

TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DEL VALLE DE OAXACA



TECNOLÓGICO
NACIONAL DE MEXICO

ASIGNATURA:
APLICACIONES MOBILES

NOMBRE DEL ALUMNO:
VÍCTOR ROMÁN PÉREZ LÓPEZ

NO. CONTROL: 22920259
GRUPO: A. SEMESTRE: 40



CARRERA: INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES.

NOMBRE DEL DOCENTE: CARDOSO JIMENEZ AMBROSIO

LUGAR Y FECHA: NAZARENO, SANTA CRUZ
XOXOCOTLÁN, OAXACA 01 DE MAYO DEL 2024

Proyecto En Clase

TAXPROPERTY

Se desea diseñar una aplicación que permita calcular el importe total que una persona debe pagar por el impuesto predial, considerando que una persona puede tener varios precios. El coto de cada predio está en función a la zona de ubicación y para ello se cuenta con un catalogo de zonas.

Clave	Zona	Costo x M ²
MAR	Marginado	2.00
RUR	Rural	8.00
URB	Urbana	10.00
RES	Residencial	25.00

El gobierno municipal está implementando el siguiente programa de descuento:

- Para las personas mayores o iguales de 70 años o madres solteras tiene un 70% de descuento si los pagos se realizan en los meses de Enero y Febrero y de un 50% en los siguientes meses.
- Para el resto de la población hay un descuento del 40% en los meses de Enero y Febrero.

EJEMPLO DE SALIDA

```
package com.itvo.taxproperty

import java.time.LocalDate

fun main() {
    val owner1 = Person(
        name = "JOSE JOSE",
        birthdate = LocalDate.parse("1950-01-01"),
        genre = "H",
        MaritalState.Single
    )
    val owner2 = Person(
        name = "Maria Belen",
        birthdate = LocalDate.parse("1980-01-01"),
        genre = "M",
        MaritalState.Married
    )
}
```

La aplicación actual resuelve efectivamente el problema planteado con una arquitectura limpia y mantenible.

El programa principal (main) muestra un ejemplo con:

- 2 propietarios con diferentes características
- 3 zonas con diferentes costos
- 3 propiedades distribuidas entre los propietarios

Como resultado:

Cálculo de impuestos para cada propietario

"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" ...
JOSE JOSE => Total: 30000.0, Discount: 15000.0, Net total: 15000.0
Maria Belen => Total: 400.0, Discount: 0.0, Net total: 400.0

Process finished with exit code 0

```
data class Tax(  
    val properties: List<Property>,  
    val owner: Person,  
    val datePayment: LocalDate  
) {  
  
    fun total() = properties.filter { it.owner == owner }.sumOf { it.tax() }  
  
    fun calculateDiscount(): Double {  
        val total = this.total()  
        val SEVENTY_YEARS = 70  
        val discount = if (owner.getAge() >= SEVENTY_YEARS ||  
            owner.maritalState == MaritalState.Single){  
            if (datePayment.month<= Month.FEBRUARY){  
                total * 0.70  
            } else total * 0.50  
        } else {  
            if (datePayment.month<= Month.FEBRUARY){  
                total * 0.40  
            } else 0.0  
        }  
        return discount  
    }  
  
    fun calculateNetTotal() = total()-calculateDiscount()  
}
```

RESOLUCIÓN DEL PROBLEMA

Estructura Básica

En la clase Tax se resuelve el problema mediante:

1. **Suma de impuestos (total()):** Calcula el valor base de todas las propiedades del dueño.

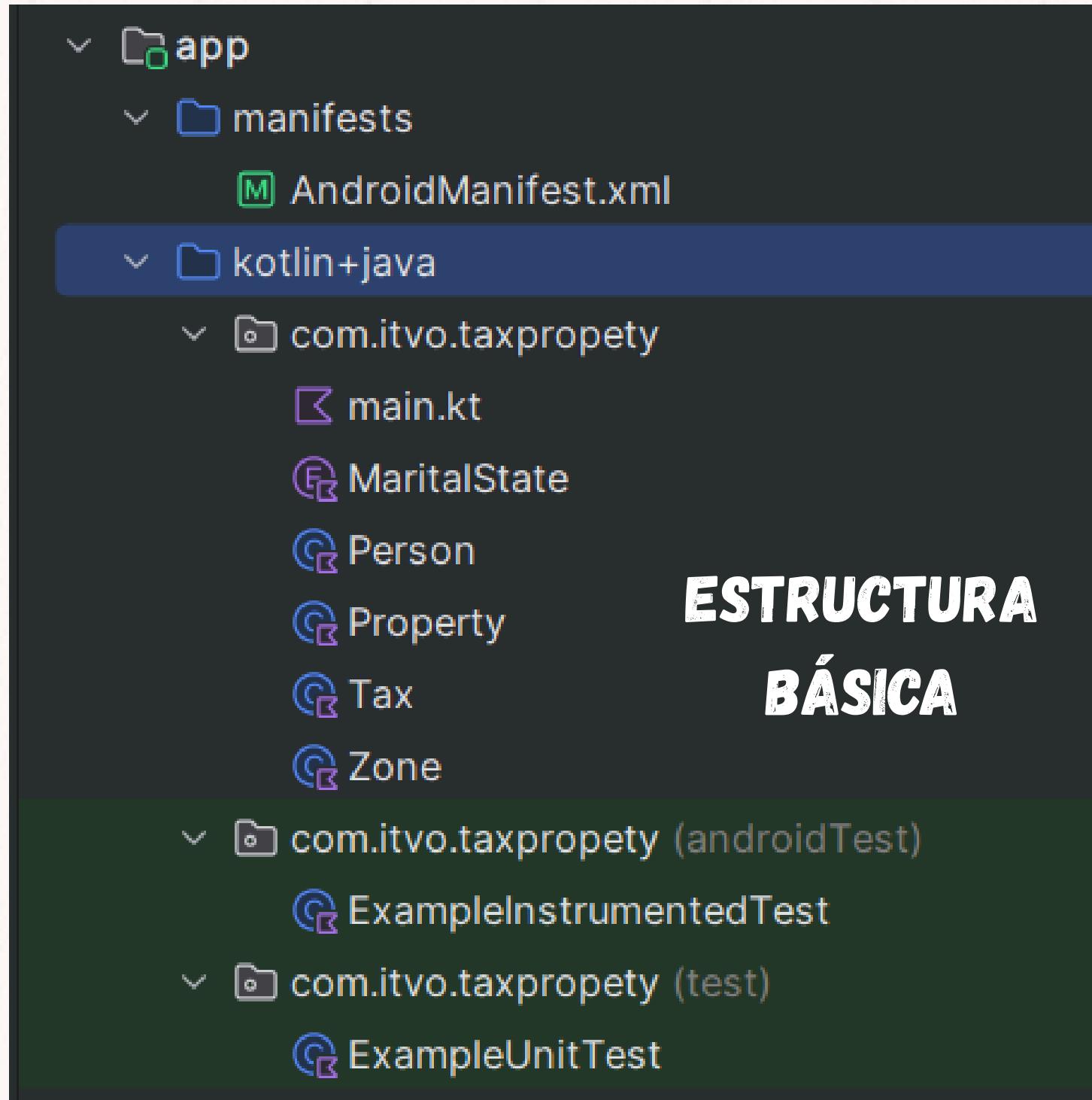
2. **Descuentos inteligentes (calculateDiscount()):**

Aplica bonificaciones según edad (≥ 70 años), estado civil (soltero) y fecha de pago (enero/febrero).

3. **Total neto (calculateNetTotal()):**

Combina ambos resultados para obtener el valor final a pagar.

"Aquí se ejecuta la lógica clave que transforma datos en resultados fiscales."



Estructura Básica

"La imagen muestra la estructura básica del proyecto Android, organizada en carpetas principales:

1. Código fuente: Clases esenciales (Person, Property, Tax, Zone, MaritalState) para gestionar datos y lógica. Punto de entrada (main.kt) con ejemplos de uso.
1. Pruebas: Carpetas androidTest (pruebas instrumentadas) y test (unitarias).
2. Recursos y configuración: Archivos estáticos y AndroidManifest.xml.

Estructura típica de Android, enfocada en separar componentes por funcionalidad."

Proyecto En Clase

SALARY

Una empresa nos ha encargado un programa para calcular las nóminas de los trabajadores. El sueldo base semanal sale aplicando la siguiente fórmula:

horasTrabajo * precioHora + horasExtra * precioHoraExtra

El precioHora es una constante = 90. El precioHoraExtra depende de las horas extras hechas:

Si son menos de 10h extras semanales, el precio es un 50% mayor que el precioHora (* 1.5).

Si se hacen entre 10 y 20h extra, el precio es un 40% mayor. Si se hacen más de 20h, el precio es un 20% mayor.

Si el trabajador es de categoría 3, el preciohora es constante.

Si es de categoria 2; el precioHora es un 25% mayor y

Si es de categoría 1 es un 45% más.



EJEMPLO DE SALIDA

```
Salary > app > src > main > java > com > itvo > salary > main.kt
```

Android Studio interface showing the project structure and code editor.

Project Structure:

- app
 - manifests
 - AndroidManifest.xml
 - kotlin+java
 - com.itvo.salary
 - Category
 - main.kt
 - Payroll
 - Worker
 - com.itvo.salary (androidTest)
 - ExampleInstrumentedTest
 - com.itvo.salary (test)
 - ExampleUnitTest
 - PayrollTest
 - WorkerTest
 - res
 - res (generated)
 - Gradle Scripts

Code Editor (main.kt):

```
4
5 fun main(){
6     val workers = listOf(
7         Worker(fullName = "Ambrosio Cardoso",
8             curp = "CAJA741207HOCRMM06",
9             dateOfHire = LocalDate.parse("2009-05-16"),
10            category = Category.UNO
11        ),
12        Worker(fullName = "Rosa Maria Lopez",
13             curp = "LOSR990817MOCSRS01",
14             dateOfHire = LocalDate.parse("2024-08-16"),
15             category = Category.TRES
16    ),
17    )
18    val payroll = Payroll (
19        worker = workers[0],
20        hoursWorked = 45
21    )
22
23    println("${workers[0].fu

```

Run Tab Output:

```
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" ...
Ambrosio Cardoso recibe
pago base: 5220.0
horas extras: 978.75
TOTAL: 6198.75

Rosa Maria Lopez recibe
pago base: 5220.0
horas extras: 978.75
TOTAL: 6198.75

Process finished with exit code 0
```

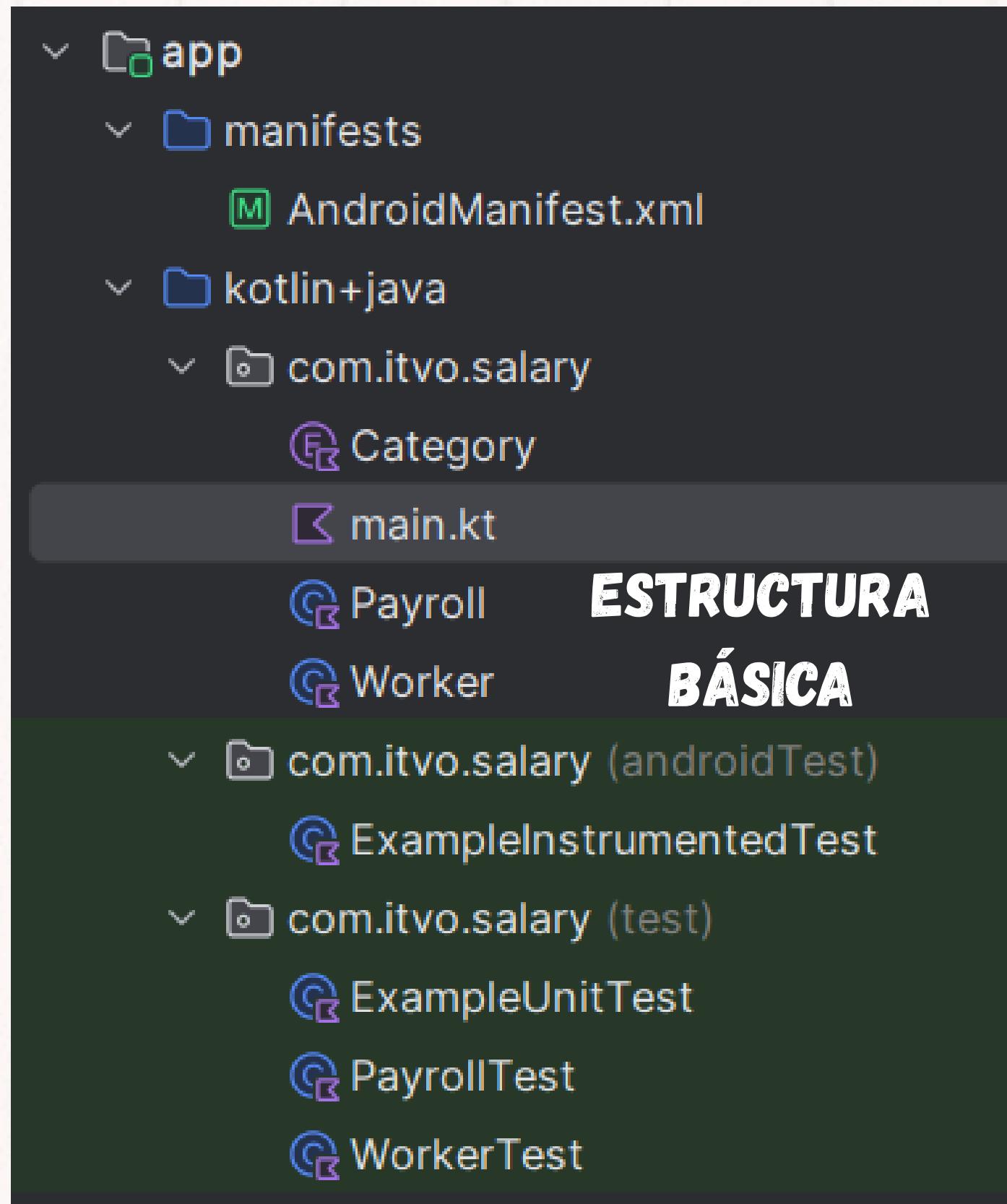
Text Callout:

La aplicación actual resuelve efectivamente el problema planteado con una arquitectura limpia y mantenible.

El programa principal (main) muestra un caso práctico con:

- 2 trabajadores con características diferentes:
 - Categoría 1 (precio/hora +45%)
 - Categoría 3 (precio/hora base)
- 2 escenarios de horas trabajadas:
 - 45 horas (5 extras)

1:1 CRLF UTF-8 4 spaces



Estructura Básica

"La imagen muestra la organización estándar del proyecto Android para el sistema de nóminas:

1. Código fuente:

- Clases principales (Worker, Payroll, Category) para gestionar empleados y cálculos salariales.
- Punto de entrada (main.kt) con ejemplos de uso.

2. Pruebas:

- Unitarias (test/): PayrollTest (lógica de cálculo) y WorkerTest (validación de datos).
- Instrumentadas (androidTest/): ExampleInstrumentedTest (pruebas en dispositivo).

3. Configuración:

- AndroidManifest.xml (configuración esencial de la app).

Estructura Android típica, separando claramente lógica de negocio, pruebas y configuración.

```
3  data class Payroll(  
4      val worker: Worker,  
5      val hoursWorked: Int  
){  
7      fun calculateHourlyRate(): Double {  
8          val normalHourlyRate = 90.0  
9          return when (this.worker.category){  
10             Category.UNO -> normalHourlyRate * 1.45  
11             Category.DOS -> normalHourlyRate * 1.25  
12             Category.TRES -> normalHourlyRate  
13         }  
14     }  
16     fun calculateOvertime(): Int {  
17         return 0.coerceAtLeast(hoursWorked - 40)  
18     }  
19     fun calculateOvertimePay(): Double {  
20         val overtime = this.calculateOvertime()  
21         val overtimePay = when (overtime){  
22             in 0 .. ≤ 10 ->{  
23                 overtime * calculateHourlyRate() * 1.5  
24             }  
25             in 11 .. ≤ 20 ->{  
26                 overtime * calculateHourlyRate() * 1.4  
27             }  
28             else -> {overtime * calculateHourlyRate() * 1.2}  
29         }  
30         return overtimePay  
31     }  
32     fun calculateBasePay (): Double {  
33         return calculateHourlyRate() * this.hoursWorked.coerceAtMost(40)
```

RESOLUCIÓN DEL PROBLEMA

Resolución del Problema

La clase *Payroll* resuelve el cálculo completo de nóminas mediante:

- 1. Tarifa horaria** (`calculateHourlyRate`): Aplica incrementos del 45% (UNO), 25% (DOS) o tarifa base (TRES) sobre \$90/h.
- 2. Horas extras** (`calculateOvertimePay`): Calcula bonificaciones escalonadas (50% para ≤ 10 h, 40% para 11-20h, 20% para > 20 h).
- 3. Integración** (`calculatePayWithOvertime`): Combina salario base (máx. 40h) + extras.



Pruebas:

El archivo WorkerTest.kt contiene pruebas unitarias para validar el cálculo de antigüedad laboral (calculateSeniority()) en diferentes escenarios temporales.

Cada prueba sigue una estructura clara (preparación de datos, ejecución y verificación) y cubre casos específicos como contrataciones recientes, exactamente un año atrás, o incluso en años bisiestos.

Gracias a los nombres descriptivos de los métodos mejoran la legibilidad, mientras que las aserciones con assertEquals garantizan que el método funcione correctamente según los años transcurridos desde la fecha de contratación.

```
1 package com.itvo.salary
2
3 import org.junit.jupiter.api.Assertions.assertEquals
4 import org.junit.jupiter.api.Test
5 import java.time.LocalDate
6
7
8 class WorkerTest{
9     @Test
10    fun `test calculateSeniority when hired over a year ago`() {
11        // Datos de entrada: trabajador con fecha de contratación hace 3 años
12        val worker = Worker(
13            fullName = "Juan Pérez",
14            curp = "JUAP901231HDFRRR04",
15            dateOfHire = LocalDate.of(2020, 2, 14),
16            category = Category.UNO
17        )
18
19        // Valor esperado: La antigüedad debe ser 3 años (2023 - 2020)
20        val expectedSeniority = 5
21
22        // Verificación
23        assertEquals(expectedSeniority, worker.calculateSeniority())
24    }
25
26    @Test
27    fun `test calculateSeniority when hired less than a year ago`() {
28        // Datos de entrada: trabajador con fecha de contratación hace 6 meses
29        val worker = Worker(
30            fullName = "Ana Gómez",
31            curp = "ANAG940612MDFLRS07",
32            dateOfHire = LocalDate.of(2024, 8, 13),
33        )
34
35        // Valor esperado: La antigüedad debe ser 0 años (2024 - 2024)
36        val expectedSeniority = 0
37
38        // Verificación
39        assertEquals(expectedSeniority, worker.calculateSeniority())
40    }
41}
```

PRUEBAS



Pruebas:

Este conjunto de pruebas valida el sistema de cálculo de nómina, verificando cada componente del salario:

- **Tarifa horaria:** Comprueba que se apliquen correctamente los incrementos del 45%, 25% o 0% según la categoría del trabajador (UNO, DOS, TRES).
- **Horas extras:** Evalúa el cálculo preciso de horas extras (solo horas trabajadas por encima de 40) y su remuneración con bonificaciones escalonadas (50% para menos de 10h, 40% entre 10-20h, 20% para más de 20h).
- **Integridad del pago:** Valida que la suma del salario base (limitado a 40h) y las horas extras coincida con el total a pagar, usando márgenes de precisión decimal (0.01) para garantizar exactitud financiera.

Las pruebas cubren escenarios críticos como trabajadores con diferentes categorías, jornadas exactas de 40h, y casos con hasta 25h extras, asegurando robustez en el sistema de nóminas.

```
9  class PayrollTest {
10
11     private val workerUno = Worker(
12         "Juan",
13         curp = "CAJA741207HOCRMM06",
14         dateOfHire = LocalDate.parse("2000-01-01"),
15         category = Category.UNO
16     )
17
18     private val workerDos = Worker(
19         "Maria",
20         curp = "CAJA741207HOCRMM06",
21         dateOfHire = LocalDate.parse("1950-01-01"),
22         Category.DOS
23     )
24
25     private val workerTres = Worker(
26         "Pedro",
27         curp = "CAJA741207HOCRMM06",
28         dateOfHire = LocalDate.parse("1974-01-01"),
29         Category.TRES
30     )
31
32     @Test
33     fun `test calculateHourlyRate should return correct rate based on category`() {
34         assertEquals(130.5, Payroll(workerUno, 40).calculateHourlyRate(), 0.01)
35         assertEquals(112.5, Payroll(workerDos, 40).calculateHourlyRate(), 0.01)
36         assertEquals(90.0, Payroll(workerTres, 40).calculateHourlyRate(), 0.01)
37     }
38
39     @Test
40     fun `test calculateOvertime should return correct overtime hours`() {
41         assertEquals(0, Payroll(workerUno, 30).calculateOvertime())
42         assertEquals(0, Payroll(workerUno, 40).calculateOvertime())
43     }
44 }
```

PRUEBAS

Proyecto En Clase

CHECKINOUT

Este proyecto implementa un sistema integral para gestionar :

1. Registro de Asistencia

- **Control preciso de check-in/out con validación de retardos (11-20 min = retardo, >20 min = falta)**
- **Cálculo automático de incidencias (retardos, faltas) considerando:**
 - Antigüedad del empleado (<10 años: 3 retardos = 1 falta)
 - Permisos justificados



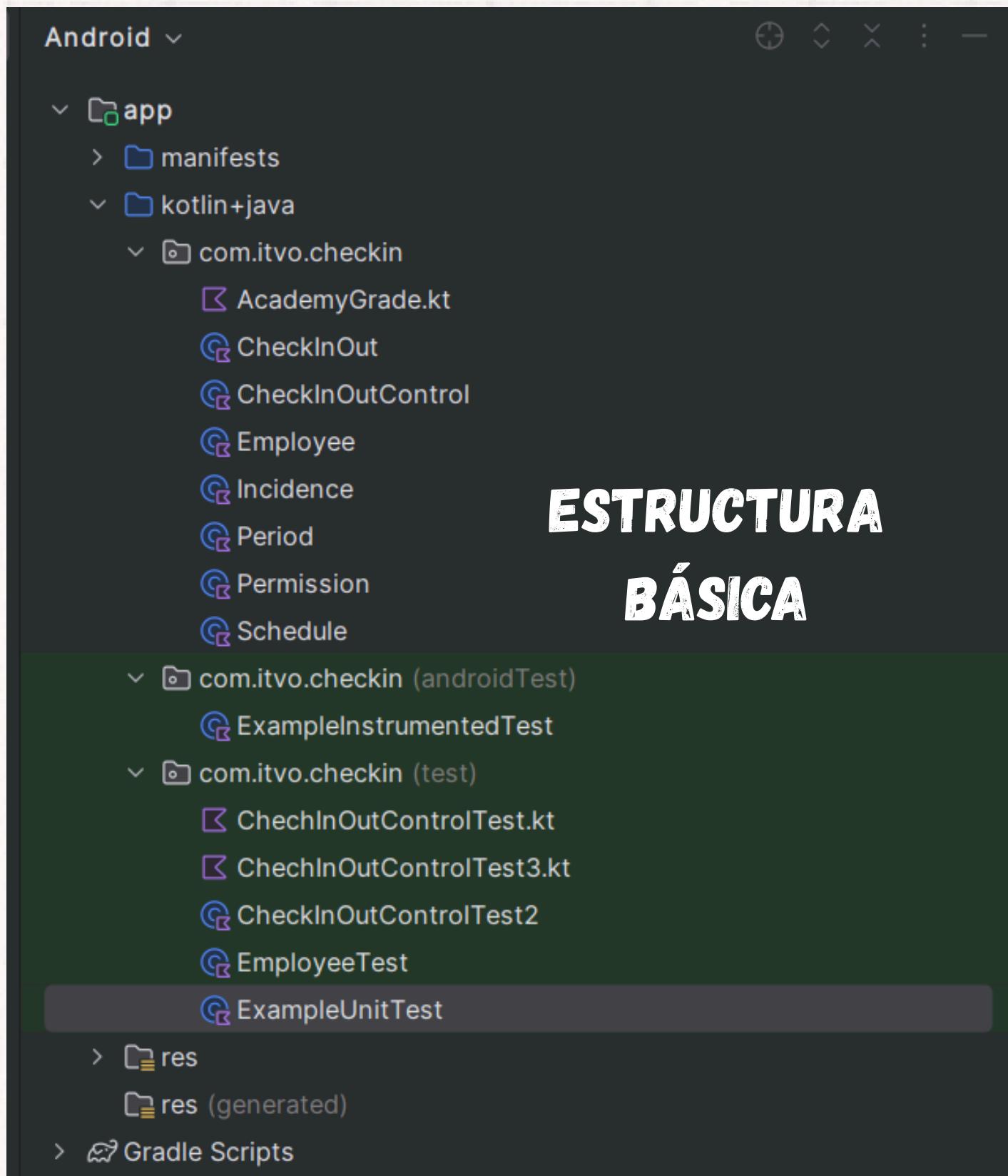
1. Gestión de Personal

- **Datos completos de empleados (CURP, grado académico, género, antigüedad)**
- **Horarios flexibles por día de la semana**

1. Pruebas Automatizadas

- **Validación de cálculos de antigüedad y género (extraído del CURP)**
- **Simulación de escenarios reales (retardos, horarios, períodos)**





Estructura Básica

La imagen muestra la estructura básica del proyecto para el sistema de control de asistencia:

Código fuente (com.itvo.checkin):

- Clases de dominio como Employee (empleados), CheckInOut (registro de entradas/salidas), Permission (permisos) y Schedule (horarios).
- Componentes de lógica como CheckInOutControl (gestión de registros) y AcademyGrade.

Pruebas:

- Unitarias (test/): CheckInOutControlTest (validación de lógica), EmployeeTest (datos de empleados).
- Instrumentadas (androidTest/): ExampleInstrumentedTest.

Estructura organizada, separando claramente la lógica, las pruebas y la configuración de la app.

RESOLUCIÓN DEL PROBLEMA

```
4
5 class CheckInOutControl() {
6     private val checkInOuts = mutableListOf<CheckInOut>()
7     private val schedules = mutableListOf<Schedule>()
8     private val permissions = mutableListOf<Permission>()
9
10    fun addSchedule(schedule: Schedule) {
11        schedules.add(schedule)
12    }
13
14    fun registerCheckInOut(checkInOut: CheckInOut) {
15        checkInOuts.add(checkInOut)
16    }
17
18    fun registerPermission(permission: Permission) {
19        permissions.add(
20            permission
21        )
22    }
23
24    private fun isDelay(checkInOut: CheckInOut, schedule: Schedule?): Boolean {
25        val minutesDifference =
26            ChronoUnit.MINUTES.between(schedule!!.checkInTime, checkInOut.checkIn)
27        return (minutesDifference in 11 .. 20)
28    }
29
30    private fun isAbsence(checkInOut: CheckInOut, schedule: Schedule?): Boolean {
31        val minutesDifference =
32            ChronoUnit.MINUTES.between(schedule!!.checkInTime, checkInOut.checkIn)
33
34        return (checkInOut.checkOut.isBefore(schedule.checkOutTime) ||
35                minutesDifference > 20)
```

Resolución del Problema

La clase `CheckInOutControl` resuelve el control de asistencia mediante:

1. Registro y filtrado: Organiza horarios (Schedule), registros (CheckInOut) y permisos (Permission) por empleado y período.

2. Detección de retardos y Ausencias:

- Retrasos (isDelay): 11-20 minutos después de la hora entrada.
- Ausencias (isAbsence): Más de 20 minutos de retraso o salida anticipada.

3. Ajuste por antigüedad: Convierte retrasos en ausencias (3 retrasos = 1 ausencia) si la antigüedad del empleado es <10 años.

4. Cálculo integrado (calculateIncidenceForPeriod): Devuelve un resumen de incidencias (delays, absences, permissions) para un período específico, combinando lógica de comparación de horarios y validación de permisos.



Pruebas:

El test

`testCalculateIncidenceForPeriodShouldReturnOneDelays` verifica que se detecte correctamente 1 retraso cuando un empleado registra una entrada a las 8:15 AM (15 minutos tarde) en un horario programado a las 8:00 AM. Se configuran horarios para lunes, martes y viernes (`testAddSchedule`), y se registra una entrada puntual (11/02) y otra tardía (14/02).

El método `calculateIncidenceForPeriod` filtra estos registros en el período especificado (febrero 2025) y devuelve un objeto `Incidence` con `delays = 1`, validando que la lógica de detección de retrasos (`isDelay`) funciona según lo esperado (11-20 minutos = retraso).

```
3 // -----TEST DE RETARDO -----
4 > import ...
10
11 class CheckInOutControlTest{
12     val checkInOutControl = CheckInOutControl()
13     val employee = Employee(
14         employeeId = 1,
15         fullName = "Rosario Martinez Gomez",
16         academicDegree = AcademicDegree.MASTER,
17         curp = "XXXX200101MOCRSDC",
18         dateOfHire = LocalDate.of(2015, 5, 16),
19         budgetKey = "123ABC"
20     )
21     val period = Period(
22         initialDate = LocalDate.of(2025, 1, 1),
23         finalDate = LocalDate.of(2025, 12, 31)
24     )
25 }
```

PRUEBAS



Pruebas:

El test

`testCalculateIncidenceForPeriodShouldReturnOneabsences` verifica la conversión automática de retrasos en ausencias para empleados con menos de 10 años de antigüedad.

Se simulan 3 registros con retrasos (19, 13 y 16 minutos) en días laborables, los cuales son contabilizados inicialmente como retardos. Dado que el empleado tiene 9 años de antigüedad (fecha de contratación: 2015), el sistema convierte cada 3 retardos en 1 ausencia.

El resultado esperado es `absences = 1` (tras la conversión) y `delays = 0`, demostrando que la regla de negocio se aplica correctamente en `calculateIncidenceForPeriod`.

```
4
3 //----- TEST 3 RETRADOS = 1 FALTA -----
4
5 > import ...
11
12 class CheckInOutControlTest3 {
13     val checkInOutControl = CheckInOutControl()
14     val employee = Employee(
15         employeeId = 3,
16         fullName = "Rosario Martinez Gomez",
17         academicDegree = AcademicDegree.MASTER,
18         curp = "XXXX200101MOCRSDC",
19         dateOfHire = LocalDate.of(2015, 5, 16),
20         budgetKey = "123ABC"
21     )
22     val period = Period(
23         initialDate = LocalDate.of(2025, 1, 1),
24         finalDate = LocalDate.of(2025, 12, 31)
25     )
}
```

PRUEBAS



Pruebas:

El test

`testCalculateIncidenceForPeriodShouldReturnOneAbsences` verifica la correcta identificación de ausencias cuando:

1. Un empleado con 9 años de antigüedad (contratado en 2015) registra:

- Un retraso de 19 minutos (11/02 a las 8:19 AM, dentro del rango de 11-20 minutos que se considera retraso)
- Una entrada a las 11:00 AM (14/02, superando el límite de 20 minutos para ser considerado retraso y clasificándose como ausencia)

2. El sistema:

- Cuenta el segundo registro como ausencia directa (por superar el umbral de 20 minutos)
- Mantiene el primer registro como retraso (no aplica conversión a ausencia pues no se alcanzan 3 retardos)
- Retorna `absences = 1` en el objeto `Incidence`, validando que la lógica de `isAbsence` funciona correctamente para entradas extremadamente tardías.

```
2 //----- FALTA -----
3
4
5 > import ...
10
11 ✓ class CheckInOutControlTest2 {
12     val checkInOutControl = CheckInOutControl()
13     val employee = Employee(
14         employeeId = 2,
15         fullName = "Rosario Martinez Gomez",
16         academicDegree = AcademicDegree.MASTER,
17         curp = "XXXX200101MOCRSDC",
18         dateOfHire = LocalDate.of(2015, 5, 16),
19         budgetKey = "123ABC"
20     )
21     val period = Period(
22         initialDate = LocalDate.of(2025, 1, 1),
23         finalDate = LocalDate.of(2025, 12, 31)
24     )
}
```

PRUEBAS



Pruebas:

Este conjunto de tests verifica dos funcionalidades clave de la clase Employee:

1. Cálculo de Antigüedad

- calculateSeniority_should_ReturnCorrectYears: Valida que el método calculateSeniority() retorne correctamente 5 años para un empleado contratado hace exactamente ese período (usando LocalDate.now().minusYears(5)).
- Importante: La prueba depende de la fecha actual del sistema, lo que garantiza precisión en el cálculo basado en la fecha de contratación (dateOfHire).

2. Extracción de Género desde CURP

- getGenre_shouldReturnCorrectGender: Confirma que el método getGender() identifique el género Mujer ('M') cuando el 10º carácter de la CURP es 'M'.
- getGender_should_ReturnCorrectGenderForMale: Verifica que retorne Hombre ('H') cuando el 10º carácter de la CURP es 'H'.
- Base legal: La CURP sigue el estándar oficial mexicano donde el 10º carácter indica el sexo (H/M).

```
12  class EmployeeTest {  
13  
14  
15  @Test  
16  fun calculateSeniority_should_ReturnCorrectYears() {  
17      // Given  
18      val hireDate = LocalDate.now().minusYears(5) // Contratado hace 5 años  
19      val employee = Employee(  
20          employeeId = 1,  
21          fullName = "Juan Pérez",  
22          academicDegree = AcademicDegree.BACHELOR,  
23          curp = "XEXX010101HNEXXXA4", // H: Hombre  
24          dateOfHire = hireDate,  
25          budgetKey = "ABC123"  
26      )  
27  
28      // When  
29      val seniority = employee.calculateSeniority()  
30  
31      // Then  
32      assertEquals(5, seniority, "La antigüedad debe ser 5 años")  
33  }
```

PRUEBAS

Proyecto En Clase

GREETINGCARD

Este proyecto es una aplicación Android desarrollada con Jetpack Compose que muestra una tarjeta de saludo.

Su objetivo principal es demostrar los conceptos básicos de la creación de interfaces declarativas en Compose, utilizando componentes fundamentales como Surface, Text y manejo de temas personalizados.



The screenshot shows the Android Studio interface with the project 'Greeting Card' open. The code editor displays the file `MainActivity.kt` containing the following Kotlin code:

```
1 package com.itvo.greetingcard
2
3 > import ...
18
19
20 </> class MainActivity : ComponentActivity() {
21     override fun onCreate(savedInstanceState: Bundle?) {
22         super.onCreate(savedInstanceState)
23         enableEdgeToEdge()
24         setContent {
25             GreetingCardTheme {
26                 Scaffold(modifier = Modifier.fillMaxSize()) {
27                     Greeting(
28                         name = "Android",
29                         modifier = Modifier.padding(16.dp))
30                 }
31             }
32         }
33     }
34 }
```

The preview window on the right shows a teal-colored card with the text "Hi, my name is ROMA!". The build output window at the bottom shows a successful build:

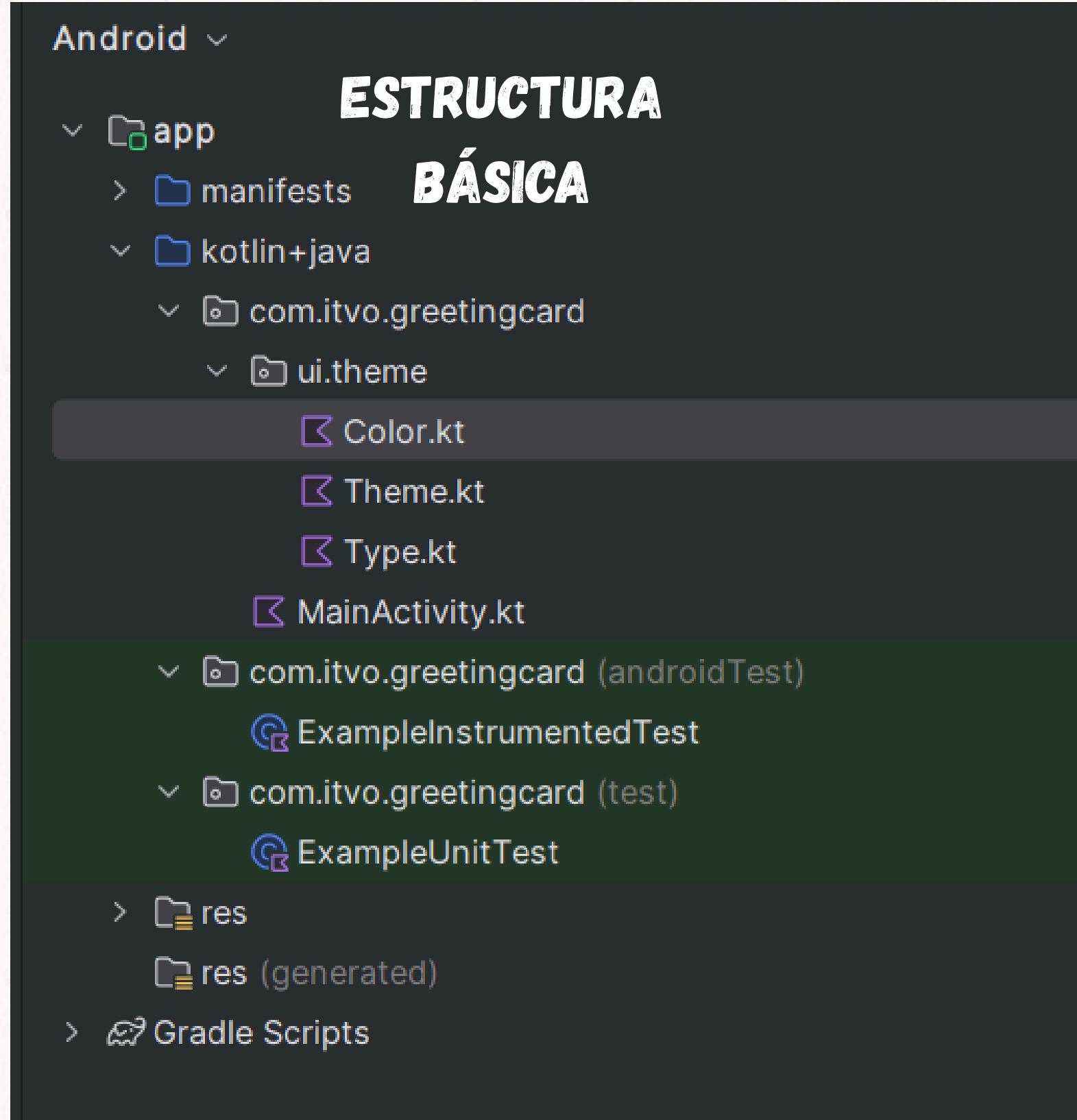
```
> Task :app:mergeDebugJavaResource
> Task :app:packageDebug
> Task :app:createDebugApkListingFileRedirect
> Task :app:assembleDebug

BUILD SUCCESSFUL in 1m 12s
32 actionable tasks: 32 executed
```

A callout bubble in the bottom right corner contains the text:

Podemos ver la visualización de lo que fue el resultado de este desarrollo.

At the bottom, the navigation bar shows the path: GreetingCard > app > src > main > java > com > itvo > greetingcard > MainActivity.kt > MainActivity > onCreate.



Estructura Básica

1. Capa de Presentación (UI) (`com.itvo.greetingcard`)

- `MainActivity.kt`:
- Componente Greeting:
 - Función `@Composable` que muestra un mensaje personalizado en un Surface con fondo cyan.
 - Uso de Modifier para espaciado y ajustes visuales.

2. Gestión de Temas y Estilos (ui.theme)

- `Color.kt`: Define la paleta de colores (ej: Purple80, Pink40).
- `Theme.kt`: Configura el tema claro/oscuro con `MaterialTheme` y esquemas de color dinámicos (Android 12+).
- `Type.kt`: Establece estilos tipográficos base (Typography).

```
1 package com.itvo.greetingcard
2
3 > import ...
18
19
20 ></> <class MainActivity : ComponentActivity() {
21     @Override fun onCreate(savedInstanceState: Bundle?) {
22         super.onCreate(savedInstanceState)
23         enableEdgeToEdge()
24         setContent {
25             GreetingCardTheme {
26                 Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
27                     Greeting(
28                         name = "Android",
29                         modifier = Modifier.padding(innerPadding)
30                     )
31                 }
32             }
33         }
34     }
35 }
36
37 @Composable
38 fun Greeting(name: String, modifier: Modifier = Modifier) {
39     Surface(color = Color.Cyan) {
40         Text(
41             text = "Hi, my name is $name!",
42             modifier = modifier.padding(24.dp)
43         )
44     }
45 }
```

ESTRUCTURA BÁSICA

Flujo de la Aplicación

- Inicio: MainActivity carga Greeting dentro de un Scaffold.
- Renderizado:
 - El componente Greeting muestra un Text en un Surface coloreado.
 - El tema aplica los colores y tipografía definidos en ui.theme.
- Previsualización: @Preview en GreetingPreview permite ver el diseño sin ejecutar la app.

Proyecto En Clase

PETS

Se desarrolló la siguiente aplicación móvil con el objetivo:

- 1. Mostrar mascotas de manera visual: Tarjetas con imágenes, nombres y descripciones.**
- 2. Implementar buenas prácticas de desarrollo: Uso de Compose, carga eficiente de imágenes y temas dinámicos.**
- 3. Garantizar escalabilidad: Estructura modular que facilita la incorporación de nuevas funcionalidades.**



```
>MainActivity.kt x Pixel 9 Pro XL API 36 +
```

1 package com.itvo.pets.presentation
2
3 > import ...
15
16 D </> class MainActivity : ComponentActivity() {
17 @↑ override fun onCreate(savedInstanceState: Bundle?) {
18 super.onCreate(savedInstanceState)
19 enableEdgeToEdge()
20 setContent {
21 PetsTheme {
22 Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
23 // Title(text = "Pets", modifier = Modifier.padding(innerPadding))
24 //PetList()
25
26 PetScreen(modifier = Modifier.padding(innerPadding))
27 }
28 }
29 }
30 }
31 }
32
33 @Composable
34 fun Greeting(name: String, modifier: Modifier = Modifier) {
35 Text(
36 text = "Hello \$name!",
37 modifier = modifier
38)
39 }
40 /*
41 @Preview(showBackground = true)
42 @Composable
43 fun GreetingPreview() {

Podemos ver la visualización de lo que fue el resultado de este desarrollo.

The screenshot shows a mobile application interface titled "I love Pets". The screen displays three items in a list format. Each item consists of a small icon followed by the pet's name. The items are: "Firulais black dog" (represented by a black dog icon), "Michi-Gan Person whit from" (represented by a cat icon), and "Big python" (represented by a snake icon). The background is white, and the list items have a light blue border.

```
om > itvo > pets > presentation > MainActivity.kt 34:5 LF UTF-8 4 spaces
```

Android



ESTRUCTURA BÁSICA

- app
 - manifests
 - kotlin+java
 - com.itvo.pets
 - data
 - local
 - remote
 - domain
 - petModel
 - PetType
 - presentation
 - composable
 - ListPet.kt
 - Title.kt
 - screen
 - PetScreen.kt
 - viewmodel
 - MainActivity.kt
 - ui.theme
 - Color.kt
 - Theme.kt
 - Type.kt
 - com.itvo.pets (androidTest)
 - ExampleInstrumentedTes
 - com.itvo.pets (test)
 - ExampleUnitTest

Estructura Básica

1. Capa de Datos (data/)

- local/: Almacena datos en memoria (petMemory), simulando una base de datos local.
- remote/: (Reservado para futuras integraciones con APIs o servicios web).

2. Capa de Dominio (domain/)

- Define los modelos esenciales (petModel, PetType) y reglas de negocio independientes del framework.

3. Capa de Presentación (presentation/)

- UI Components (composable):
 - ListPet.kt: Muestra el listado de mascotas en tarjetas con bordes redondeados.
 - Title.kt: Componente reutilizable con icono y texto.
- Pantallas (screen):
 - PetScreen.kt: los componentes para formar la vista principal.
- Lógica (viewmodel):
 - Gestiona el estado de la UI y comunica con la capa de datos (ejemplo: MainActivity.kt).

4. Temas y Estilos (ui.theme/)

- Color.kt: Paleta de colores (ejemplo: azul para bordes).
- Theme.kt: Configura temas claros/oscuros y colores dinámicos.
- Type.kt: Define estilos de texto (tamaños, fuentes).

```
1 package com.itvo.pets.presentation.composable
2
3 > import ...
4
5 @Composable
6 fun PetList(pets: List<petModel>, modifier: Modifier = Modifier) {
7     LazyColumn(
8         modifier = Modifier
9             .fillMaxSize()
10            .padding(16.dp)
11    ) {
12        items(pets) { pet ->
13            Row(
14                modifier
15                    .fillMaxWidth()
16                    .border(
17                        width = 2.dp,
18                        color = Color(3, 169, 244, 255),
19                        shape = RoundedCornerShape(8.dp)
20                    )
21            )
22            .padding(4.dp)
23        }
24        AsyncImage(
25            model = pet.image,
26            contentDescription = null, modifier
27                .width(75.dp)
28                .height(75.dp)
29        )
30        Spacer(modifier.width(8.dp))
31        Text(
32            text = pet.name + " " + pet.description,
33        )
34    }
35 }
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
```

1. Listado Visual de Mascotas:

- Se implementó mediante el componente ListPet.kt (en presentation/composable/), que muestra tarjetas con bordes redondeados, imágenes cargadas eficientemente con Coil (AsyncImage), y detalles como nombre y descripción.
- La lógica de renderizado se centró en:
 - Uso de LazyColumn para listas optimizadas.
 - Estructura de datos proporcionada por petModel (en domain/).

2. Lógica Clave:

- Capa de Dominio:
 - Modelos (petModel, PetType) definieron la estructura de datos de las mascotas.
- Capa de Presentación:
 - PetScreen.kt (en presentation/screen/) integró los componentes y consumió datos desde petMemory (en data/local/).