Formulario para añadir un contacto en agenda

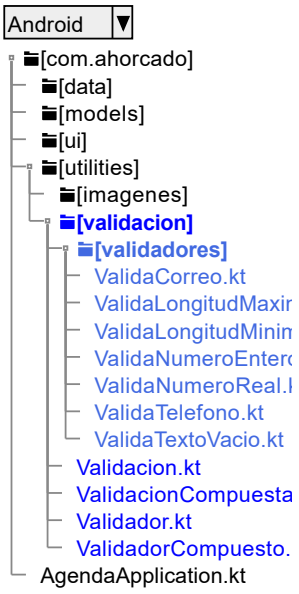
Descargar estos apuntes

'*Codelab*' para crear formularios

En el siguiente ejercicio partiremos del ejercicio 3 del Tema 3.2 donde definíamos un componente imagen para la Agenda. La idea del mismo es proponer una metodología para crear formularios en Compose y validarlos en el `ViewModel` de manera que reutilicemos el código de validación y el `ViewModel` quede lo más limpio posible.

Paso 1.- Añadiendo clases de utilidad para introducir campos de texto

Descarga el recurso **valizadion.zip**. Si lo descomprimes te generará una carpeta `validacion` que debes arrastrar a tu proyecto al bajo el paquete `com.pmdm.agenda.ui.utilities`.



Nos creará el paquete validación en el que podremos encontrar las siguientes clases que se ven en el esquema de la derecha. Algunas de ellas ya las hemos tratado cuando hablamos de `TextField`.

```
// Validacion.kt ------------------------------------------------
// Describe el resultado de una validación.
// Si hay error, se indica el mensaje de error.
// Será el UIState que reciben nuestros TextField para indicar si hay error o no.
data class Validacion(
    val hayError: Boolean,
    val mensajeError: String? = null
)
```

```
// ValidacionCompuesta.kt ---------------------------------------------
// Es una clase de utilidad que tiene una lista de validaciones que debemos pasar antes
// de dar por válidos los datos de un formulario.
// Rehusamos para ello los objetos que representan los estados de validación de cada campo y si alguno
// de ellos tiene error lo indicaremos en la propiedad calculada de solo lectura hayError de esta clase
class ValidacionCompuesta {
    private val validaciones = mutableListOf<Validacion>()
    fun add(validacion: Validacion): ValidacionCompuesta {
        validaciones.add(validacion)
        return this
    }
    val hayError: Boolean
        get() = validaciones.any { it.hayError }
}
```

```kotlin
// Validador.kt ------------------------------------------------
// Abstracción (SAM) de una función de validación de campos de formulario
// como por ejemplo un email, un teléfono, etc.
// Devuelve un objeto Validacion que devolverá un estado de validación para un TextField.
fun interface Validador {
    fun valida(texto: String): Validacion
}
```

```kotlin
// ValidadorCompuesto.kt ---------------------------------------------
// Implementa la interfaz Validador y es una clase de utilidad que tiene
// una lista de validadores que debemos pasar para un único TextField
// antes de dar po válido el dato introducido.
// Por ejemplo para un teléfono, podemos tener una validación que compruebe
// que no está vacío, otra que compruebe que tiene una longitud mínima.
// Nota: Las validaciones se ejecutan en orden y si alguna de ellas tiene error
// se devuelve el error y no se ejecutan las siguientes.
class ValidadorCompuesto(validador: Validador) : Validador {
    private val validadores = mutableListOf(validador)
    fun add(validador: Validador): ValidadorCompuesto {
        validadores.add(validador)
        return this
    }

    override fun valida(texto: String): Validacion =
        validadores.firstOrNull { it.valida(texto).hayError }?.valida(texto)
            ?: Validacion(false)
}
```

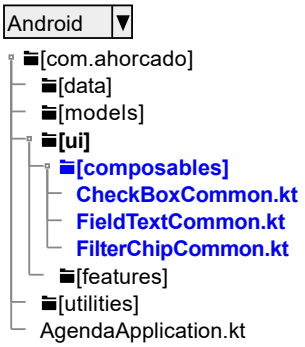
Dentro del paquete **`validacion`** encontramos otro paquete **`validadores`** que contiene las clases que implementan la interfaz **`Validador`**. Estas clases son las que se encargan de validar un campo de texto concreto. Por ejemplo, la clase **`ValidaCorreo`** se encarga de validar que un campo de texto es un correo electrónico válido.

```kotlin
class ValidaCorreo(
    val mensajeError: String = "Correo no válido"
) : Validador {
    override fun valida(texto: String): Validacion =
        if (Regex("^[A-Za-z](.*)([@]{1})(.{1,})(\\.)(.{1,})$").matches(texto))
            Validacion(false)
        else
            Validacion(true, mensajeError)
}
```

Así vamos a tener validadores para validar que un campo de texto es un correo, un teléfono, un número entero, un número real, etc.

Paso 2.- Definiendo composables reutilizables

Dentro del paquete `ui` vamos a crear el paquete `composables`. Aquí irán aquellos composables 'genéricos' creados por nosotros que podamos reutilizar en diferentes partes de nuestra aplicación o candidatos en otras aplicaciones.



Dependerán únicamente de la **capa de compose** de `material` y por tanto podremos usar definiciones `fundation`, `ui` y `runtime`.

> 📌 **Nota**: Lo ideal es que se implementarán en un proyecto a parte y se publicarán como librerías para que puedan ser usadas en otros proyectos.

En nuestro caso tenemos:

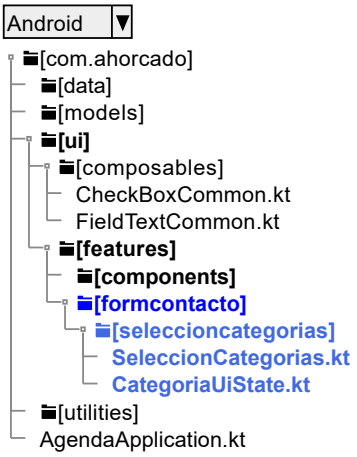
- **CheckBoxCommon.kt**: Composable que nos permite crear un `CheckBox` con una etiqueta.
- **FieldTextCommon.kt**: Composables de utilidad que me van a permitir introducir un campo de texto con una etiqueta, una validación y en ocasiones un icono. Como por ejemplo...
  - `OutlinedTextFieldEmail` para introducir un email.
  - `OutlinedTextFieldPhone` para introducir un teléfono.
  - `OutlinedTextFieldPassword` para introducir una contraseña.
  - etc.

  Puedes ver en la función `@Composable` con la previsualización para tests un ejemplo de como usar estos componentes junto a las clase con la **validacion** y los diferentes **validadores** que hemos definido en el paso anterior.
- **FilterChipCommon.kt**: Composable que nos permite crear un `Chip` con un texto y un icono para filtrar, tal y como indica en Material3. De momento no lo vamos a usar pero añádelo también a tu proyecto.

Estos *composables* junto a `ImagenContacto.kt` que ya debería estar incluido en el proyecto, los usaremos en el siguiente paso para crear el formulario de contacto.

Paso 3.- Creando componente de seleccion de categorias

En ocasiones necesitamos crear componentes que no son genéricos y que solo se van a usar en un sitio concreto de nuestra aplicación. Este es el caso de la selección de categorías que vamos a usar en el formulario de contacto. Sin embargo, el hacerlo nos va a permitir reutilizarlo y descomponer un estado complejo en estados más simples como pudiera ser el que guarda la selección de categorías.



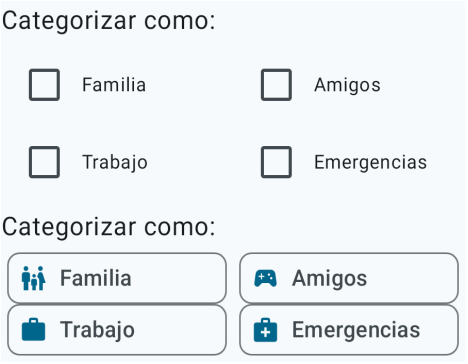
En este caso vamos a crearlo dentro de la carpeta `features` y dentro **vamos a crear el paquete** `formcontacto` donde va a ir la implementación del mismo. Como la selección de categorías es un sub-componente del mismo, lo vamos a crear a su vez en un paquete denominado `formcontacto.seleccioncategorias`.

Para ello decarga el recurso **seleccioncategorias.zip**. Que contiene los fuentes que se ven en el esquema arriba a la derecha. Desomprime el recurso y arrastra la carpeta `seleccioncategorias` a tu proyecto en la carpeta `formcontacto`.

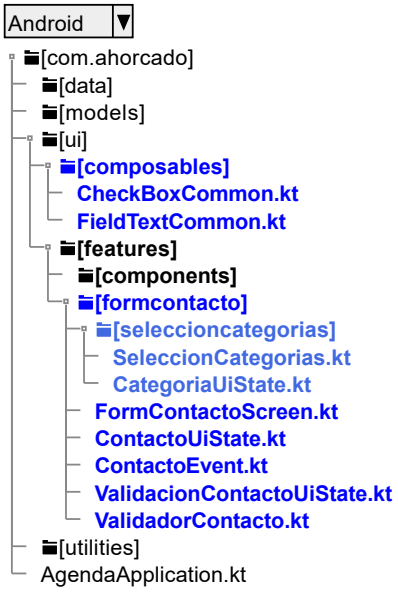
- **CatergoriaUiState.kt**: Nos define un estado para dicho componente.

```
data class CatergoriaUiState(
    val amigos: Boolean = false,
    val amigosIcon : ImageVector = Icons.Filled.SportsEsports,
    val trabajo: Boolean = false,
    val trabajoIcon : ImageVector = Icons.Filled.Work,
    val familia: Boolean = false,
    val familiaIcon : ImageVector = Icons.Filled.FamilyRestroom,
    val emergencias: Boolean = false,
    val emergenciasIcon : ImageVector = Icons.Filled.MedicalServices
)
```

- **SeleccionCategorias.kt**: Nos define un estado para dicho componente.



TODO:AQUI EXPLICARLO y LAZYGRID


Android
[com.ahorcado]
[data]
[models]
[ui]
[composables]
CheckBoxCommon.kt
FieldTextCommon.kt
[features]
[components]
[formcontacto]
[seleccioncategorias]
SeleccionCategorias.kt
CategoriaUiState.kt
FormContactoScreen.kt
ContactoUiState.kt
ContactoEvent.kt
ValidacionContactoUiState.kt
ValidadorContacto.kt
[utilities]
AgendaApplication.kt