MVC and Android

Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong, and Dr. Abram Hindle

Model/View/Controller Roles

- Model:
  - Entity layer
    - Complete, self-contained representation of the data managed by the application
    - Provides services to manipulate this data
    - “The back end”
  - Main responsibilities
    - Representation and computation issues
    - Sometimes persistence

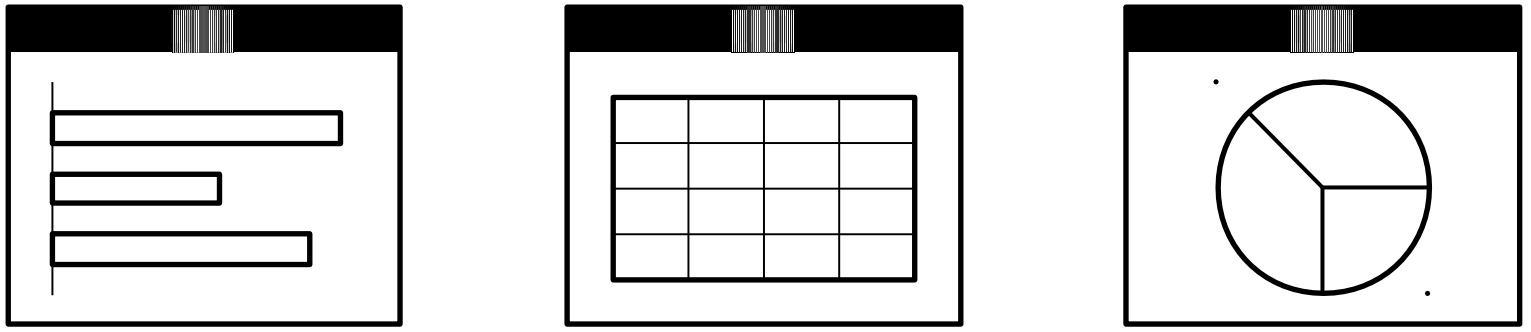
MVC Roles

- View:
  - Boundary layer
    - Set of user interface components determines what is needed for a particular perspective of the data
    - “The front end” or what the user sees
  - Main responsibility
    - Display the current data
    - Presentation issues

MVC Roles

- Controller:
  - Control layer
    - Handles events and uses appropriate information from user interface components to modify the model
  - Main responsibility
    - Interaction issues

Views



Need to maintain consistency in the views

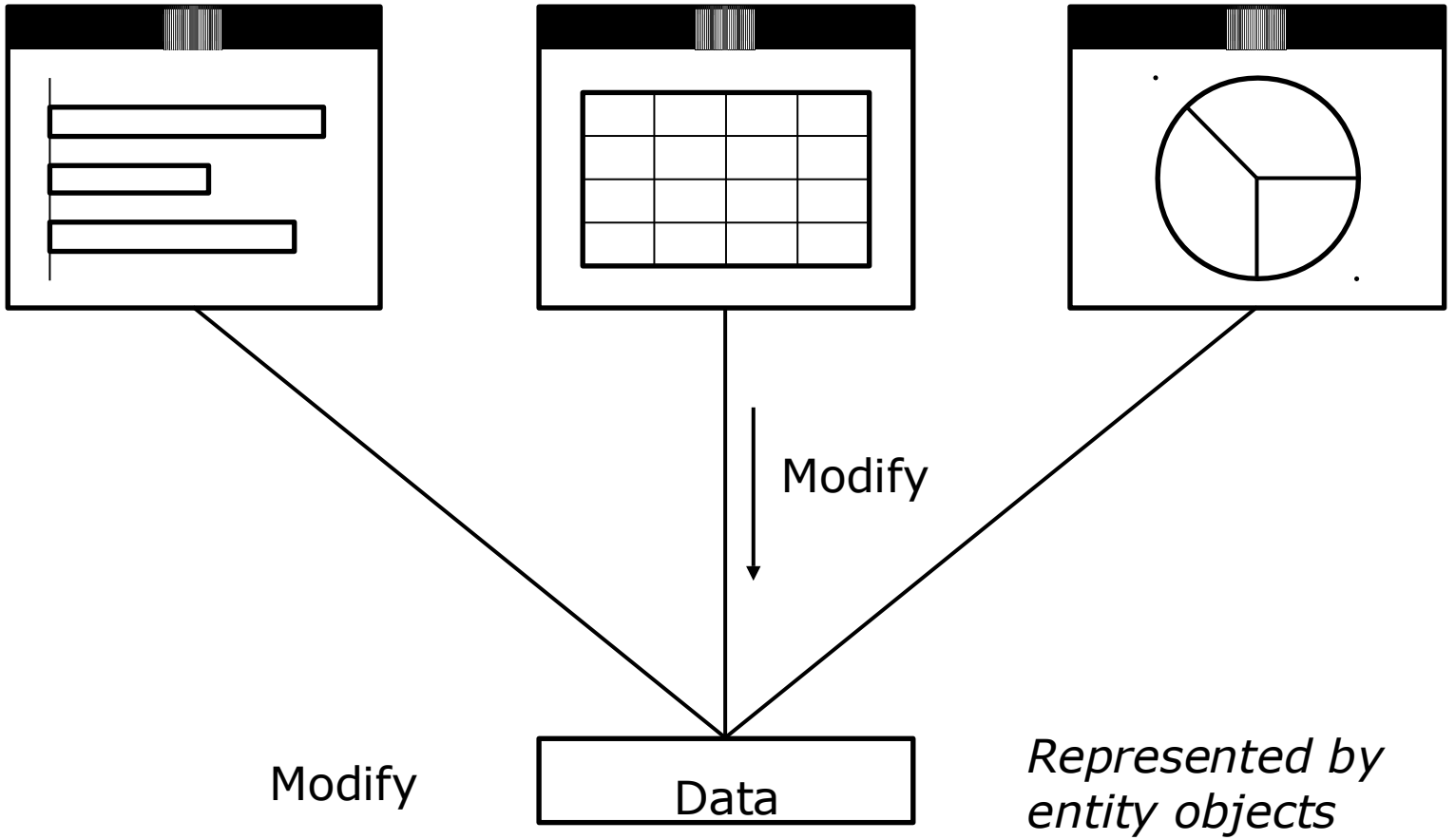
Want clear, separate responsibilities for presentation, interaction, computation, and representation

Data

Need to update multiple views of the common data model

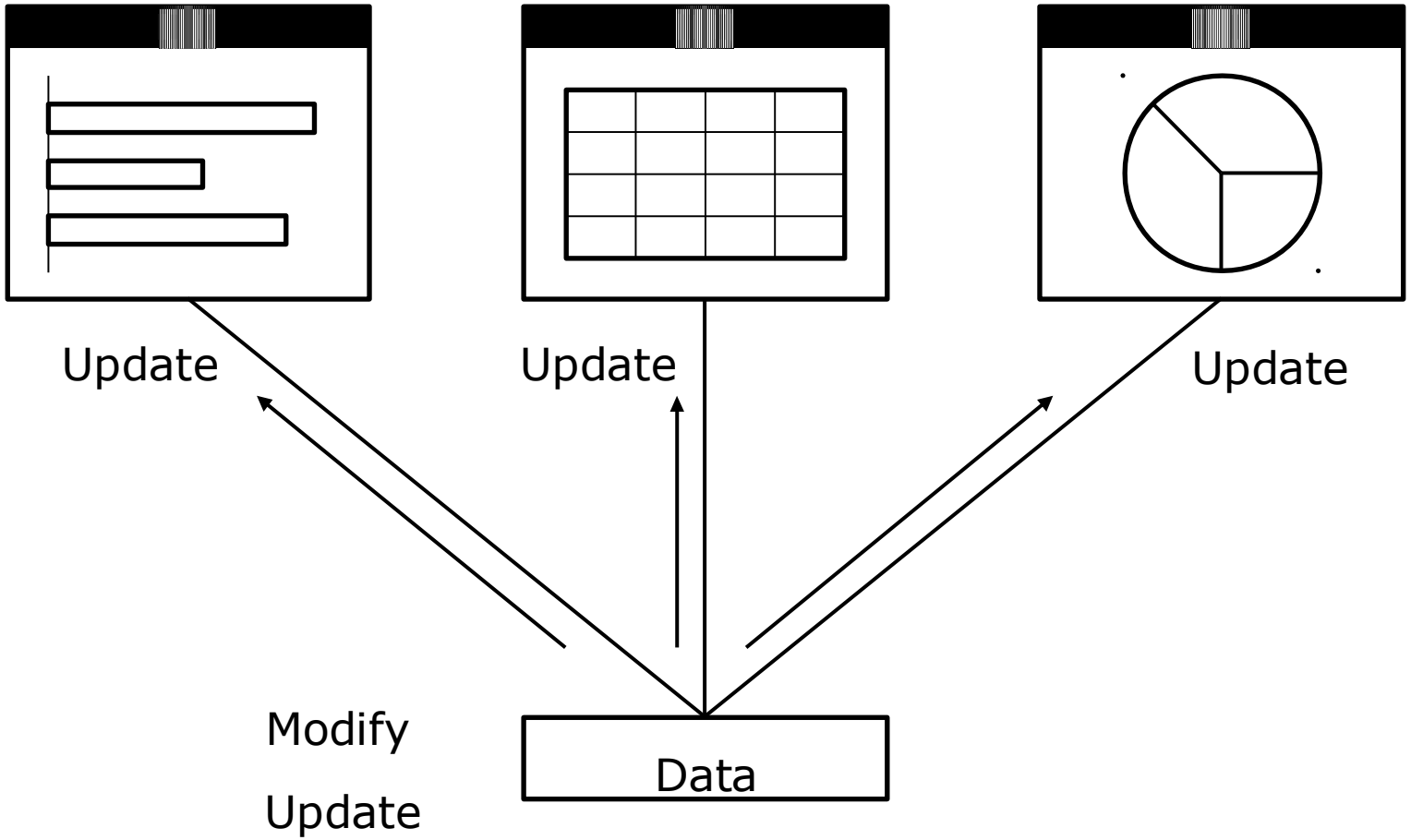
Model

Views (i.e., observers, clients)



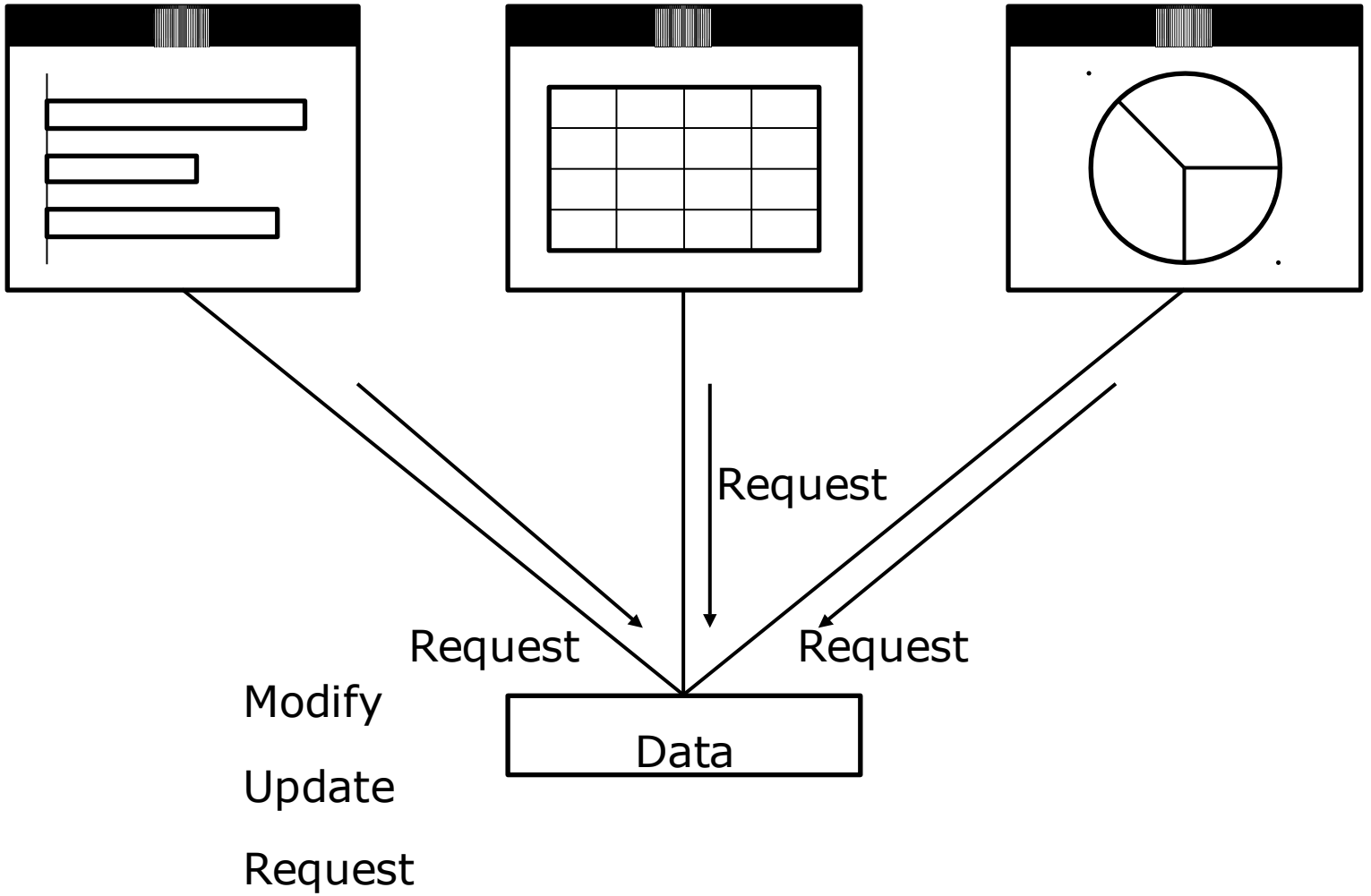
Model (i.e., subject, server)

Views (i.e., observers, clients)



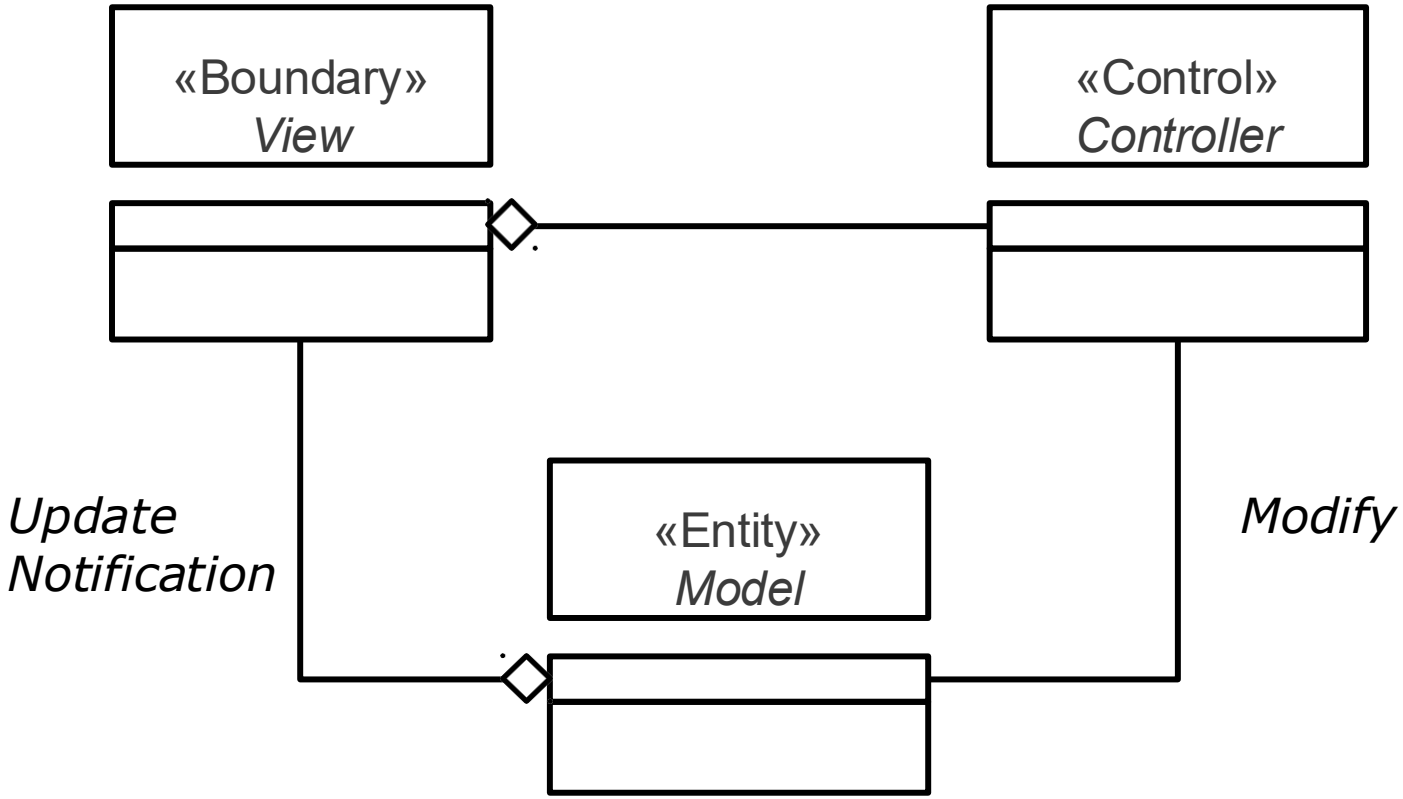
Model (i.e., subject, server)

Views (i.e., observers, clients)



Model (i.e., subject, server)

Can create new, specific types of views without changing the model



The model should not need to know the particulars of a specific view

The model should not need to know about any controllers

MVC Sequence

- User interacts with the View
- Controller handles the interaction
- Controller modifies the Model
- Model changes state
- View updates based on the Model

MVC Design Issues

- Android UI-Dependent part:
  - **Views** contain Android UI components
    - XML layouts, Compose UI, Button, TextView, RecyclerView, etc.
  - **Controllers** handle Android UI events
    - Click listeners, text-change listeners, menu actions, navigation events
- Android UI-Independent part:
  - **Model** should be as Android-free as possible
    - E.g., avoid using Android UI types in entity/data classes
      - Avoid TextView, Button, Context, Activity, Fragment, etc. in Model classes

“MV” Design

- Generalization:
  - Use “model” superclass and “view” interface
    - All models keep track of their views
    - When changed, all models notify their views to update
    - All views update themselves when notified
  - Have application-specific model and view classes

Model-View Update Example

```
interface ViewObserver<M> {
    fun update( model: M )
}

open class ObservableModel<M> {
    private val observers = mutableListOf< ViewObserver<M> >()

     // “all models keep track of their views”
    fun addObserver( o: ViewObserver<M> ) { … }
    fun removeObserver( o: ViewObserver <M> ) { … }

     // “all models notify their views to update”
    protected fun notifyObservers( model: M ) { … }

}
```

Model-View Update Example

```
// MyModel.kt
class MyModel : ObservableModel<MyModel>() {
    var message: String = “”
        private set

    fun updateMessage( newMessage: String ) {
        message = newMessage
        notifyObservers(this) // notify views to update
    }
}
```

Model-View Update Example

```
// MyView.kt
class MyView : ViewObserver<MyModel> {
     override fun update( model: MyModel ) {
          println( model.message() )
     }
}
```

Model-View Update Example

```
// MyApp.kt
fun main() {
        val theModel = MyModel()
        val aView = MyView()
        val anotherView = MyView()

        theModel.addObserver( aView )
        theModel.addObserver( anotherView )

        theModel.updateMessage( "hello" )
}
```

MVC Design

- Approach:
  - Use a framework that supports MVC to help structure an interactive application
  - Framework is a set of cooperating classes, interfaces, and conventions that forms a reusable design in a particular domain
  - Reusable design *and* code

MVC Framework

Who is in Control?

- Class library reuse
  - Application developers:
    - Write the main body of the application
    - Reuse library code by calling it
- Framework reuse
  - Application developers:
    - Reuse the main body of the application
    - Write code that the framework calls
    - Reuse library code by calling it

Framework

- Separation of concerns:
  - Framework
    - Skeletal application code
    - General superclasses and interfaces
  - Your “customizations”
    - Specific subclasses and implementations
    - Define app-specific models, views, and controllers

Exercise

- Design an MVC framework for building interactive applications.

Generic MVC Structure

```
// MVC.kt
…
interface View< M > {
    fun render( model: M )
}

interface Controller {
    fun onUserAction() // Handles user actions
}
```

Generic Model

- Generic Model Rules:
  - The Model stores application state
  - The Model contains application rules and calculations
  - The Model should avoid Android UI types
  - The Model may notify Views when its state changes

Generic Model

```
// ObservableModel.kt
…

open class ObservableModel<M> {
    private val views = mutableListOf<View<M>>()

    fun addView( view: View<M> ) {
        if ( view !in views ) {
            views.add( view )
        }
    }
```

Generic Model

```
fun removeView(view: View<M>) {
    views.remove(view)
}

protected fun notifyViews( model: M ) {
    views.forEach { view ->
        view.render( model )
    }
}
```

Generic Controller

```
// Controller.kt
…
class MyController(
    private val model: MyModel
) {
    fun onUserAction() {
        // validate input
        // modify the model
    }
}
```

Generic View

```
// MyView.kt

…
class MyView : View<MyModel> {
    override fun render( model : MyModel) {
        // update the UI using model state
    }
}
```

Connecting MVC

```
// MyApp.kt
…

    val model = MyModel()
    val view = MyView()
    val controller = MyController(model)

    model.addView(view)

    // UI event delegates to the controller
    // e.g., when a button is clicked, call
       controller.onUserAction()
}
```

Multiple Views

```
// MyApp.kt
…

    val model = MyModel()

    model.addView( summaryView )
    model.addView( detailView )
    model.addView( chartView )
    // One Model can update multiple Views
}
```

Generic MVC Flow

- User action:
  - User interacts with View
  - View delegates interaction to Controller
- Controller response:
  - Controller validates and handles the interaction
  - Controller invokes changes on the Model
- Model update:
  - Model changes state
  - Model notifies registered Views
- View refresh:
  - Views update based on the Model

“Code Reuse”

- http://www.dilbert.com/strips/comic/1996-01-31/

Example Template

```
class SomeModel : ObservableModel<SomeModel>() {
    fun changeState() {
        // update state
        notifyViews(this)
    }
}
class SomeView : View<SomeModel> {
    override fun render( model: SomeModel ) {
        // display model state
    }
}
class SomeController(
    private val model: SomeModel
) {
    fun onUserAction() {
        model.changeState()
    }
}
```

MVC Responsibility Checklist Example

Question	Belongs in
What data does the application store?	Model
What rules protect or change the data?	Model
What methods change application state?	Model
How is the current state displayed?	View
Which UI components are shown?	View
What happens when the user clicks or types?	Controller
Which Model method should be called for this action?	Controller
Who notifies registered Views after state changes?	Model
Who updates the screen after notification?	View
Who delegates the user interaction to the Controller?	View

Implementing MVC

- Framework:
  - Implement using generic base classes and interfaces for Model and View

TView

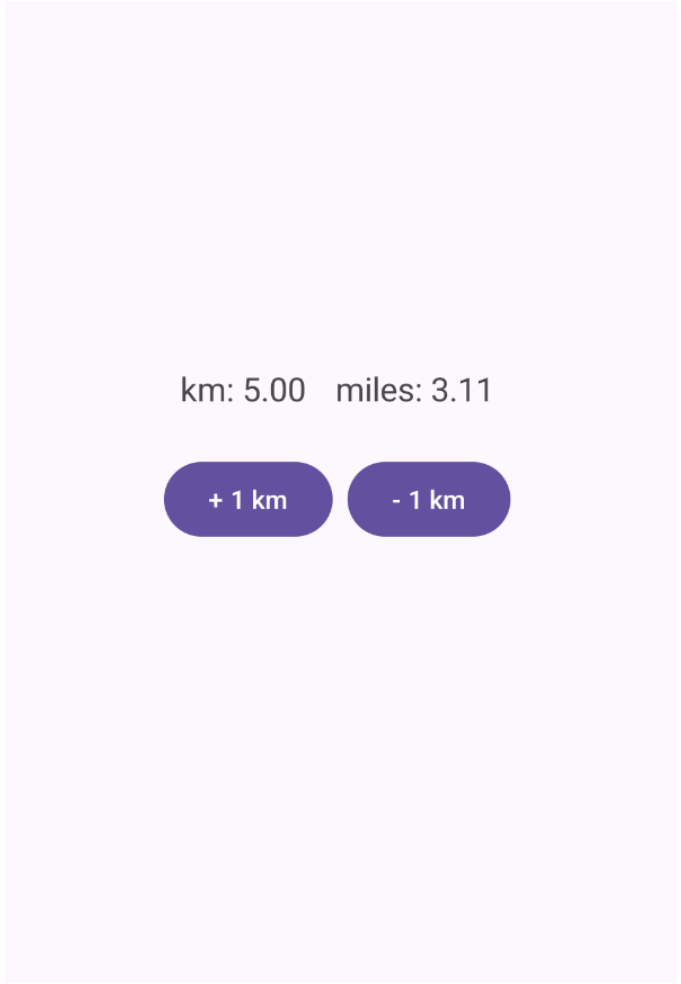
```
// TView.kt
interface TView<M> {
    fun update( model: M )
}
```

TModel

```
// TModel.kt

abstract class TModel<M, V: TView<M> > {
    private val views = mutableListOf<V>()

    fun addView( view: V ) {
        if ( view !in views ) {
            views.add( view )
        }
    }
```

```
    fun deleteView( view: V ) {
        views.remove( view )
    }

    protected fun notifyViews(model: M) {
        views.forEach { view ->
            view.update( … )
        }
    }

    …
}
```

Example Custom Application



Custom View Interface

```
// TView.kt

…
interface TView<M> {
    fun update( model: M )
}
```

Custom Model

```
// TModel.kt
…
abstract class TModel<M> {
    private val views = mutableListOf< TView<M> >()

    fun addView( view: TView<M> ) {
        if ( view !in views ) {
            views.add( view )
        }
    }
```

Custom Model

```
    fun deleteView( view: TView<M> ) {
        views.remove( view )
    }

    protected fun notifyViews(model: M) {
        views.forEach { view ->
            view.update(model)
        }
    }
}
```

Custom Model

```
// MyModel.kt
…
class MyModel: TModel<MyModel>() {
     var value: Int = 0
          private set

     fun setValue ( newValue: Int ) {
          value = maxOf( 0, newValue )
          notifyViews( this )
     }
}
```

Custom View

```
// MyView.kt
…
class MyView(
    private val prefixText: String,
    private val multiplier: Double
) : TView< MyModel > {

        var displayedText: String = “”
            private set
```

Custom View

```
    override fun update( model: MyModel ) {
        val value = model.value * multiplier
        displayedText =
              “$prefixText${ “%.2f”.format( value ) }”
        println( displayedText )
    }
}
```

Generic Command

```
// TCommand.kt
…
fun interface TCommand {
    fun execute()
}
```

Custom Controller

```
// MyController.kt
…
class MyController(
    private val model: MyModel
) {

    val increment = TCommand {
        model.setValue( model.value + 1 )
    }

    val decrement = TCommand {
        model.setValue( model.value - 1 )
    }
}
```

Custom Application

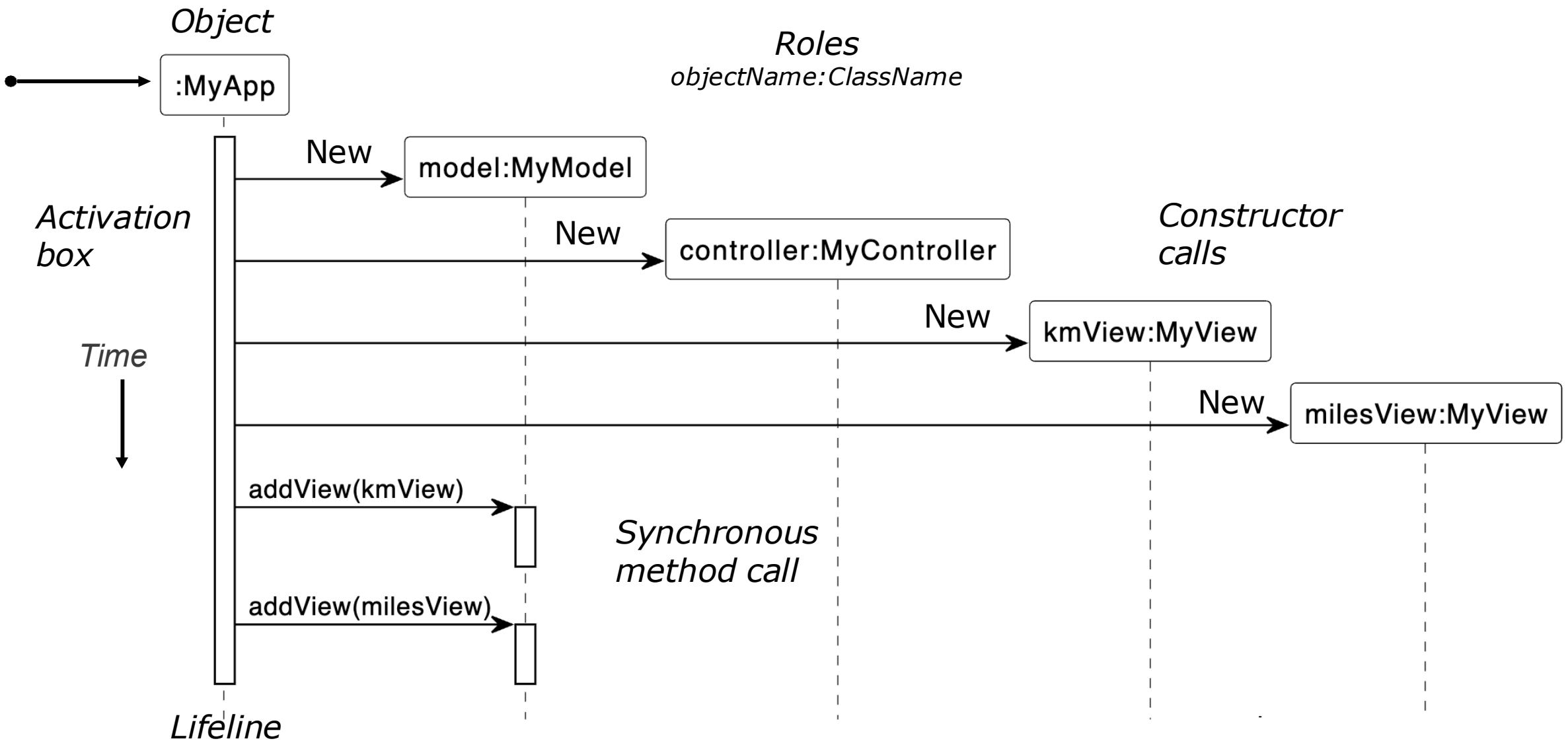
```
// MyApp.kt
…
fun main() {
  // create model, views, and controller
    val model = MyModel()
    val controller = MyController( model )
    val kmView = MyView( “km: ”, 1.0 )
    val milesView = MyView(
        “miles: ”,
        0.621371192
    )
```

Custom Application

```
    // register views with model
     model.addView(kmView)
     model.addView(milesView)

    // UI event handling
     incrementButton {
         controller.increment.execute()
     }

     decrementButton {
         controller.decrement.execute()
     }
}
```

The controller executes the increment or decrement command, updates the model, and the model notifies its registered views.

UML Sequence Diagram



Exercise

- Draw a UML sequence diagram for the behaviour when the increment button is clicked in the example application.

Events

- Interactive applications are event driven:
  - Receive an event (e.g., initiated from user)
  - Check event and system state
  - Respond by changing state and display
  - Return and wait for another event
- Event handling is done via:
  - Explicit event loops, event queues, and dispatchers
  - Callbacks or event handlers
  - *Registered callback through listeners*

More Information

- Books:
  - Head First Kotlin
    - D. Griffiths, D. Griffiths
    - O'Reilly, 2019
  - Kotlin in Action, Second Edition
    - S. Aigner, R. Elizarov, S. Isakova, D. Jemerov
    - Manning, 2024

More Information

- Books:
  - The Elements of UML 2.0 Style
    - S. W. Ambler
    - Cambridge, 2005
  - UML Distilled
    - M. Fowler
    - Addison-Wesley, 2003

More Information

- Links:
  - Get started with Kotlin
    - https://kotlinlang.org/docs/getting-started.html
  - Kotlin: An Illustrated Guide
    - https://typealias.com/start/
  - Kotlin API References
    - https://kotlinlang.org/docs/api-references.html
  - Android App Architecture
    - https://developer.android.com/topic/architecture