



**INSTITUTO TECNOLÓGICO DEL VALLE
DE OAXACA**



TRABAJO:

**BITÁCORA DEL DESARROLLO DE UNA
APLICACIÓN MÓVIL IMPLEMENTANDO LOS CINCO
PRINCIPIOS SOLID.**

MATERIA:

APLICACIONES MOVILES II

DOCENTE

AMBROSIO CARDOSO JIMÉNEZ

ALUMNO:

EMMANUEL JOSUE LOPEZ ZERMEÑO

GRUPO: "A"

SEMESTRE: 9

PROBLEMA 1 A RESOLVER: “PREDIAL”

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 predios. El costo de cada predio está en función a la zona de ubicación y para ello se cuenta con un catálogo **de zonas**.

Clave	Zona	Costo
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 sean 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.**

Objetivo: El objetivo principal del problema es desarrollar una solución que cumpla los principios SOLID, este caso dada la estructura de la problemática, es difícil poder cumplir todos los ya mencionados, los cuales son:

principio de responsabilidad única: Se refiere a que una clase u objeto debe centrarse y tener una responsabilidad única, cubrir un rubro único en el sistema.

principio de open/ close: Una entidad de software debería estar abierta a extensión, pero cerrada a modificación

principio de sustitución de liskov: Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

principio de segregación de interfaces: Ninguna clase debería depender de métodos que no usa.

Herramientas:

Android versión: 3.2.24

Windows 10 home.

Repositorio: <https://github.com/josue242/Proyecto-Predial.git>

Proceso de elaboración

Paso1. Para comenzar debimos haber identificado cada una de las clases, en este caso las entidades que ocuparemos dentro de la solución a nuestra problemática, comenzamos por nuestra clase “Persona”, ya que será la entidad de donde extraeremos información, declaramos los atributos necesarios.

Dentro de la misma clase crearemos nuestra función calcular edad, ya que es una función propia de este modo no estaríamos incumpliendo el principio de responsabilidad limitada, pues aún sigue cumpliendo una única función, en esta como podemos observar calculamos la edad con base a la fecha de nacimiento restando a la fecha actual la fecha de nacimiento y retornamos el resultado. Todo esto mostrado en la imagen 1.1

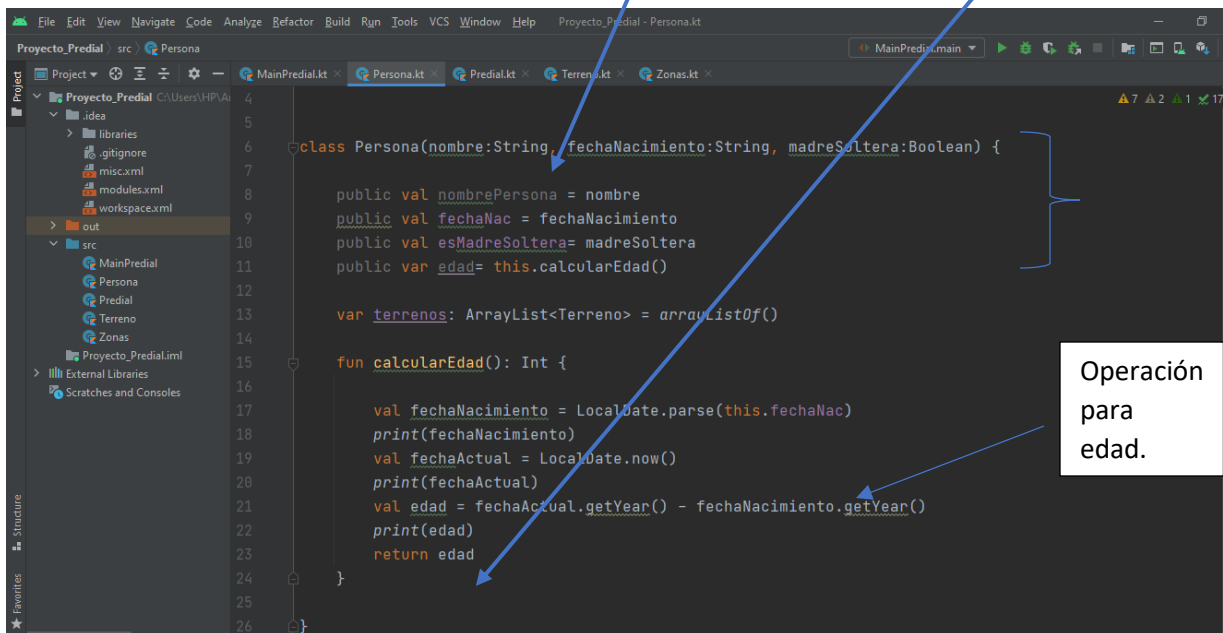


Imagen 1.1. clase Persona

Paso2: Una vez acabado con la clase Persona procederemos a continuar con nuestra siguiente entidad detectada, la cual es zona: en este caso, solamente declaramos sus atributos que a su vez serán sus atributos para poder trabajar con ellos. Como podemos observar utilizaremos costo, clave y zona, los 3 atributos para el catálogo de zonas.

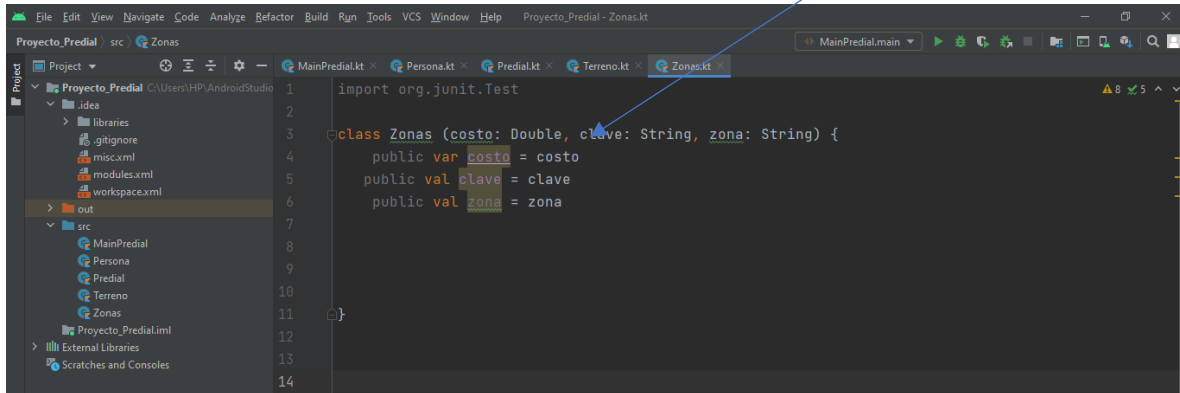


Imagen 1.2. Clase Zona

Paso3: Terreno juega un papel importante dentro de la problemática, pues con base a el podremos obtener los datos para calcular el importe del predial a pagar, pues nos otorgará tamaño, por lo que declaramos las siguientes variables, verificando la zona y el tamaño.

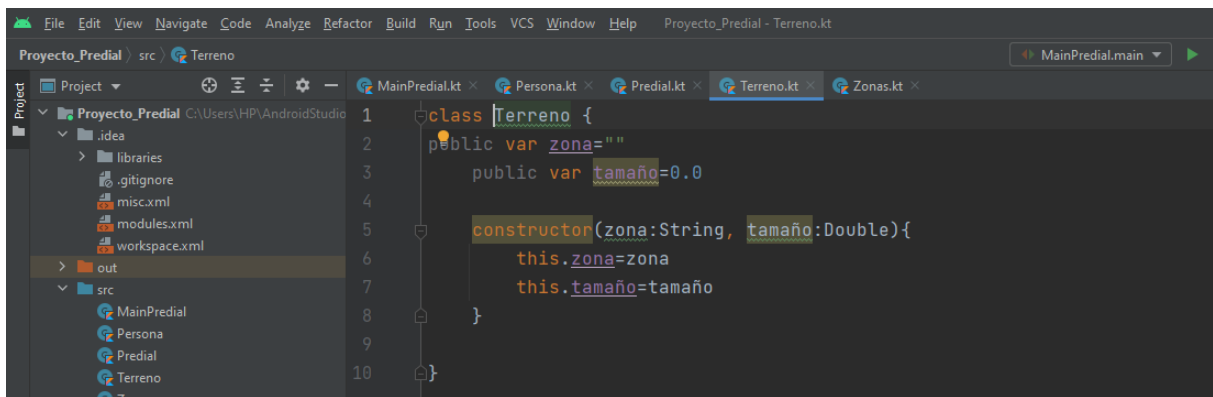
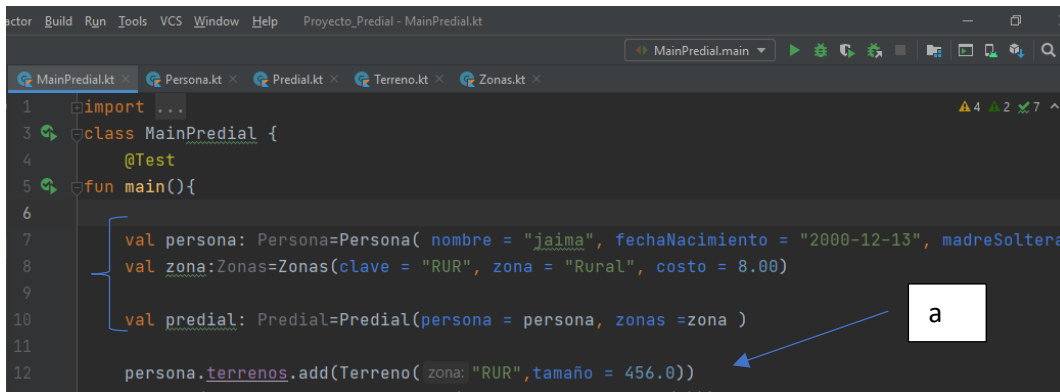


Imagen 1.3. Clase Terreno

Paso 4: En este caso para poder extraer información y tener una clase principal en donde imprimir nuestro resultado, tenemos nuestra clase MainPredial, donde crearemos las variables que almacenaran los datos a utilizar en la solución. Tal es el ejemplo de nombre, zona y predial.

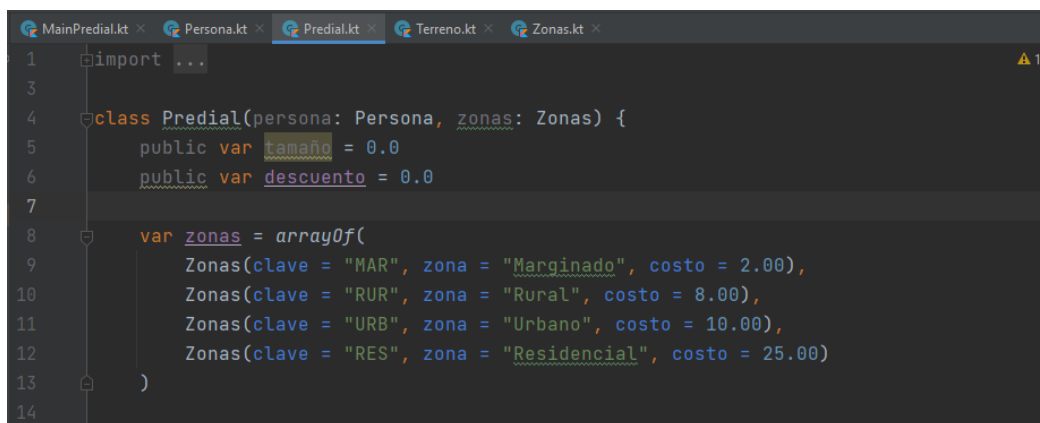
a) Posteriormente estamos agregando los atributos de zona y tamaño a terrenos, un arreglo de la clase persona, en el cual se guardaran la cantidad de terrenos o predios que la persona pueda tener.



```
1 import ...
2
3 class MainPredial {
4     @Test
5     fun main(){
6
7         val persona: Persona=Persona( nombre = "jaima", fechaNacimiento = "2000-12-13", madreSoltera
8         val zona:Zonas=Zonas(clave = "RUR", zona = "Rural", costo = 8.00)
9
10        val predial: Predial=Predial(persona = persona, zonas =zona )
11
12        persona.terrenos.add(Terreno( zona: "RUR", tamaño = 456.0))
```

Imagen 1.4. Clase MainPredial

Paso 5: Continuamos con la clase que realiza los cálculos correspondientes, en este caso especificaremos que clases ocuparemos como parámetros, posteriormente crearemos un arreglo de zonas con sus datos correspondientes, cabe aclarar que este fue ocupado por el main.



```
1 import ...
2
3
4 class Predial(persona: Persona, zonas: Zonas) {
5     public var tamaño = 0.0
6     public var descuento = 0.0
7
8     var zonas = arrayOf(
9         Zonas(clave = "MAR", zona = "Marginado", costo = 2.00),
10        Zonas(clave = "RUR", zona = "Rural", costo = 8.00),
11        Zonas(clave = "URB", zona = "Urbano", costo = 10.00),
12        Zonas(clave = "RES", zona = "Residencial", costo = 25.00)
13    )
14 }
```

Imagen 1.5. Clase Predial

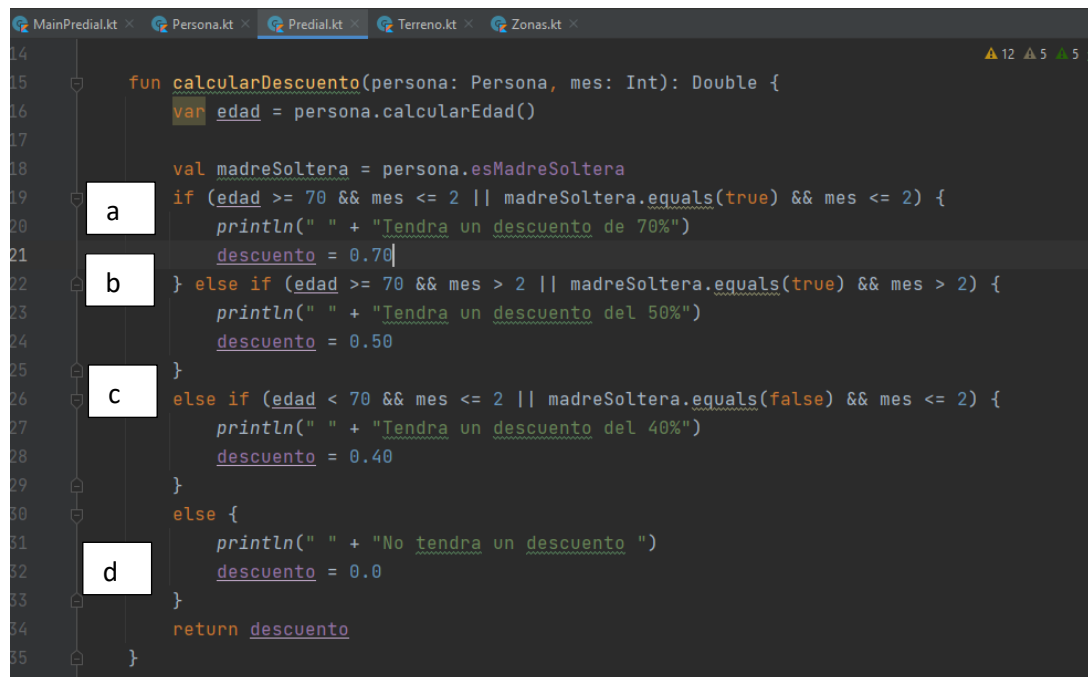
Paso 6: Dentro de la misma clase crearemos una función para poder calcular el descuento, esto a través de condiciones. Para esto tendremos como parámetros a la clase persona y mes, a su vez mandaremos a llamar el método calcular edad.

a) En el primer if, verificamos si la edad es igual o mayor a 70 y el mes igual o menos a 2, en este caso mes equivale a 1 = enero, por lo que sería igual o menor a febrero o si la persona es madre soltera e hizo el pago en los mismos términos. Si se cumpliese la condición, tendría un descuento del 70%.

b) En el primer else if, si el mes de pago es mayor a 2 “febrero” tendrá un descuento de 50%.

c) En el segundo else if, si no cumple con la edad y no es madre soltera, pero realizo el pago antes en febrero o menos, tendrá un descuento del 40%.

d) Si no aplica todas las anteriores no tiene descuento. Y retorna el valor.



```
14  
15 fun calcularDescuento(persona: Persona, mes: Int): Double {  
16     var edad = persona.calcularEdad()  
17  
18     val madreSoltera = persona.esMadreSoltera  
19     if (edad >= 70 && mes <= 2 || madreSoltera.equals(true) && mes <= 2) {  
20         println(" " + "Tendra un descuento de 70%")  
21         descuento = 0.70  
22     } else if (edad >= 70 && mes > 2 || madreSoltera.equals(true) && mes > 2) {  
23         println(" " + "Tendra un descuento del 50%")  
24         descuento = 0.50  
25     }  
26     else if (edad < 70 && mes <= 2 || madreSoltera.equals(false) && mes <= 2) {  
27         println(" " + "Tendra un descuento del 40%")  
28         descuento = 0.40  
29     }  
30     else {  
31         println(" " + "No tendra un descuento ")  
32         descuento = 0.0  
33     }  
34     return descuento  
35 }
```

Imagen 1.6. Clase Predial- calcularDescuento

Paso 7: Ahora para calcular el impuesto debemos obtener el dato de la zona, para extraer el valor del catalogo y el tamaño del terreno, en este caso, en este caso si la clave del catalogo de zona es igual a la zona del terreno realiza la operación del costo de la zona por el tamaño del terreno y retorna el resultado.

```
fun calcularImpuesto(terreno: Terreno): Double {  
    var resultado: Double = 0.0  
    zonas.forEach { zona: Zonas ->  
        if (zona.clave == terreno.zona) {  
            resultado = zona.costo * terreno.tamaño  
            return resultado  
        }  
    }  
    return resultado  
}
```

Imagen 1.7. Clase Predial- calcularImpuesto

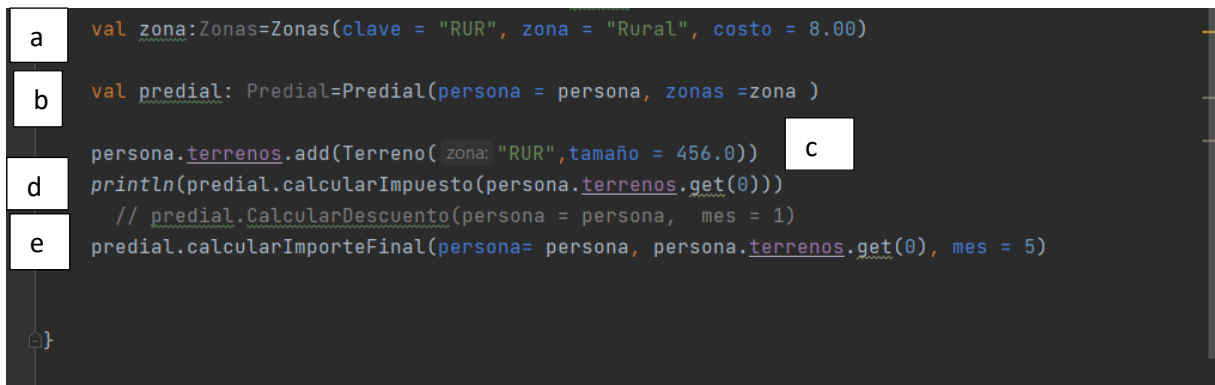
Paso 8: Para calcular el importe final, agregamos como atributos a persona, terreno y el mes, donde realizamos la operación de multiplicar el valor que retorna calcular descuento por el valor que retorna el calcular impuesto de esta forma obtenemos el descuento final, esto simplemente se le resta el valor del calcular impuesto, obteniendo el importe final.

```
50 }  
51 }  
52  
53     return resultado  
54 }  
55  
56 fun calcularImporteFinal(persona: Persona, terreno: Terreno, mes: Int): Double {  
57     var descuentoFinal = calcularDescuento(persona, mes) * calcularImpuesto(terreno)  
58     println("su descuento es: " + descuentoFinal)  
59     println("su importe total es: " + (calcularImpuesto(terreno) - descuentoFinal))  
60  
61     return calcularImpuesto(terreno) - descuentoFinal  
62 }  
63  
64  
65  
66 }  
67
```

Imagen 1.8. Clase Predial- calcularImporteFinal

Paso 9: Para esto, dentro del main, construimos a zona, predial asignándole datos para poder operar los métodos correspondientes.

- a) Asignamos a zonas valores para clave, zona y costo.
- b) Asignamos parámetros para predial.
- c) Asignamos datos en los parámetros de la clase de Terreno con base al arreglo de terrenos de la persona.
- d) Imprimimos el impuesto del primer terreno de la persona
- e) Imprimimos el importe final.



```

a  val zona:Zonas=Zonas(clave = "RUR", zona = "Rural", costo = 8.00)
b  val predial: Predial=Predial(persona = persona, zonas =zona )

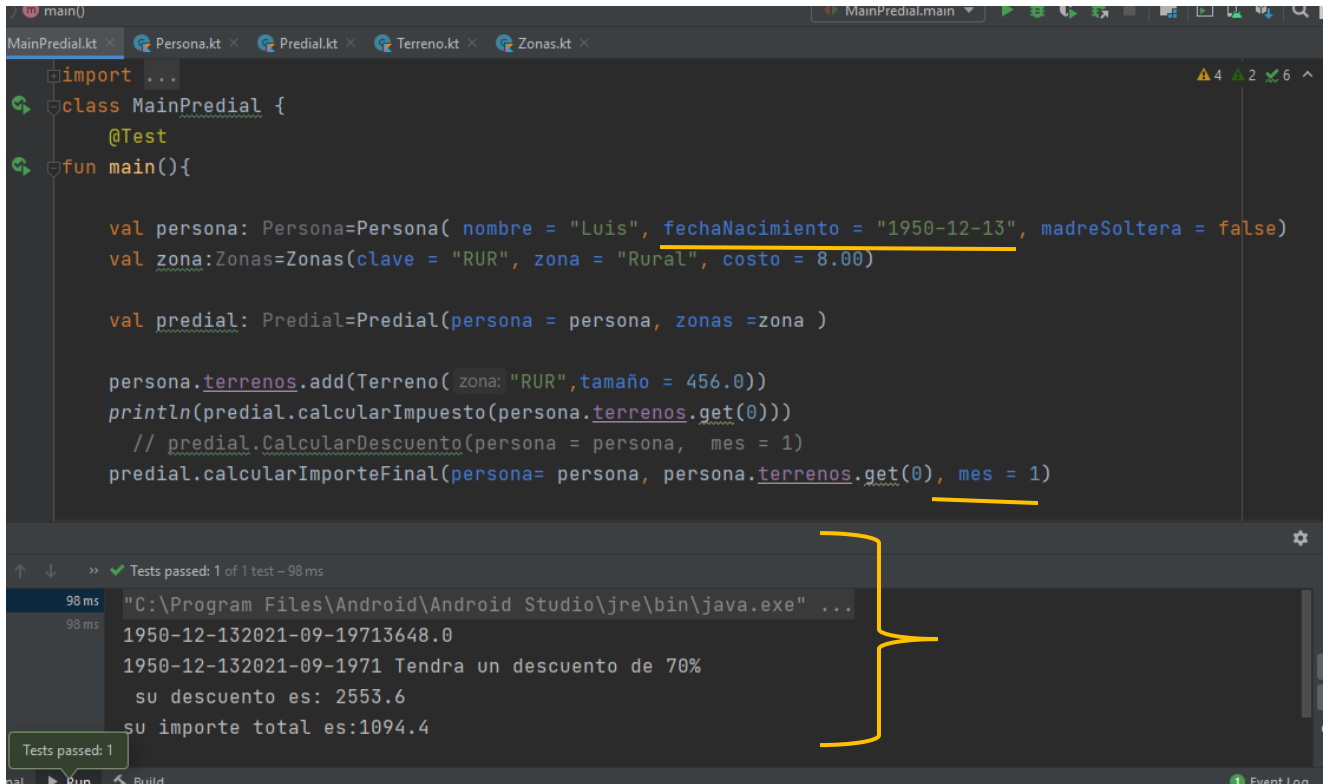
  persona.terrenos.add(Terreno( zona: "RUR", tamaño = 456.0)) c
d  println(predial.calcularImpuesto(persona.terrenos.get(0)))
  // predial.CalcularDescuento(persona = persona, mes = 1)
e  predial.calcularImporteFinal(persona= persona, persona.terrenos.get(0), mes = 5)

```

Imagen 1.9. Clase MainPredial-impresión

PRUEBAS

1. Cumpliendo con la edad “70 años o más”, con el mes de pago, en este caso debe tener un descuento del 70%, lo cual se muestra en la imagen con su descuento e importe final.



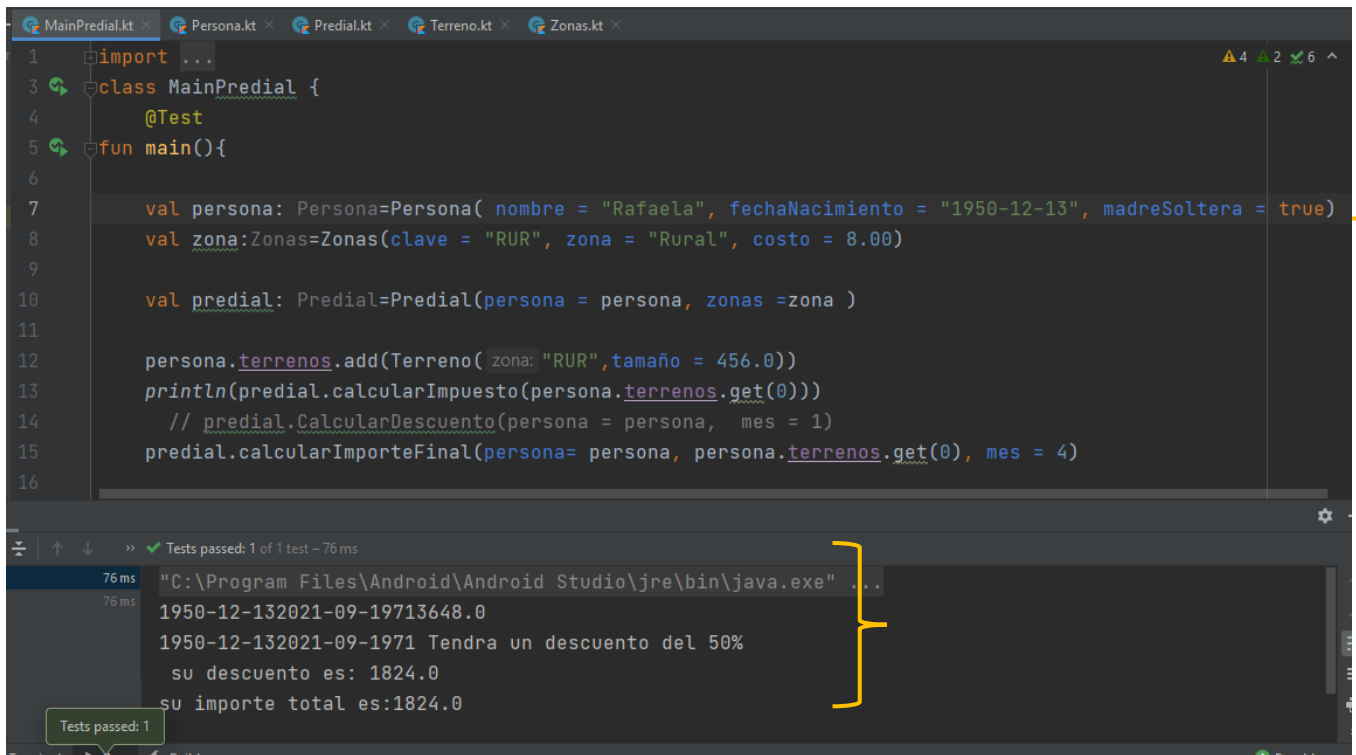
The screenshot shows an IDE with a Kotlin file named `MainPredial.kt`. The code defines a `MainPredial` class with a `@Test` annotated `main` function. Inside `main`, a `persona` object is created with `nombre = "Luis"` and `fechaNacimiento = "1950-12-13"`. A `zona` object is created with `clave = "RUR"`, `zona = "Rural"`, and `costo = 8.00`. A `predial` object is then created using `persona` and `zona`. The code then adds a `terreno` with `zona = "RUR"` and `tamaño = 456.0`, prints the calculated tax, and finally prints the final import and discount for `mes = 1`.

The output window shows the following results:

```
Tests passed: 1 of 1 test - 98 ms
98 ms "C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
98 ms 1950-12-132021-09-19713648.0
1950-12-132021-09-1971 Tendra un descuento de 70%
su descuento es: 2553.6
su importe total es:1094.4
```

Imagen 1.10. Prueba 1

2. Para madres solteras aplica el mismo descuento independientemente de la edad.



```
1 import ...
2
3 class MainPredial {
4     @Test
5     fun main(){
6
7         val persona: Persona=Persona( nombre = "Rafaela", fechaNacimiento = "1950-12-13", madreSoltera = true)
8         val zona:Zonas=Zonas(clave = "RUR", zona = "Rural", costo = 8.00)
9
10        val predial: Predial=Predial(persona = persona, zonas =zona )
11
12        persona.terrenos.add(Terreno( zona: "RUR",tamaño = 456.0))
13        println(predial.calcularImpuesto(persona.terrenos.get(0)))
14        // predial.CalcularDescuento(persona = persona, mes = 1)
15        predial.calcularImporteFinal(persona= persona, persona.terrenos.get(0), mes = 4)
16    }
17 }
```

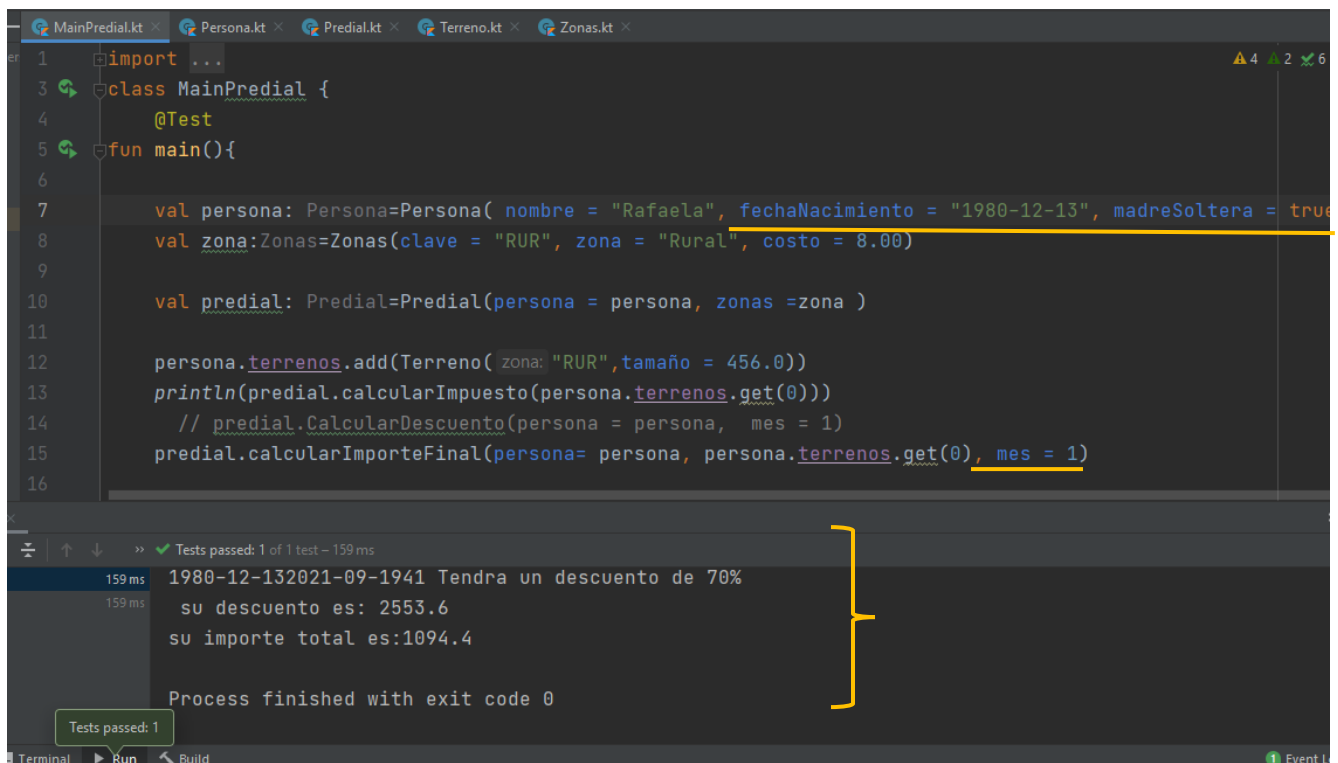
Tests passed: 1 of 1 test - 76 ms

76 ms "C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
76 ms 1950-12-132021-09-19713648.0
1950-12-132021-09-1971 Tendra un descuento del 50%
su descuento es: 1824.0
su importe total es:1824.0

Tests passed: 1

Imagen 1.11. Prueba 2

3. Si cumple con la edad o el estado civil pero no con el mes de pago entonces es un 50% de descuento.



```
1 import ...
2
3 class MainPredial {
4     @Test
5     fun main(){
6
7         val persona: Persona=Persona( nombre = "Rafaela", fechaNacimiento = "1980-12-13", madreSoltera = true)
8         val zona:Zonas=Zonas(clave = "RUR", zona = "Rural", costo = 8.00)
9
10        val predial: Predial=Predial(persona = persona, zonas =zona )
11
12        persona.terrenos.add(Terreno( zona: "RUR",tamaño = 456.0))
13        println(predial.calcularImpuesto(persona.terrenos.get(0)))
14        // predial.CalcularDescuento(persona = persona, mes = 1)
15        predial.calcularImporteFinal(persona= persona, persona.terrenos.get(0), mes = 1)
16    }
17 }
```

Tests passed: 1 of 1 test - 159 ms

159 ms 1980-12-132021-09-1941 Tendra un descuento de 70%
159 ms su descuento es: 2553.6
su importe total es:1094.4

Process finished with exit code 0

Tests passed: 1

```
import ...
class MainPredial {
    @Test
    fun main(){

        val persona: Persona=Persona( nombre = "Rafaela", fechaNacimiento = "2000-12-13", madreSoltera = false)
        val zona:Zonas=Zonas(clave = "RUR", zona = "Rural", costo = 8.00)

        val predial: Predial=Predial(persona = persona, zonas =zona )

        persona.terrenos.add(Terreno( zona: "RUR",tamaño = 456.0))
        println(predial.calcularImpuesto(persona.terrenos.get(0)))
        // predial.CalcularDescuento(persona = persona, mes = 1)
        predial.calcularImporteFinal(persona= persona, persona.terrenos.get(0), mes = 5)
    }
}
```

Tests passed: 1 of 1 test - 173 ms

173 ms "C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...

173 ms 2000-12-132021-09-19213648.0

2000-12-132021-09-1921 No tendra un descuento

su descuento es: 0.0

su importe total es:3648.0

Imagen 1.12. Prueba 3

4. Si no cumple con la edad o el estado civil pero sí con el mes de pago, tendrá un 40% de descuento.

```
1 import ...
3 class MainPredial {
4     @Test
5     fun main(){
6
7         val persona: Persona=Persona( nombre = "Rafaela", fechaNacimiento = "2000-12-13", madreSoltera = false)
8         val zona:Zonas=Zonas(clave = "RUR", zona = "Rural", costo = 8.00)
9
10        val predial: Predial=Predial(persona = persona, zonas =zona )
11
12        persona.terrenos.add(Terreno( zona: "RUR",tamaño = 456.0))
13        println(predial.calcularImpuesto(persona.terrenos.get(0)))
14        // predial.CalcularDescuento(persona = persona, mes = 1)
15        predial.calcularImporteFinal(persona= persona, persona.terrenos.get(0), mes = 2)
16    }
}
```

Tests passed: 1 of 1 test - 100 ms

100 ms "C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...

100 ms 2000-12-132021-09-19213648.0

2000-12-132021-09-1921 Tendra un descuento del 40%

su descuento es: 1459.2

su importe total es:2188.8

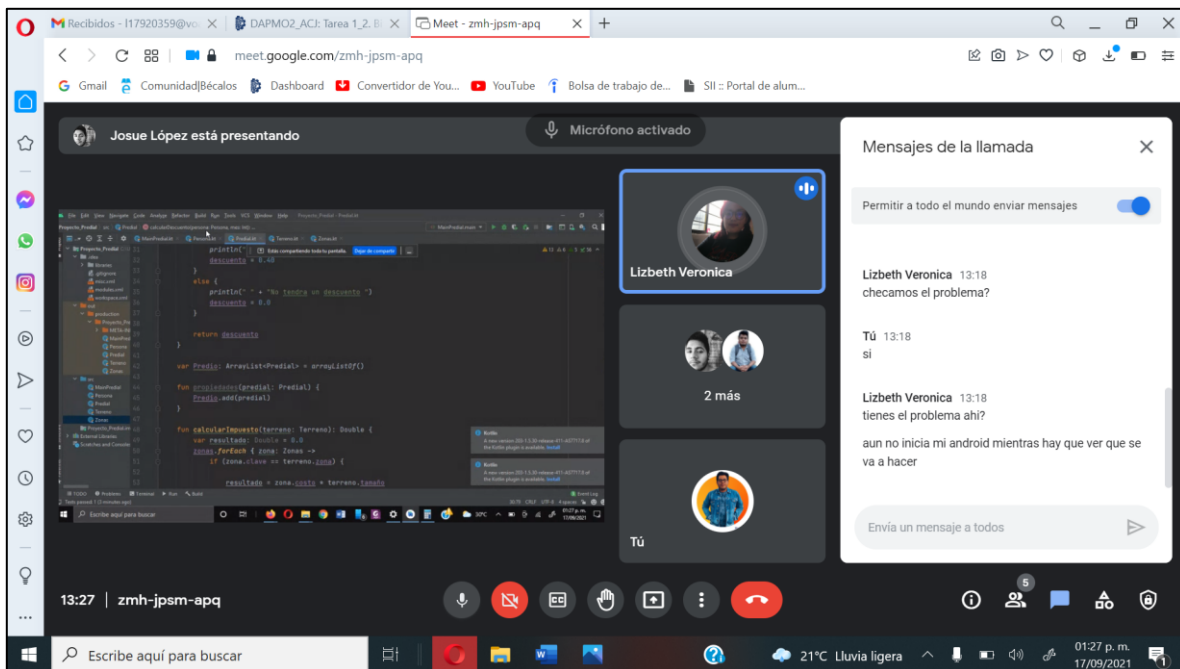
5. Si no cumple con ninguna de las anteriores no tiene descuento.

Principios: Dentro del proceso de elaboración pudimos observar que cada una de las clases se encarga de una función específica, dividiendo la solución en varias clases con responsabilidad directa, esto a su vez se puede verificar en los parámetros de cada clase, ya que cada una ocupa lo que necesita sin jalar datos de otro lado. En este caso se cumple el primer método: **responsabilidad única**.

Así también se puede observar que el código está abierto a agregación o extensión del proyecto sin afectar o modificar el código para que esto suceda. Por lo que se cumpliría el principio de **open/close**, cabe aclarar que también todas las clases y funciones se utilizan, todas tienen una funcionalidad.

Evidencia de trabajo

Debido a nuestra falta de experiencia, la problemática se nos complicó en cuanto a la sintaxis de Kotlin así como algunos aspectos relacionales como la lógica, por lo que decidimos colaborar con algunos compañeros (Gustavo Romero Monterroza, Lizbeth Verónica y Damián Martínez) retroalimentarnos, cabe recalcar que considero que es una muy buena forma de trabajar y adquirir nuevos conocimientos, ya que hay más opiniones, debates y autonomía para poder dar solución a la problemática.



Recibidos - 117920359@v... INF_DISINT_ECO: EJERCICIO... Meet - zmh-jpsm-apq

meet.google.com/zmh-jpsm-apq

Lizbeth Veronica está presentando

The screenshot shows a Google Meet interface. On the left, a code editor displays Java code for a class named `MainControl`. The code includes an import statement, a class definition, and a `main` method that creates a `Personal` object. The right side of the screen shows a grid of participant avatars: Josue López, Lizbeth Veronica, DAMIAN MARTINEZ JIMENEZ, and 'Tú' (You). The bottom of the screen features a taskbar with various application icons and a system tray showing the time as 02:47 p.m. on 17/09/2021.

```
1 import org.junit.Test
2 class MainControl {
3
4     val personal: Personal = Personal(id=PersonalId(), nombre = "Jaine", correo="jpd@u",
5     fecha_ingreso = "1966-03-12", genero="M", clavePresupuestal="a")
6
7
8
9
10
11 }
```

Activar micrófono (Ctrl + D)

14:47 | zmh-jpsm-apq

Escribe aquí para buscar

23°C 02:47 p.m. 17/09/2021

PROBLEMA 2 A RESOLVER: "CONTROL REGISTRO"

```
13 2. Se desea crear un programa para el control de registro de entrada y salida de personal de un centro educativo
14 los datos del personal son: identificador, nombre completo, grado academico (Bachillerato, licenciatura, maestría o
15 doctorado), curp
16 fecha de ingreso, genero y clave presupuestal. El personal tiene asignado previamente un horario de trabajo y en
17 función a ello
18 se va a determinar según el registro de entrada si tiene retardo o en su caso sino registró entrada podría ser una
19 falta o permiso justificado
20 El sistema cada quincena debe generar el total de inasistencias, retardos o permisos justificados de cada personal;
21 se considera retardo
22 si el registro se realizó en un intervalo entre 11 a 20 minutos después del horario establecido. También se
23 considera inasistencia si la salida se registró antes del horario de salida establecida
24
25 Los horarios previamente establecidos debe tener el id del personal, día, hora entrada y hora salida y desde luego
26 la fecha inicial y final que aplica ese horario (Se asume que solo habrá un horario por día).
27
28 Si la antigüedad es menor a 10 años cada 3 retardos a la quincena serán contabilizados como 1 falta.
```

Objetivo: El objetivo principal del problema es desarrollar una solución que cumpla los principios SOLID, este caso dada la estructura de la problemática, es difícil poder cumplir todos los ya mencionados, los cuales son:

principio de responsabilidad única: Se refiere a que una clase u objeto debe centrarse y tener una responsabilidad única, cubrir un rubro único en el sistema.

principio de open/ close: Una entidad de software debería estar abierta a extensión, pero cerrada a modificación

principio de sustitución de liskov: Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

principio de segregación de interfaces: Ninguna clase debería depender de métodos que no usa.

Herramientas:

Android versión: 3.2.24

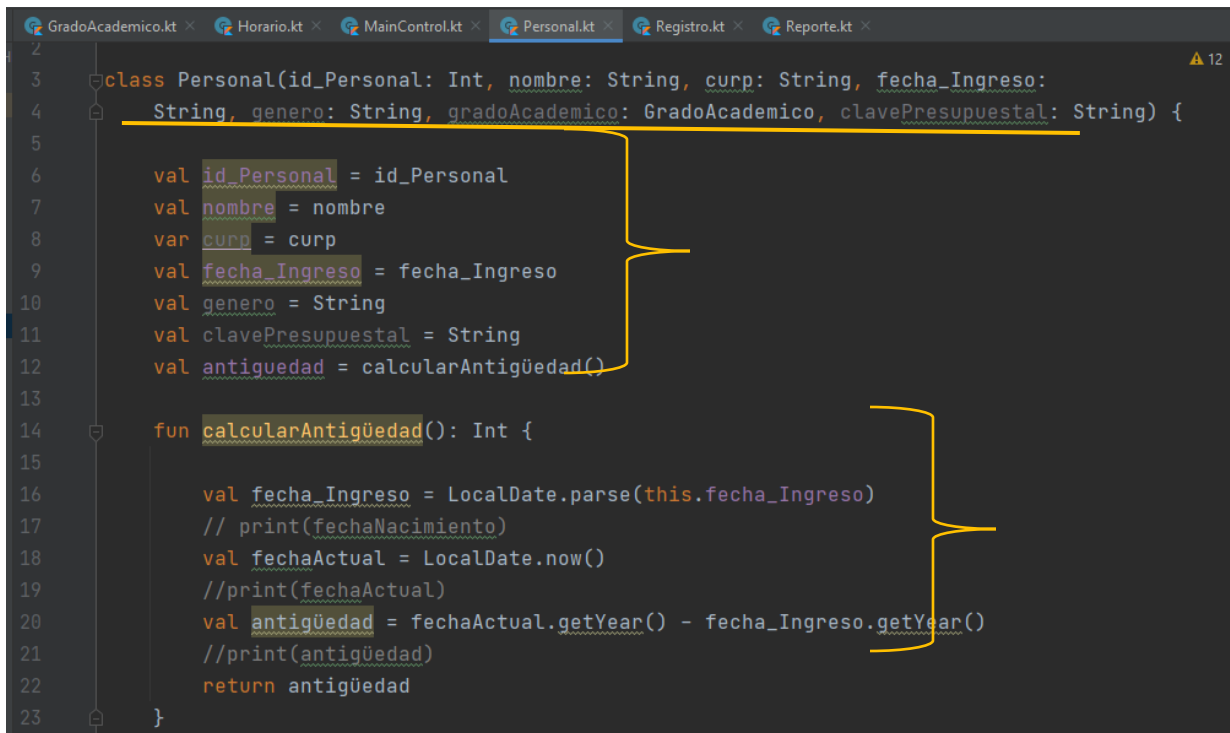
Windows 10 home.

Repositorio: <https://github.com/josue242/RegistroPersonal.git>

Proceso de elaboración

Paso 1: Una vez que analizamos y identificamos las entidades que formaran parte de nuestra solución comenzamos por la clase Personal, donde utilizaremos como parámetros los siguientes datos, cada uno de ellos fungirá de manera importante en el proceso.

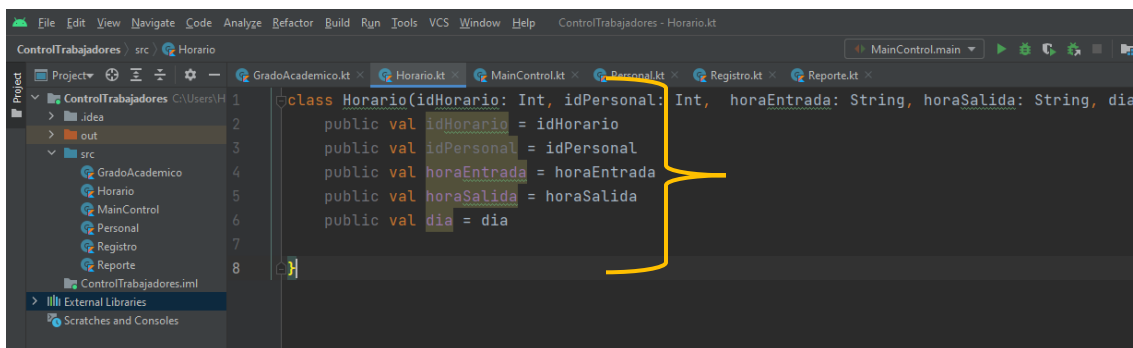
Posteriormente creamos la función calcularAntigüedad el cual pertenece a personal en donde con base a la fecha de ingreso y la fecha actual calcularemos la antigüedad.



```
2
3 class Personal(id_Personal: Int, nombre: String, curp: String, fecha_Ingreso:
4 String, genero: String, gradoAcademico: GradoAcademico, clavePresupuestal: String) {
5
6     val id_Personal = id_Personal
7     val nombre = nombre
8     var curp = curp
9     val fecha_Ingreso = fecha_Ingreso
10    val genero = String
11    val clavePresupuestal = String
12    val antigüedad = calcularAntigüedad()
13
14    fun calcularAntigüedad(): Int {
15
16        val fecha_Ingreso = LocalDate.parse(this.fecha_Ingreso)
17        // print(fechaNacimiento)
18        val fechaActual = LocalDate.now()
19        //print(fechaActual)
20        val antigüedad = fechaActual.getYear() - fecha_Ingreso.getYear()
21        //print(antigüedad)
22        return antigüedad
23    }
```

Figura 1.1, Clase Personal.

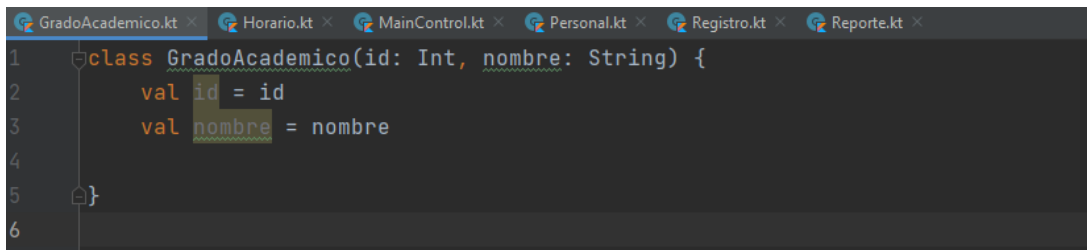
Paso 2: Continuamos con otra entidad, la cual es Horario, contendrá los datos necesarios para poder generar los horarios del personal y se pueda controlar con base a ello.



```
1 class Horario(idHorario: Int, idPersonal: Int, horaEntrada: String, horaSalida: String, dia
2
3     public val idHorario = idHorario
4     public val idPersonal = idPersonal
5     public val horaEntrada = horaEntrada
6     public val horaSalida = horaSalida
7     public val dia = dia
8
```

Figura 1.2, Clase Horario.

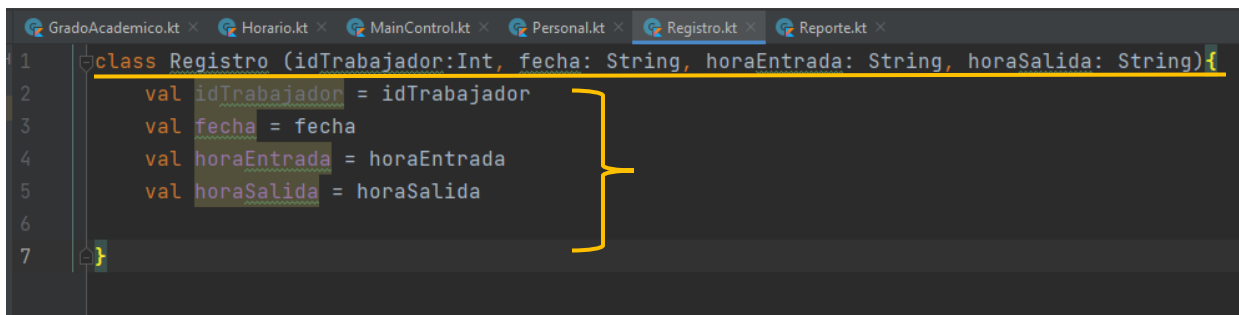
Paso 3: Creamos nuestra clase de GradoAcademico, esto para asignar un grado academico al usuario.



```
1 class GradoAcademico(id: Int, nombre: String) {
2     val id = id
3     val nombre = nombre
4
5 }
6
```

Figura 1.3, Clase GradoAcademica.

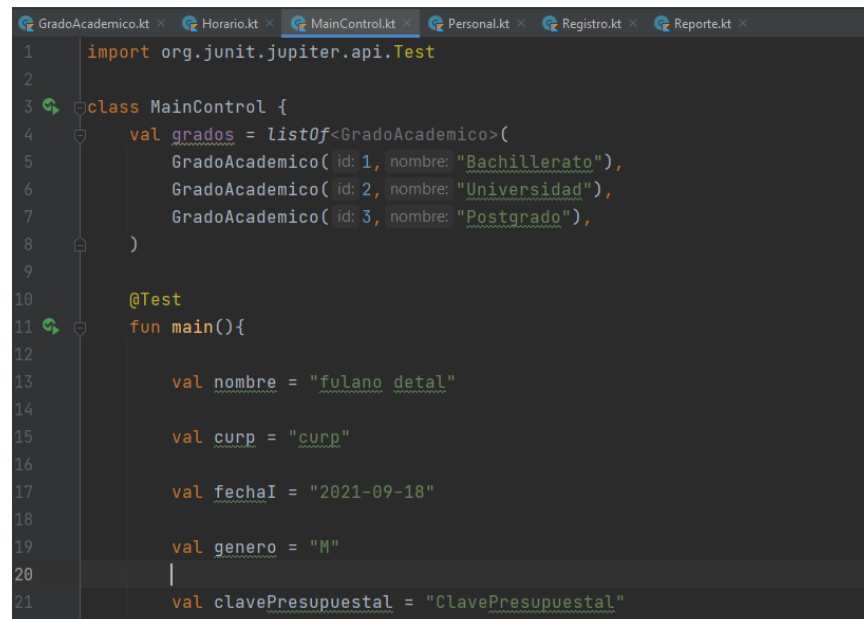
Paso 4: Pasamos a crear la clase registro, la cual tendrá los siguientes parámetros, los cuales serán las bases para poder hacer los registros y generar el reporte.



```
1 class Registro(idTrabajador: Int, fecha: String, horaEntrada: String, horaSalida: String) {
2     val idTrabajador = idTrabajador
3     val fecha = fecha
4     val horaEntrada = horaEntrada
5     val horaSalida = horaSalida
6
7 }
```

Figura 1.4, Clase Registro.

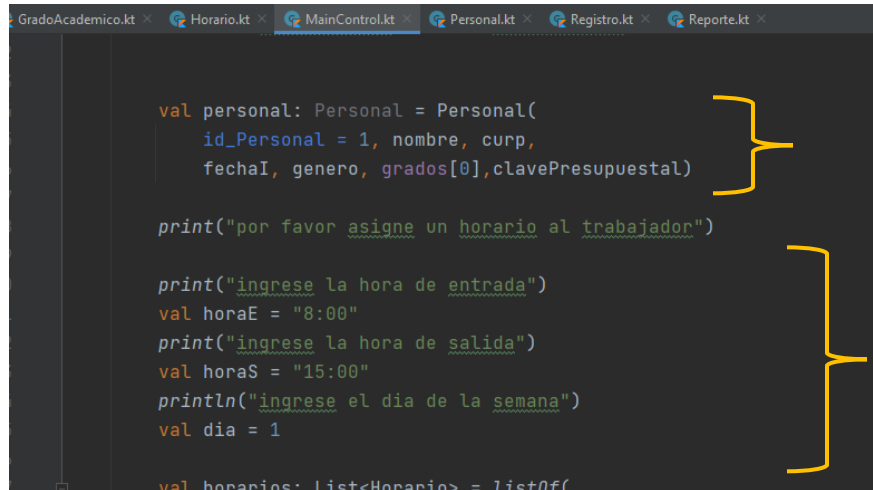
Paso 5: Creamos una clase en la cual construiremos nuestras variables y testaremos nuestro proyecto.



```
1 import org.junit.jupiter.api.Test
2
3 class MainControl {
4     val grados = listOf<GradoAcademico>(
5         GradoAcademico(id: 1, nombre: "Bachillerato"),
6         GradoAcademico(id: 2, nombre: "Universidad"),
7         GradoAcademico(id: 3, nombre: "Postgrado"),
8     )
9
10    @Test
11    fun main() {
12
13        val nombre = "fulano detal"
14
15        val curp = "curp"
16
17        val fechaI = "2021-09-18"
18
19        val genero = "M"
20
21        val clavePresupuestal = "ClavePresupuestal"
22    }
```

Figura 1.5, Clase MainControl.

Paso 7: Ahora contruímos la variable peronsal, donde asignamos el id, el nombre, fecha inicial, genero, grados y la clave presupuestal. Como podemos observar comenzamos a crear el horario, asignando una hora de entrada, hora de salida y el día de la semana, con base a esto se harán los registros.



```
val personal: Personal = Personal(
    id_Personal = 1, nombre, curp,
    fechaI, genero, grados[0], clavePresupuestal)

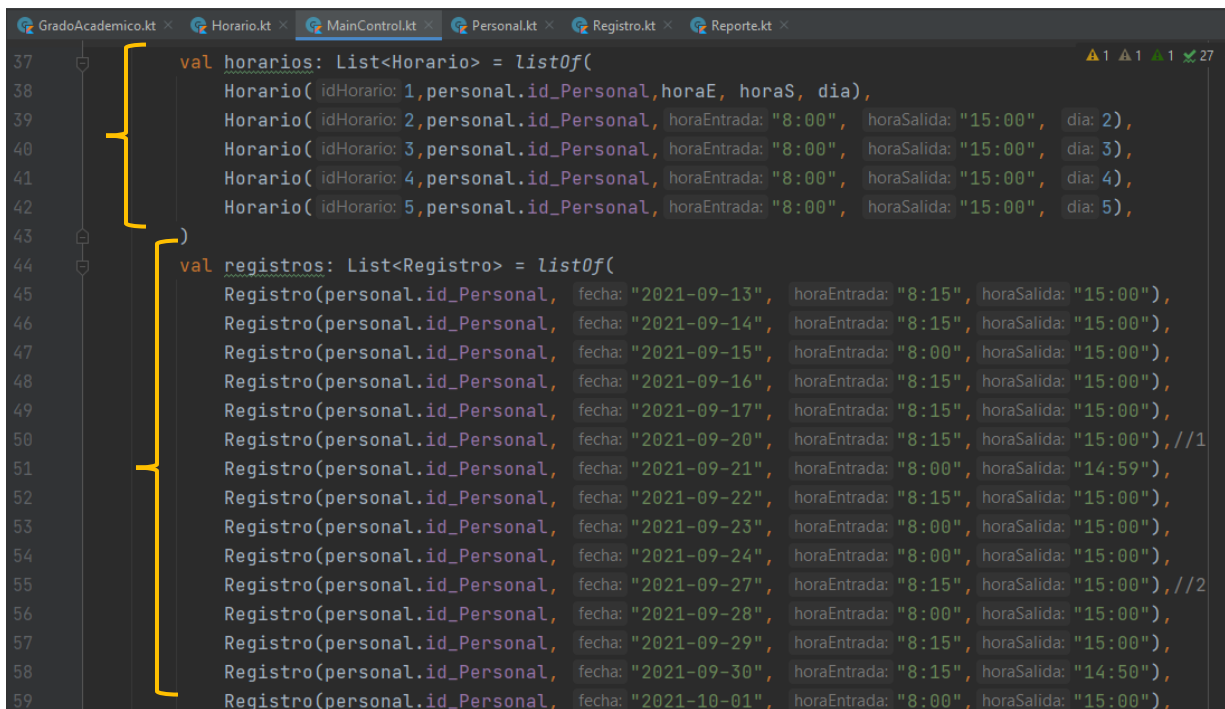
print("por favor asigne un horario al trabajador")

print("ingrese la hora de entrada")
val horaE = "8:00"
print("ingrese la hora de salida")
val horaS = "15:00"
println("ingrese el día de la semana")
val dia = 1

val horarios: List<Horario> = listOf(
```

Figura 1.6, Clase MainControl-persona.

Paso 8: Posteriormente, para los registros, crearemos un arreglo de registros tanto para horarios y registros de entrada los cuales serán 15, para abarcar la quincena. Asignando id del personal, hora entrada, hora salida y por supuesto la fecha.

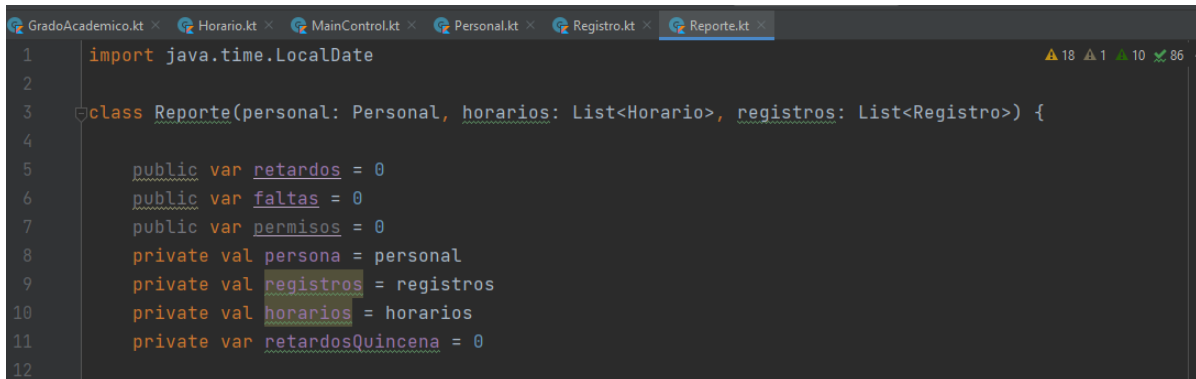


```
val horarios: List<Horario> = listOf(
    Horario(idHorario: 1, personal.id_Personal, horaE, horaS, dia),
    Horario(idHorario: 2, personal.id_Personal, horaEntrada: "8:00", horaSalida: "15:00", dia: 2),
    Horario(idHorario: 3, personal.id_Personal, horaEntrada: "8:00", horaSalida: "15:00", dia: 3),
    Horario(idHorario: 4, personal.id_Personal, horaEntrada: "8:00", horaSalida: "15:00", dia: 4),
    Horario(idHorario: 5, personal.id_Personal, horaEntrada: "8:00", horaSalida: "15:00", dia: 5),
)

val registros: List<Registro> = listOf(
    Registro(personal.id_Personal, fecha: "2021-09-13", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-14", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-15", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-16", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-17", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-20", horaEntrada: "8:15", horaSalida: "15:00"), //1
    Registro(personal.id_Personal, fecha: "2021-09-21", horaEntrada: "8:00", horaSalida: "14:59"),
    Registro(personal.id_Personal, fecha: "2021-09-22", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-23", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-24", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-27", horaEntrada: "8:15", horaSalida: "15:00"), //2
    Registro(personal.id_Personal, fecha: "2021-09-28", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-29", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-30", horaEntrada: "8:15", horaSalida: "14:50"),
    Registro(personal.id_Personal, fecha: "2021-10-01", horaEntrada: "8:00", horaSalida: "15:00"),
)
```

Figura 1.7, Clase MainControl-Listas.

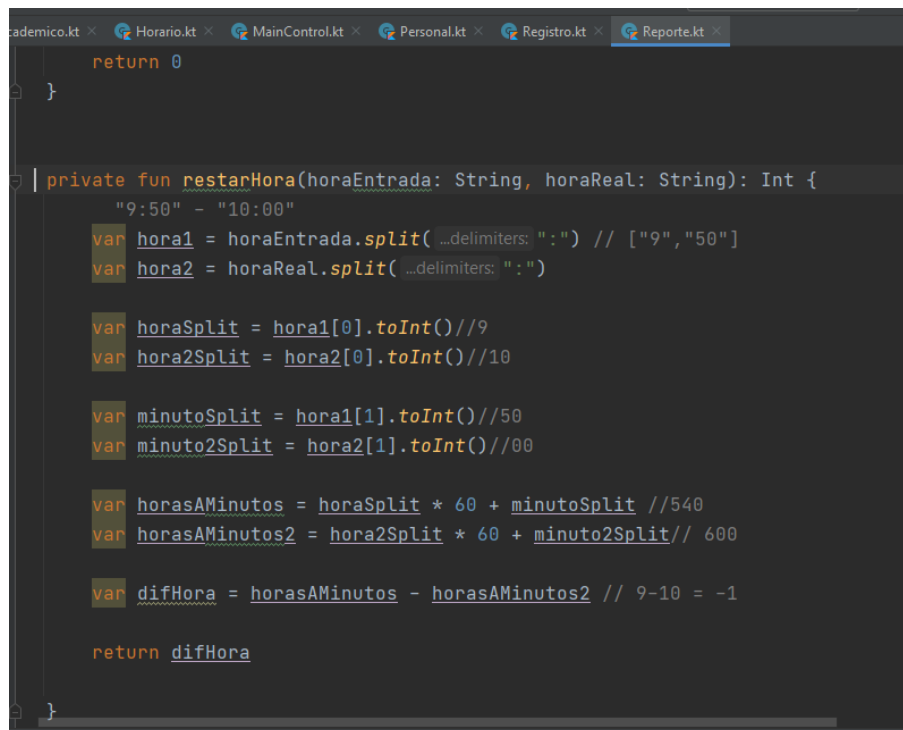
Paso 9: La clase reporte sera la encargada de manejar todas las operaciones, para esto ocupamos las variables que veremos en la siguiente imagen, así tambien como parametros como personal, la lista de horarios y lista de registros.



```
1 import java.time.LocalDate
2
3 class Reporte(personal: Personal, horarios: List<Horario>, registros: List<Registro>) {
4
5     public var retardos = 0
6     public var faltas = 0
7     public var permisos = 0
8     private val persona = personal
9     private val registros = registros
10    private val horarios = horarios
11    private var retardosQuincena = 0
12 }
```

Figura 1.8, Clase Reporte.

Paso 10: Para poder calcular si hay faltas o retados, debemos saber la diferencia de tiempo, para esto, creamos un metodo en el cual nuestra hora en String la separemos con un split y la convertimos a Int para así poder realizar operaciones, sin mencionar que convertimos el timepo total en minutos. Para esto necesitamos la hora Entrada marcada en el horario y la hora del personal.



```
return 0
}

private fun restarHora(horaEntrada: String, horaReal: String): Int {
    "9:50" - "10:00"
    var hora1 = horaEntrada.split( ...delimiters: ":" ) // ["9","50"]
    var hora2 = horaReal.split( ...delimiters: ":" )

    var horaSplit = hora1[0].toInt()//9
    var hora2Split = hora2[0].toInt()//10

    var minutoSplit = hora1[1].toInt()//50
    var minuto2Split = hora2[1].toInt()//00

    var horasAMinutos = horaSplit * 60 + minutoSplit //540
    var horasAMinutos2 = hora2Split * 60 + minuto2Split// 600

    var difHora = horasAMinutos - horasAMinutos2 // 9-10 = -1

    return difHora
}
```

Figura 1.9, Clase Personal – restarHora

Paso 11: Para calcular el retardo necesitaremos los parametros que podemos observar a continuación, junto con ello utilizaremos la antigüedad previamente calculada, y restarHora que de igual manera ya esta creada, en donde si la diferencia de horario es mayor a 10 y menos de 20 minutos, habrá retardo y si la antigüedad es menor a 10 años y los retardos de la quincena al dividirlos entre 3 da un residuo de 0, entonces hay una falta.

```
fun calcularRetardo(horaE: String, horaIH: String): Int {  
    val antigüedad = persona.antigüedad  
  
    val diferenciaHorario = restarHora(horaE, horaIH)  
    if (diferenciaHorario > 10 && diferenciaHorario < 20) {  
        println("retardo")  
        retardos++  
        retardosQuincena++  
        if ((antigüedad < 10) && (retardosQuincena % 3 == 0)) {  
            println("falta")  
            faltas++  
        }  
    }  
  
    return 0  
}
```

Figura 2.1, Clase Personal – calculoRetardo

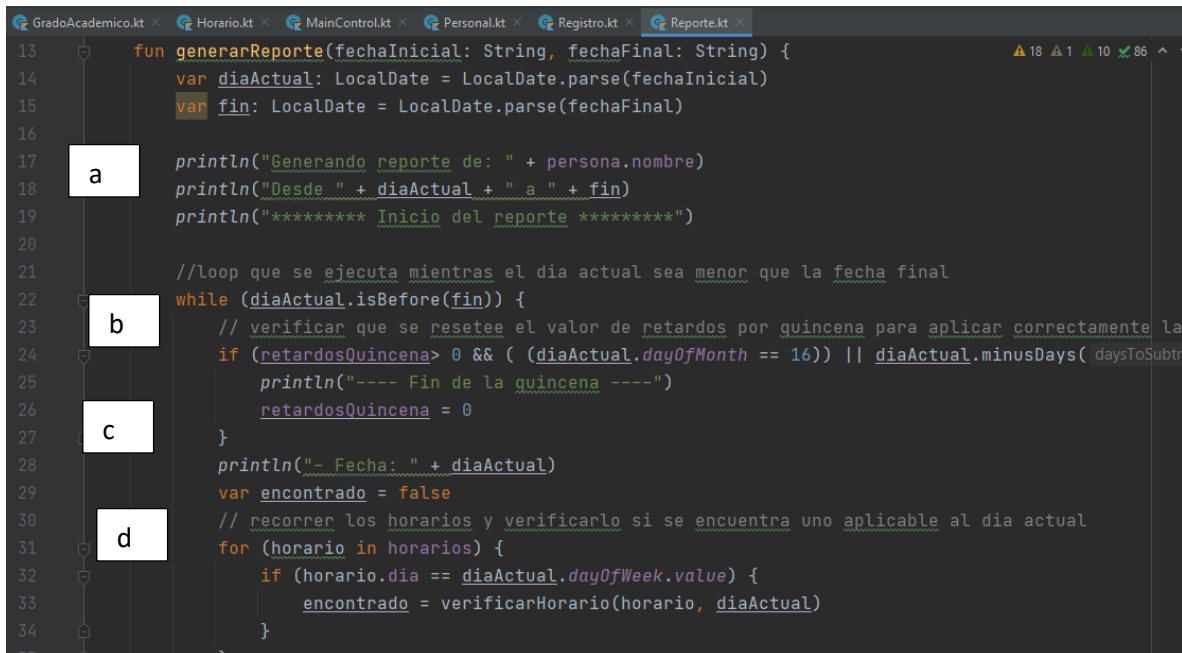
Paso 12 : En el siguiente metodo, verificaremos que el día registrado sea menor que la fecha ifnal, para esto necesitamos el día actual y la fecha fin.

a) Mensaje de inicio reporte.

b) Loop para verificar que el día actual registrado este antes de la fecha final

c) Recetear el valor de los retardos al verificar que los 3 retardos por falta se aplican correctamente

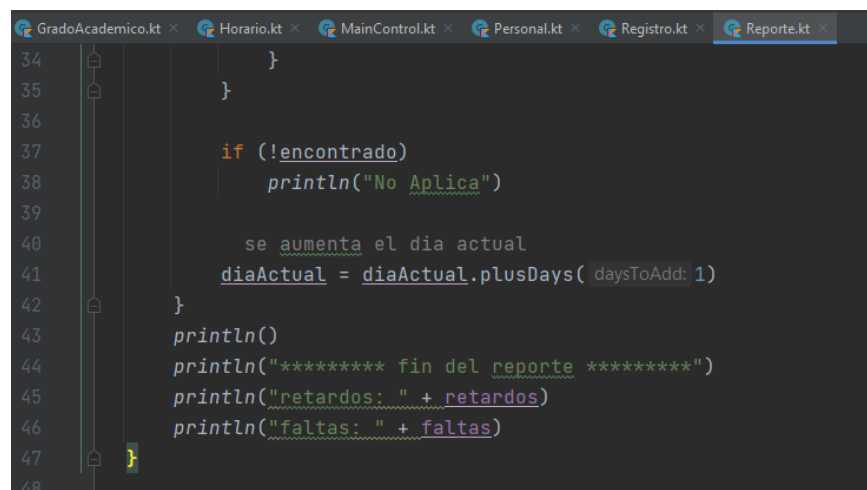
d) Verificar que el registro se encuentre dentro de las fechas registradas en la lista.



```
13 fun generarReporte(fechaInicial: String, fechaFinal: String) {
14     var diaActual: LocalDate = LocalDate.parse(fechaInicial)
15     var fin: LocalDate = LocalDate.parse(fechaFinal)
16
17     a    println("Generando reporte de: " + persona.nombre)
18         println("Desde " + diaActual + " a " + fin)
19         println("***** Inicio del reporte *****")
20
21         //loop que se ejecuta mientras el dia actual sea menor que la fecha final
22         b    while (diaActual.isBefore(fin)) {
23             // verificar que se resetee el valor de retardos por quincena para aplicar correctamente la
24             if (retardosQuincena > 0 && (diaActual.dayOfMonth == 16) || diaActual.minusDays(daysToSubtra
25                 println("---- Fin de la quincena ----")
26                 retardosQuincena = 0
27         }
28         c    println("- Fecha: " + diaActual)
29         var encontrado = false
30         // recorrer los horarios y verificarlo si se encuentra uno aplicable al dia actual
31         d    for (horario in horarios) {
32             if (horario.dia == diaActual.dayOfWeek.value) {
33                 encontrado = verificarHorario(horario, diaActual)
34             }
35         }
36     }
```

Figura 2.2, Clase Personal – generarReporte

e) Si no es encontrada la fecha de registro dentro de las fechas registradas en el horario, no aplicará y simplemente mostrara los retos junto con las faltas.



```
34     }
35 }
36
37 if (!encontrado)
38     println("No Aplica")
39
40     se aumenta el dia actual
41     diaActual = diaActual.plusDays(daysToAdd: 1)
42 }
43 println()
44 println("***** fin del reporte *****")
45 println("retardos: " + retardos)
46 println("faltas: " + faltas)
47 }
48 }
```

Figura 2.3, Clase Personal – generarReporte

Paso 13: Una vez que tengamos los registros y hayamos calculado los retardos y faltas procederemos a imprimir los registros, para esto necesitamos a personal, arreglo de horarios y arreglo de registros.

```
val registroAsistencias = Reporte(personal, horarios, registros)

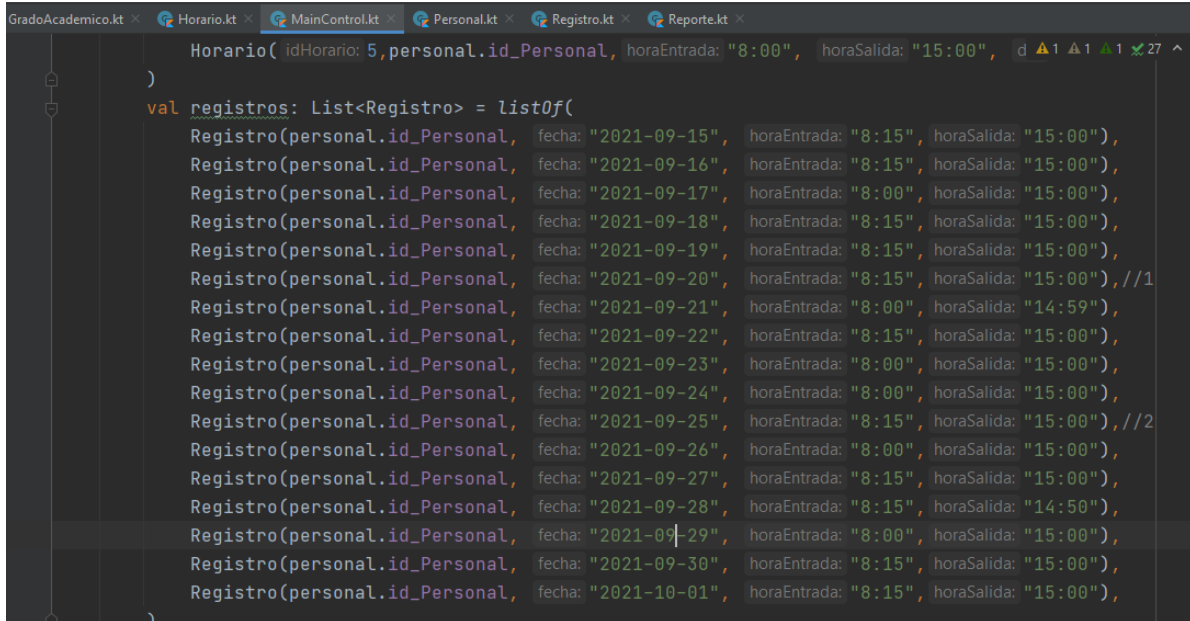
println("Reporte")
registroAsistencias.generarReporte( fechaInicial: "2021-09-16", fechaFinal: "2021-09-30")

}
```

Figura 2.3.1, Clase Personal – generarReporte

Pruebas

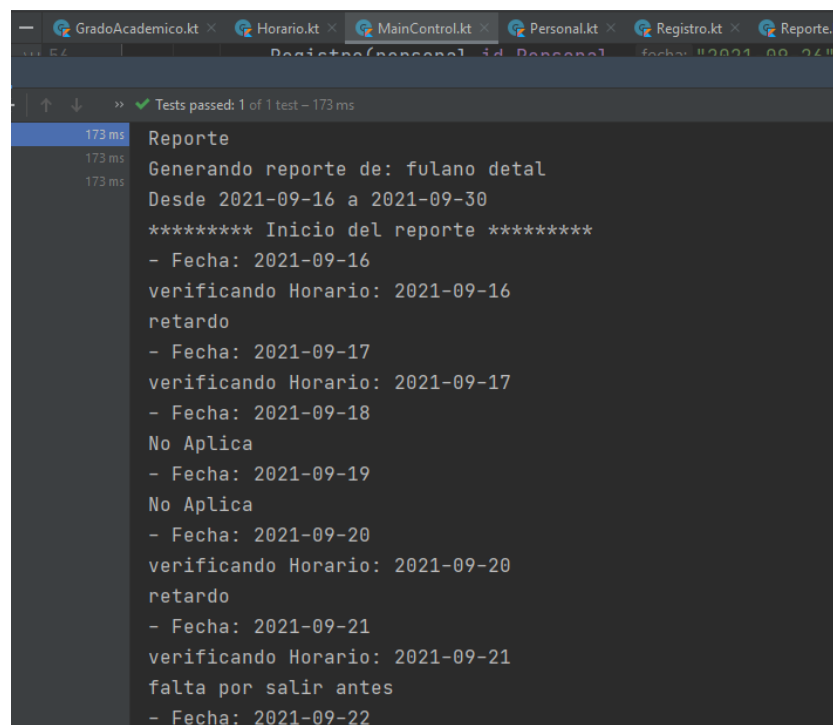
Como podemos observar el periodo va del 15 de septiembre al 01 de octubre, dentro de los registros, se verificará la cantidad de retardos y con base a ello se corroborará su antigüedad para contabilizar las faltas y las faltas directamente.



```
Horario( idHorario: 5, personal.id_Personal, horaEntrada: "8:00", horaSalida: "15:00", d 1 1 1 1 27 ^
)
val registros: List<Registro> = listOf(
    Registro(personal.id_Personal, fecha: "2021-09-15", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-16", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-17", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-18", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-19", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-20", horaEntrada: "8:15", horaSalida: "15:00"), //1
    Registro(personal.id_Personal, fecha: "2021-09-21", horaEntrada: "8:00", horaSalida: "14:59"),
    Registro(personal.id_Personal, fecha: "2021-09-22", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-23", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-24", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-25", horaEntrada: "8:15", horaSalida: "15:00"), //2
    Registro(personal.id_Personal, fecha: "2021-09-26", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-27", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-28", horaEntrada: "8:15", horaSalida: "14:50"),
    Registro(personal.id_Personal, fecha: "2021-09-29", horaEntrada: "8:00", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-09-30", horaEntrada: "8:15", horaSalida: "15:00"),
    Registro(personal.id_Personal, fecha: "2021-10-01", horaEntrada: "8:15", horaSalida: "15:00"),
)
```

Figura 2.4, Prueba

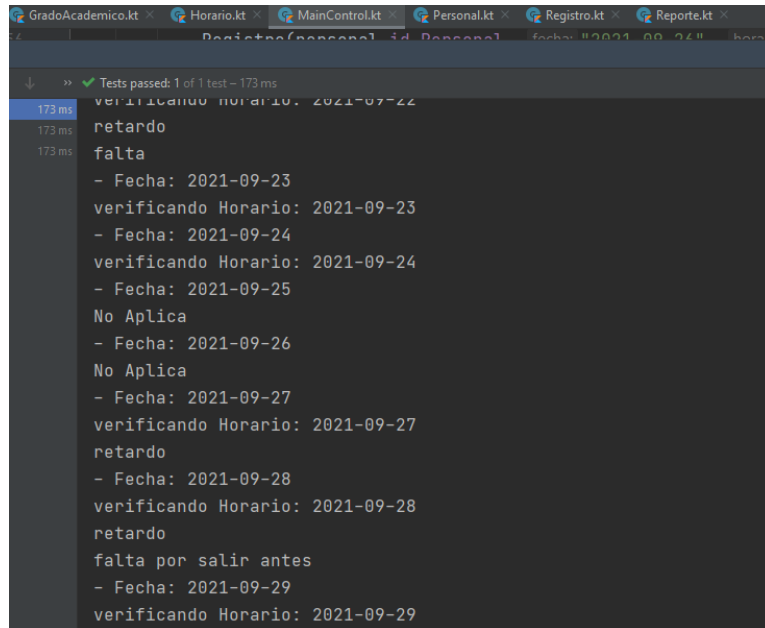
Al correr el programa nos aparece lo siguiente, donde nos indica el inicio del reporte, verifica la fecha y nos muestra si es retardo, falta o no aplica.



```
Tests passed: 1 of 1 test - 173 ms
173 ms Reporte
173 ms Generando reporte de: fulano detal
173 ms Desde 2021-09-16 a 2021-09-30
***** Inicio del reporte *****
- Fecha: 2021-09-16
verificando Horario: 2021-09-16
retardo
- Fecha: 2021-09-17
verificando Horario: 2021-09-17
- Fecha: 2021-09-18
No Aplica
- Fecha: 2021-09-19
No Aplica
- Fecha: 2021-09-20
verificando Horario: 2021-09-20
retardo
- Fecha: 2021-09-21
verificando Horario: 2021-09-21
falta por salir antes
- Fecha: 2021-09-22
```

Figura 2.5.1, Prueba

Así también nos indicará si es falta por salir antes.

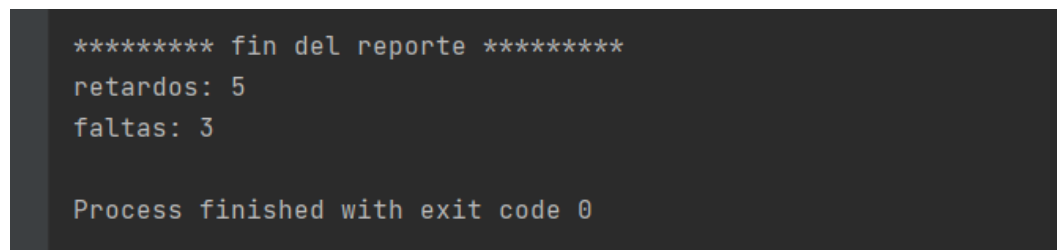


```
GradoAcademico.kt  Horario.kt  MainControl.kt  Personal.kt  Registro.kt  Reporte.kt
Registro(Personal id: Personal, fecha: "2021-09-24", hora: 10)

Tests passed: 1 of 1 test - 173 ms
verificando Horario: 2021-09-23
retardo
falta
- Fecha: 2021-09-23
verificando Horario: 2021-09-23
- Fecha: 2021-09-24
verificando Horario: 2021-09-24
- Fecha: 2021-09-25
No Aplica
- Fecha: 2021-09-26
No Aplica
- Fecha: 2021-09-27
verificando Horario: 2021-09-27
retardo
- Fecha: 2021-09-28
verificando Horario: 2021-09-28
retardo
falta por salir antes
- Fecha: 2021-09-29
verificando Horario: 2021-09-29
```

Figura 2.5.2, Prueba

Nos mostrará el fin del reporte, indicando la cantidad de retardos y faltas de la persona.



```
***** fin del reporte *****
retardos: 5
faltas: 3

Process finished with exit code 0
```

Figura 2.5.3, Resultado

Principios: Dentro del proceso de elaboración pudimos observar que cada una de las clases se encarga de una función específica, dividiendo la solución en varias clases con responsabilidad directa, esto a su vez se puede verificar en los parámetros de cada clase, ya que cada una ocupa lo que necesita sin jalar datos de otro lado. En este caso se cumple el primer método: **responsabilidad única**.

Así también se puede observar que el código está abierto a agregación o extensión del proyecto sin afectar o modificar el código para que esto suceda. Por lo que se cumpliría el principio de **open/close**, cabe aclarar que también todas las clases y funciones se utilizan, todas tienen una funcionalidad.

Evidencia de trabajo

Debido a nuestra falta de experiencia, la problemática se nos complicó en cuanto a la sintaxis de kotlin así como algunos aspectos relacionales como la lógica, por lo que decidimos colaborar con algunos compañeros (Gustavo Romero Monterroza, Lizbeth Verónica y Damián Martínez) retroalimentarnos, cabe recalcar que considero que es una muy buena forma de trabajar y adquirir nuevos conocimientos, ya que hay más opiniones, debates y autonomía para poder dar solución a la problemática.

