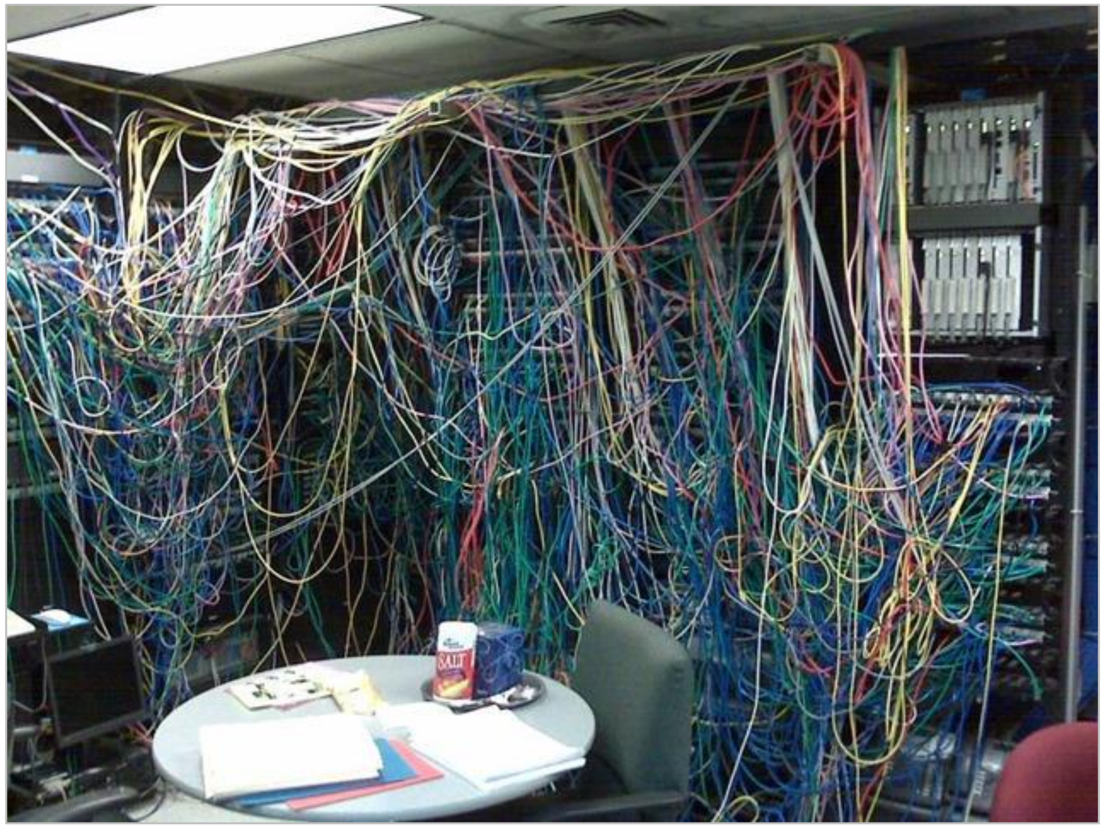
- Which components are responsible for what, and how the responsible components interact
- These problems not new to mobile development. Can be addressed by adopt the right design patterns
  - that foster the clear separation of concerns between
    - The business logic (or Model)
      - what the app provides in terms of data or services
      - which is generally core and independent
    - The UI (or View)
      - the way the app presents the data/service
      - which can change according to the context of use

So, which Patterns ?

MVC
MVP
Reactive
Cairngorm
Flux
fdsafdsa
MVVM
CLEAN

(Architectural) Design Patterns

- Architectural patterns help organise the entire code structure
  - typically implement multiple software patterns
- Your goal: Scalable, maintainable, testable
- Scalable
  - Add new features quickly
- Maintainable
  - No spaghetti code
  - Do not cross the streams
- Testing
  - Easy to separate and mock

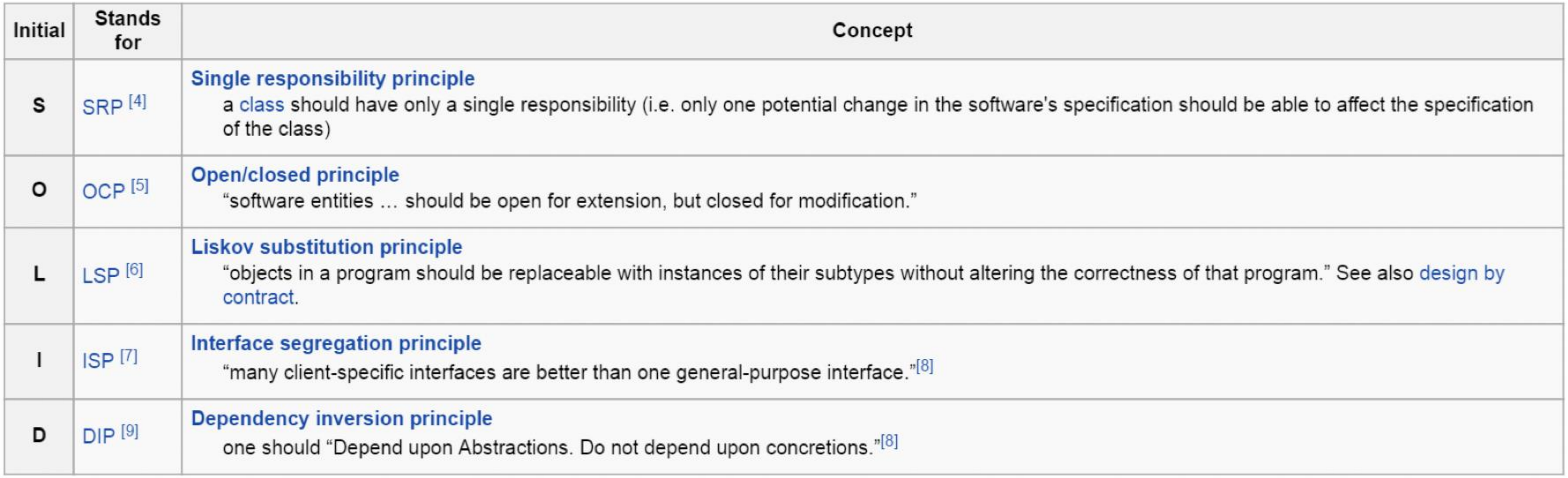


This Photo by Unknown Author is licensed under CC BY-SA

But it works!?!

Why ?

- Easier to add new features
- Easier to understand
- Easier to police
- Makes our lives easier
- Make Unit Testing easier



Initial	Stands for	Concept
S	SRP [4]	**Single responsibility principle** a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	OCP [5]	**Open/closed principle** “software entities … should be open for extension, but closed for modification.”
L	LSP [6]	**Liskov substitution principle** “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also design by contract.
I	ISP [7]	**Interface segregation principle** “many client-specific interfaces are better than one general-purpose interface.”[8]
D	DIP [9]	**Dependency inversion principle** one should “Depend upon Abstractions. Do not depend upon concretions.”[8]

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

Why ?

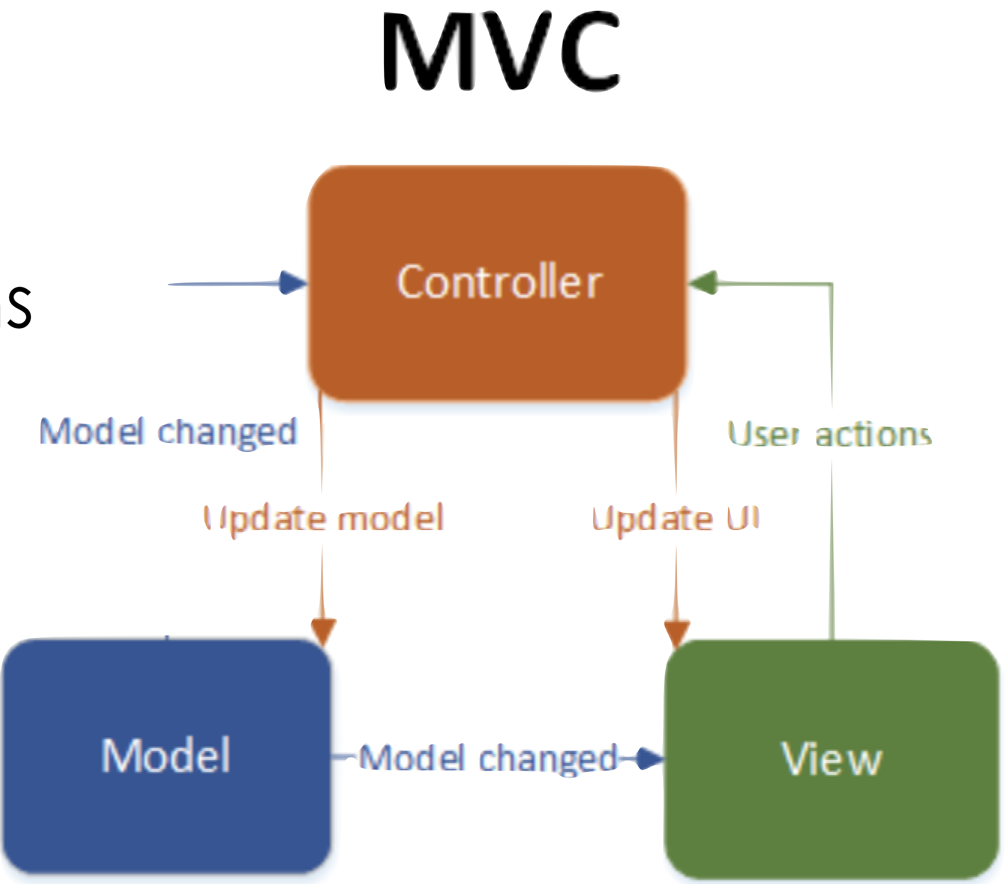
- Using patterns later in the development lifecycle will increase the cost of developing your app.
- Used right, yield low maintenance cost.
- The use of patterns is essential for extensibility and maintenance.
- Most app will become complex with time. Keep the app simple, understandable and use pattens

Android (Architectural) Design Patterns

- The ones used in Mobile Development tends to be based on two main modules:
  - The **Model**
    - The model is the Data + State + Business logic of our application
      - It's the brains of our application so to speak. It is not tied to the view or controller, and because of this, it is reusable in many contexts
  - The **View**
    - Best kept dumb
      - No knowledge of the underlying model
      - No understanding of state
      - No understanding of how to respond to user interactions i.e. clicking a button, typing a value, etc.
      - **in short: an XML file**
  - The third element differs in different patterns
    - Model View **Controller** (MV**C**)
    - Model View **Presenter** (MV**P**)
    - Model View **View Model** (MV**VM**)

MVC[ontroller]

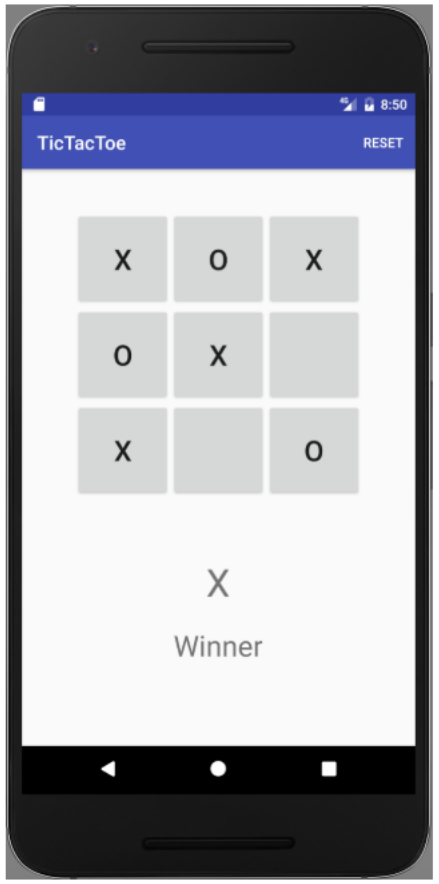
- The Controller
  - The controller is the glue that ties the app together – it controls what happens in the application
- When the View tells the controller that a user clicked a button, the controller decides what to do, i.e. how to interact with the model
- If data changed in the model, the controller decides what to do, i.e. may choose to update the state of the view
- In the case of an Android application, the controller is almost always represented by an **Activity or Fragment**.



https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

Tic Tac Toe app

- In this application, the MVC is implemented in the following way

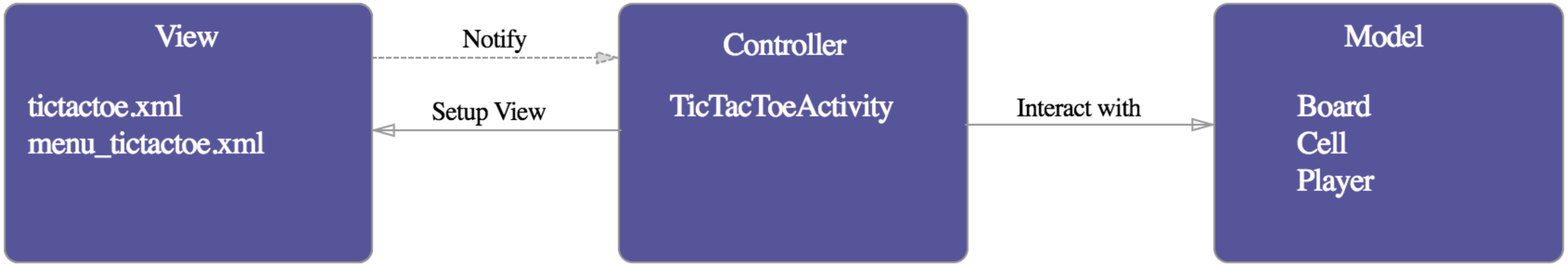


X	O	X
O	X	
X		O

X

Winner

It owns the Activity/Fragment



The view is just in the XML files (declarative)

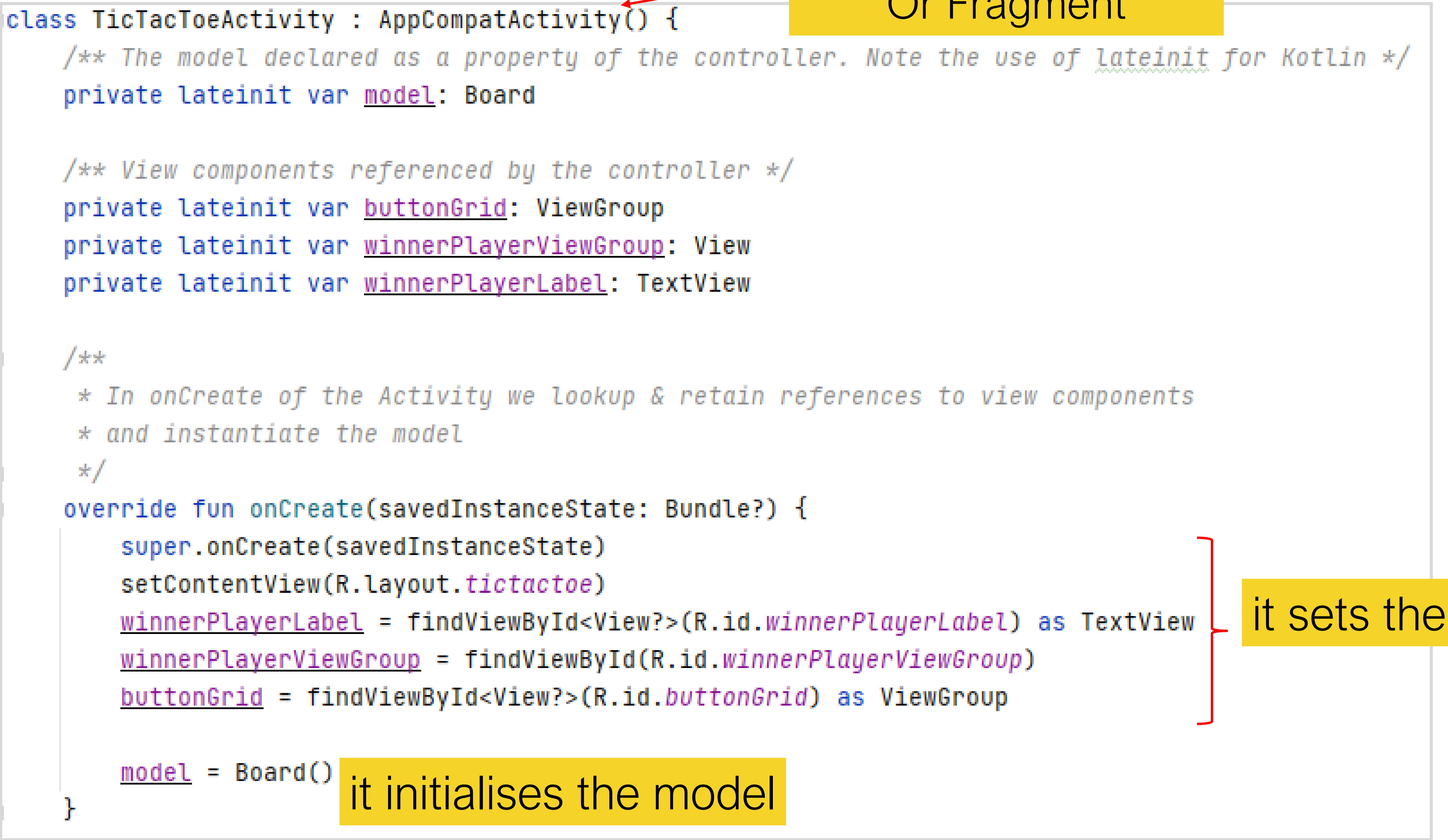
Connects the UI action to the model's Data and computations. e.g. click on a cell on the UI will call a check to the model to

1. check if the cell is not taken
2. check if the user has won/lost

The model is about how to play the game it does not know anything about the View or Controller

https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

Controller



```
class TicTacToeActivity : AppCompatActivity() {
    /** The model declared as a property of the controller. Note the use of lateinit for Kotlin */
    private lateinit var model: Board

    /** View components referenced by the controller */
    private lateinit var buttonGrid: ViewGroup
    private lateinit var winnerPlayerViewGroup: View
    private lateinit var winnerPlayerLabel: TextView

    /**
     * In onCreate of the Activity we lookup & retain references to view components
     * and instantiate the model
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.tictactoe)
        winnerPlayerLabel = findViewById<View?>(R.id.winnerPlayerLabel) as TextView
        winnerPlayerViewGroup = findViewById(R.id.winnerPlayerViewGroup)
        buttonGrid = findViewById<View?>(R.id.buttonGrid) as ViewGroup

        model = Board()
    }
```

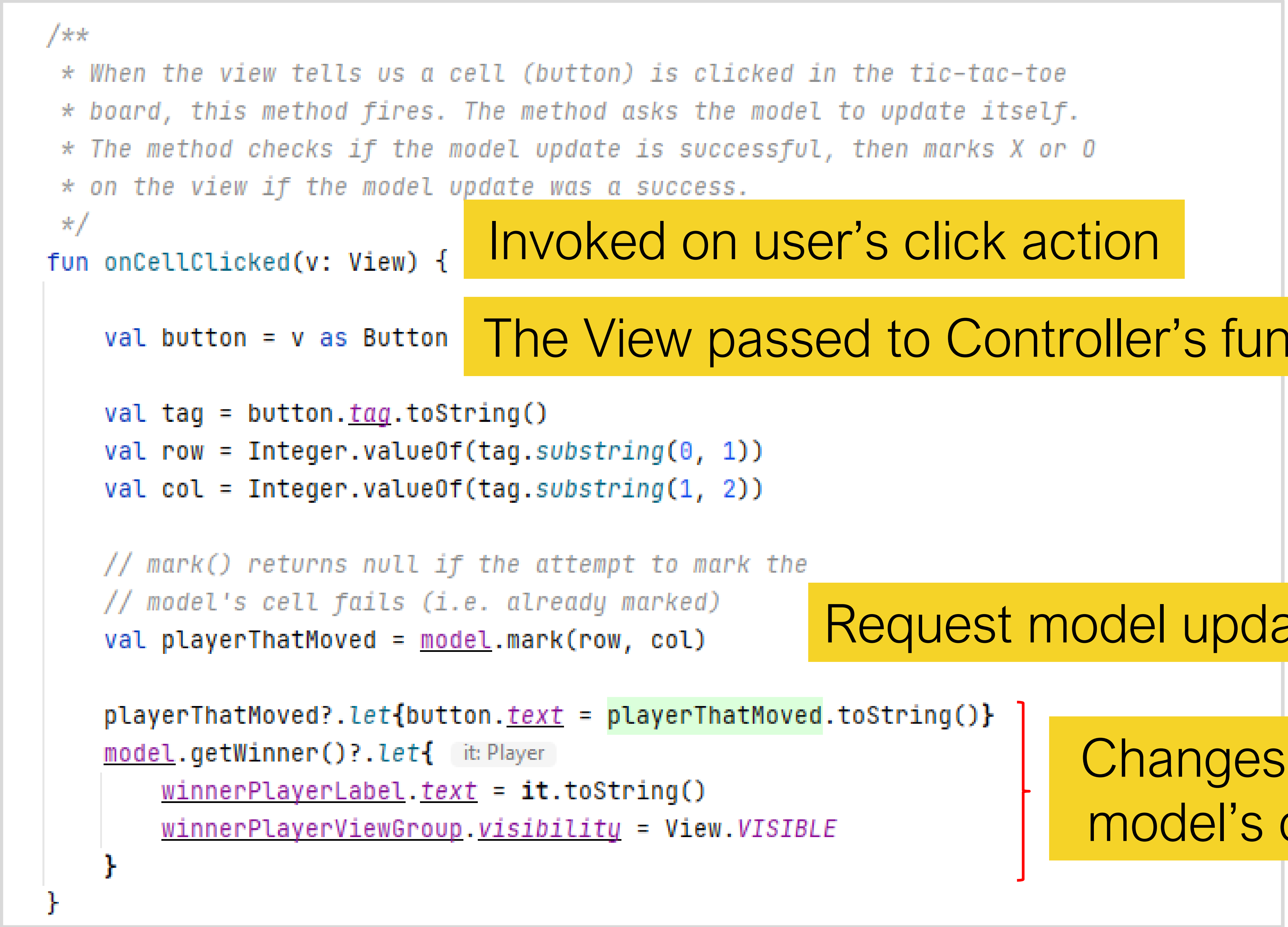
It is always an Activity
Or Fragment

it sets the UI

it initialises the model

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

Controller



```
/**
 * When the view tells us a cell (button) is clicked in the tic-tac-toe
 * board, this method fires. The method asks the model to update itself.
 * The method checks if the model update is successful, then marks X or O
 * on the view if the model update was a success.
 */
fun onCellClicked(v: View) {

    val button = v as Button

    val tag = button.tag.toString()
    val row = Integer.valueOf(tag.substring(0, 1))
    val col = Integer.valueOf(tag.substring(1, 2))

    // mark() returns null if the attempt to mark the
    // model's cell fails (i.e. already marked)
    val playerThatMoved = model.mark(row, col)

    playerThatMoved?.let{button.text = playerThatMoved.toString()}
    model.getWinner()?.let{ it: Player
        winnerPlayerLabel.text = it.toString()
        winnerPlayerViewGroup.visibility = View.VISIBLE
    }
}
```

Invoked on user's click action

The View passed to Controller's function is the button

Request model update

Changes UI based on model's current state

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

MVC Pros vs Cons?

- Pros:
  - MVC does a great job of separating the model and view
  - The model can be easily unit tested - it's not tied to anything
  - The view is passive and there is nothing to test
- Cons:
  - The controller is tied so tightly to the Android APIs that it is difficult to unit test.
  - The controller is tightly coupled to the views
    - it might as well be an extension of the view - change the view, and you have to change the controller (Violates Single responsibility principle)
  - Over time, particularly in applications with anemic models (models that have state, but no behaviour e.g. Java POJO or Kotlion Data classes), more and more code starts getting transferred into the controllers, making them bloated and brittle

https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

MVP[resenter]

- MVP breaks the controller up
  - so that the natural view/activity coupling can occur
  - there is still separation between model and view
  - activity/fragment is owned by the view
  - improved separation of concerns – better maintainability
  - Presenter
    - stills plays the middle-man between the model & view (like the controller)
    - but provides an interface (contract) to prevent deep coupling with the view
    - thus, it can be Junit tested

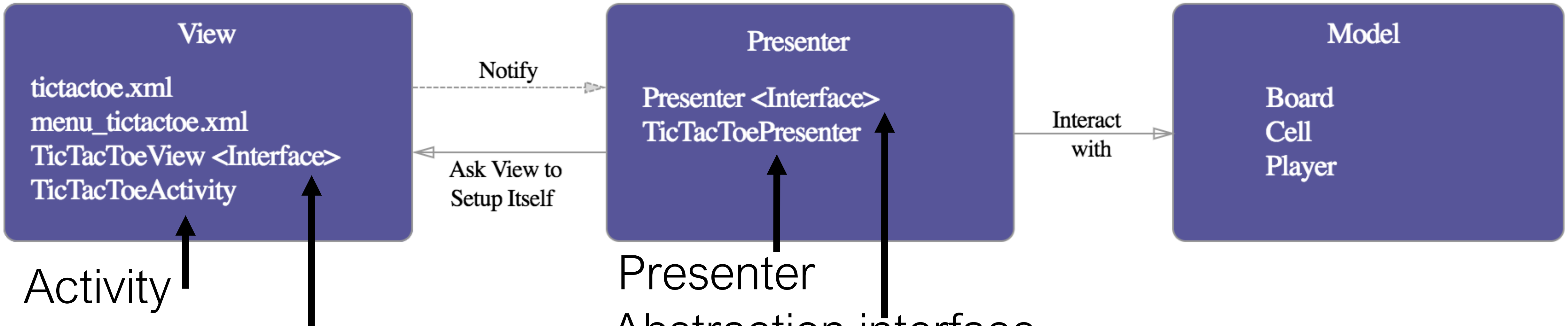
https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

Changes from MVC to MVP

- Model does not change
- View is still XML
  - However, the Activity/Fragment is now logically considered part of the view
    - in MVC Activity/Fragment is part of the controller
  - it implement a view interface so that the presenter can call abstract actions to be performed on the View.
- Presenter:
  - same as the controller from MVC but not tied to the View
    - tied to its interface (which is merely a contract)

Tic Tac Toe app

View owns the Activity/Fragment

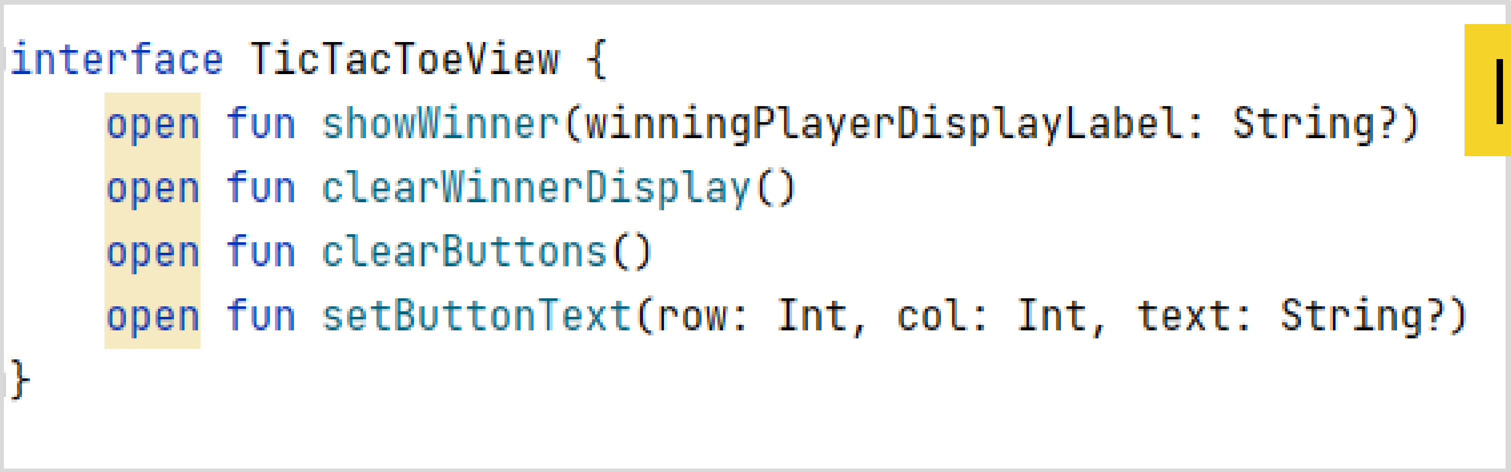


Activity

Abstraction interface

Presenter

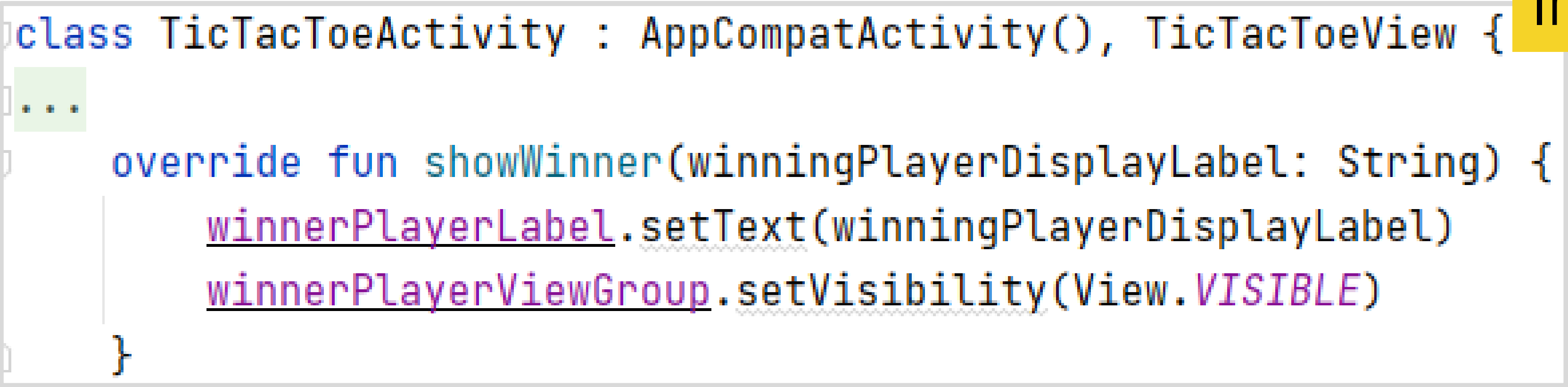
Abstraction interface



```
interface TicTacToeView {
    open fun showWinner(winningPlayerDisplayLabel: String?)
    open fun clearWinnerDisplay()
    open fun clearButtons()
    open fun setButtonText(row: Int, col: Int, text: String?)
}
```

Interface declaration

View (Activity) Implements the Interface

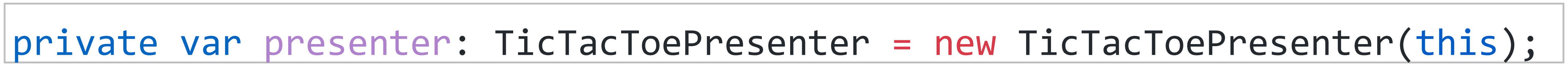


```
class TicTacToeActivity : AppCompatActivity(), TicTacToeView {
...
    override fun showWinner(winningPlayerDisplayLabel: String) {
        winnerPlayerLabel.setText(winningPlayerDisplayLabel)
        winnerPlayerViewGroup.setVisibility(View.VISIBLE)
    }
```

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

View and Presenter

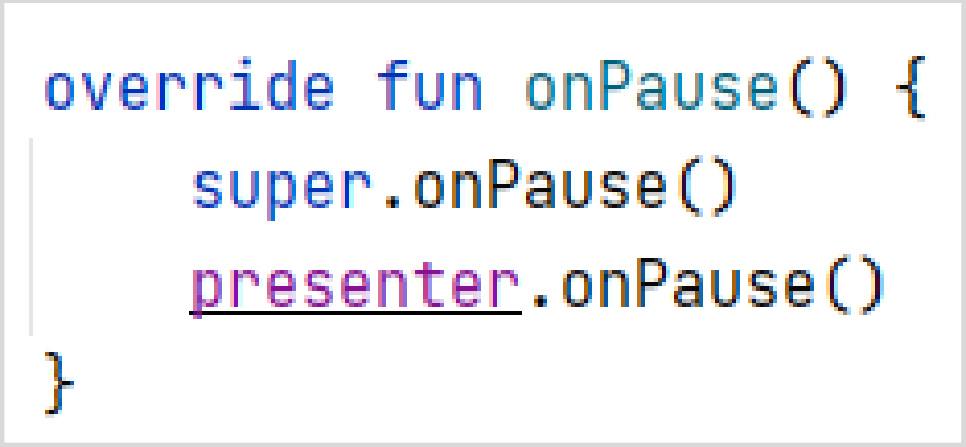
- As the Activity is now part of the view, the presenter must be instantiated in it:



```
private var presenter: TicTacToePresenter = new TicTacToePresenter(this);
```

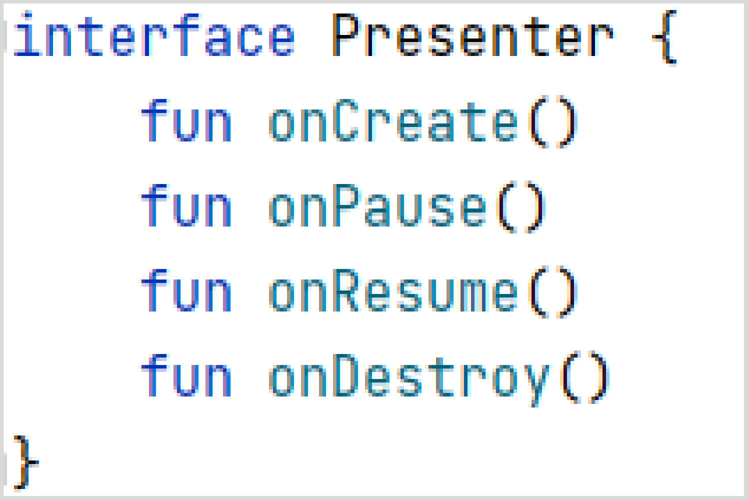
- Moreover it must inform the presenter of lifecycle event (create/resumed/paused/destroyed)
  - this is why the presenter exposes an interface with names with the same methods

In the view



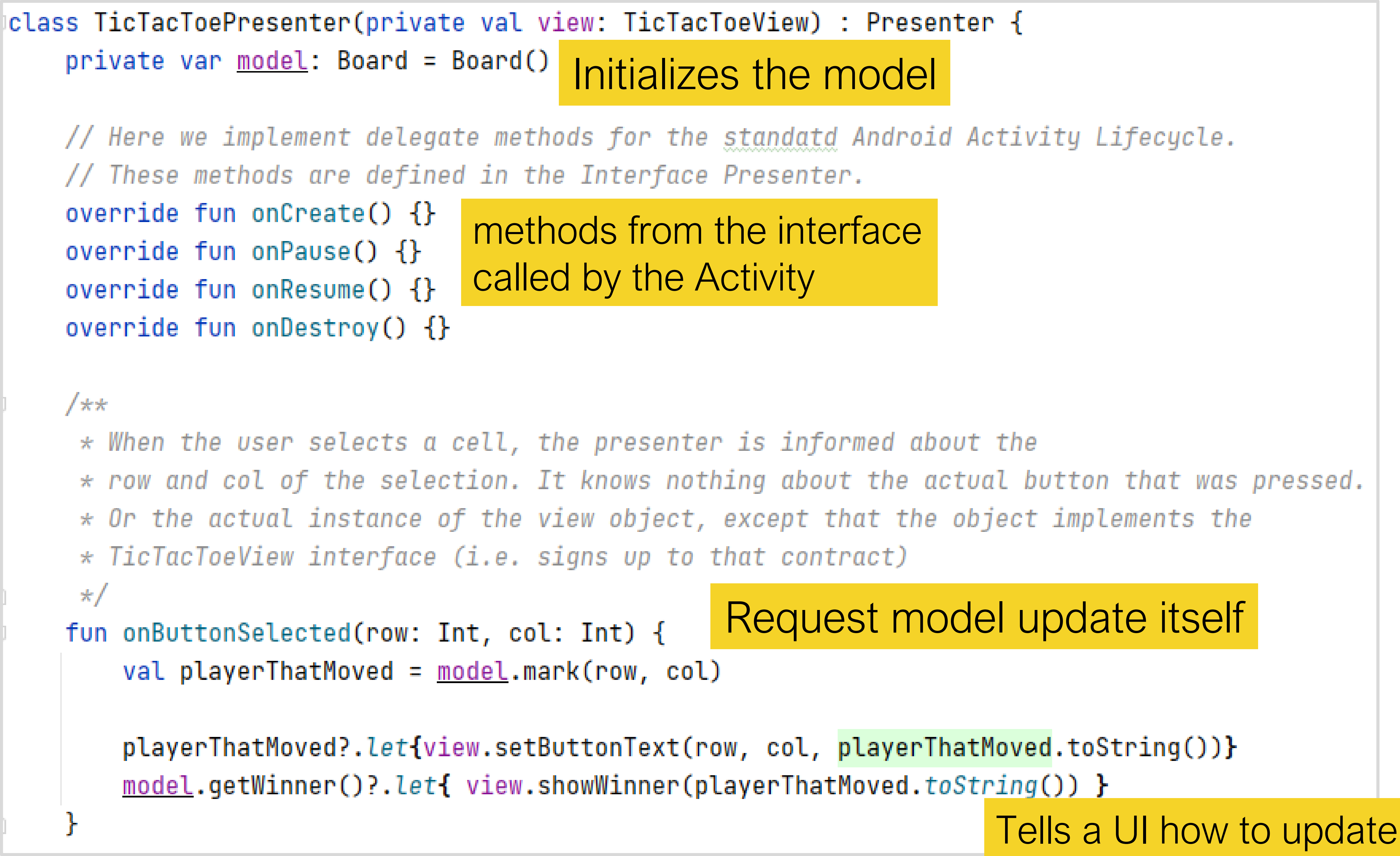
```
override fun onPause() {
    super.onPause()
    presenter.onPause()
}
```

The presenter Interface



```
interface Presenter {
    fun onCreate()
    fun onPause()
    fun onResume()
    fun onDestroy()
}
```

Presenter



```
class TicTacToePresenter(private val view: TicTacToeView) : Presenter {
    private var model: Board = Board()

    // Here we implement delegate methods for the standatd Android Activity Lifecycle.
    // These methods are defined in the Interface Presenter.
    override fun onCreate() {}
    override fun onPause() {}
    override fun onResume() {}
    override fun onDestroy() {}

    /**
     * When the user selects a cell, the presenter is informed about the
     * row and col of the selection. It knows nothing about the actual button that was pressed.
     * Or the actual instance of the view object, except that the object implements the
     * TicTacToeView interface (i.e. signs up to that contract)
     */
    fun onButtonSelected(row: Int, col: Int) {
        val playerThatMoved = model.mark(row, col)

        playerThatMoved?.let{view.setButtonText(row, col, playerThatMoved.toString())}
        model.getWinner()?.let{ view.showWinner(playerThatMoved.toString()) }
    }
```

Initializes the model

methods from the interface called by the Activity

Request model update itself

Tells a UI how to update based on the agreed interface contract (abstraction)

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

MVP Pros and Cons

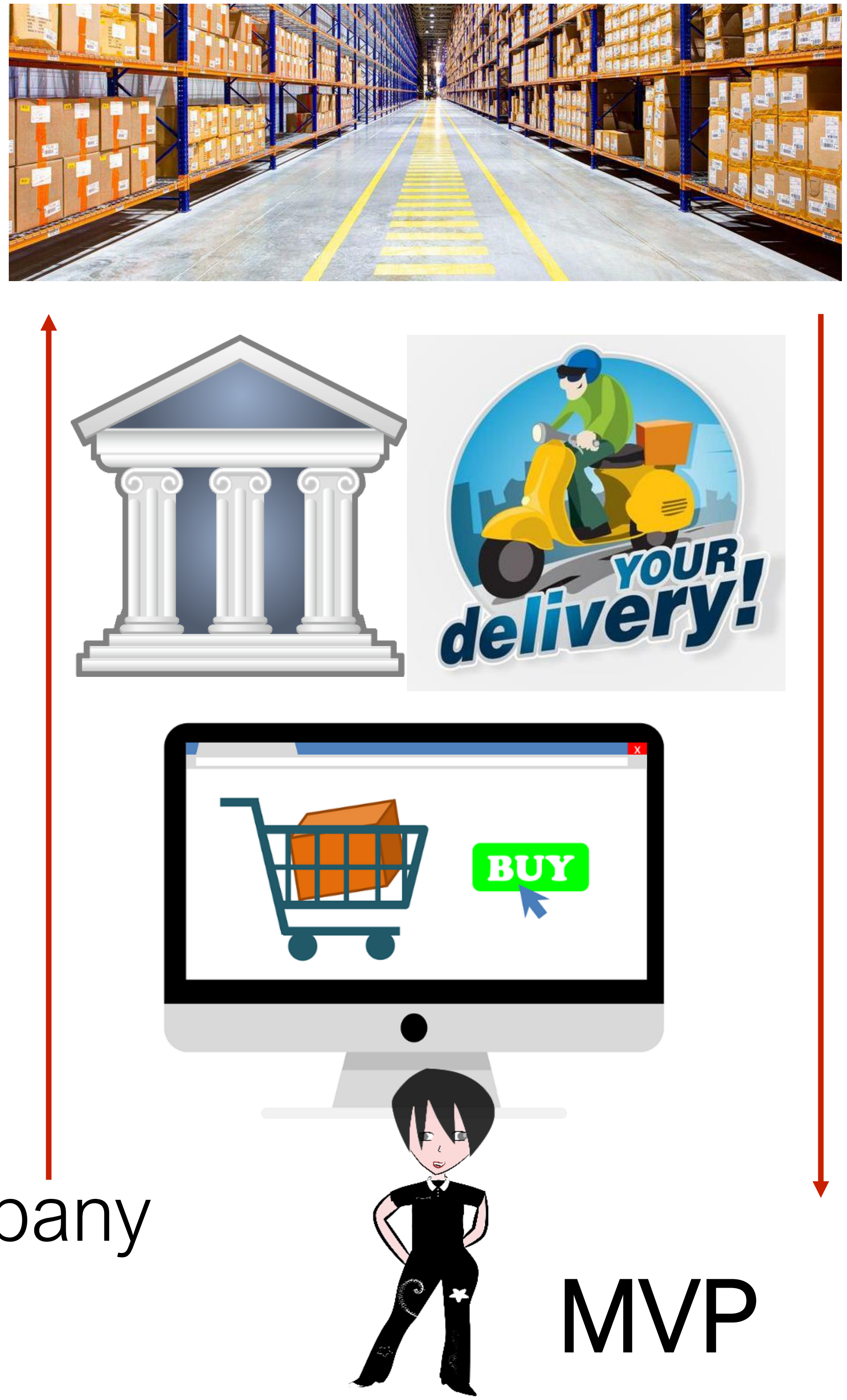
- Pros
  - Complex Tasks split into simpler tasks
  - Smaller objects, less bugs, easier to debug
  - Testable
    - Can test render logic (View) interaction with presenter. Mock presenter
    - Can test view events invoke the right model methods. Mock view and model.
    - Can test the business logic. Mock data source and presenter
- Cons
  - Boiler plate to wire up the layers.

https://speakerdeck.com/rallat/androiddevlikeaprodroidconsf

MVP Pros and Cons (cont.)

- Cons
  - Model cannot be reused, tied to specific use case.
  - View and Presenter are tied to data objects since they share the same type of object with the Model.
  - Presenters (just like Controllers) may start to collect business logic

MVC vs. MVP



MVC gets the job done

MVP gets the job done but reduces coupling

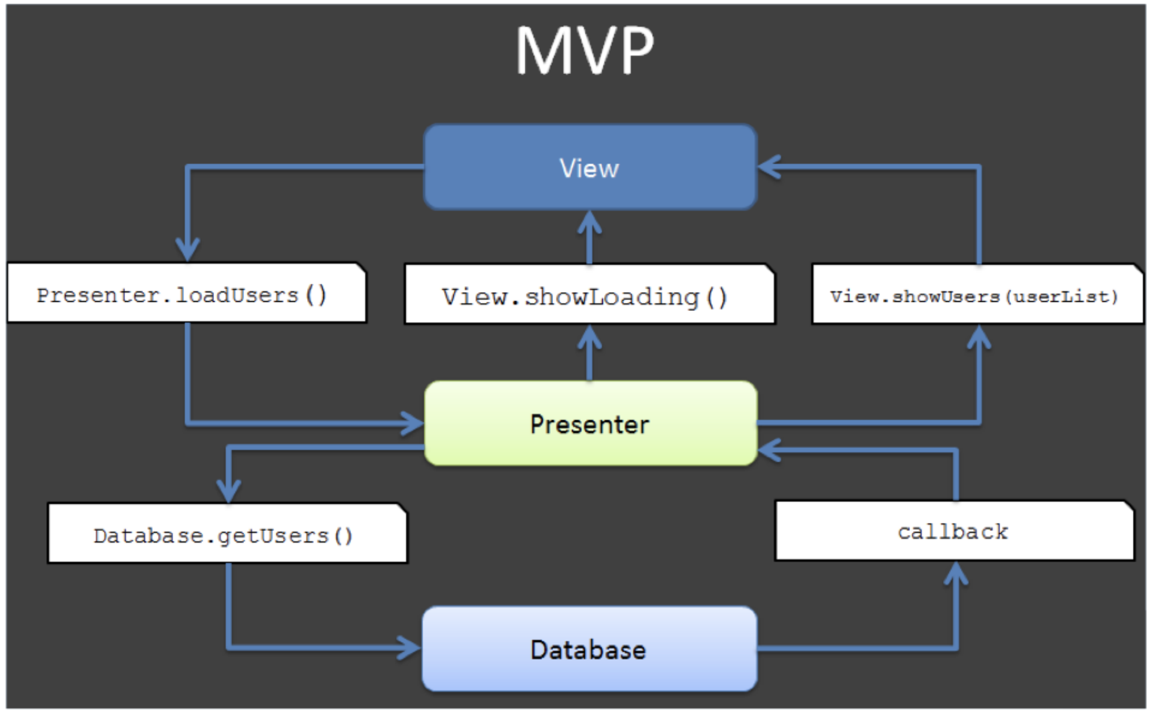
- You can change banks and delivery company
- You can test each part independently
- But…

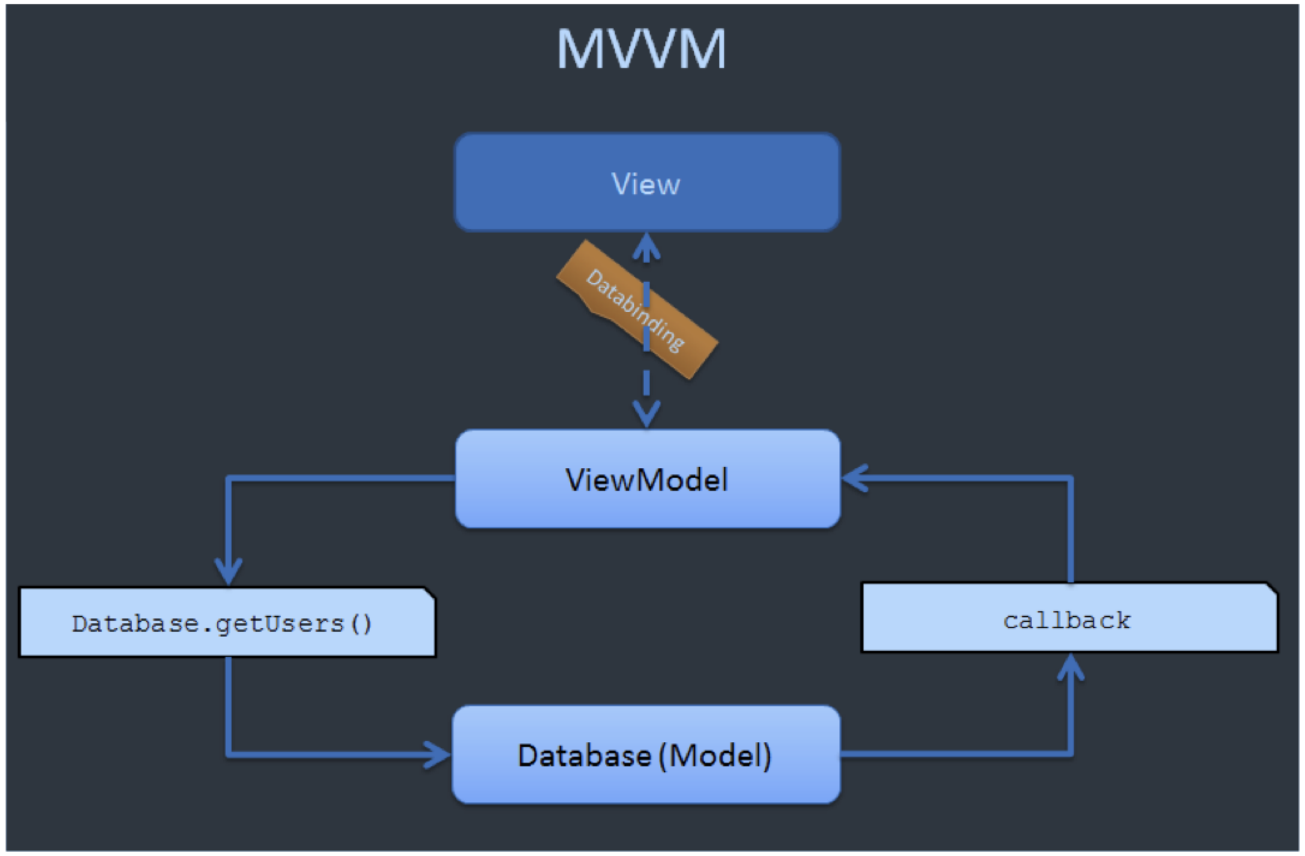
The imagery is not a perfect analogy, but stick with me

MVV[iew]M[odel]

- Microsoft Pattern
- Removes UI code from Activities/ Fragments
- View has no knowledge of model
- Android implementation uses Data Binding = binds ViewModel to Layout without tight coupling
- Goodbye Presenter, hello ViewModel

MVP vs MVVM



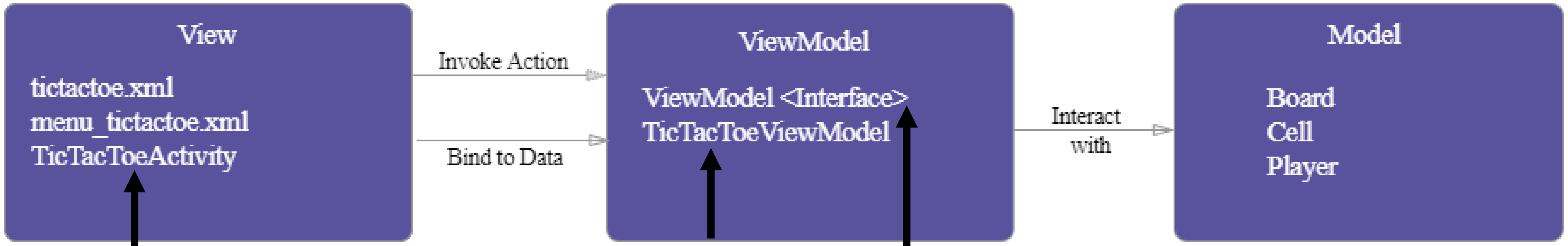


https://schibsted.com/blog/android-databinding-goodbye-presenter-hello-viewmodel/

Changes to MVVM

- Model does not change, same as MVC
- View
  - The view binds to observable variables and actions exposed by the ViewModel in a flexible way.
- ViewModel
  - The ViewModel is responsible for wrapping the model and preparing *Observable fields* needed by the view
  - It provides hooks for the view to pass events to the model
  - The ViewModel is not tied to the view however.

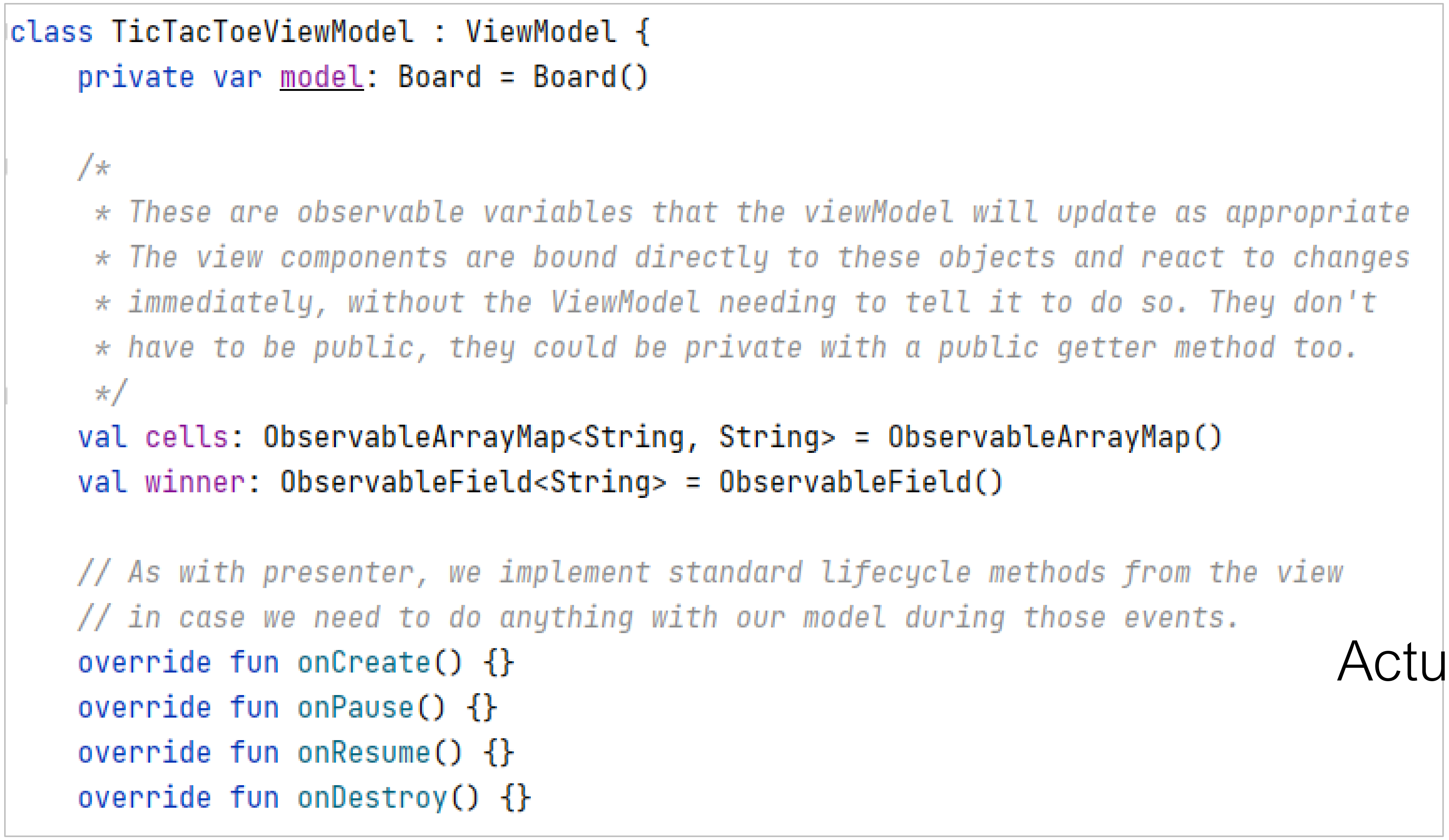
Tic Tac Toe app



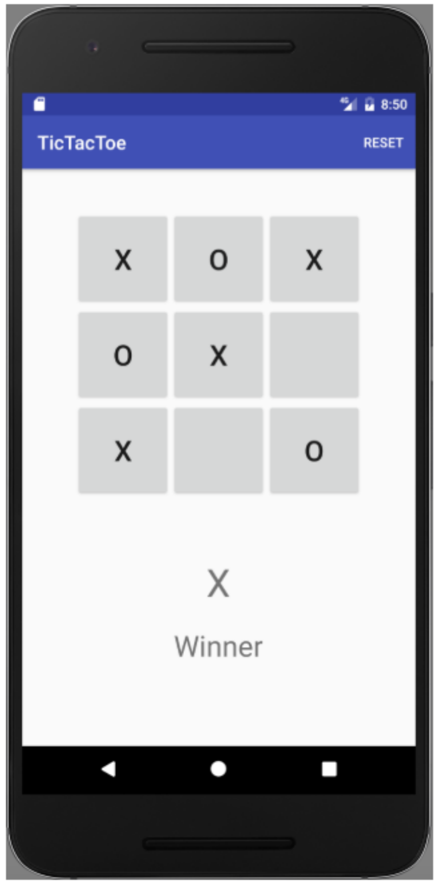
Activity

View Model

Abstraction interface



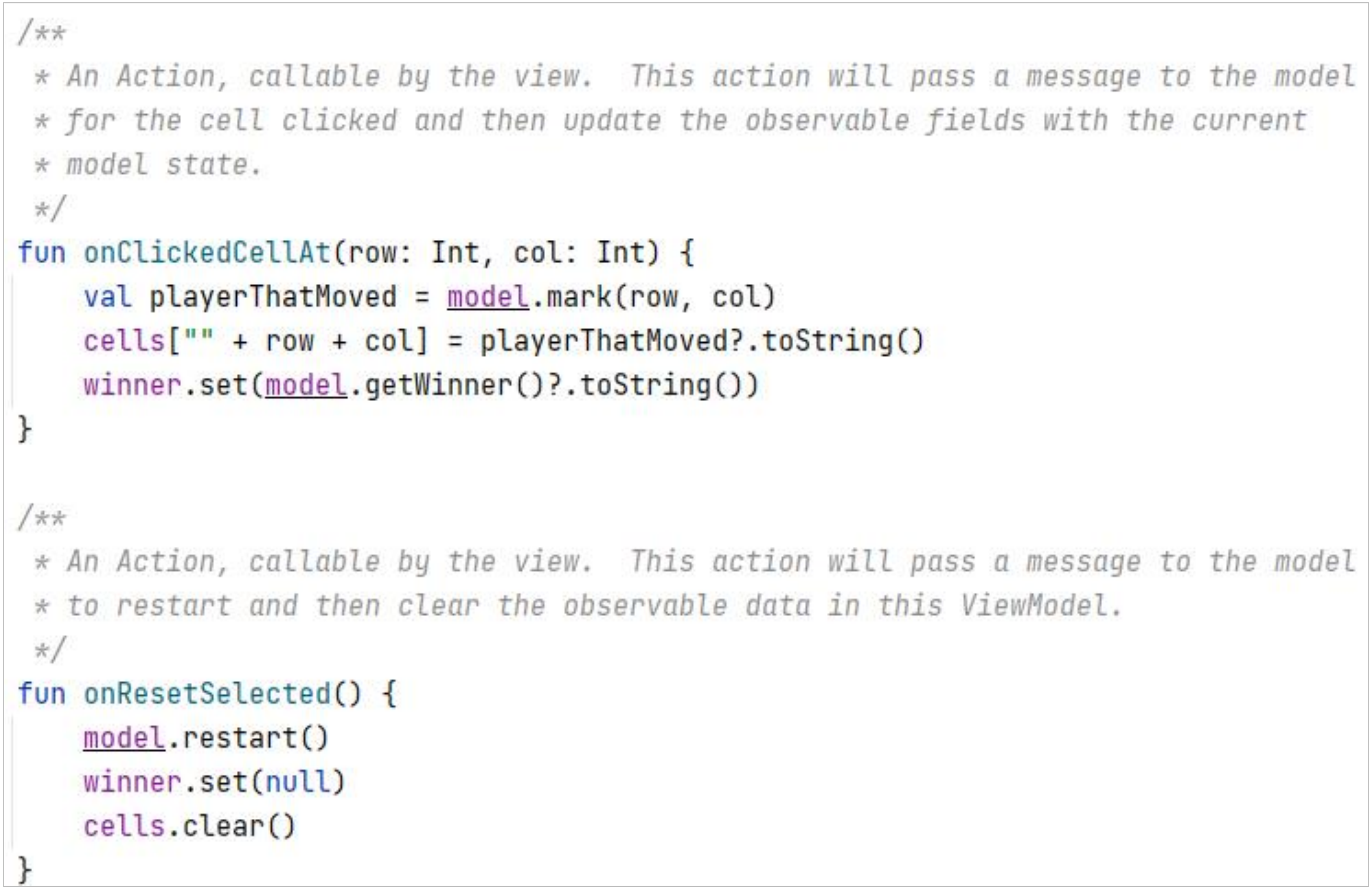
```
class TicTacToeViewModel : ViewModel {
    private var model: Board = Board()

    /*
     * These are observable variables that the viewModel will update as appropriate
     * The view components are bound directly to these objects and react to changes
     * immediately, without the ViewModel needing to tell it to do so. They don't
     * have to be public, they could be private with a public getter method too.
     */
    val cells: ObservableArrayMap<String, String> = ObservableArrayMap()
    val winner: ObservableField<String> = ObservableField()

    // As with presenter, we implement standard lifecycle methods from the view
    // in case we need to do anything with our model during those events.
    override fun onCreate() {}
    override fun onPause() {}
    override fun onResume() {}
    override fun onDestroy() {}
```



Actual Implementation

https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

Implementation



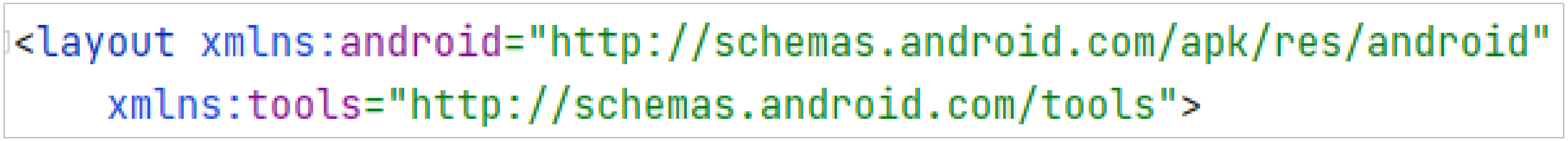
```
/**
 * An Action, callable by the view.  This action will pass a message to the model
 * for the cell clicked and then update the observable fields with the current
 * model state.
 */
fun onClickedCellAt(row: Int, col: Int) {
    val playerThatMoved = model.mark(row, col)
    cells["" + row + col] = playerThatMoved?.toString()
    winner.set(model.getWinner()?.toString())
}

/**
 * An Action, callable by the view.  This action will pass a message to the model
 * to restart and then clear the observable data in this ViewModel.
 */
fun onResetSelected() {
    model.restart()
    winner.set(null)
    cells.clear()
}
```

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

Data Binding in XML

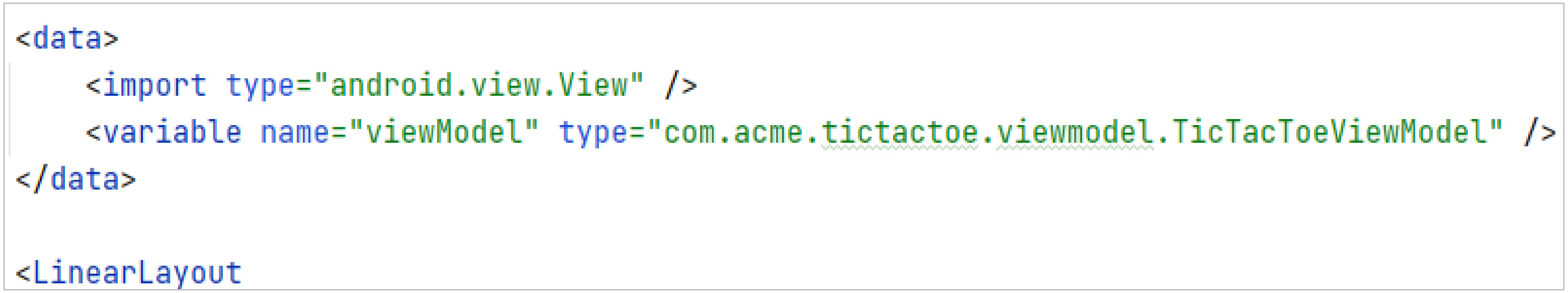
Important changes to note about the databinding layout XML:

1. The root element is **<layout>** and this element is also the visual root layout of the view.



```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
```

2. The <layout> tag is followed by a **<data>** element: used to define the variable we wish to use in binding expressions and import any other classes need for reference, like android.view.View.



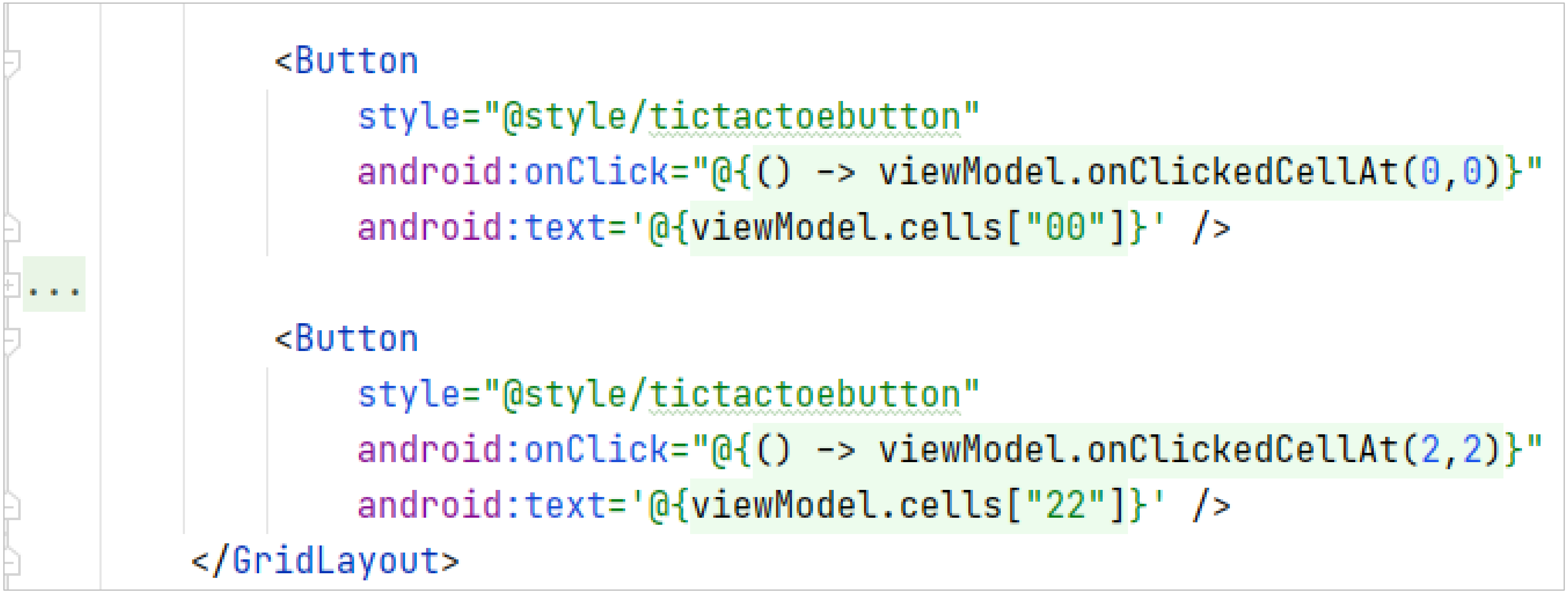
```
<data>
    <import type="android.view.View" />
    <variable name="viewModel" type="com.acme.tictactoe.viewmodel.TicTacToeViewModel" />
</data>


<LinearLayout
```

Note the variable named ***viewModel*** with type ***TicTacToeViewModel***

3. The *view root* element follows the <data> element. This is normally the root element in non-binding layout files (*LinearLayout* in above case).

Data Binding in XML

In layout, the button clicked will invoke the onClickedCellAt method with its row, col. The display value comes from the ObservableArrayMap defined in the ViewModel.

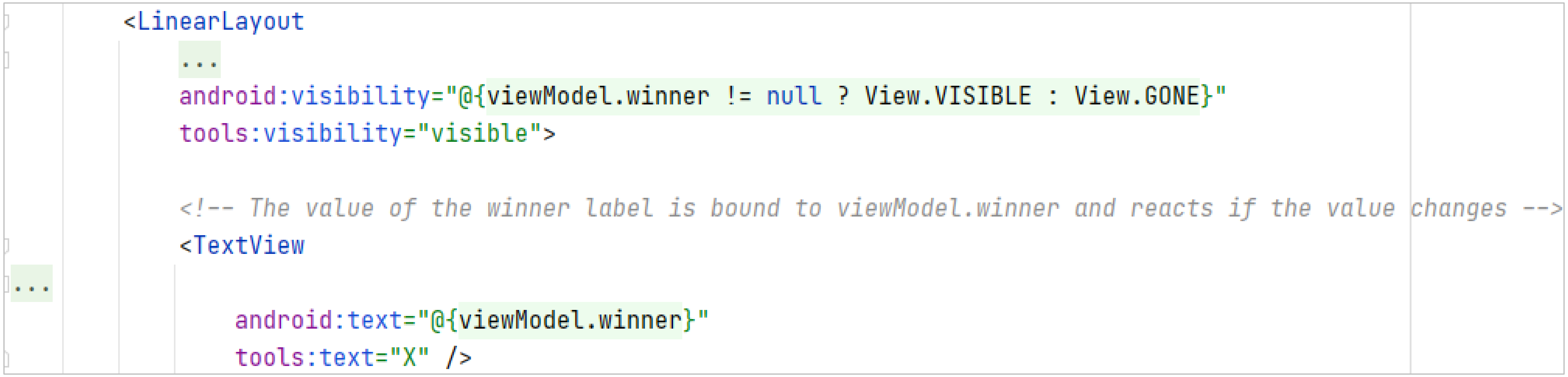


```
        <Button
            style="@style/tictactoebutton"
            android:onClick="@{() -> viewModel.onClickedCellAt(0,0)}"
            android:text='@{viewModel.cells["00"]}' />
...

        <Button
            style="@style/tictactoebutton"
            android:onClick="@{() -> viewModel.onClickedCellAt(2,2)}"
            android:text='@{viewModel.cells["22"]}' />
    </GridLayout>
```

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

Data Binding in XML

The visibility of the winner view group is based on whether or not the winner value is null.



```
    <LinearLayout
        ...
        android:visibility="@{viewModel.winner != null ? View.VISIBLE : View.GONE}"
        tools:visibility="visible">

        <!-- The value of the winner label is bound to viewModel.winner and reacts if the value changes -->
        <TextView
...
            android:text="@{viewModel.winner}"
            tools:text="X" />
```

Please check out the Android Data Binding documentation for more.

https://developer.android.com/topic/libraries/data-binding/index.html

Original Java code version available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/
Adapted Kotlin version can be downloaded on Black Board

Implication for Testing

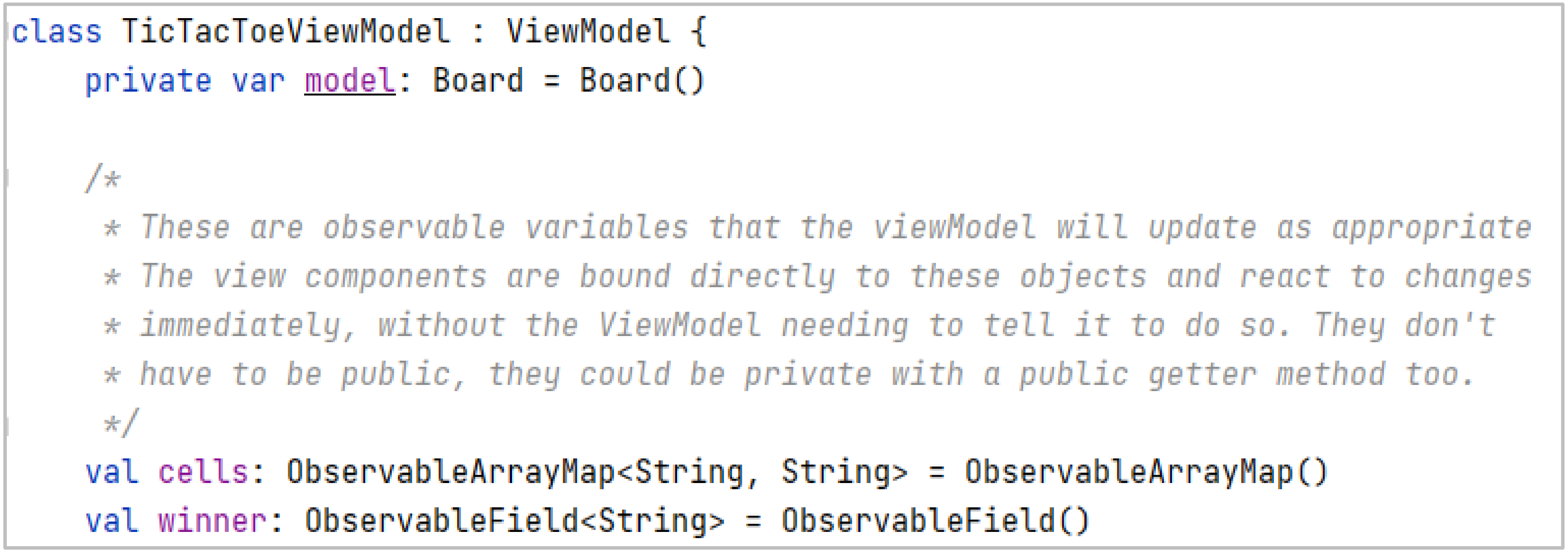
- Unit testing is even easier now, ViewModel has no dependency on the view.
- Only need to verify that the observable variables are set appropriately when the model changes.
- No need to mock out the view for testing as there was with the MVP pattern.

MVVM Highlights

- Maintenance
  - Views can bind to both variables and expressions.
  - Extraneous presentation logic can creep in over time.
  - Use of expression language effectively adds code to XML.
  - To avoid this, always get values directly from the observable field (which again can be unit tested) rather than attempt to compute or derive them in the views binding expression

MVVM Highlights

- Tries to remove the amount of glue code that we have to write to connect the view + model
- The view binds to observable variables and actions exposed by the ViewModel in a flexible way.



```
class TicTacToeViewModel : ViewModel {
    private var model: Board = Board()

    /*
     * These are observable variables that the viewModel will update as appropriate
     * The view components are bound directly to these objects and react to changes
     * immediately, without the ViewModel needing to tell it to do so. They don't
     * have to be public, they could be private with a public getter method too.
     */
    val cells: ObservableArrayMap<String, String> = ObservableArrayMap()
    val winner: ObservableField<String> = ObservableField()
```

MVVM Pros & Cons

- Pros
  - First Party Library
  - Compile time checking
  - Presentation layer in XML
  - Testable
  - Less code, no more Butterknife etc
    https://www.vogella.com/tutorials/AndroidButterknife/article.html
- Cons
  - Data Binding is not always appropriate
  - Android Studio integration was flaky (keyword was)

Conclusions

- Both MVP and MVVM do a better job than MVC in breaking down your app into modular.
- Single purpose components, but they can also been seen as adding more complexity to simple app.
- For a very simple application with only one or two screens and none complex behaviour, MVC may do just fine.
- MVVM with data binding is attractive as it follows a more reactive programming model and produces cleaner code.

Note: app templates in Android studio typically are structured for MVC, MVVM or neither.

Summary

- Android design patterns: repeatable solution to common software problems
- MVC – best for smaller apps
- MVP – separates view logic from business logic, but requires some boiler plate coding, improved testing
- MVVM – improved MVP, reduced boiler plate coding, highly testable