Universidad Nacional de Quilmes

CONSTRUCCIÓN DE INTERFACES DE USUARIO

APLICACIONES DESKTOP

Desventajas

- Muy atadas al SO
  - Linux (muchas distros)
  - Windows 10 / 8 / 7
  - OSX
- Mucho consumo de recursos
- Es difícil hacerlas “*online*”
  - Mantener la información sincronizada entre clientes
  - No se pueden acceder “desde cualquier lado”

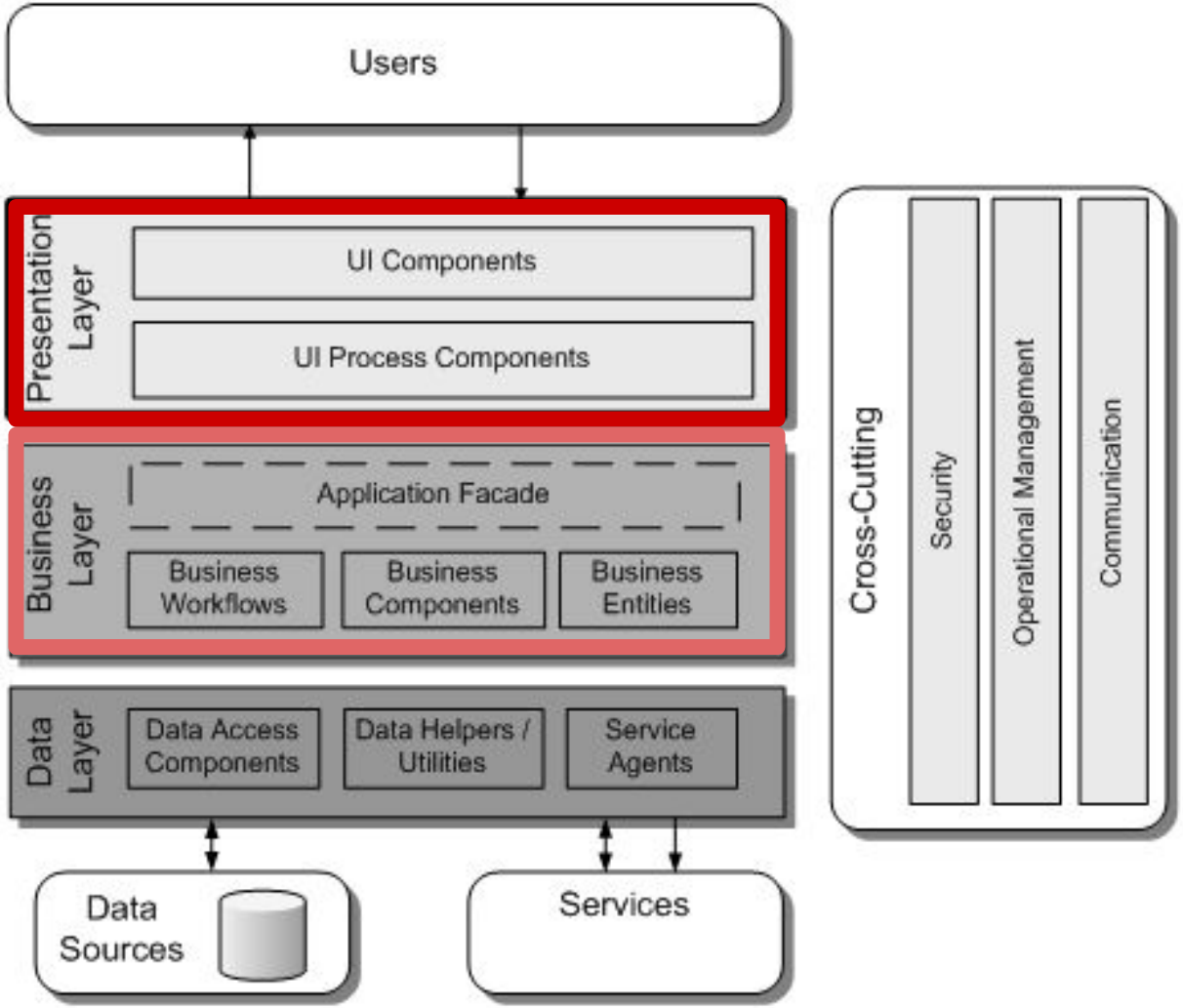
¿Entonces?

- Todavía son necesarias
- Tienen buena performance
- Pueden llegar a ser más útiles
- Arquitectura más simple
- Se pueden usar offline
  - Útil en situaciones críticas
  - Menor probabilidad de hackeo

Diseño vs Diseño

- Diseño de Software
  - Lógica de Negocio
  - Modelo de Datos
  - Aplicación de Patrones
  - Responsabilidades
  - Tecnologías
- Diseño Gráfico
  - Imágen de Negocio
  - Identificación de la Marca
  - Paleta de Colores
  - Imágenes, Iconografía
  - Usabilidad

Capas de una Aplicación



Arena MVVM Framework

Propiedades del Framework:

- Separación Modelo-Presentación
- Binding bidireccional
- Controllers
  - Simples
  - Reusable
  - Concepto de Single-Responsability

Fuertemente alineadas con los principales conceptos del diseño orientado a objetos, aplicado al desarrollo de UIs.

Stack Tecnológico

- Windows / Linux / OSX
- (IntelliJ / Eclipse) + JRE + JDK + Maven
- Java / **Kotlin** / Scala / Groovy / Xtend

Página oficial

http://arena.uqbar-project.org/

Qué provee el framework

- Manejo de Ventanas
- Componentes Visuales
- Manejo de eventos
- Binding *automágico* Vista-Modelo
- Acceso a disco

Componentes ⇒ Labels

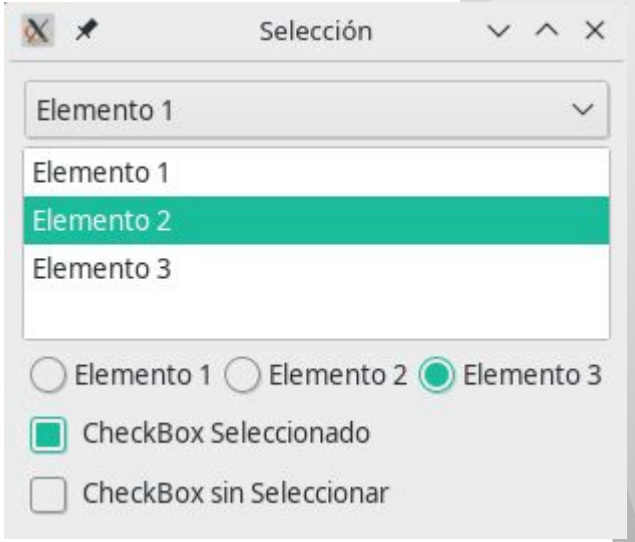
- Permiten visualizar contenido
  - Texto
  - Imágen
- Customizables
  - Tamaño de fuente
  - Color del texto
  - Color de fondo
  - Alineación

Componentes ⇒ Inputs

- Permiten ingresar información
  - `TextBox` (texto libre)
  - `NumericField` (sólo núméros)
  - `PasswordField` (********)
  - `KeyWordTextArea` (multilínea)
  - `Spinner` (numérico con flechitas)
- Customización media

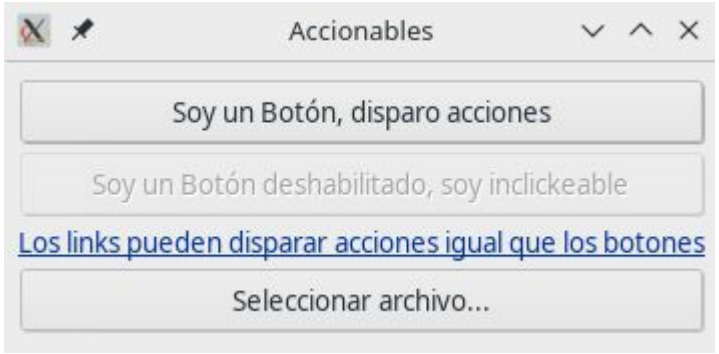
Componentes ⇒ Selectores

- Permiten elegir opciones
  - `Selector` (Dropdown)
  - `List`
  - `RadioSelector`
  - `CheckBox`
- Requieren colecciones
  - salvo el `CheckBox`
- Customización media/baja



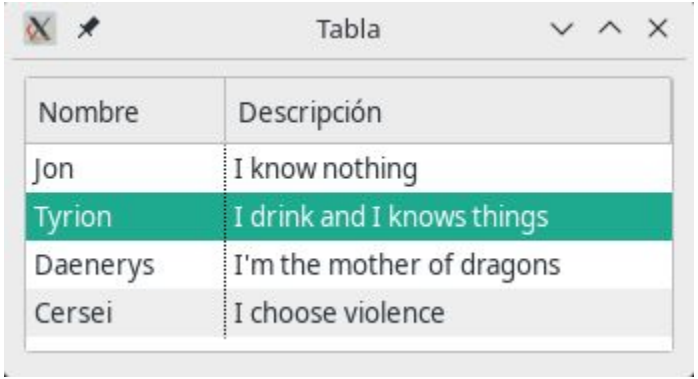
Componentes ⇒ Accionables

- Permiten generar acciones
  - `Button`
  - `Link`
  - `FileSelector`
- Customización baja
- Pueden contener imágenes en vez de texto



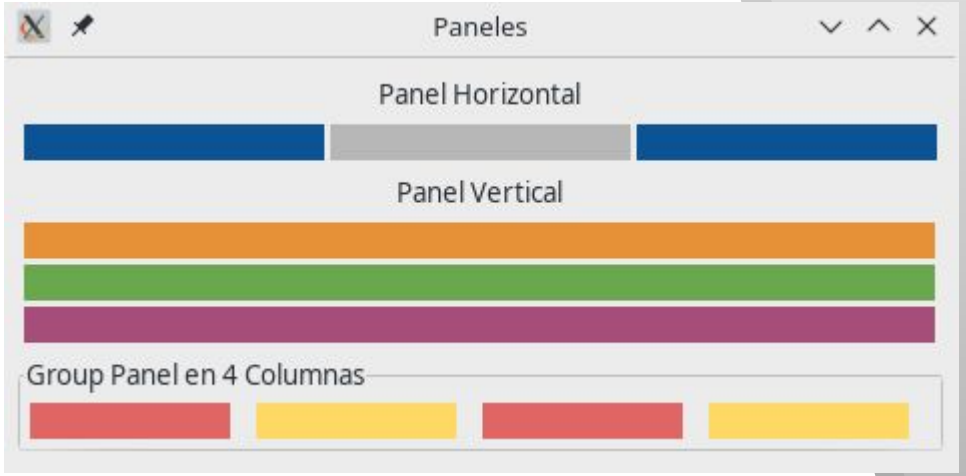
Componentes ⇒ Tablas

- Permiten listar información en columnas
  - Se espera una lista de elementos
  - Cada columna muestra info de ese elemento
- Customización baja



Componentes ⇒ Paneles

- Son “invisibles”
- Contienen al resto de los elementos
  - Y a otros paneles
- Permiten 3 distribuciones
  - Horizontal
  - Vertical
  - En columnas (n)



+Info de Componentes

- Extensiones para Kotlin
  - https://github.com/unq-ui/arena-kotlin-extensions
  - Se simplifica el uso
- Documentación oficial
  - http://arena.uqbar-project.org/documentation/components.html
  - Ejemplos “a la java”

MVC | MVVM

- **M**odel **V**iew **C**ontroller
- **M**odel **V**iew **V**iew**M**odel
- Son patrones de diseño aplicables a UI
- Permiten separar *Vista* de *Modelo* agregando un componente intermedio
  - El cual se encarga de que el modelo y la vista se comuniquen
  - Viene en dos sabores:
    - **Controller** (MVC)
    - **ViewModel** (MVVM)

Ejemplo MVC

```
fun main() = LoginController().run()
```

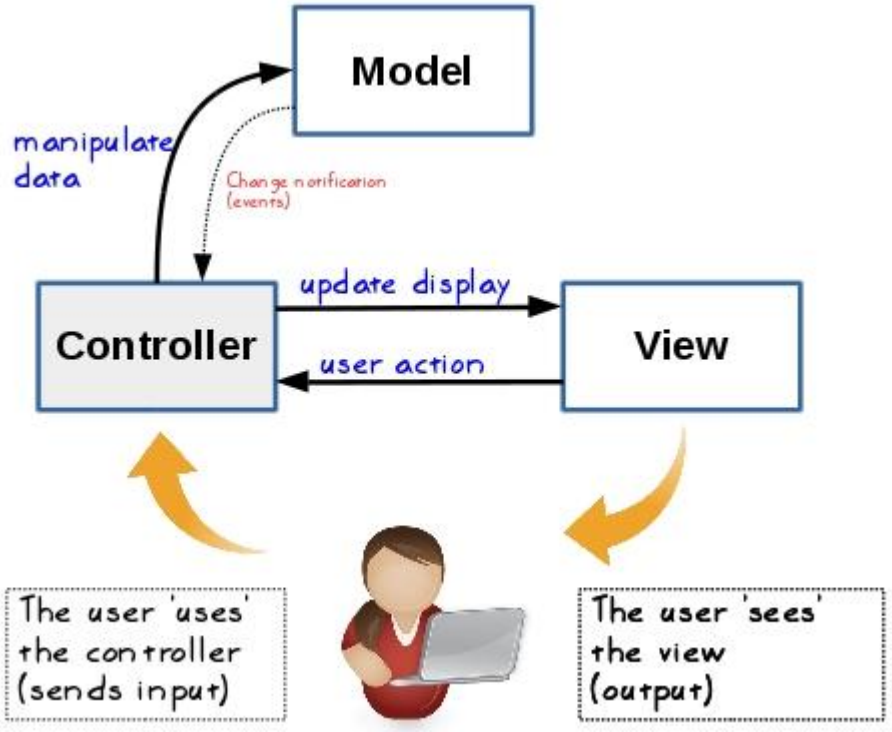
```
class LoginModel(val user, val pass) {
  fun isValid(): Boolean {
    return DB.exists(user, pass)
  }
}
```

```
class LoginController() {
  fun run() = LoginFormView(this).render()
  fun validate(user: String, pass: String) {
    if (LoginModel(user, pass).isValid())
      HomeController(user).run()
    else this.run()
}

class HomeController(val user) {
  fun run() = HomePageView(this).render()
}
```

pseudocódigo

```
class LoginFormView(val login: LoginController){
  fun render() {
    title = "Login"
    val user = input { name = "username" }
    val pass = input { name = "password" }
    button("Entrar") {
      onClick {
        login.validate(user.value, pass.value)
      }
    }
  }
}

class HomePageView(val user: UserModel) {
  fun render() {
    title = "Hola ${user.username}"
    content = "Bienvenido a la Home Page"
  }
}
```

MVC ⇒ Diagrama conceptual



Acerca de MVC

- Patrón de Diseño de UI
- Desacopla objetos y define responsabilidades
- Xerox introdujo el concepto en los 70s
- Se implementó como *library* en el Smalltalk-80
- Se popularizó en 1988 con el artículo "A cookbook for using the model–view controller user interface paradigm in Smalltalk-80".

Leer más: https://en.wikipedia.org/wiki/Model–view–controller

Ejemplo MVVM

```
fun main() = LoginFormView(LoginVM()).render()
```

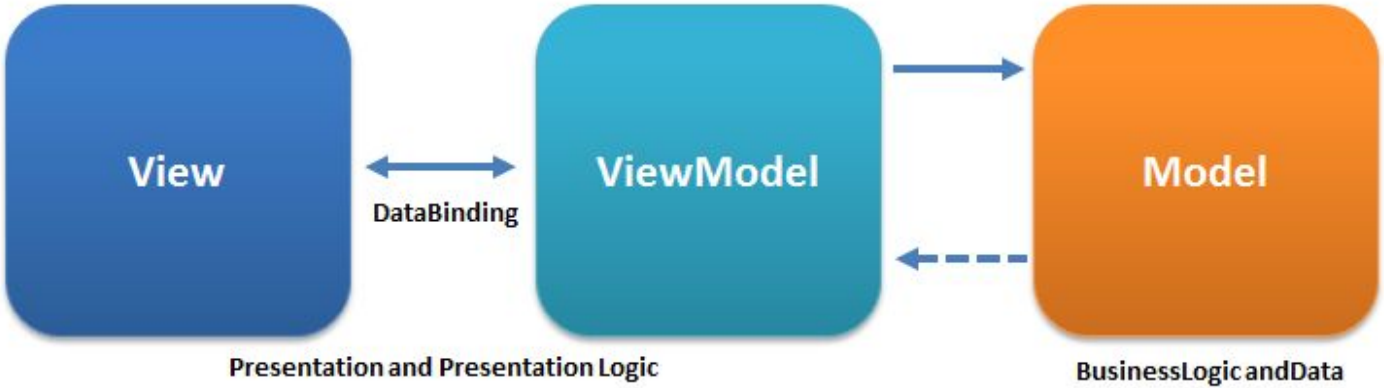
```
class LoginModel(val user, val pass) {
  fun isValid(): Boolean {
    return DB.exists(user, pass)
  }
}
```

```
class LoginViewModel() {
  var user: String = ""
  var pass: String = ""

  fun validate() {
    val login = LoginModel(user, pass)
    return login.isValid()
}

class UserViewModel(val username)
```

pseudocódigo

```
class LoginFormView(val user: LoginViewModel) {
  fun render() {
    title = "Login"
    input{ name="username"; bindTo(login.user) }
    input{ name="password"; bindTo(login.pass) }
    button("Entrar") {
      onClick {
        if (login.validate())
         HomePageView(UserViewModel(login.user))
            .render()
        else throw Error("Invalid user")
      }
    }
  }}

class HomePageView(val user: UserViewModel) {
  fun render() {
    title = "Hola ${user.username}"
    content = "Bienvenido a la Home Page"
  }}
```

MVVM ⇒ Diagrama conceptual



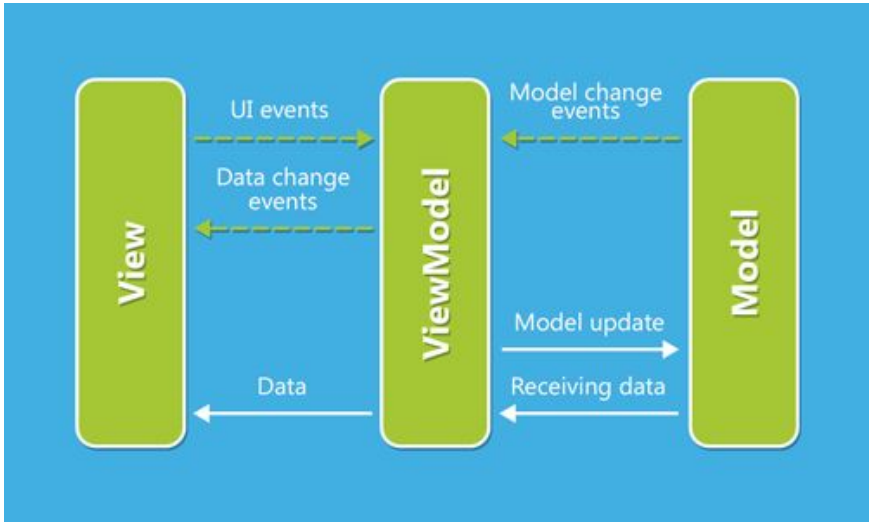
Acerca de MVVM

- Patrón de Diseño de UI
- Facilita la separación de componentes
- El *ViewModel* es un puente entre el Modelo de Negocio y la Presentación
- Hace fuerte uso del concepto de *Binding*
- Fue diseñado por Microsoft para Silverligth en 2005
- Es una variación de "The Presentation Model Design Pattern" que planteó Martin Folwer en 2004.

Leer más: https://en.wikipedia.org/wiki/Model–view–viewmodel

Binding

- Patrón de diseño que permite vincular componentes de la vista con propiedades del modelo.
- La mayoría de los frameworks MVVM lo provee en mayor o menor medida.
- En general se implementa utilizando el patrón *Observer*.
- Puede ser
  - **Bidireccional:** si se modifica la vista se actualiza el modelo y viceversa.
  - **Unidirección:** la vista actualiza el modelo o el modelo actualiza la vista, pero no ambos.



Arena Framework

* MVVM
* Binding

- Una de las grandes diferencias entre MVC y MVVM es que MVVM hace uso de Binding y MVC no.
- Cuando la aplicación procesa todo de un mismo lado conviene MVVM con Binding.
- Cuando se requiere mucho flujo de comunicación cliente-servidor conviene MVC.
- **MVVM** simplifica el `ViewModel` pero complejiza la `View`
- **MVC** da mayor responsabilidad al `Controller` pero la `View` es “tonta”

Hello Arena

```
// Start App
fun main() = HelloWindow(HelloViewModel()).startApplication()
// Modelo
@Observable
class HelloViewModel(var msg: String = "Saludos desde Arena!"){
  var logo = "arena.png"
}

// Vista
class HelloWindow(model: HelloViewModel) : MainWindow<HelloViewModel>(model) {
  override fun createContents(mainPanel: Panel) {
    title = "Hello Window"
    Label(mainPanel) with { text = this.modelObject.msg }
    Label(mainPanel) with {
      bindTo("bindMsg"); color = Color.decode("#0b5394")
    }
    Label(mainPanel) with { bindImageTo("logo") }
    Button(mainPanel) with { caption = "Cerrar"; onClick { close() } }
  }
}
```

Pasos para crear una App Arena

1. Desde Eclipse/IntelliJ crear un nuevo proyecto **Maven**
2. Editar el `pom.xml` agregando dependencias
3. Crear una clase que extienda de `MainWindow`
4. Crear un `ViewModel` con la *annotation* `@Observable` para esa `Window`
5. Importar las extensiones para *kotlin*
6. Definir un `main()` que instancie esa `Window` y la levante
7. Asegurarse de tener bien el *Classloader* en **VM Options**

pom.xml

```
<!-- Agregar -->
<repositories>
  <repository>
    <id>jitpack.io</id>
    <url>https://jitpack.io</url>
  </repository>
</repositories>

<!-- Agregar dentro de <dependencies> -->
<dependency>
  <groupId>com.github.unq-ui</groupId>
  <artifactId>arena-kotlin-extensions</artifactId>
  <version>1.4.0</version>
</dependency>
```

Classloader

Tanto en Eclipse como en IntelliJ, para correr aplicaciones Arena necesitamos configurar el Classloader en el IDE.

```
-Djava.system.class.loader=org.uqbar.apo.APOClassLoader
```

Repaso de lo visto

- Aplicaciones Desktop
- Capas de una Aplicación
- Patrones de Diseño en UI
  - MVC
  - MVVM
- Data binding
- Arena MVVM Framework
  - Componentes
  - Patrones
  - Hello world

¿Preguntas?