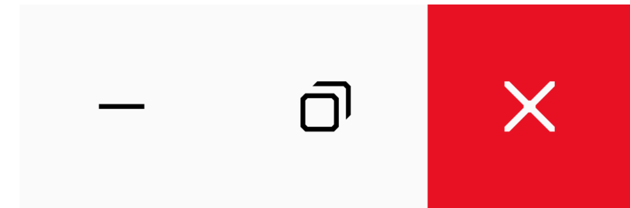
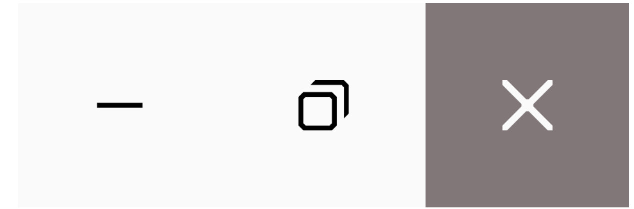


CS349: User Interfaces

Architectural Patterns for User Interfaces
Implementing MVC
Model–View–ViewModel

May 31

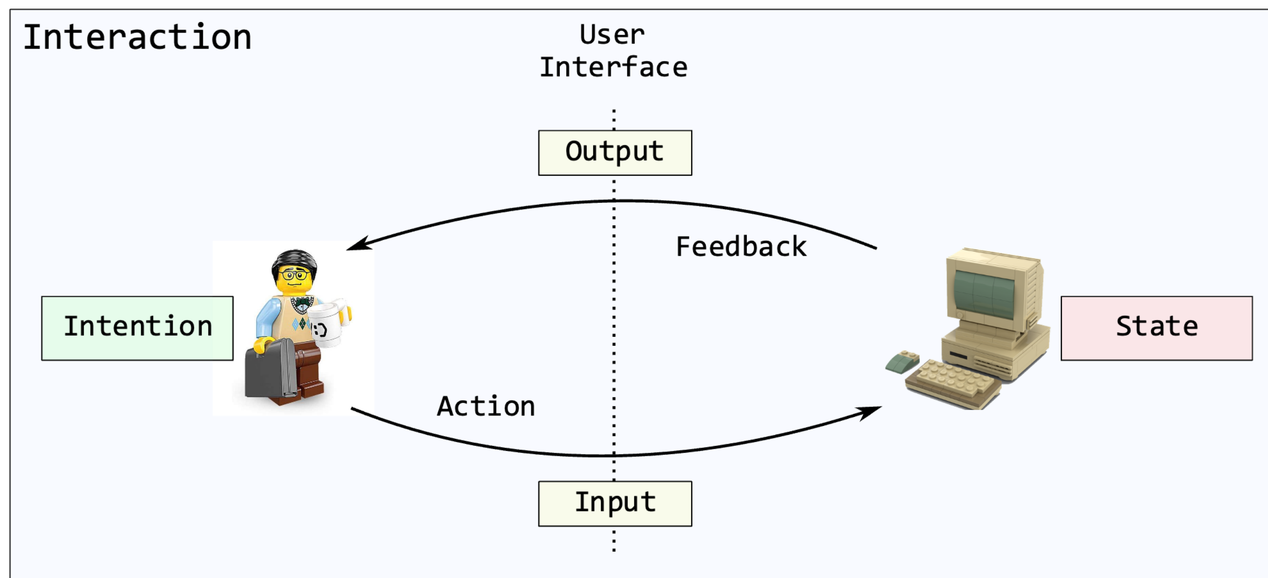


Architectural Patterns for User Interfaces

Recall: User Interaction

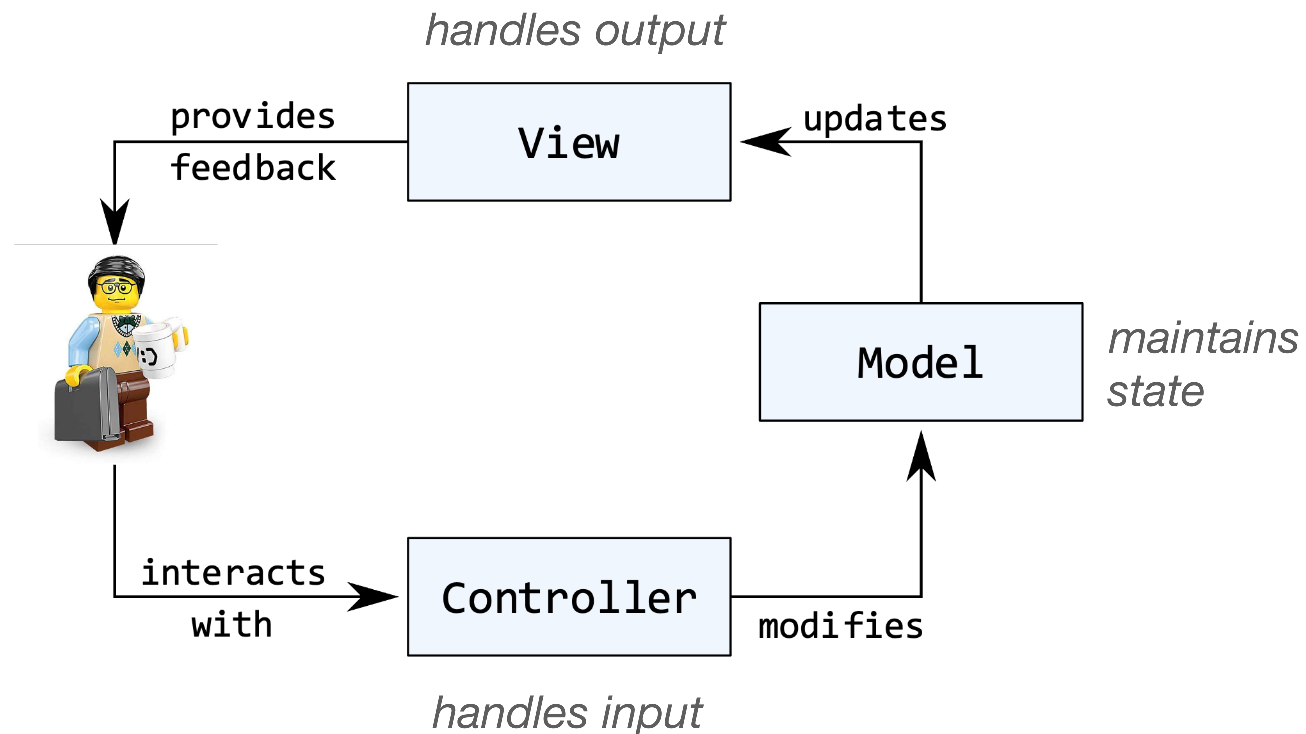
People typically interact with technology to reach a goal.

They determine what they need to do, and perform actions to the UI, which gives feedback in return.



User Interaction

We often design our systems to directly support this style of interaction. This architectural pattern is called *Model–View–Controller* (*MVC*).



Model-View-Controller (MVC)

Developed at Xerox PARC in 1979 by Trygve Reenskaug for Smalltalk-80, the precursor to Java (which is the precursor to Kotlin).

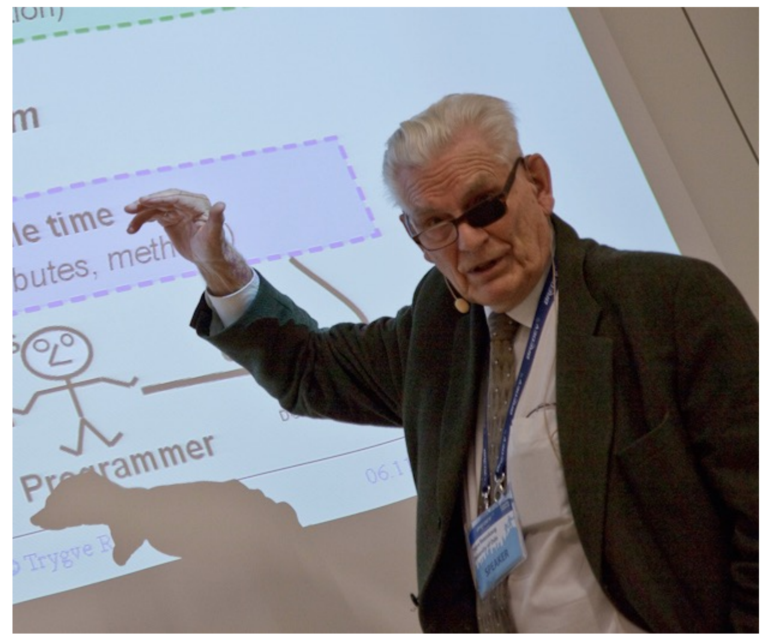
MVC *used* to be the standard design pattern for GUIs.

Used at many levels

- Overall application design
- Individual components

Variations of MVC have emerged:

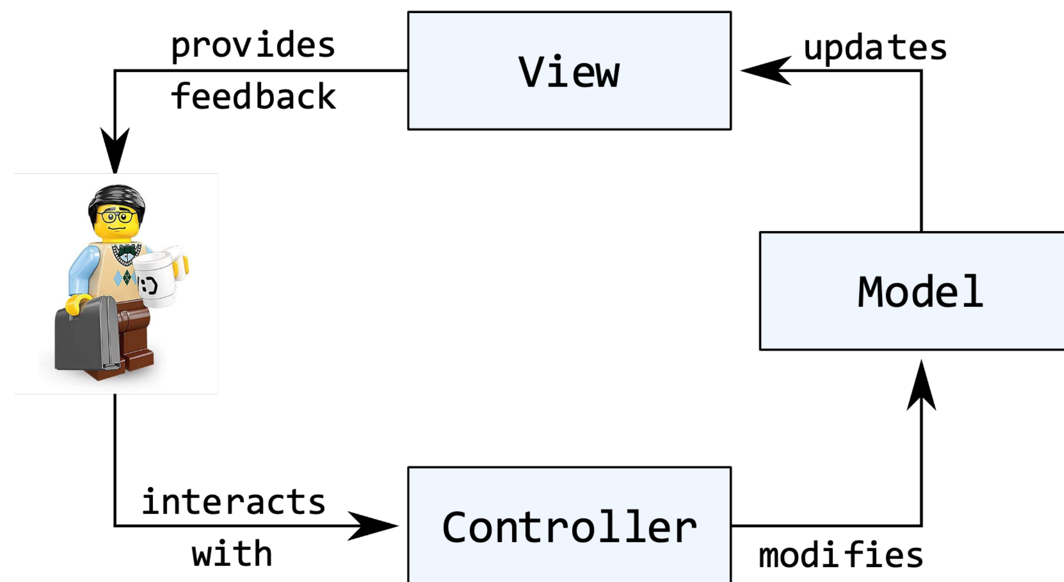
- Model–View–Presenter
- Model–View–Adapter
- Hierarchical Model–View–Controller



Why use MVC?

Separation of “business logic” and the user-interface logic.

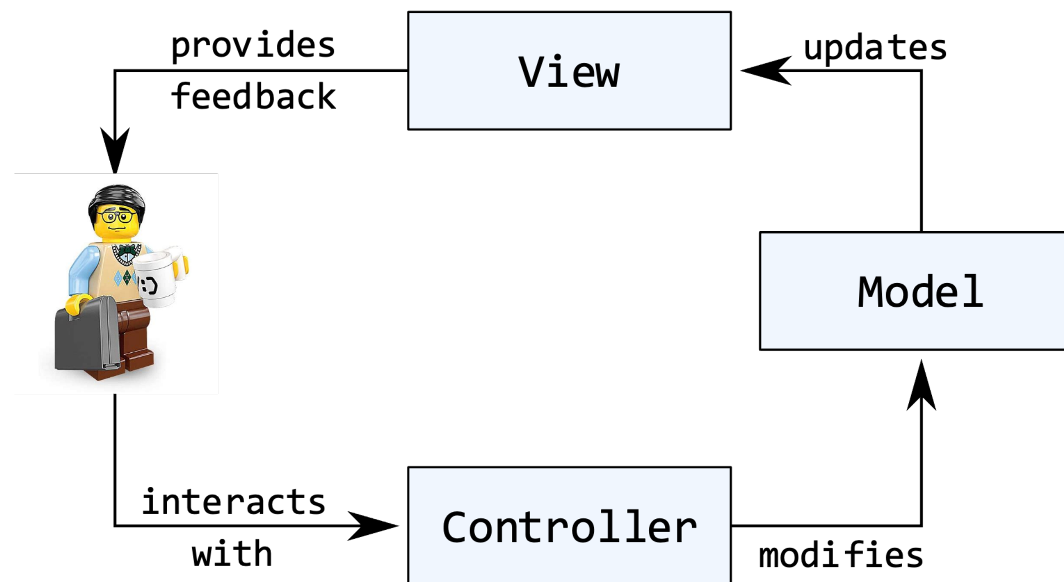
- View & Controller can be changed without changing the underlying Model.
- This is useful when
  - adding support for a new input device (e.g., voice control, touchscreen)
  - adding support for a new type of output device (e.g., different size screen)



Why use MVC?

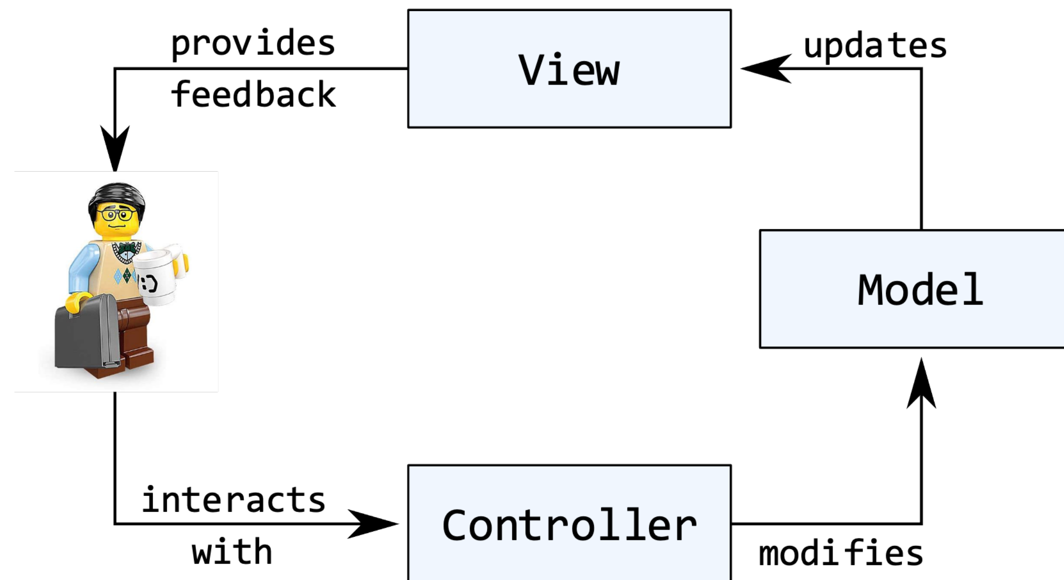
Supports multiple views of the underlying data / model. This is useful when

- viewing numeric data as a table, a line graph, a pie chart, etc.
- displaying simultaneous “overview” and “detail” views
- enabling “edit” and “preview” views



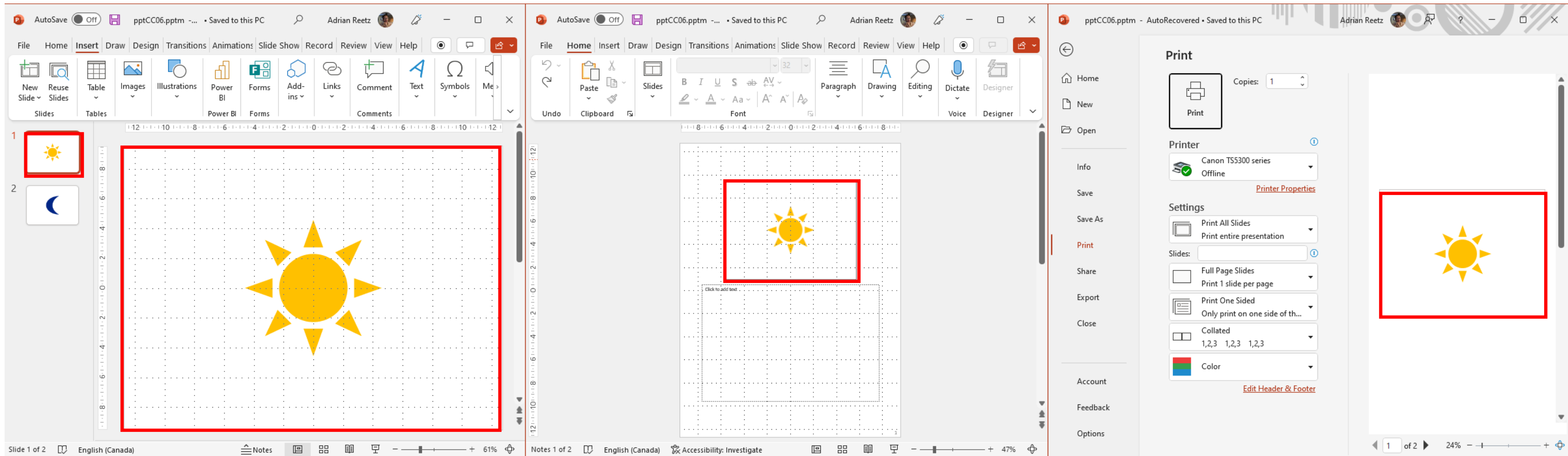
Why use MVC?

Separation of concerns in code (code reuse, ease of testing)



Why use MVC?

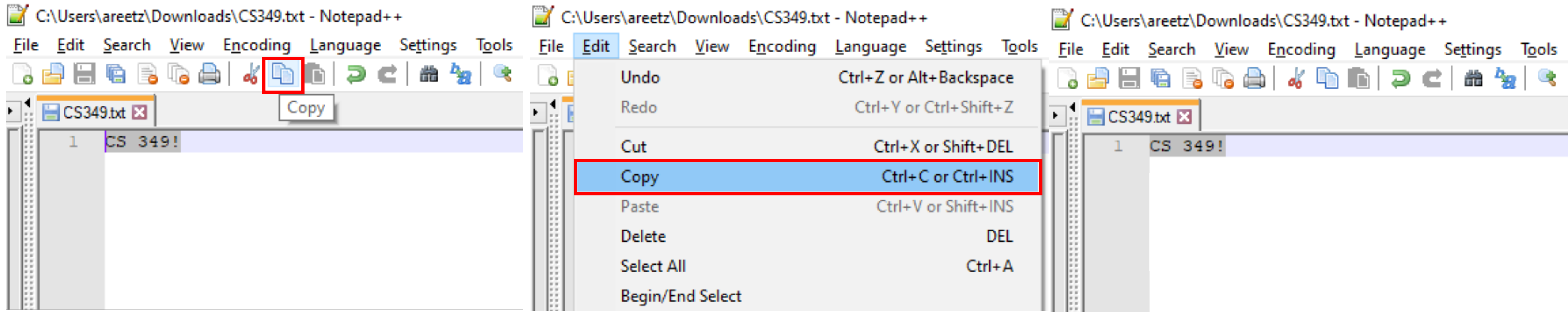
Four views of the same data:

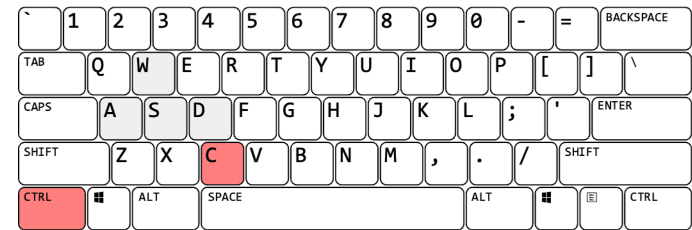


When one view changes, the others should change as well.

Why use MVC?

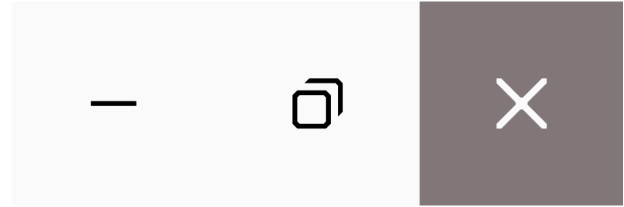
Three controllers for the same action:





All controllers should leverage the same underlying implementation.

When possible, we reuse code for common functionality.



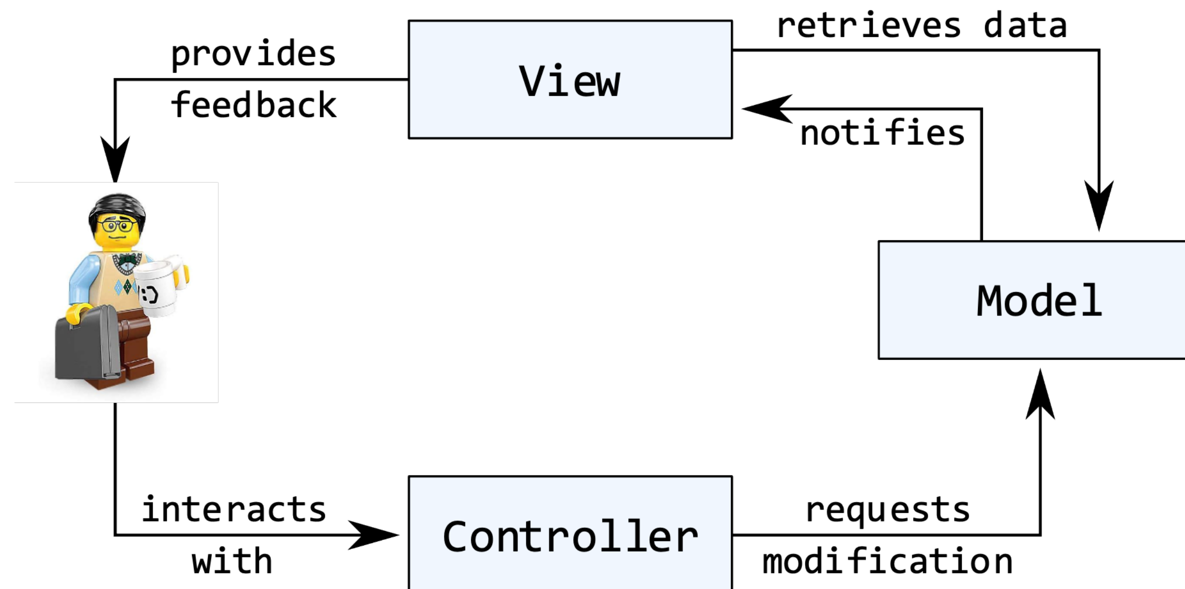
Implementing MVC

MVC Implementation

MVC implementations typically expand the **Observer** design pattern.

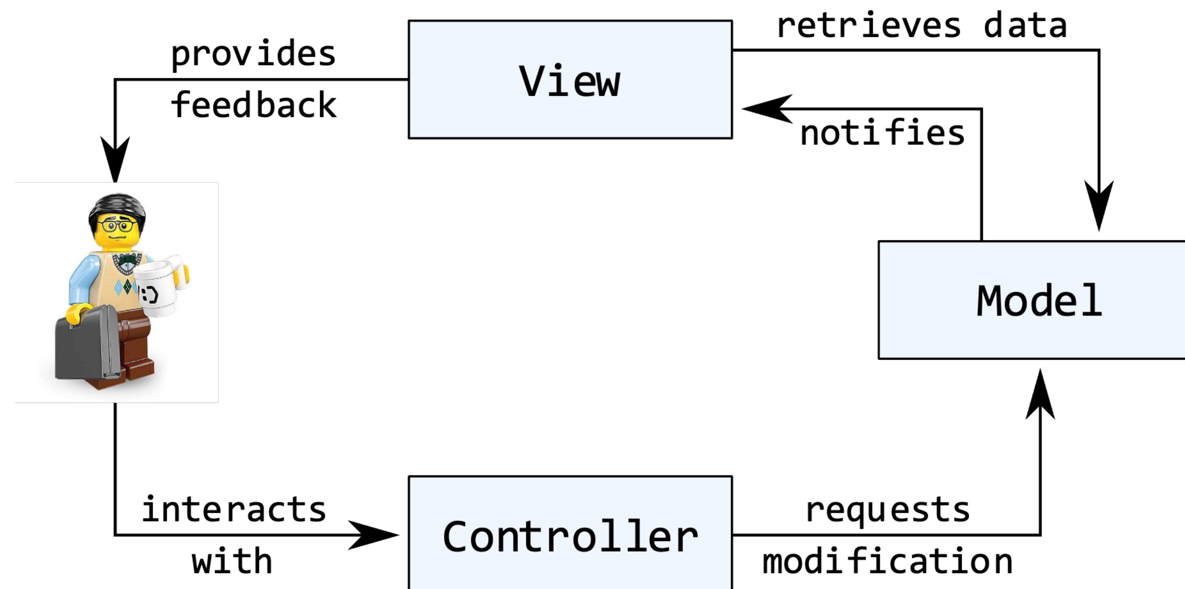
Interface architecture is decomposed into three parts:

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



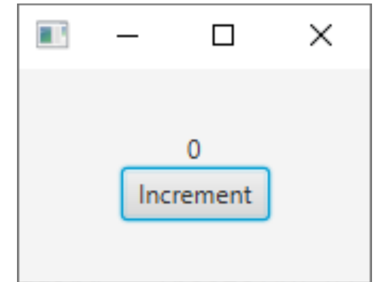
MVC Implementation

1. User performs action on Controller.
2. Controller asks Model to act upon input event.
3. Model might change state and notify View that state change occurred.
4. View retrieves updated state from Model and visualizes it.
5. User analyzes new state of the system based on feedback from View.



No MVC

No separation between Model, View, and Controller



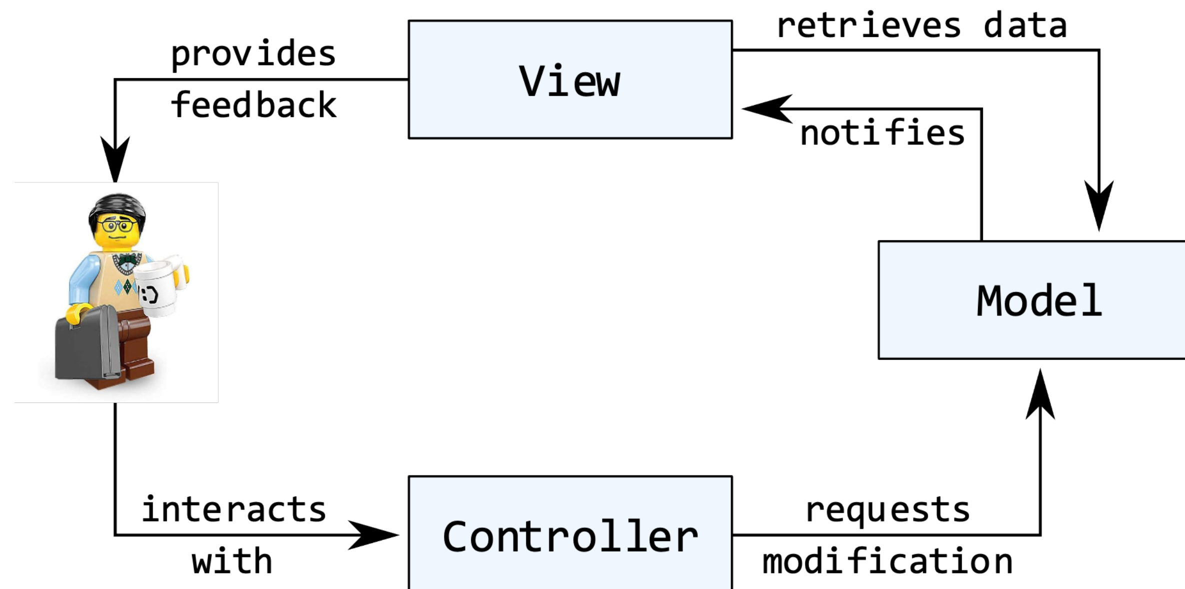

```
class NoMVC: VBox() {
    init {
        val countLbl = Label(0.toString())
        children.addAll(
            countLbl,
            Button("Increment").apply {
                onAction = EventHandler {
                    countLbl.text = (countLbl.text.toInt() + 1).toString()
                }
            })
        alignment = Pos.CENTER
    }
}
```

DEMO: JavaFX.MVC/01.NoMVC

MVC Implementation

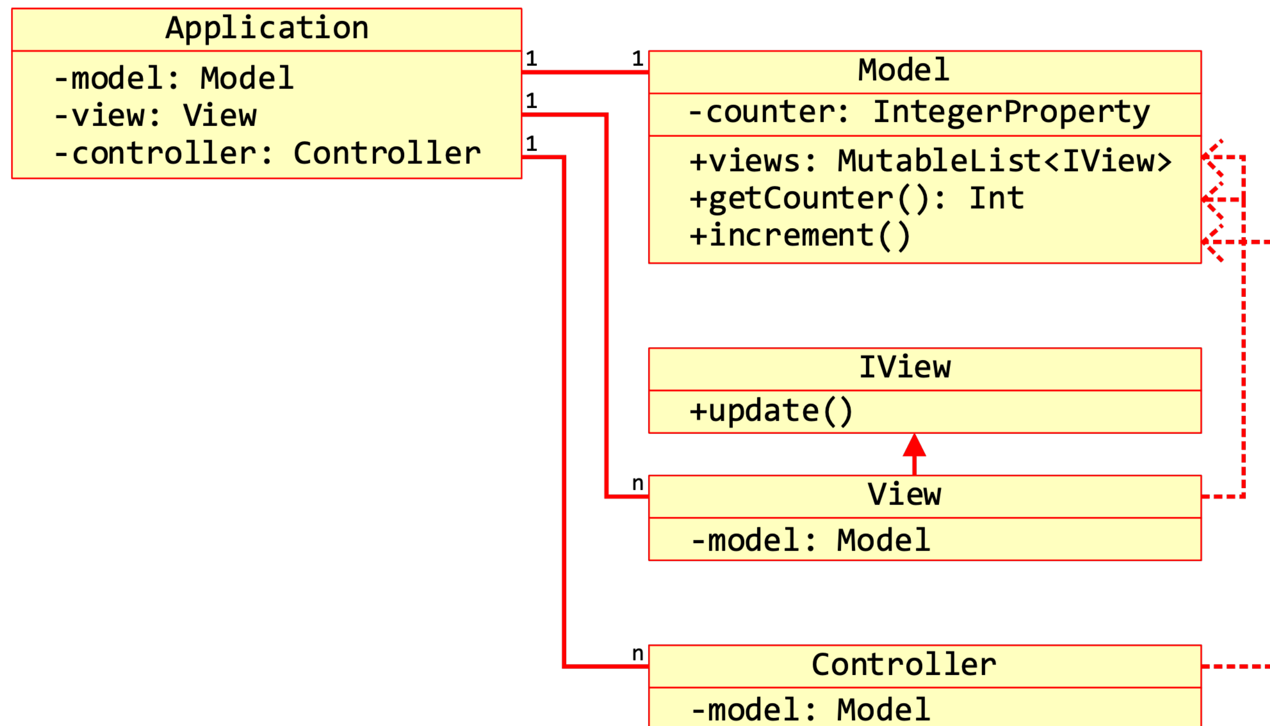
Interface architecture consisting of three parts:

- **Model**: stores system state and manages its modification
- **View**: presents system state and feedback
- **Controller**: allows for input to manipulate system state



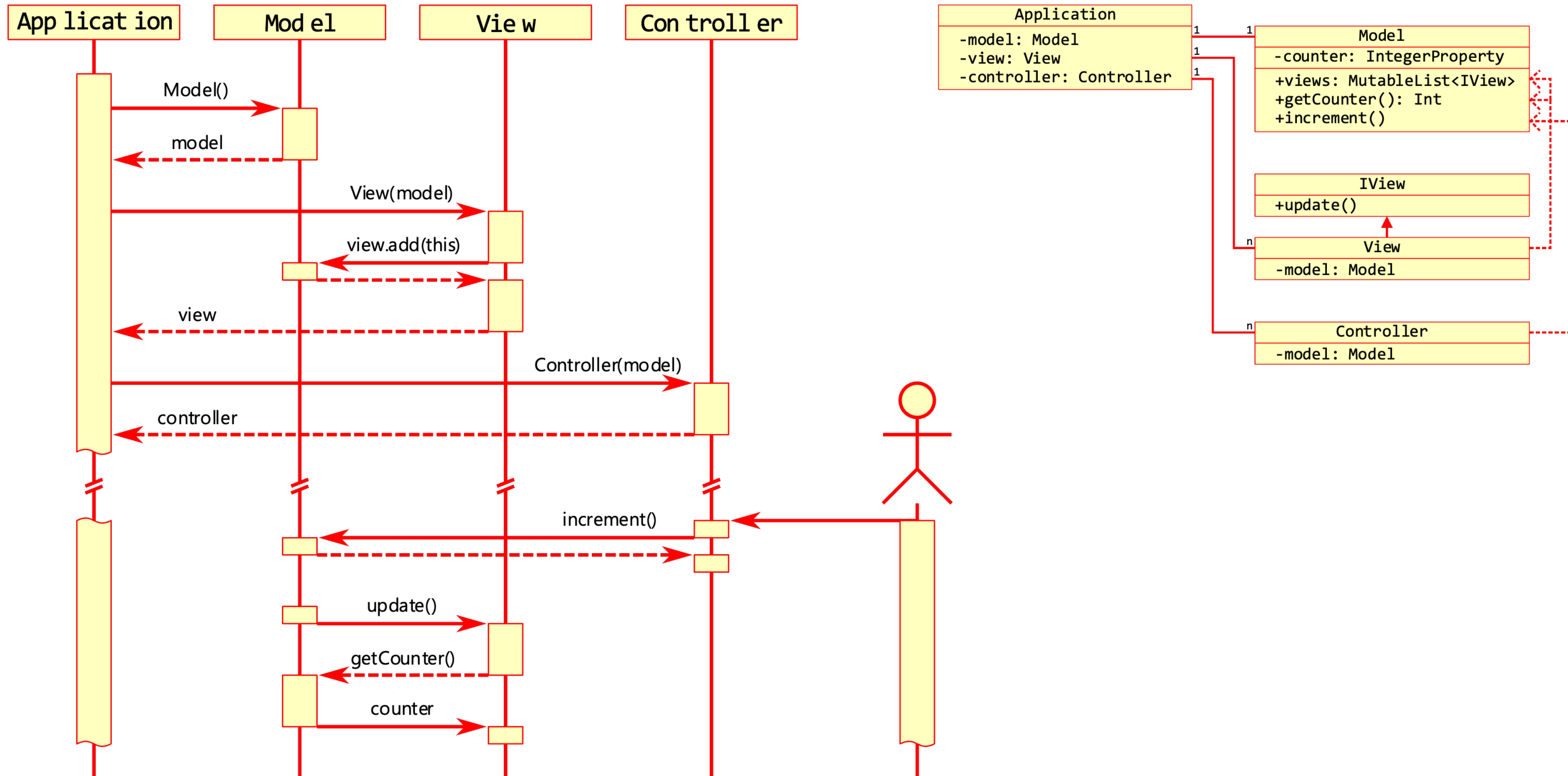
With MVC – Basic

Maximal separation between Model, View, and Controller



With MVC – Basic

Maximal separation between Model, View, and Controller



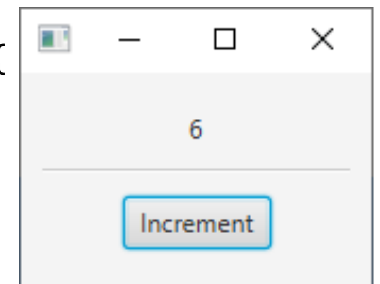
With MVC – Basic

```
class Model {

    private var counter = 0

    val views = mutableListOf<IView>()

    fun increment() {
        ++counter
        views.forEach { it.update() }
    }

    fun getCounter(): Int {
        return counter
    }
}
```

```
interface IView {
    fun update()
}

class View(private val model: Model): Label(), IView {

    init {
        model.views.add(this)
        update()
    }

    override fun update() {
        text = model.getCounter().toString()
    }
}
```

```
class Controller(model: Model):
 Button("Increment") {
    init {
        onAction = EventHandler {
            model.increment()
        }
    }
}
```

```
override fun start(stage: Stage) {

    val model = Model()
    val view = View(model)
    val ctrl = Controller(model)
    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(Vbox(view, Separator(), ctrl) ,
                        165.0, 100.0)
    }.show ()
}
```



DEMO: JavaFX.MVC/02. MVC.Basic

With MVC – Basic

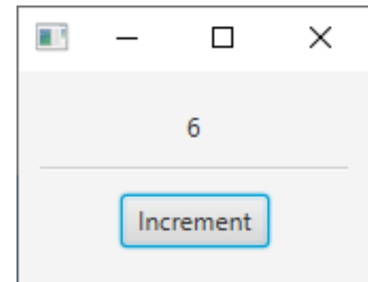
```kotlin
interface IView {
    fun update()
}
```

```kotlin
class View(private val model: Model): Label(), IView {

    init {
        model.views.add(this)
        update()
    }

    override fun update() {
        text = model.getCounter().toString()
    }
}
```

```kotlin
class Model {

    private var counter = 0          // data, here just a single integer

    val views = mutableListOf<IView>() // list of views that listen to a change in data

    fun increment() {                // allows a controller to manipulate the model
        ++counter
        views.forEach { it.update() }  // notifies listeners that the model has changed
    }

    fun getCounter(): Int {          // allows for views to retrieve the data
        return counter
    }
}
```

```kotlin
class Controller(model: Model):
 Button("Increment") {
    init {
        onAction = EventHandler {
            model.increment()
        }
    }
}
```

```kotlin
override fun start(stage: Stage) {

    val model = Model()
    val view = View(model)
    val ctrl = Controller(model)
    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(Vbox(view, Separator(), ctrl) ,
                      165.0, 100.0)
    }.show ()
}
```



DEMO: JavaFX.MVC/02. MVC.Basic

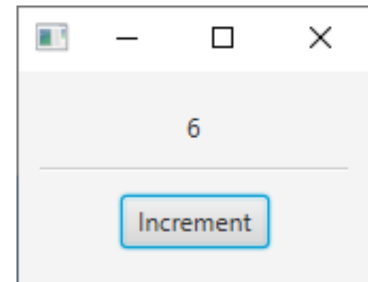
With MVC – Basic

```
class Model {

    private var counter = 0

    val views = mutableListOf<IView>()

    fun increment() {
        ++counter
        views.forEach { it.update() }
    }

    fun getCounter(): Int {
        return counter
    }
}
```

```
interface IView { // interface for all views
    fun update()  // function that the model calls
}                 //  every time its state changes

class View(private val model: Model): Label(), IView {

    init {
        model.views.add(this) // adds this view as
        update()              //  listener to the model
    }

    override fun update() {
        text = model.getCounter().toString()
    }
}
```

```
class Controller(model: Model):
 Button("Increment") {
    init {
        onAction = EventHandler {
            model.increment()
        }
    }
}
```

```
override fun start(stage: Stage) {

    val model = Model()
    val view = View(model)
    val ctrl = Controller(model)
    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(Vbox(view, Separator(), ctrl) ,
                                  165.0, 100.0)
    }.show ()
}
```



DEMO: JavaFX.MVC/02. MVC.Basic

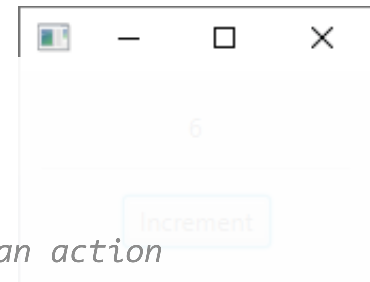
With MVC – Basic

```
class Model {

    private var counter = 0

    val views = mutableListOf<IView>()

    fun increment() {
        ++counter
        views.forEach { it.update() }
    }

    fun getCounter(): Int {
        return counter
    }
}
```

```
class Controller(model: Model):
 Button("Increment") {
    init {
        onAction = EventHandler {
            model.increment()     // call to the model after a user has performed an action
        }
    }
}
```

```
interface IView {
    fun update()
}

class View(private val model: Model): Label(), IView {

    init {
        model.views.add(this)
        update()
    }

    override fun update() {
        text = model.getCounter().toString()
    }
}
```

```
override fun start(stage: Stage) {

    val model = Model()
    val view = View(model)
    val ctrl = Controller(model)
    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(Vbox(view, Separator(), ctrl) ,
                            165.0, 100.0)
    }.show ()
}
```

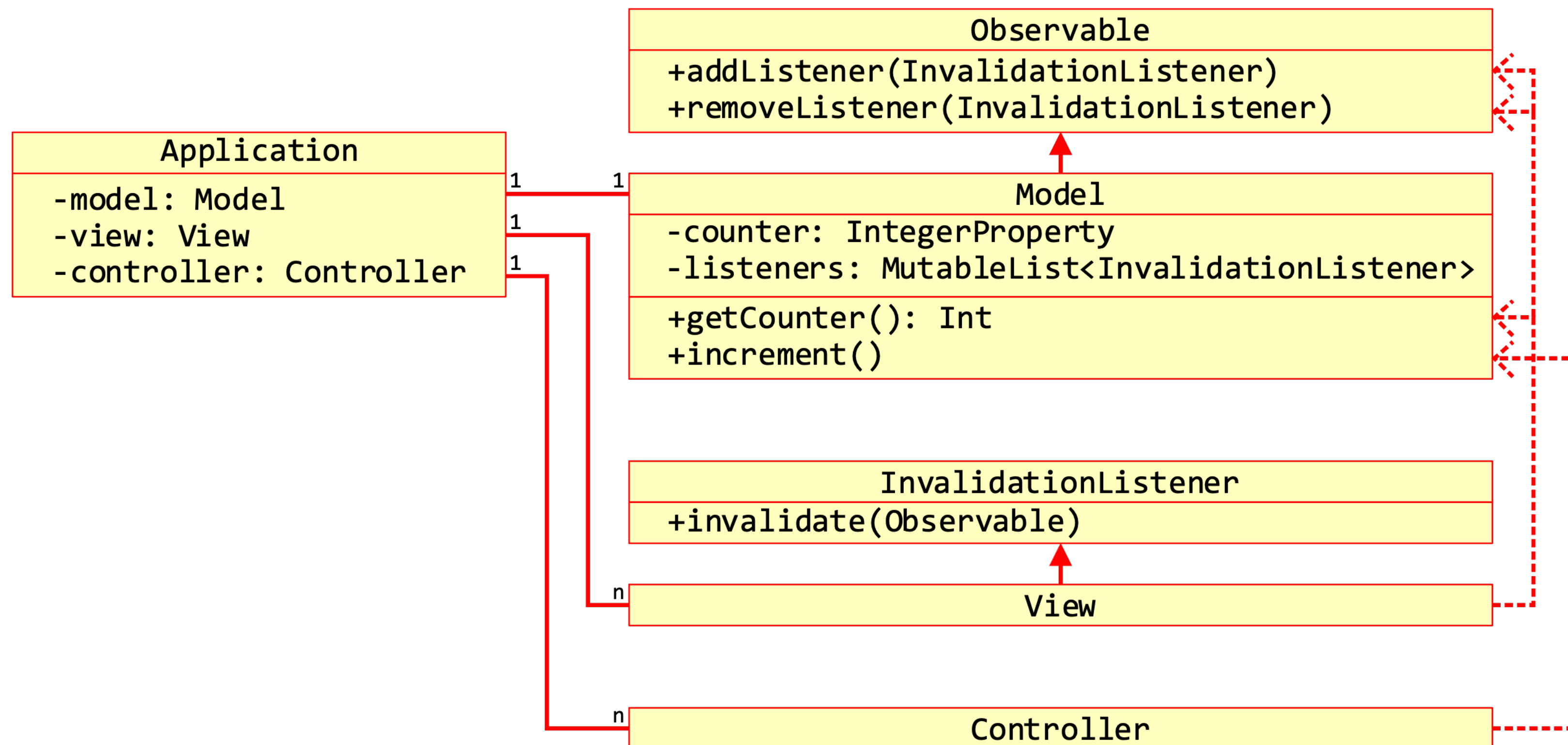


DEMO: JavaFX.MVC/02. MVC.Basic

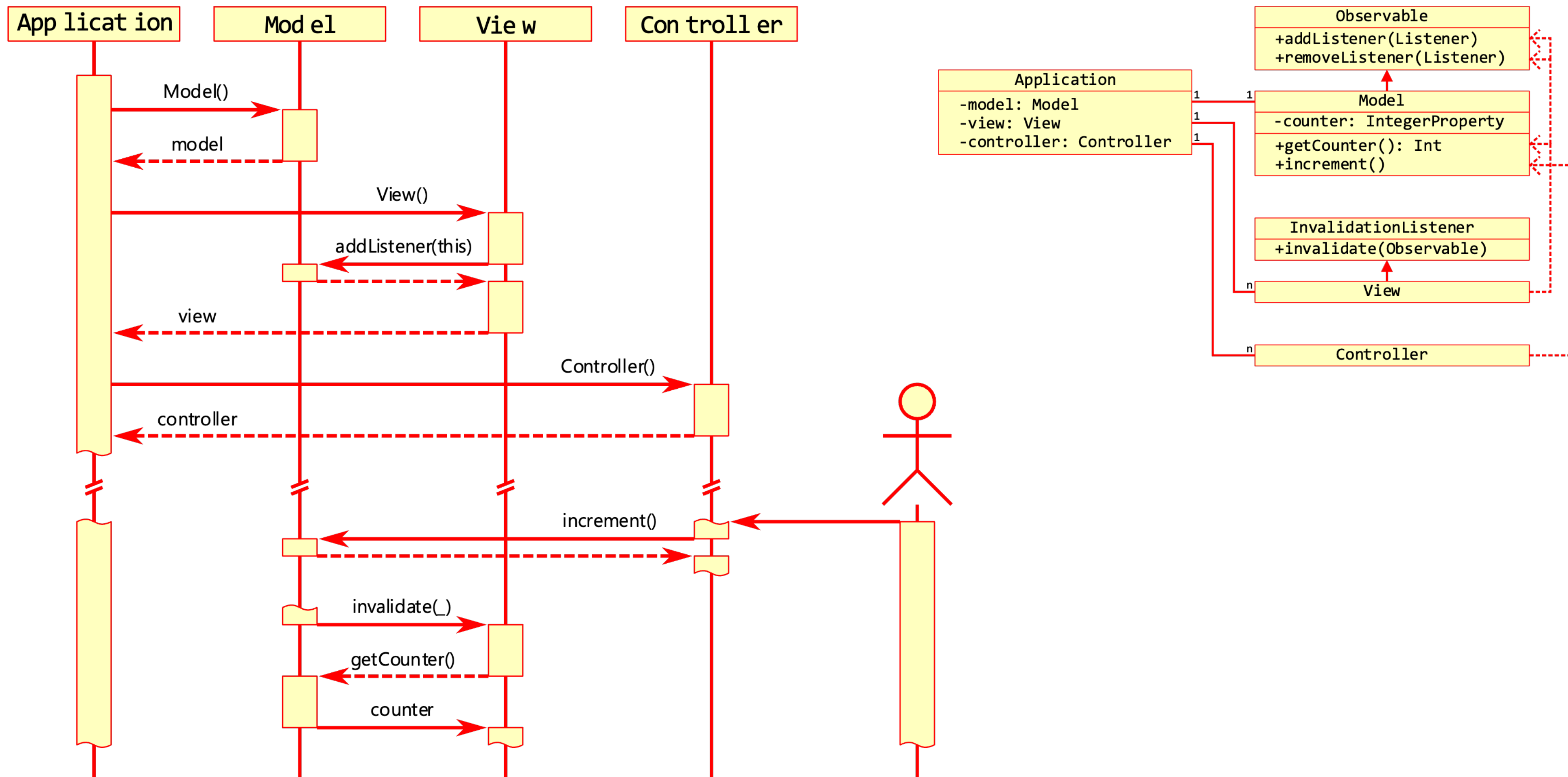
With MVC – JavaFX

Same approach as before, just using built-in functionality:

- Model is object instead of class: makes Model singleton
- Model is using existing `Observable`
- Views are using existing `InvalidationListener`



With MVC – JavaFX



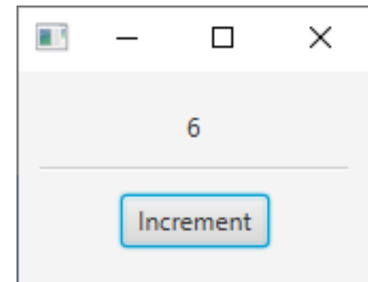
With MVC – JavaFX

```
object Model: Observable {

    private var counter = 0

    fun increment() {
        ++counter
        listeners.forEach { it?.invalidated(this) }
    }
    fun getCounter(): Int {
        return counter
    }

    private val listeners = mutableListOf<InvalidationListener?>()

    override fun addListener(listener: InvalidationListener?) {
        listeners.add(listener)
    }
    override fun removeListener(listener: InvalidationListener?) {
        listeners.remove(listener)
    }
}
```

```
class View() : Label(), InvalidationListener {

    init {
        Model.addListener(this)
        invalidate(null)
    }

    override fun invalidated(observable: Observable?) {
        text = Model.getCounter().toString()
    }
}
```

```
class Controller: Button("Increment") {
    init {
        onAction = EventHandler {
            Model.increment()
        }
    }
}
```

```
override fun start(stage: Stage) {

    val view = View()
    val ctrl = Controller()
    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(Vbox(view, ctrl) , 165.0, 100.0)
    }.show ()
}
```



DEMO: JavaFX.MVC/04. MVC.JavaFX

Optimizing View Updates

With each notification, *everything* in *every view* is refreshed from the model. To avoid this inefficiency, one could

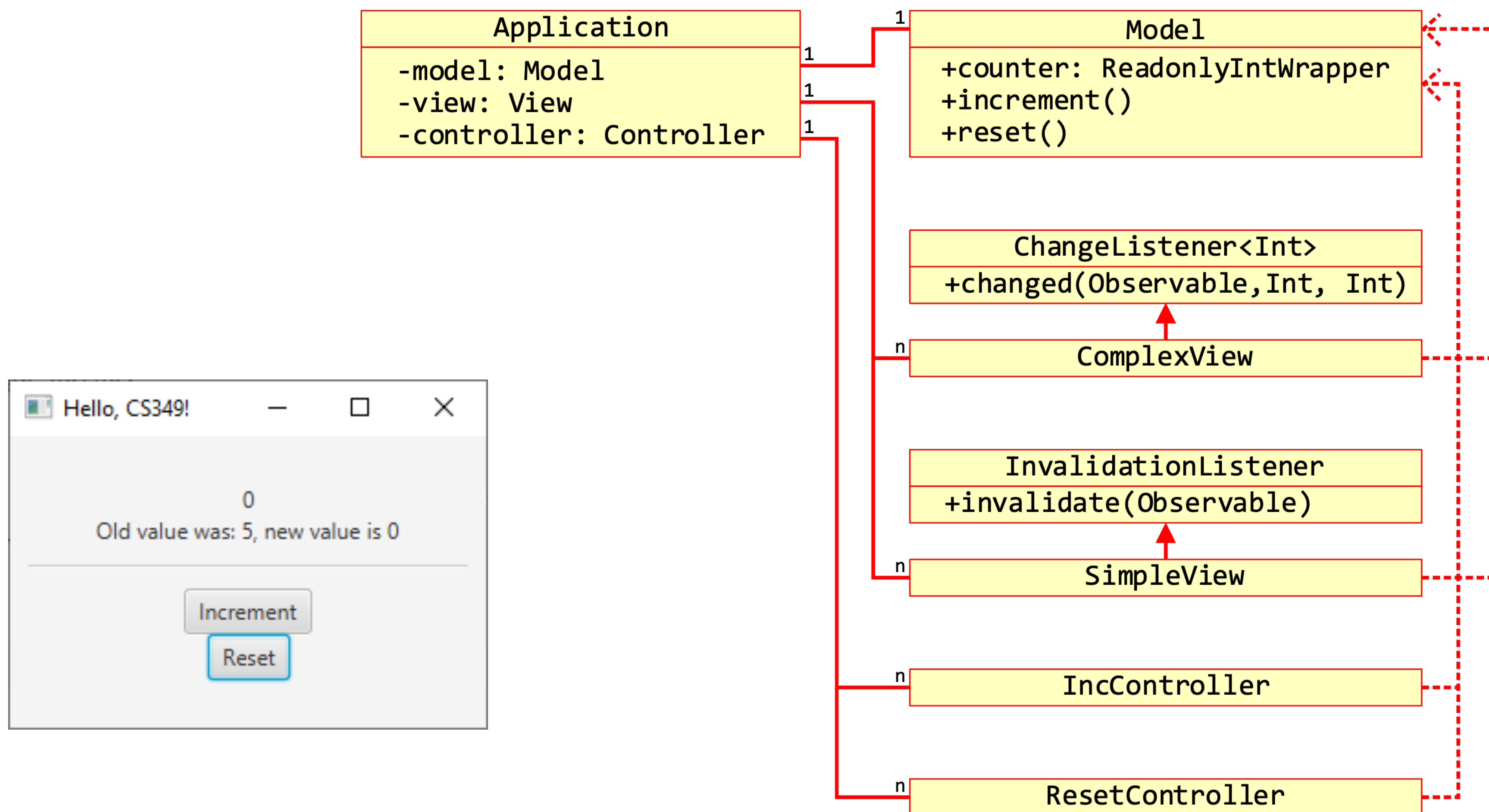
- Add parameters to view notifications to indicate *what* changed
- Be more granular in what views can listen to

In CS349 we do not worry about efficiency for now: it is okay to just update the entire interface.

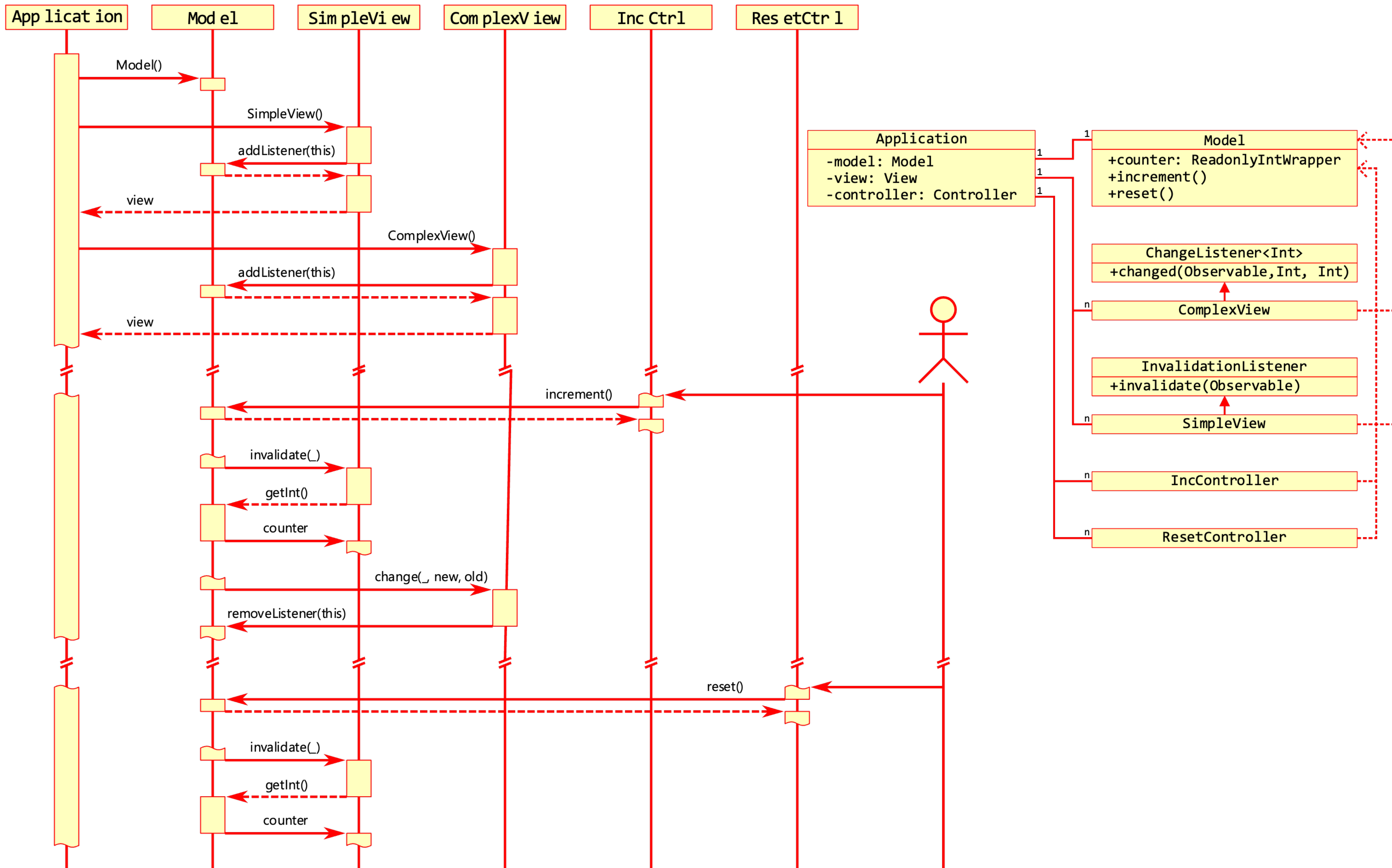
However, it's not just about performance. You may also want different types of notifications to indicate what type of change occurred. This is slightly beyond what we cover here.

With MVC – JavaFX / Read-only Properties

Listening to properties with InvalidationListeners or ChangeListeners.



With MVC – JavaFX / Read-only Properties



With MVC – JavaFX / Read-only Properties

```
object Model {

    private val counter = ReadOnlyIntegerWrapper(0)
    val Counter = counter.readOnlyProperty

    fun incrementCounter() {
        ++counter.value
    }

    fun resetCounter() {
        counter.value = 0
    }
}


class IncController : Button("Increment") {
    init {
        onAction = EventHandler {
            Model.incrementCounter()
        }
    }
}


class ResetController : Button("Reset") {
    init {
        onAction = EventHandler {
            Model.resetCounter()
        }
    }
}
```

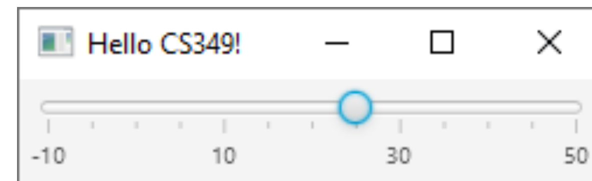
```
class ComplexView : Label(), ChangeListener<Number> {

    init {
        Model.Counter.addListener(this)
        changed(null, null, Model.Counter.value)
    }

    override fun changed(
        observable: ObservableValue<out Number>?,
        oldValue: Number?,
        newValue: Number?) {
        text = "Old: $oldValue, new: $newValue"
    }
}


class SimpleView : Label(), InvalidationListener {

    init {
        Model.Counter.addListener(this)
        invalidated(null)
    }

    override fun invalidated(observable: Observable?) {
        text = Model.Counter.value.toString()
    }
}
```

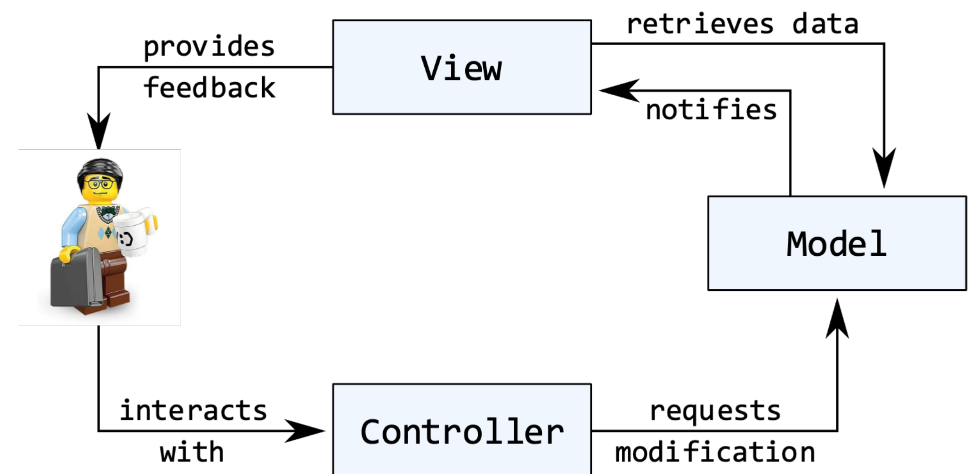
DEMO: JavaFX.MVC/06. MVC.Property

MVC Separation – Theory and Practice

In theory, View and Controller are separate and uncoupled. In practice, View and Controller are often tightly coupled.

In this case, separation between view and controller makes little sense.

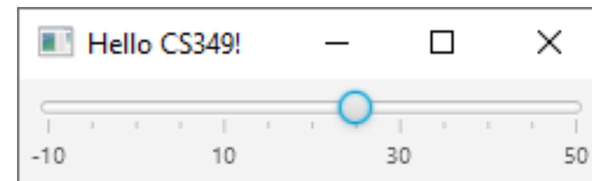


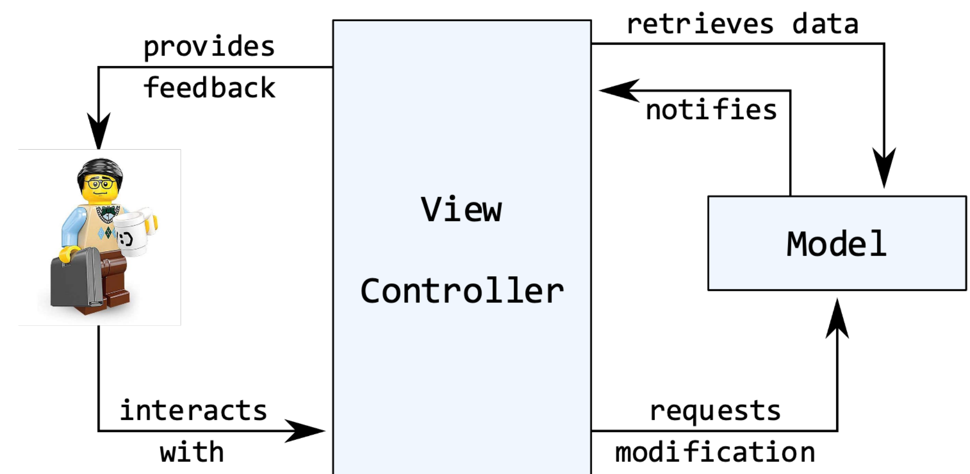


MVC Separation – Theory and Practice

Instead, the view and the controller are oftentimes combined into a “ViewController”.

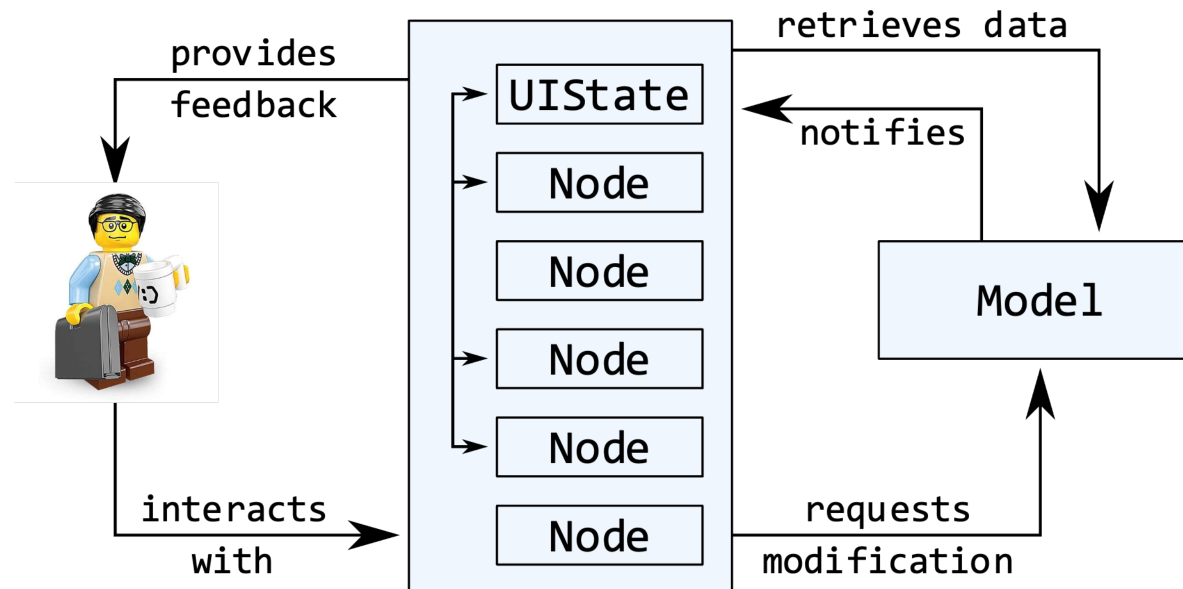
```
class MySlider(private val model: Model) : Slider(), IView {

    init {
        min = model.minValue
        max = model.maxValue
        model.addView(this)
        valueProperty().addListener { _ ->
            model.setValue(value)
        }
        update()
    }

    override fun update() {
        value = model.getValue()
    }
}
```

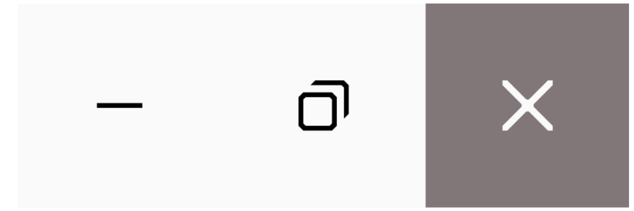




MVC – Models and States

The Model holds the *system state*, but the View itself can have different *UI states*, e.g., showing a status bar or not, showing English or French labels, etc.





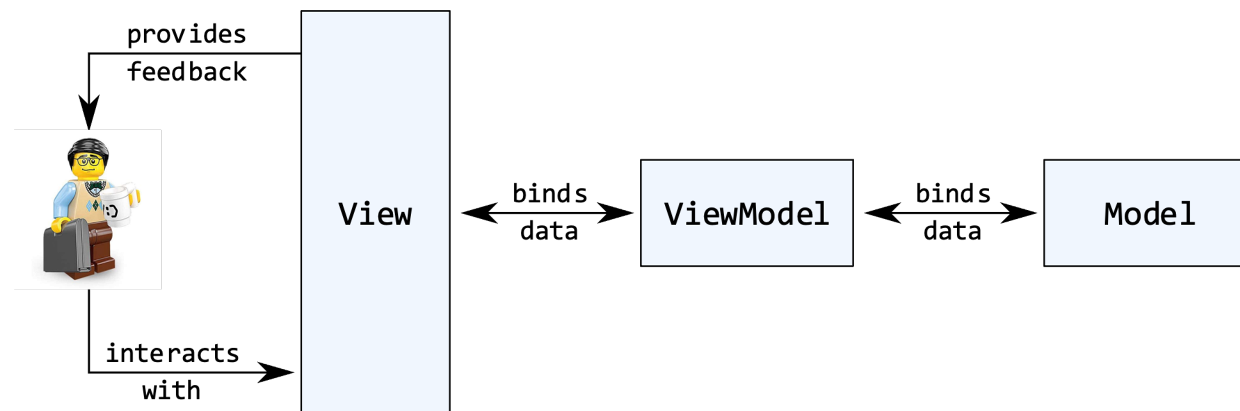
Model–View–ViewModel

Model–View–ViewModel

The `ViewModel` mediates between the `Model` and `View`.

- Manages the view's display logic
- Display-independent logic is relegated to the `Model`.

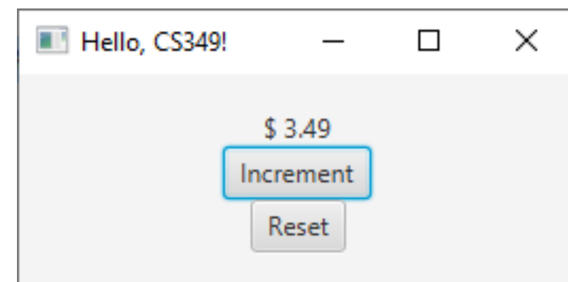
Useful in scenarios where you may have view-dependent state, e.g., a localized UI that uses time and date formats specific to your region; universal data would be in the model, and the data format conversion would be managed by the `ViewModel`.



With MVVM

```
object Model {

    private val counter = ReadOnlyIntegerWrapper(0)
    val Counter = counter.readOnlyProperty

    fun incrementCounter() {
        ++counter.value
    }

    fun resetCounter() {
        counter.value = 0
    }
}
class ViewModel {

    val countProperty = SimpleStringProperty()

    init {
        Model.Counter.addListener { _, _, new ->
            new as Int
            countProperty.value =
              "$ ${new / 100}.${new % 100 / 10}${new % 10}"
        }
    }
    fun incrementCounter() {
        Model.incrementCounter()
    }
    fun resetCounter() {
        Model.resetCounter()
    }
}
```

```
class View(viewModel: ViewModel) : VBox() {

    init {
        children.addAll(
            Label().apply {
            textProperty().bind(viewModel.countProperty)
            },
            Button("Increment").apply {
                onAction = EventHandler {
                    viewModel.incrementCounter() }
            },
            Button("Reset").apply {
                onAction = EventHandler {
                    viewModel.resetCounter() }
            })
        alignment = Pos.CENTER
    }
}
```



End of the Chapter

Please make sure to

- Understand how the MVC architecture (in theory) works
- Understand the shortcomings of the "pure" MVC architecture
- Understand the different implementation variations of M(VC) in JavaFX.

Any further questions?