



Universidad
Nacional
de Quilmes

CONSTRUCCIÓN DE INTERFACES DE USUARIO

2do Cuatrimestre de 2018

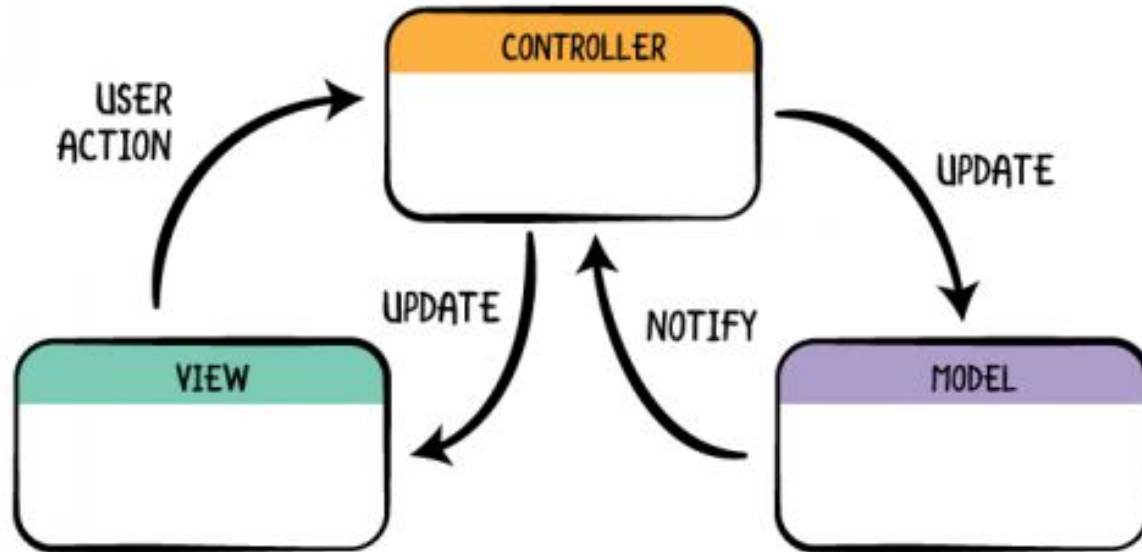


Repaso Binding y Eventos

- ▶ Eventos
 - ▷ Tanto la vista como el modelo producen eventos.
 - ▷ Inversión relación de conocimiento
 - ▷ Similar patron Observer
- ▶ Binding
 - ▷ Muchos frameworks proveen mecanismos de binding
 - ▷ Relaciona dos variables manteniendolas vinculadas
 - ▷ Unidireccional y Bidireccional

Patrón Model-View-Controller

Patrón de arquitectura



Patrón Model-View-Controller

- ▶ Vista
 - ▷ Capa de presentación / interfaz de usuario
 - ▷ Ventanas, botones, inputs.

Patrón Model-View-Controller

- ▶ Modelo
 - ▷ Capa que contiene la lógica de la aplicación - TP1
 - ▷ Contiene la lógica de negocio/reglas de negocio
 - ▷ Modelo de objetos
 - ▷ Ejemplo: objetos que representen:
 - ▷ Un contacto de una agenda
 - ▷ Un movimiento de una caja de ahorro.

Patrón Model-View-Controller

- ▶ Controlador
 - ▷ Intermediario entre la capas Vista y Modelo.
 - ▷ Orquestadores del modelo o decisores de la operación
 - ▷ Gestiona el flujo de información entre el las capas.
 - ▷ Adapta la vista con el modelo.

Patrón Model-View-Controller

- ▶ Modelo - Errores comunes
 - ▷ Aparición de lógica de negocio en otras capas
 - ▷ Modelo con elementos propios de interacción con usuario

Patrón Model-View-Controller

► Ejemplo Arena - Conversor

View

```
class ConversorWindow extends MainWindow<BasicConverter> {  
    new() { super(new BasicConverter)}  
  
    override createContents(Panel mainPanel) {  
        this.title = "Conversor de millas a kilómetros (XTend)"  
        mainPanel.layout = new VerticalLayout  
        new ErrorsPanel(mainPanel, "Listo para convertir")  
        new Label(mainPanel).text = "Ingrese la longitud en millas"  
        new NumericField(mainPanel) => [  
            value <=> "miles"  
        ]  
        new Button(mainPanel) => [  
            caption = "Convertir a kilómetros"  
            onClick [ | this.modelObject.convert ]  
        ]  
        new Label(mainPanel) => [  
            background = Color.ORANGE  
            value <=> "kilometers"  
        ]  
        new Label(mainPanel).text = " kilómetros"  
    }  
    def static main(String[] args) {  
        new ConversorWindow().startApplication  
    }  
}
```

Patrón Model-View-Controller

- ▶ Ejemplo Arena - Conversor

Modelo

```
@Accessors
@Observable
class BasicConverter{
    val FACTOR = 1.60934
    double miles
    double kilometers

    def convert() {
        kilometers = miles * FACTOR
    }
}
```

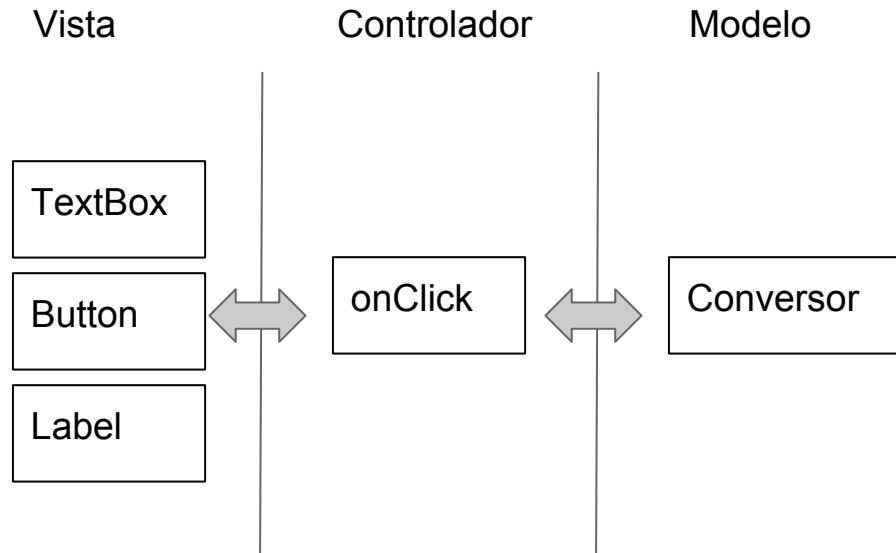
Patrón Model-View-Controller

► Ejemplo Arena - Conversor - Controller

```
class ConversorWindow extends MainWindow<BasicConverter> {  
    new() { super(new BasicConverter()) }  
  
    override createContents(Panel mainPanel) {  
        this.title = "Conversor de millas a kilómetros (XTend)"  
        mainPanel.layout = new VerticalLayout  
        new ErrorsPanel(mainPanel, "Listo para convertir")  
        new Label(mainPanel).text = "Ingresa la longitud en millas"  
        new NumericField(mainPanel) => [  
            value <=> "miles"  
        ]  
        new Button(mainPanel) => [  
            caption = "Convertir a kilómetros"  
            onClick [ | this.modelObject.convert ]  
        ]  
        new Label(mainPanel) => [  
            background = Color.ORANGE  
            value <=> "kilometers"  
        ]  
        new Label(mainPanel).text = " kilómetros"  
    }  
    def static main(String[] args) {  
        new ConversorWindow().startApplication  
    }  
}
```

```
@Accessors  
@Observable  
class BasicConverter {  
    val FACTOR = 1.60934  
    double miles  
    double kilometers  
  
    def convert() {  
        kilometers = miles * FACTOR  
    }  
}
```

Patrón Model-View-Controller



Patrón Model-View-Controller

- ▶ Beneficios
 - ▷ Reutilización de código
 - ▷ Separación de conceptos - Legibilidad
 - ▷ Facilitar el desacople para el desarrollo
 - ▷ Facilitar el mantenimiento
 - ▷ Aparición de perfiles especializados en cada capas
 - ▷ Desacople de tecnologías
 - ▷ Mantenimiento - Testing

Validaciones

- ▶ Validaciones a nivel de Vista (View)
 - ▷ Aquellas que deben ser realizadas sin la necesidad de intervención del modelo
 - ▷ La validación se visualiza en la misma capa.
- ▶ Validaciones a nivel de Modelo
 - ▷ Aquellas que deben ser realizadas por el modelo dado que implican reglas de negocio.
 - ▷ Deben ser informadas a la capa View.

Validaciones a nivel Vista

- ▶ “Los km solo pueden ser valores numéricos”
- ▶ “El formato de fecha es inválido; debe ingresar el formato dd/mm/aaaa”
- ▶ Debe ingresar un máximo de 20 dígitos ”

```
new NumericField(mainPanel) => [  
  value <=> "numero"  
  width = 100  
]
```

```
new TextBox(mainPanel) => [  
  withFilter [ event | event.potentialTextResult.matches("[0-9,.*]") ]  
  bindValueToProperty("numero")  
  width = 100  
]
```

Validaciones a nivel Modelo

- ▶ “Debe ingresar un número de cliente válido”
- ▶ “Ya existe un capítulo con igual identificador para la serie seleccionada”

```
def void validarClientesDuplicados(Celular celular) {  
    val numero = celular.numero  
    if (this.search(numero).isEmpty) {  
        throw new UserException("Ya existe otro cliente con el mismo número")  
    }  
}
```

En Arena las ventanas del tipo SimpleWindows manejan UserException mostrando el mensaje de error

Validaciones a nivel Modelo

Regla importante:

Mensajes descriptivos que permitan al usuario identificar claramente cuál regla de negocio no se está cumpliendo y le permita continuar con el proceso.

Manejo de errores

Son errores propios de la programación, que hace que no sean modelados, simplemente ocurren y que identificarlas y tratarlas

Ejemplos de errores:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

```
try {  
    //hacer algo  
} catch (ArithmeticException exception) {  
    new UserException("No puede dividirse por cero");  
} catch (ArrayIndexOutOfBoundsException exception) {  
    println("Error del programador")  
} catch (TuExceptionDeNegocio exception) {  
}
```

Application Model vs Model Simple

Cada vista necesita un modelo pero cuando la complejidad de interacción entre el usuario y el modelo crece aparece el concepto de Application Model

Consiste en crear un objeto que tenga la representación del comportamiento global de la aplicación. Generalmente representan un Caso de Uso

Application Model vs Model Simple

Diferentes Ciclos de vida

Los objetos de dominio se crear, se persisten, se modifican, etc.

Los objetos de aplicación se usan una sola vez.

Application Model vs Model Simple

Esquema MMVC

V - Vista: la pantalla *Window

C - Controller: adapta la vista con el modelo, en este caso con el application model, es el encargado de resolver el binding

M - Modelo de la vista/modelo de aplicación/application model: modela al caso de uso, mantiene simple el binding con la vista

M - Modelo de dominio

Application Model vs Model Simple

Ejemplo