



**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.**

## 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 como parámetro de la clase para que esta sea utilizada.

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. Todo esto mostrado en la imagen 1.1

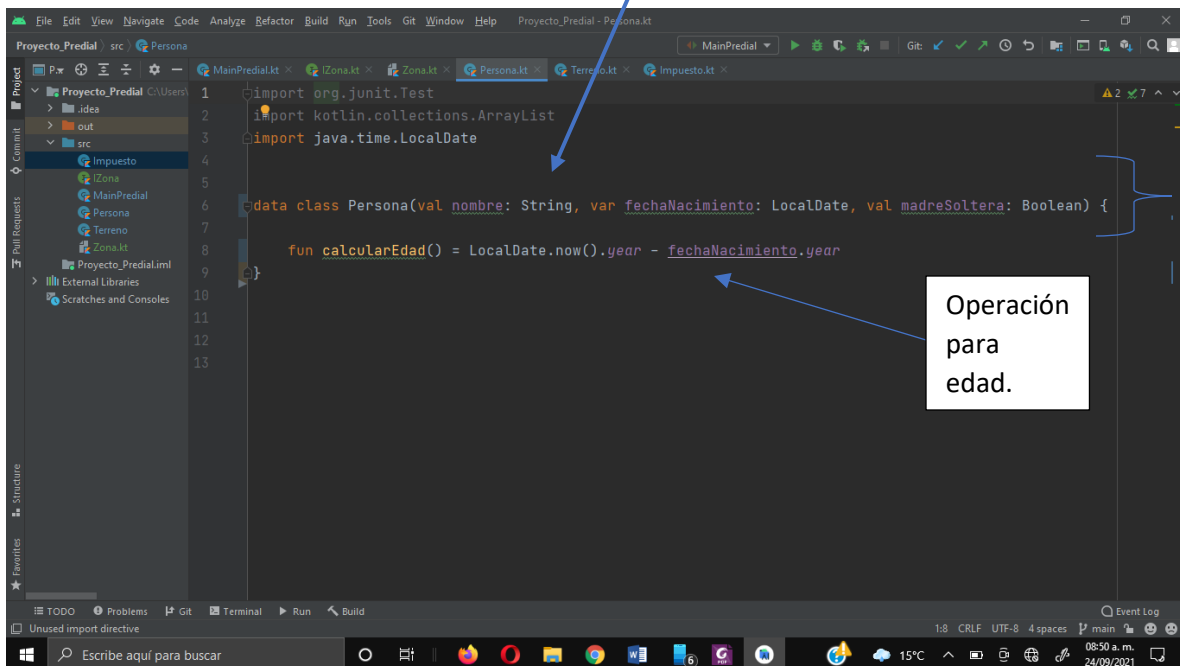


Imagen 1.1. clase Persona

**Paso2:** Una vez acabado con la clase Persona procederemos a continuar con nuestra siguiente entidad detectada, la cual es zona la cual será una interfaz: 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 descripción.

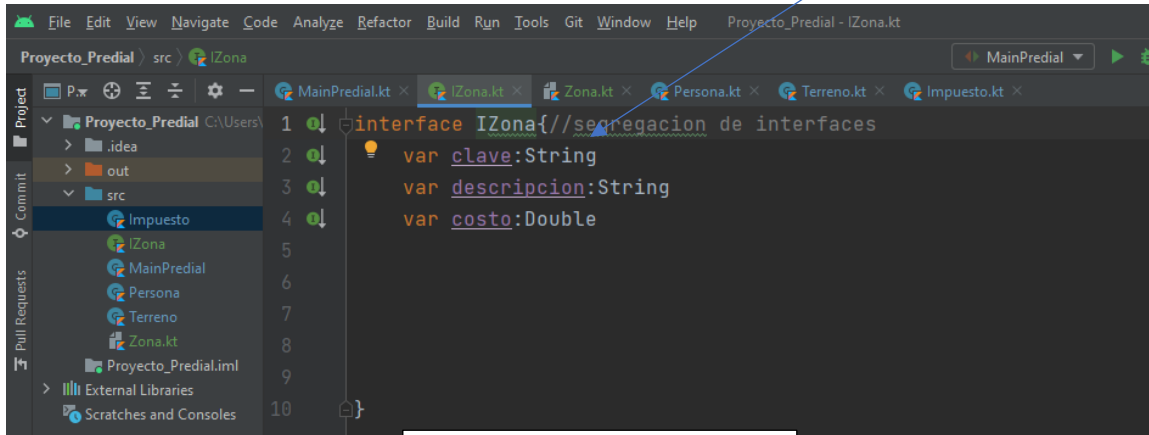


Imagen 1.2. Interfaz Zona

**Paso3:** Como podemos ver creamos una clase por cada zona, especificando sus atributos y junto con ellos segregando la interfaz de “Zona”, con esto cada zona podrá usarse por separado.

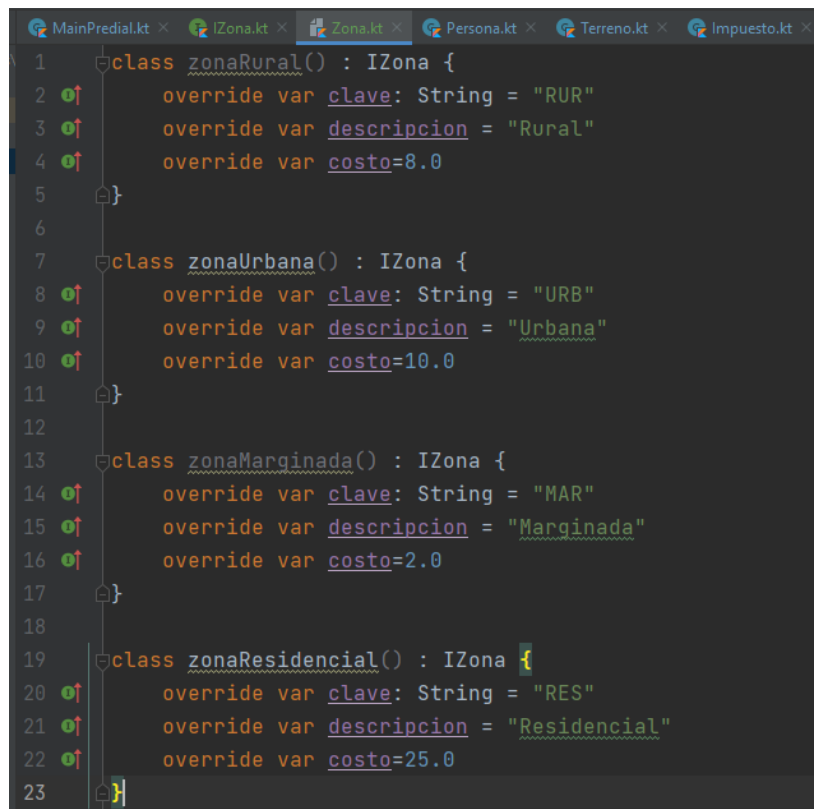


Imagen 1.3. Clase Zona

**Paso4:** Terreno juega un papel importante dentro de la problemática, pues con base a el podremos obtener los datos para calcular el costo del predial a pagar, pues nos otorgará tamaño, por lo que declaramos las siguientes variables, zona la cual hace referencia nuestra interfaz y extensión.

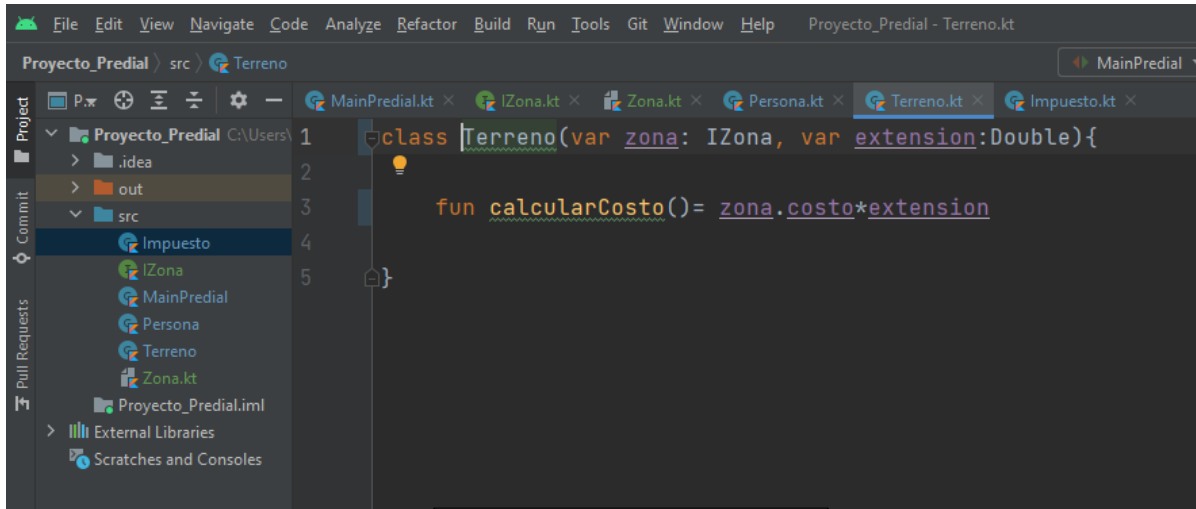
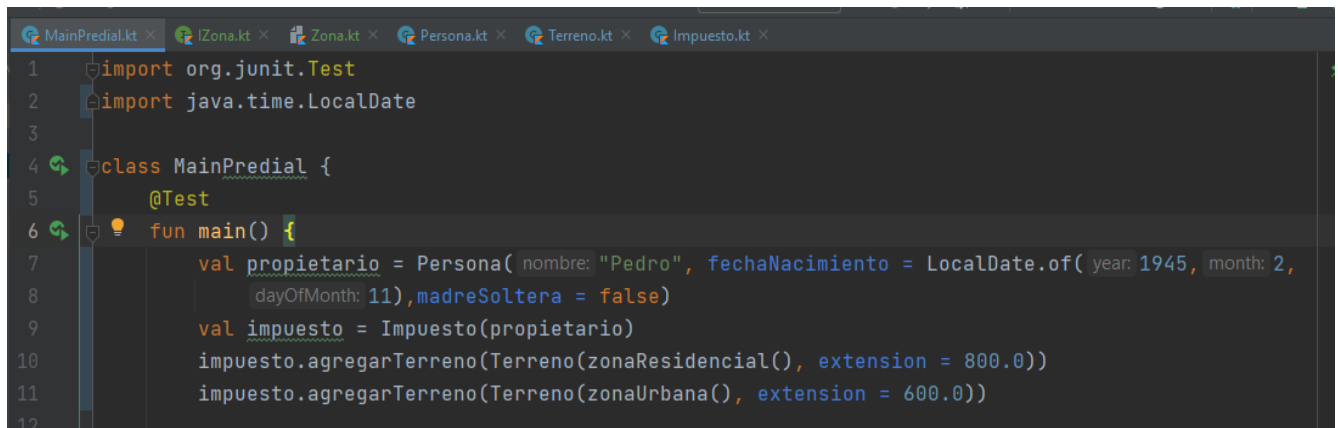
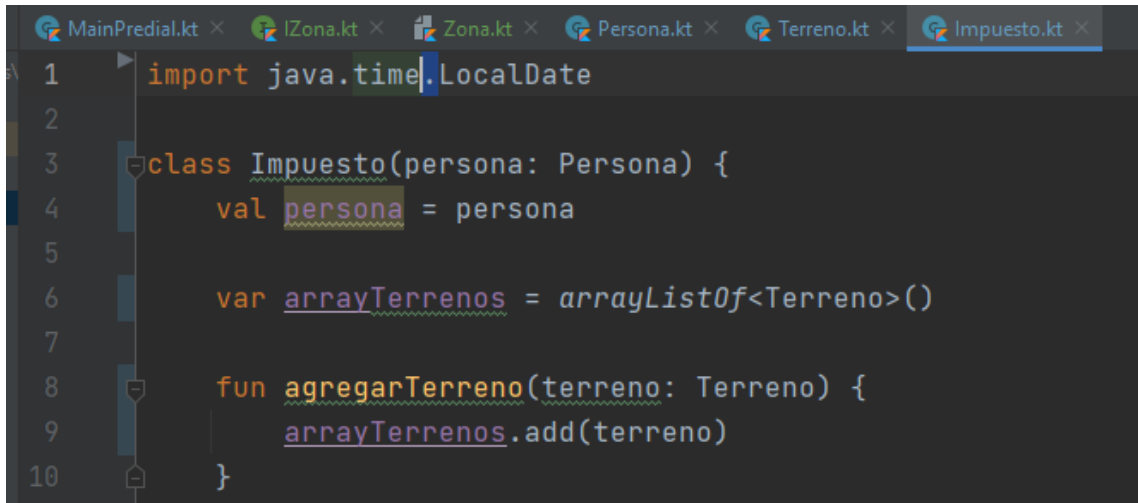


Imagen 1.3. Clase Terreno

**Paso 5:** 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 propietario e impuesto



**Paso 5:** Continuamos con la clase que realiza los cálculos correspondientes, en este caso especificaremos que clases ocuparemos como parámetros, la cual será persona, por lo que no habrá problemas si ocupamos parámetros de esa clase para la entidad Impuesto, posteriormente crearemos un arreglo de terrenos basados en los atributos de la clase Terreno, esto lo utilizaremos en una función “agregar terreno”.

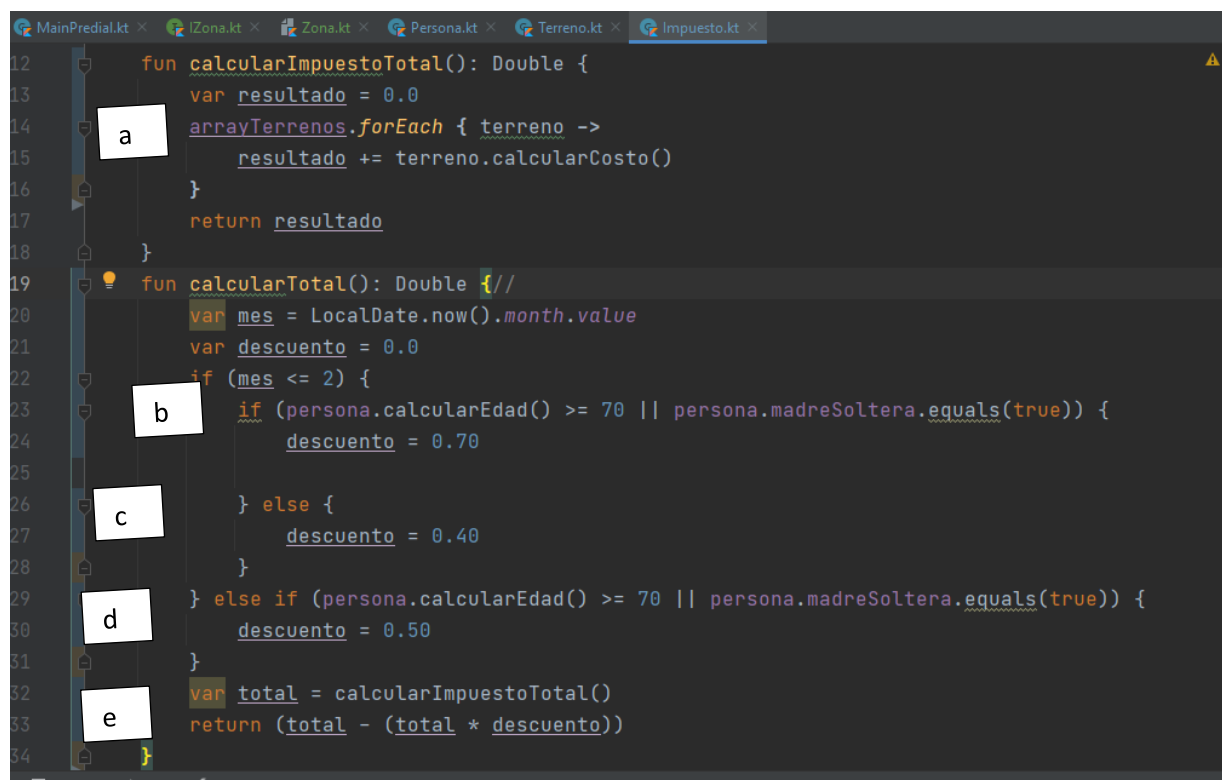
The image shows a screenshot of an IDE with several tabs open at the top: MainPredial.kt, lZona.kt, Zona.kt, Persona.kt, Terreno.kt, and Impuesto.kt. The Impuesto.kt tab is active, displaying the following Kotlin code:

```
1 import java.time.LocalDate
2
3 class Impuesto(persona: Persona) {
4     val persona = persona
5
6     var arrayTerrenos = arrayListOf<Terreno>()
7
8     fun agregarTerreno(terreno: Terreno) {
9         arrayTerrenos.add(terreno)
10    }
```

Imagen 1.5. Clase Predial

**Paso 6:** Dentro de la misma clase crearemos una función para poder calcular el impuesto total Para esto tendremos:

- a) Crear un `forEach` y utilizarlo para recorrer el arreglo de terrenos que previamente habíamos inicializado dentro del proceso a la variable `resultado` se le asignará el impuesto del método `calcularCosto` de cada terreno recorrido y retornará el resultado.
- b) 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%.
- c) 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) En el `else if`, si el mes de pago es mayor a 2 “febrero” tendrá un descuento de 50%.
- e) Asignaremos el método de `calcularImpuestoTotal` a la variable `total` que como recordaremos retorna el resultado del arreglo, a la cual le restaremos el descuento multiplicando el total por el descuento indicado, de esta forma obtendremos el total a pagar.

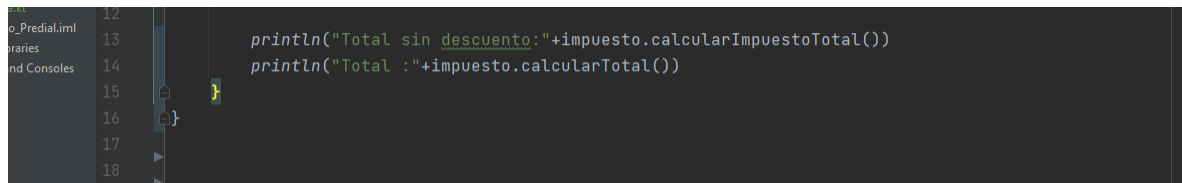


```
12 fun calcularImpuestoTotal(): Double {
13     var resultado = 0.0
14     arrayTerrenos.forEach { terreno ->
15         resultado += terreno.calcularCosto()
16     }
17     return resultado
18 }
19 fun calcularTotal(): Double {
20     var mes = LocalDate.now().month.value
21     var descuento = 0.0
22     if (mes <= 2) {
23         if (persona.calcularEdad() >= 70 || persona.madreSoltera.equals(true)) {
24             descuento = 0.70
25         } else {
26             descuento = 0.40
27         }
28     } else if (persona.calcularEdad() >= 70 || persona.madreSoltera.equals(true)) {
29         descuento = 0.50
30     }
31     var total = calcularImpuestoTotal()
32     return (total - (total * descuento))
33 }
34 }
```

The image shows a screenshot of an IDE with several Kotlin files open. The file `Impuesto.kt` is active, showing the implementation of `calcularImpuestoTotal` and `calcularTotal`. Annotations a-e are placed over the code to correspond with the steps in the text above:

- a** points to the `forEach` loop in `calcularImpuestoTotal`.
- b** points to the first `if` block in `calcularTotal`.
- c** points to the `else` block inside the first `if` in `calcularTotal`.
- d** points to the `else if` block in `calcularTotal`.
- e** points to the final calculation `return (total - (total * descuento))` in `calcularTotal`.

**Paso 8:** Una vez hecho todo esto, solo imprimiremos el impuesto a través de nuestro método `ImpuestoTotal` y el importe total a pagar a través del método `calcularTotal`.



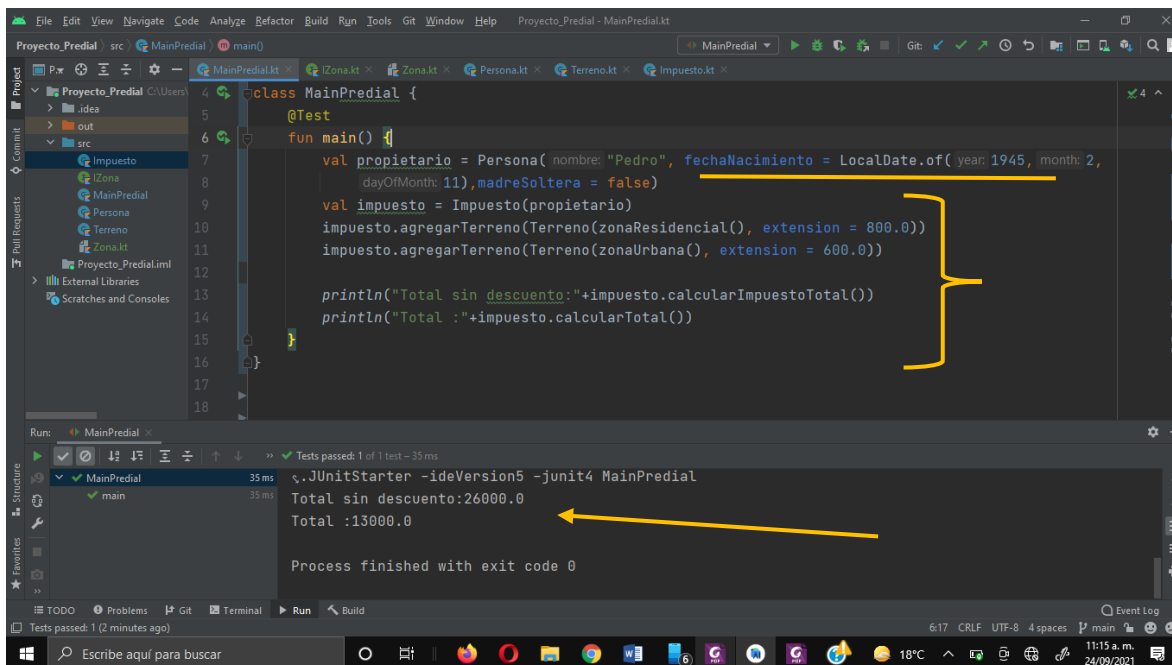
```
12  
13     println("Total sin descuento:"+impuesto.calcularImpuestoTotal())  
14     println("Total :"+impuesto.calcularTotal())  
15  
16  
17  
18
```

Imagen 1.8. Clase Predial- calcularImporteFinal



## PRUEBAS

1. Cumpliendo con la edad “70 años o más”, pero no con el mes de pago, en este caso debe tener un descuento del 50%, lo cual se muestra en la imagen con su descuento e importe final.



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

**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 y en cada caso hubo un data class para no redundar entre parámetros y atributos. 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.

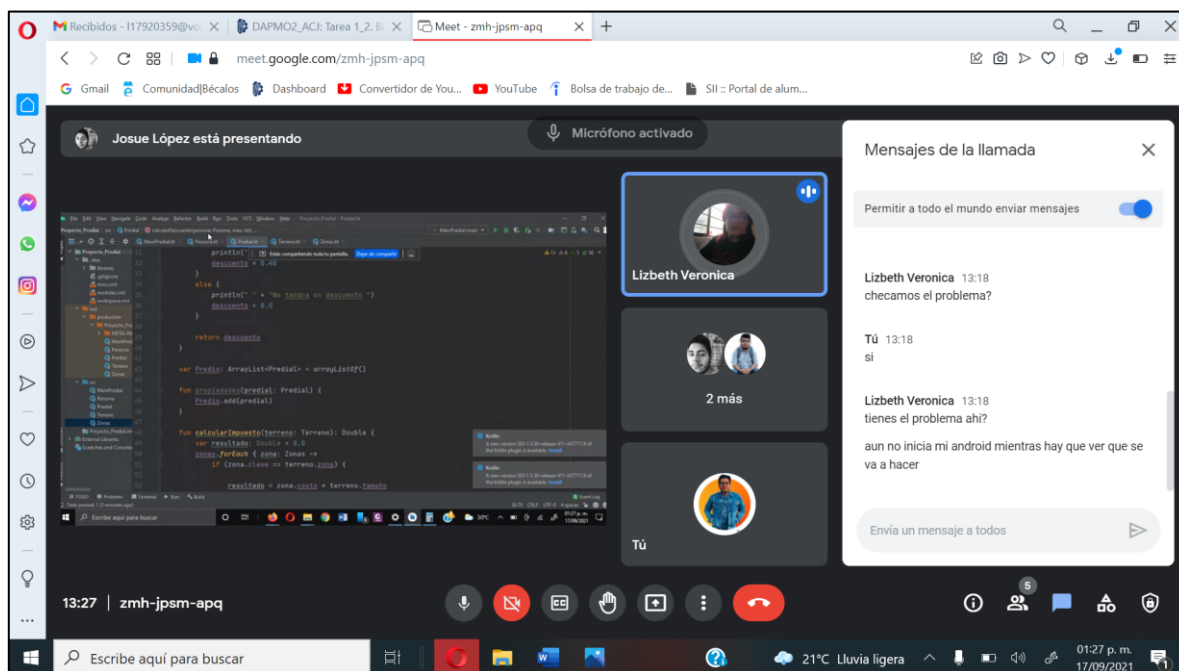
Podemos comprobar que dentro del proceso tuvimos que realizar una interfaz utilizando la **segregación de interfaces** para las clases de las zonas.

Imagen 1.12. Prueba 3

Imagen 1.14. Prueba 5

## Evidencia de trabajo

Debido a nuestra falta de experiencia, la problemática se nos complico 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 mas opiniones, debates y autonomía para poder dar solución a la problemática.

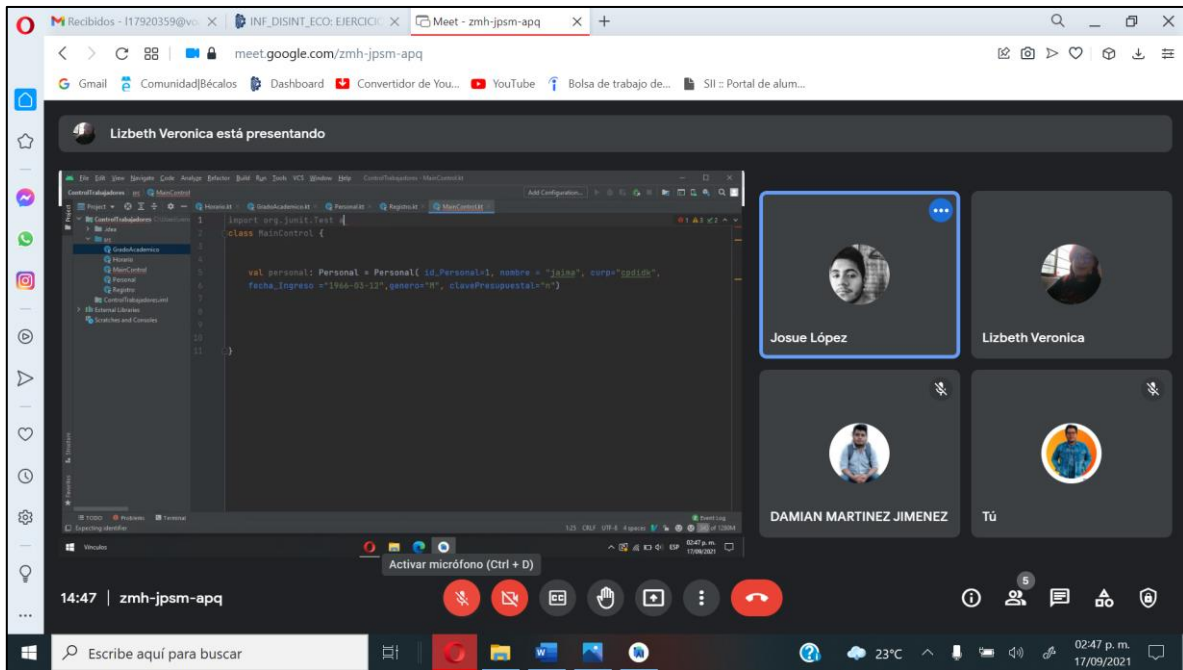


Recibidos - 117920359@v... INF\_DISINT\_ECO: EJERCICIO... Meet - zmh-jpsm-apq

meet.google.com/zmh-jpsm-apq

Gmail ComunidadBécalos Dashboard Convertidor de You... YouTube Bolsa de trabajo de... SII - Portal de alum...

Lizbeth Veronica está presentando



The screenshot shows a Google Meet interface. On the left, a code editor displays Java code for a class named `Personal`. The code includes an `import` statement, a class definition, and a `main` method that creates a `Personal` object. The code is as follows:

```
1 import org.junit.Test;
2 class MainControl {
3
4     val personal: Personal = Personal(10, "Jairo", "corp", "jpd@n",
5     fecha_ingreso = "1966-03-12", genero="M", clavePresupuestal="n")
6
7
8
9
10
11 }
```

On the right, there are four participant avatars: Josue López, Lizbeth Veronica, DAMIAN MARTINEZ JIMENEZ, and Tú. The bottom of the screen shows a Windows taskbar with the time 14:47, the meeting ID zmh-jpsm-apq, and system icons including temperature (23°C) and date (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 Los horarios previamente establecidos debe tener el id del personal, día, hora entrada y hora salida y desde luego
la fecha inicial y final que aplica ese horario (Se asume que solo habrá un horario por día).
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:**

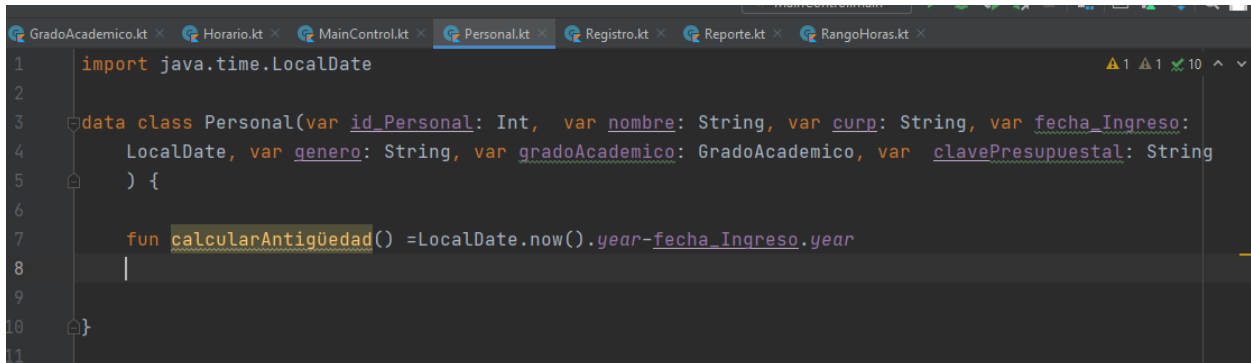
**Android versión: 3.2.24**

**Windows 10 home.**

## 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.

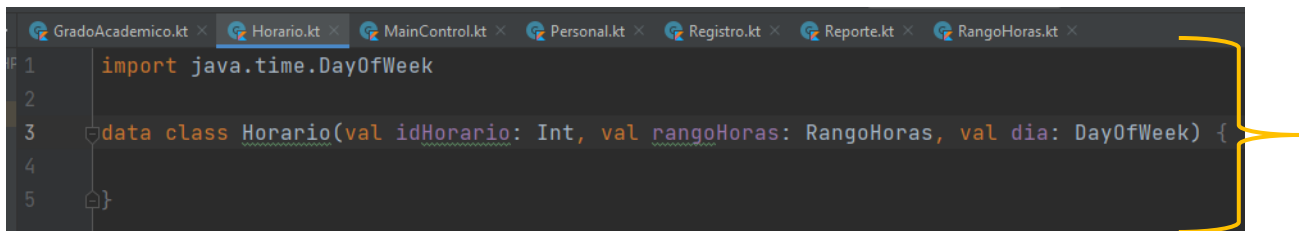
Posteriormente creamos la función calcularAntigüedad el cual pertenece a personal en donde con base a la fecha de ingreso y la fecha actual calcularemos la antigüedad.



```
1 import java.time.LocalDate
2
3 data class Personal(var id_Personal: Int, var nombre: String, var curp: String, var fecha_Ingreso:
4     LocalDate, var genero: String, var gradoAcademico: GradoAcademico, var clavePresupuestal: String
5 ) {
6
7     fun calcularAntigüedad() = LocalDate.now().year - fecha_Ingreso.year
8
9
10 }
11
```

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.

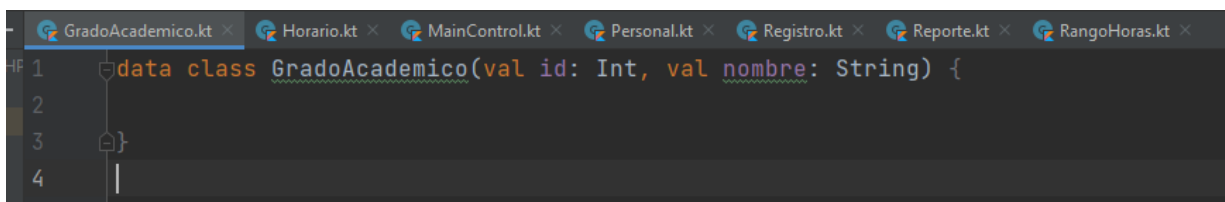


```
1 import java.time.DayOfWeek
2
3 data class Horario(val idHorario: Int, val rangoHoras: RangoHoras, val dia: DayOfWeek) {
4
5 }

```

Figura 1.2, Clase Horario.

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

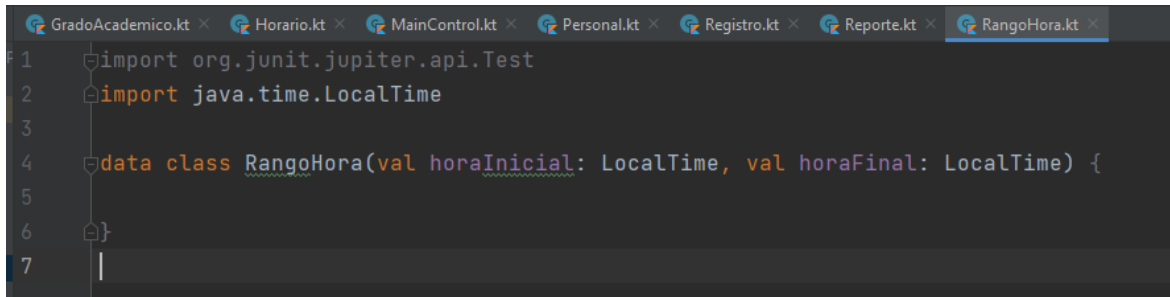


```
1 data class GradoAcademico(val id: Int, val nombre: String) {
2
3 }
4

```

Figura 1.3, Clase GradoAcademica.

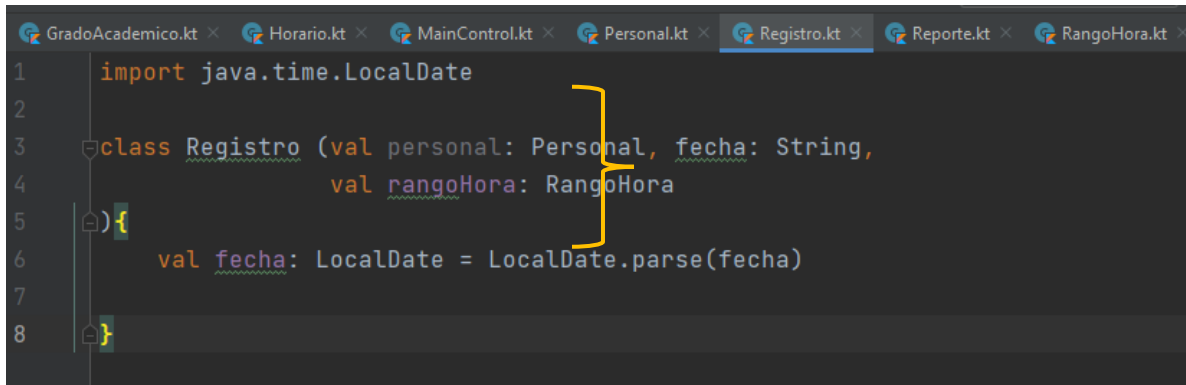
**paso 4:** posteriormente continuamos con la clase “RangoHora”, la cual es vital para establecer y controlar el horario manejando las 2 variables claves.



```
1 import org.junit.jupiter.api.Test
2 import java.time.LocalDateTime
3
4 data class RangoHora(val horaInicial: LocalDateTime, val horaFinal: LocalDateTime) {
5
6 }
7
```

Figura 1.4, Clase RangoHora.

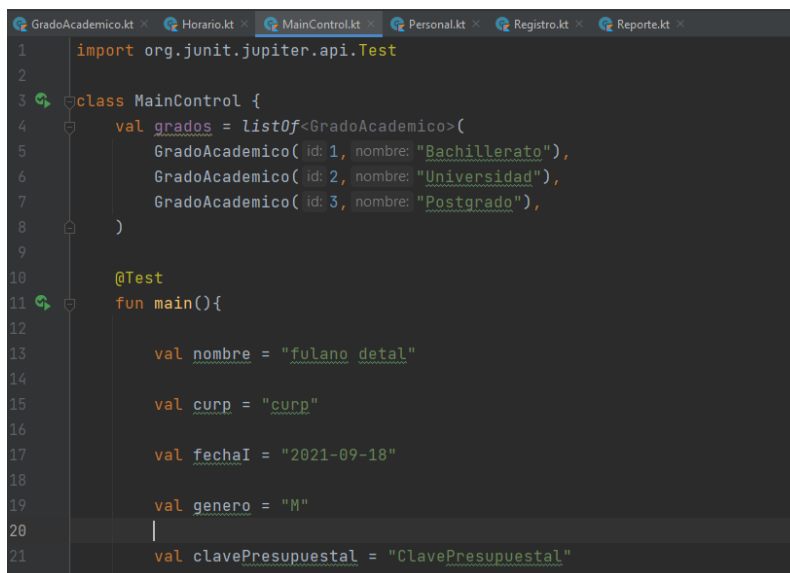
**Paso 5:** Pasamos a crear la clase registro, la cual tendrá los siguientes parámetros, los cuales serán las bases para poder hacer los registros y generar el reporte.



```
1 import java.time.LocalDate
2
3 class Registro (val personal: Personal, fecha: String,
4               val rangoHora: RangoHora
5 ) {
6     val fecha: LocalDate = LocalDate.parse(fecha)
7
8 }
```

Figura 1.5, Clase Registro.

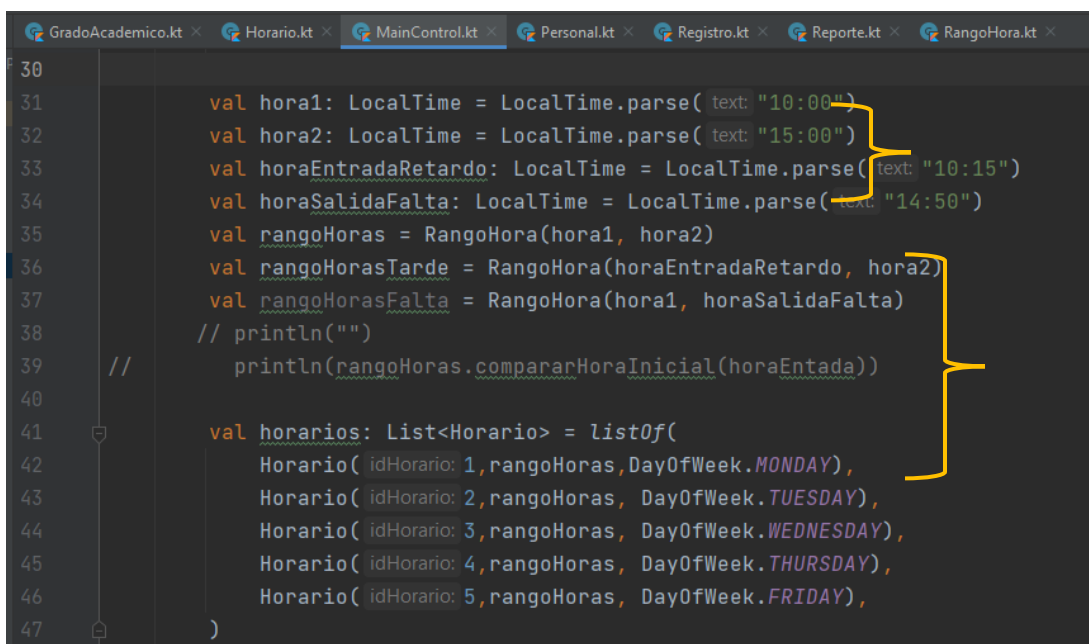
**Paso 6:** Creamos una clase en la cual construiremos nuestras variables y testaremos nuestro proyecto.



```
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"
```

Figura 1.6, Clase MainControl.

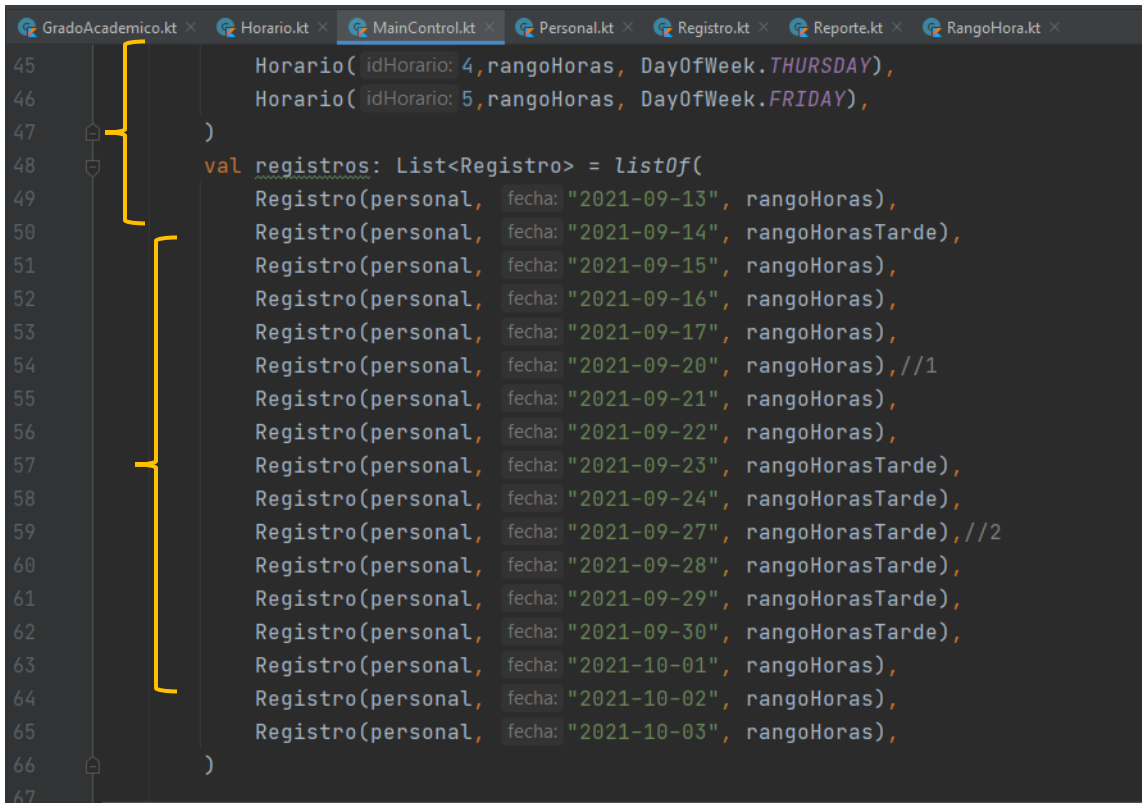
**Paso 7:** Asignaremos valores a cada variable establecida en horario y agregaremos otras para hacer factible la solución, todo basado en el horario del personal. Así también realizaremos un arreglo con el horario del usuario, especificando que horario es y el rango de horas junto con el día correspondiente.



```
30
31     val hora1: LocalTime = LocalTime.parse(text: "10:00")
32     val hora2: LocalTime = LocalTime.parse(text: "15:00")
33     val horaEntradaRetardo: LocalTime = LocalTime.parse(text: "10:15")
34     val horaSalidaFalta: LocalTime = LocalTime.parse(text: "14:50")
35     val rangoHoras = RangoHora(hora1, hora2)
36     val rangoHorasTarde = RangoHora(horaEntradaRetardo, hora2)
37     val rangoHorasFalta = RangoHora(hora1, horaSalidaFalta)
38     // println("")
39     // println(rangoHoras.compararHoraInicial(horaEntada))
40
41     val horarios: List<Horario> = listOf(
42         Horario(idHorario: 1, rangoHoras, DayOfWeek.MONDAY),
43         Horario(idHorario: 2, rangoHoras, DayOfWeek.TUESDAY),
44         Horario(idHorario: 3, rangoHoras, DayOfWeek.WEDNESDAY),
45         Horario(idHorario: 4, rangoHoras, DayOfWeek.THURSDAY),
46         Horario(idHorario: 5, rangoHoras, DayOfWeek.FRIDAY),
47     )
```

Figura 1.6.1, Clase MainControl-horario.

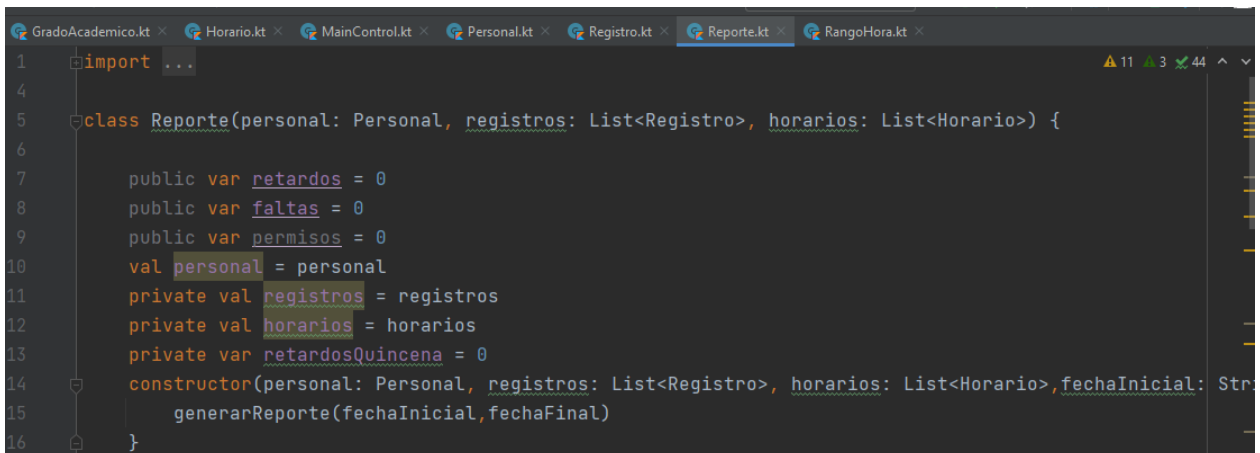
**Paso 8:** Posteriormente, para los registros, crearemos un arreglo de registros en los cuales verificará el personal, la fecha y el rango de horas del horario.



```
45     Horario( idHorario: 4,rangoHoras, DayOfWeek.THURSDAY),
46     Horario( idHorario: 5,rangoHoras, DayOfWeek.FRIDAY),
47 )
48 val registros: List<Registro> = listOf(
49     Registro(personal, fecha: "2021-09-13", rangoHoras),
50     Registro(personal, fecha: "2021-09-14", rangoHorasTarde),
51     Registro(personal, fecha: "2021-09-15", rangoHoras),
52     Registro(personal, fecha: "2021-09-16", rangoHoras),
53     Registro(personal, fecha: "2021-09-17", rangoHoras),
54     Registro(personal, fecha: "2021-09-20", rangoHoras), //1
55     Registro(personal, fecha: "2021-09-21", rangoHoras),
56     Registro(personal, fecha: "2021-09-22", rangoHoras),
57     Registro(personal, fecha: "2021-09-23", rangoHorasTarde),
58     Registro(personal, fecha: "2021-09-24", rangoHorasTarde),
59     Registro(personal, fecha: "2021-09-27", rangoHorasTarde), //2
60     Registro(personal, fecha: "2021-09-28", rangoHorasTarde),
61     Registro(personal, fecha: "2021-09-29", rangoHorasTarde),
62     Registro(personal, fecha: "2021-09-30", rangoHorasTarde),
63     Registro(personal, fecha: "2021-10-01", rangoHoras),
64     Registro(personal, fecha: "2021-10-02", rangoHoras),
65     Registro(personal, fecha: "2021-10-03", rangoHoras),
66 )
67
```

Figura 1.6.2, Clase MainControl-ListaRegistro

**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 ...
4
5 class Reporte(personal: Personal, registros: List<Registro>, horarios: List<Horario>) {
6
7     public var retardos = 0
8     public var faltas = 0
9     public var permisos = 0
10    val personal = personal
11    private val registros = registros
12    private val horarios = horarios
13    private var retardosQuincena = 0
14    constructor(personal: Personal, registros: List<Registro>, horarios: List<Horario>, fechaInicial: Str
15        generarReporte(fechaInicial, fechaFinal)
16 }
```

Figura 1.7, Clase Reporte.



**Paso 10:** En este paso la función estará encargada de calcular el retardo todo con base al horario y sus variables pues de cumplir la condición los retardos aumentan y se retorna true, en caso contrario false.

```
40 fun calcularRetardo(diaActual: LocalDate, horario: Horario): Boolean {
41     val registro = obtenerRegistro(diaActual)
42     if (registro != null) {
43         var llegada = registro.rangoHora.horaInicial.minus(
44             horario.rangoHora.horaInicial.getLong(ChronoField.MINUTE_OF_DAY),
45             ChronoUnit.MINUTES)
46         if (llegada.minute in 5..15){
47             retardos++
48             retardosQuincena++
49             return true
50         }
51     }
52     return false
53 }
54 }
```

Figura 1.9, Clase Reporte – calculoRetardo

**Paso 11 :** Para calcular la falta, ocuparemos el siguiente método, donde si el registro es nulo se suma una falta, también se evalúa con base a la antigüedad del usuario y el registro de su hora entrada y salida.

```
GradoAcademico.kt x Horario.kt x MainControl.kt x Personal.kt x Registro.kt x Reporte.kt x RangoHora.kt x
54 }
55
56 fun calcularFalta(diaActual: LocalDate, horario: Horario, retardo: Boolean): Boolean {
57
58     val registro = obtenerRegistro(diaActual)
59     if (registro == null) {
60
61         faltas++
62         return true
63     }
64     if (personal.calcularAntigüedad() > 10)
65         return false
66     if (registro.rangoHora.horaFinal.isBefore(horario.rangoHora.horaFinal) || retardo && retardosQui
67         faltas++
68         return true
69     }
70     return false
71 }
72
73 private fun obtenerRegistro(diaActual: LocalDate): Registro? {
74     return registros.find { registro -> registro.fecha == diaActual }
75 }
76 }
```

Figura 2.1, Clase Reporte – calcularFalta

**Paso 12:** Una vez que tengamos los registros y hayamos calculado los retardos y faltas procederemos a obtener los registros, evaluando con que la fecha de registro sea igual a la fecha del día actual, así mismo evaluamos la quincena si el día actual llega a un acumulado de 15, posterior, se reinian los datos de la quincena a 0.

```
private fun obtenerRegistro(diaActual: LocalDate): Registro? {
    return registros.find { registro -> registro.fecha == diaActual }
}

private fun esQuincena(diaActual: LocalDate): Boolean{
    return diaActual.dayOfMonth == 15 || diaActual.minusDays( daysToSubtract: 1).month > diaActual.month
}

private fun reiniciarRetardosPorQuincena() {
    retardosQuincena = 0
}
```

Figura 2.2, Clase Reporte – obtenerRegistros

**Paso 13:** Ahora generaremos el reporte con base la fecha inicial, final, fecha actual y claro el mismo horario, basándonos con lo obtenido en los métodos anteriores.

```
fun generarReporte(fechaInicial: String, fechaFinal: String): Reporte {
    var fechaActual: LocalDate = LocalDate.parse(fechaInicial)
    var fin: LocalDate = LocalDate.parse(fechaFinal)

    //loop que se ejecuta mientras el dia actual sea menor que la fecha final
    while (!fechaActual.isAfter(fin)) {
        var diaActual = fechaActual.dayOfWeek
        if (esQuincena(fechaActual))
            reiniciarRetardosPorQuincena()
        horarios.forEach { horario ->
            if (horario.dia == diaActual ){
                var retardo = calcularRetardo( fechaActual, horario)
                calcularFalta(fechaActual, horario, retardo)
            }
        }
        fechaActual = fechaActual.plusDays( daysToAdd: 1)
    }
    return this
}
```

Figura 2.3, Clase Reporte – generarReporte

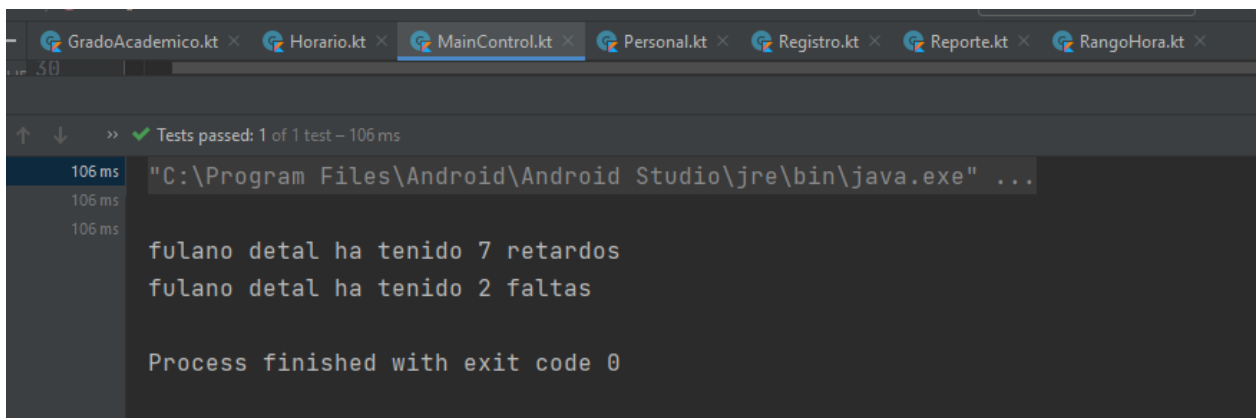
## Pruebas

Ahora procederemos a correr la aplicación, de modo que calcule el total de retardos y faltas en el periodo que marca la quincena.

```
val registroAsistencias = Reporte(personal,registros,horarios, fechaInicial: "2021-09-13", fechaFinal: "2021-10-03")
println(registroAsistencias.personal.nombre + " ha tenido " + registroAsistencias.retardos + " retardos")
println(registroAsistencias.personal.nombre + " ha tenido " + registroAsistencias.faltas + " faltas")
```

Figura 2.4, Imprimir

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



```
↑ ↓ >> ✓ Tests passed: 1 of 1 test - 106 ms
106 ms "C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
106 ms
106 ms
fulano detal ha tenido 7 retardos
fulano detal ha tenido 2 faltas

Process finished with exit code 0
```

Figura 2.5, 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**.

## 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.

