Trabajo Práctico Grupal 2021

Taller de Programacion 1

Universidad Nacional de Mar del Plata

Facultad de Ingeniería

Grupo 7

Alumnos

- Andrade, Tobias Ezequiel
- Chalop, Franco
- Dell Aguila Ureña, Franco
- Di Leo Santos, Juan Federico

Presentación del Informe

Alumnos:

- Andrade, Tobias Ezequiel - tobiaseltoti5@gmail.com
- Chalop, Franco - chalopfranco@gmail.com
- Dell Aguila Ureña, Franco - francodellaguila@gmail.com
- Di Leo Santos, Juan Federico- juanfedileo@gmail.com

Repositorio Github:
https://github.com/FrancoDellAguila/TallerDeProgramacion1

Temas tratados

- Manejo de proyectos en Java y Maven.
- Serialización y Persistencia binaria de archivos.
- Creación de Interfaces Gráficas usando el patrón de MVC.
- Test de Caja Negra o Test Unitarios
- Test de Caja Blanca.
- Test de Persistencia.
- Test de GUI para las interfaces gráficas.
- Test de Integración

Introducción

El Trabajo Práctico Grupal 2021 consistió en realizar tareas de codificar y testear en un proyecto de Java. En primera instancia se recibió un proyecto de clínica hecho durante la cursada de Programación 3 en el primer cuatrimestre de 2021. A partir del mismo, documentar y codificar 3 diferentes módulos de persistencia de los pacientes, el personal médico y el sistema de facturación y diferentes interfaces gráficas para el manejo de estos tres módulos de persistencia siguiendo el patrón de MVC.
Una vez que se terminó esta etapa de codificación, se intercambió el proyecto con otro proyecto realizado por nuestros compañeros. Y con ese proyecto se comenzó a realizar los diferentes tipos de Test.

Desarrollo

Modificaciones al proyecto dado por la catedra:
En base al proyecto dado por la cátedra se modificó la estructura de colecciones y clases donde se guardaban los datos de pacientes y médicos para poder hacer posible la serialización y persistencia de los datos. Se codificaron y agregaron dichos módulos de persistencia binaria de pacientes, médicos y facturas. Y se les armó una interfaz gráfica con los requisitos pedidos para el manejo de ingreso y egreso de datos. Se implementó todo lo anterior en base al modelo de MVC.

Modificaciones al proyecto dado por nuestros compañeros:
En base al proyecto dado por nuestros compañeros no se lo modificó de manera significativa. Se agregaron ciertos getters y setter para hacer posible las pruebas de caja negra y blanca, y en especial para el desarrollo de los test de GUI.

Caja Negra:
Para los Test de Caja Negra realizamos un documento Excel al cual se puede acceder desde aca(gdrive) o aca(gitHub).

Caja Blanca:
Para el Test de Caja Blanca se tomó un método que agregaba una nueva funcionalidad el cual consiste en el cálculo de un importe adicional a la facturación.
Este método resuelve el cálculo de un importe adicional que se le cobraría a ciertos pacientes.
Se partió en base a un pseudocódigo dado por la cátedra y en base al mismo se codeó en lenguaje Java. Con el mismo luego se construyó un grafo ciclomatico y se describieron los diferentes caminos. Así mismo se establecieron los escenarios y los casos de prueba

- El codigo del metodo es el siguiente:

```
/**
    * Devuelve el importe adicional que se le va a cobrar al paciente
    * <b> Pre: </b> numeroDeFactura > 0
    * fechaDeSolicitud != null y valido<br>
        * <b> Post: </b> Devuelve el importe adicional a partir de la existencia de la factura, rango etario y la lista de insumos.<br>
    * <br>
* @param numeroDeFactura: numero de factura de la que se va a calcular el importe.
* @param fechaDeSolicitud: fecha de solicitud.
* @param listaDeInsumos: lista de double con los costes de los insumos utilizados.
    * */
```

```
public double calculoImporteAdicionales(int numeroDeFactura, Calendar fechaDeSolicitud, ArrayList<Double> listaDeInsumos)
   {
        double importeTotal = 0, importeParcial = 0;
        double A = 0.7 /*Menor a 1*/, B = 0.5/*Menor a 1 y menor a A*/, C = 1.2 /*Entre 1 y 2*/, D = 0.2 /*Menor a 1 y mayor a A*/;
        Factura factura = null;

        Iterator<Factura> it;
        Iterator<Double> itDouble;

        it = facturas.iterator();
        while (it.hasNext() && factura == null) {
            factura = it.next();
            if (factura.getNroFactura() != numeroDeFactura)
            {
                factura = null;
            }
        }
        if (factura != null)
        {
            if (ChronoUnit.DAYS.between(fechaDeSolicitud.toInstant(), factura.getFecha().toInstant()) < 10)
            {
                importeParcial = factura.getTotal() - (factura.calcularSubTotalImpar() * A);
            }
            else
```

```
                {
                        importeParcial = factura.getTotal() * B;
                }

                if (factura.getPaciente().esMayor())
                {
                        importeTotal = importeParcial * C;
                }
                else
                {
                        importeTotal = importeParcial * D;
                }
                  if (listaDeInsumos != null && Math.random() * 30 + 1 !=
factura.getFecha().get(Calendar.DAY_OF_MONTH))
                {
                        itDouble = listaDeInsumos.iterator();
                        while (itDouble.hasNext()) {
                                importeTotal += itDouble.next();
                        }
                }
        }
        return importeTotal;
   }
```

- El cálculo de la complejidad es:

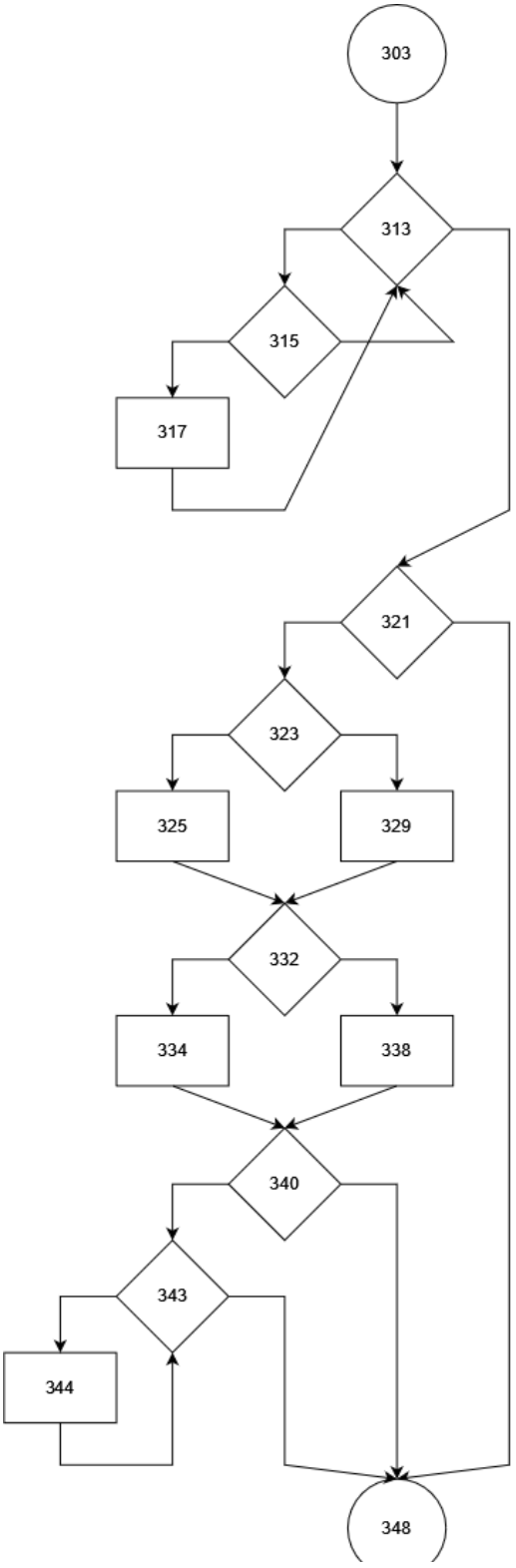
Nodos Totales: 15
Nodos Condición: 7
Arcos: 21
Áreas Cerradas: 7

Complejidad:
Nodos Condición + 1 = 8
Áreas Cerradas +1 = 8
Arcos - Nodos + 2 = 8

Caminos:
-C1=(303) - (313) - (321) - (348)
-C2=(303) - (313) - (315) - (317) - (313) - (321) - (348)

-C3=(303) - (313) - (315) - (313) - (321) - (323) - (325) - (332) - (334) - (340) - (348)
-C4=(303) - (313) - (315) - (313) - (321) - (323) - (329) - (332) - (338) - (340) - (343) - (344) - (343) - (348)

-K5=(303) - (313) - (315) - (313) - (321) - (323) - (329) - (332) - (334) - (340) - (343) - (344) - (343) - (348)
-K6=(303) - (313) - (315) - (313) - (321) - (323) - (325) - (332) - (338) - (340) - (343) - (344) - (343) - (348)
K7=(303) - (313) - (315) - (313) - (321) - (323) - (325) - (332) - (338) - (340) - (348)
K8=(303) - (313) - (315) - (313) - (321) - (323) - (329) - (332) - (334) - (340) - (348)



Casos de prueba:

Escenarios:

Escenario 1:

Facturas == null

Escenario 2:

Facturas != null y ListaDeInsumos == null

Escenario 3:

Facturas != null y ListaDeInsumos != null

Camino	Escenario	Caso	Salida Esperada
C1	Escenario 1		importeTotal = 0
C2	Escenario 2	factura!=null, pero NumerodeFactura no existe en Facturas	importeTotal = 0
C3	Escenario 2	Diferencia de fechas < 10, RangoEtario == Mayor, listaDeInsumos== null	importeTotal = (totalFactura - (subtotalImpar*A))*C
C4	Escenario 3	Diferencia de fechas > 10, RangoEtario != Mayor, listaDeInsumos != null, ALEATORIO != FechadeFactura	importeTotal = totalFactura*B *D + Insumos

- Calculamos la cobertura del código con el plugin EclEMMA el cual nos devolvió los siguientes resultados:

(Nos encontramos con el inconveniente de no poder realizar mock con Math random ya que este último es estático y por eso el cálculo de la cobertura dio un número menor al que esperamos ya que no entró a partes de código que tendría que haber entrado)

```java
public double calculoImporteAdicionales(int numeroDeFactura, Calendar fechaDeSolicitud, ArrayList<Double> listaDeInsumos)
{
    double importeTotal = 0, importeParcial = 0;
    double A = 0.7 /*Menor a 1*/, B = 0.5/*Menor a 1 y menor a A*/, C = 1.2 /*Entre 1 y 2*/, D = 0.2 /*Menor a 1 y mayor a A*/;
    Factura factura = null;

    Iterator<Factura> it;
    Iterator<Double> itDouble;

    it = facturas.iterator();
    while (it.hasNext() && factura == null) {
        factura = it.next();
        if (factura.getNroFactura() != numeroDeFactura)
        {
            factura = null;
        }
    }

    if (factura != null)
    {
        if (ChronoUnit.DAYS.between(fechaDeSolicitud.toInstant(), factura.getFecha().toInstant()) < 10)
        {
            importeParcial = factura.getTotal() - (factura.calcularSubTotalImpar() * A);
        }
        else
        {
            importeParcial = factura.getTotal() * B;
        }

        if (factura.getPaciente().esMayor())
        {
            importeTotal = importeParcial * C;
        }
        else
        {
            importeTotal = importeParcial * D;
        }
        if (listaDeInsumos != null && Math.random() * 30 + 1 != factura.getFecha().get(Calendar.DAY_OF_MONTH))
        {
            itDouble = listaDeInsumos.iterator();
            while (itDouble.hasNext()) {
                importeTotal += itDouble.next();
            }
        }
    }
```

Problems | Javadoc | Declaration | Console | Debug | Coverage

Código_Clinica (26 nov. 2021 23:35:05)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TestUnidadCalculoImporteAdicionales.j	69,9 %	190	82	272
TestUnidadCalculoImporteAdicionale	69,9 %	190	82	272
testHayFacturasYExisteElNroEnFa	12,8 %	12	82	94
setUp()	100,0 %	11	0	11
tearDown()	100,0 %	11	0	11
testHayFacturasNoExisteElNroEnF	100,0 %	40	0	40
testHayFacturasYExisteElNroEnFa	100,0 %	84	0	84
testNoHayFactura()	100,0 %	29	0	29

```java
public void testHayFacturasNoExisteElNroEnFacturas() {
    Factura f1= mock(Factura.class);
    when(f1.getNroFactura()).thenReturn(10);
    this.facturas.add(f1);
    Clinica.getInstance().setFacturas(facturas);
    Calendar f= new GregorianCalendar(2021,11,11);
    double rta= Clinica.getInstance().calculoImporteAdicionales(0, f, listaDeInsumos);
    assertEquals(rta, 0, 0);
}

@Test
public void testHayFacturasYExisteElNroEnFacturasYListaDeInsumosNull() {
    Paciente p = mock(Paciente.class);
    when(p.esMayor()).thenReturn(true);
    Factura f1= mock(Factura.class);
    when(f1.getNroFactura()).thenReturn(10);
    when(f1.getTotal()).thenReturn(100.0);
    when(f1.calcularSubTotalImpar()).thenReturn(50.0);
    when(f1.getFecha()).thenReturn(new GregorianCalendar(2021,11,10));
    when(f1.getPaciente()).thenReturn(p);
    this.facturas.add(f1);
    Clinica.getInstance().setFacturas(facturas);
    Calendar f= new GregorianCalendar(2021,11,11);
    double rta= Clinica.getInstance().calculoImporteAdicionales(10, f, null);
    assertEquals(rta, 78, 0);//importeTotal= (100-50*0,7)*1,2=78
}

@Test //esto implica que insumos sea distinto de null
public void testHayFacturasYExisteElNroEnFacturasYRandomDistintoFecha() {
    Paciente p = mock(Paciente.class);
    when(p.esMayor()).thenReturn(false);
    Math m = mock(Math.class);
    when(m.random()).thenReturn(0.2);
    this.listaDeInsumos.add(100.0);
    Factura f1= mock(Factura.class);
    when(f1.getNroFactura()).thenReturn(10);
    when(f1.getTotal()).thenReturn(100.0);
    when(f1.getFecha()).thenReturn(new GregorianCalendar(2021,11,10));
    when(f1.getPaciente()).thenReturn(p);
    this.facturas.add(f1);
    Clinica.getInstance().setFacturas(facturas);
    Calendar f= new GregorianCalendar(2021,11,11);
    double rta= Clinica.getInstance().calculoImporteAdicionales(10, f, this.listaDeInsumos);
    assertEquals(rta, 110, 0);//importeTotal= (100*0,5*0,2) + 100 = 110
}
```

Test de Persistencia:

Para el Test de Persistencia se eligió el módulo que se encarga de la persistencia de Pacientes.
La realización del test se llevó a cabo a partir de la estructura con la que se enseñó en la cursada, cubriendo los 4 posibles casos ante un posible fallo en la realización de la persistencia.
Los casos cubiertos fueron la creación de un archivo con el nombre con el cual se realiza dentro del código. También se cubrieron los casos en que la persistencia se realiza sin datos o con datos. Y por último se cubrió el caso en el que se intente leer desde un archivo inexistente, verificando que esto no se realiza y arroje la excepción adecuada.
Dichos casos fueron testeados con éxito, y no se encontraron fallas.

Test de GUI:

Para el Test de GUI de las interfaces gráficas se eligió la interfaz de médicos. Se crearon métodos @Test para evaluar todos los casos posibles con los que se llenaron los diferentes TextField y los diferentes botones.

Para realizar estos test nos vimos obligados a usar la clase robot, debido a la complejidad de ciertas características de la interfaz, tales como el ingreso de datos en distintos campos y la selección de los distintos botones radiales (especialidad, contratación, posgrado).

Se implementó una clase llamada TestUtils, la cual fue presentada por la cátedra, con ciertas modificaciones adaptadas para el caso de esta interfaz. Esto nos permitió acceder a varios métodos generales que realizan acciones simples, como obtener el centro de un componente o determinar el botón radial que se desea seleccionar, los cuales facilitan el desarrollo de los test y permiten una mayor legibilidad del código.

Para el caso de obtener la referencia de los objetos de las visuales se intentó implementar la forma que recomienda la cátedra, la cual es setear los nombres para cada componente y luego obtener el objeto a través de una función que los lee. Tal desarrollo se vió imposibilitado a causa de la implementación de los ButtonGroup en la ventana, los cuales carecen de los métodos setName() y getName() o de un método similar que los permita identificar, por lo cual nos vimos obligados a crear getters para obtener la referencia a los objetos. Estos fueron aplicados a todos los elementos para mantener la consistencia en la obtención de los mismos.

También se creó la clase Mensajes dentro del paquete Util, la cual nos permitió aplicar el tipo Enum para el caso del mensaje de error por matrícula, el único mensaje que se muestra por pantalla con la interfaz de los médicos. Se podría haber aplicado el mismo procedimiento para el caso de todos los textos que aparecen en las ventanas pero no nos pareció oportuno modificar el código a ese nivel.

Para realizar los test, supusimos que los paneles se comportan de forma independiente, y las modificaciones sobre uno no influyen en los demás.
No se tuvo en cuenta el orden en el que los campos fueron llenados.
Tampoco se han considerado todas las posibles combinaciones de ingreso de datos, ya que la ventana tiene un total de 7 campos de texto, por lo tanto el número de combinaciones de cuadros llenos / vacíos sería 2^7 = 128, y eso sin contar las opciones de botones radiales. Por este motivo se tuvieron en cuenta únicamente casos específicos, como que solo esté completo un campo, dos campos, o todos los campos, que tenga una matrícula errónea (no sea un número) o que tenga un dni erróneo (sea un número con menos de 7 cifras).

GuiTestEnabledDisabled

testVacio()		Verifica que estén desactivados Agregar y Eliminar si no hay ningun campo completo
testCompleto()		Verifica que estén desactivados Eliminar y Agregar esté activado si se completan todos los datos correctamente
testCompletoErrorDni()		Verifica que estén desactivados Eliminar y Agregar si se completan todos los datos correctamente, pero el dni de forma incorrecta (menos de 7 caracteres)
testCompletoErrorTelefono()		Verifica que estén desactivados Eliminar y Agregar si se completan todos los datos correctamente, pero el telefono de forma incorrecta (menos de 8 caracteres)
testErrorMatricula()		Verifica que estén desactivados Eliminar y Agregar esté activado si se completan todos los datos correctamente, pero la matrícula de forma incorrecta
testSoloApellido()		Verifican que no estén activados Agregar y Eliminar si hay solo un campo completo
testSoloCirugia()		
testSoloCiudad()		
testSoloClinica()		
testSoloDni()		
testSoloDoctor()		
testSoloDomicilio()		
testSoloMagister()		
testSoloMatricula()		
testSoloNinguno()		
testSoloNombre()		
testSoloPediatria()		
testSoloPermanente()		
testSoloResidente()		
testSoloTelefono()		
testNombreApellido()		Verifican que no estén activados Agregar y Eliminar si hay solo dos campos completos, siendo el nombre uno de ellos
testNombreCiudad()		
testNombreDni()		
testNombreDomicilio()		
testNombreMatricula()		
testNombreTelefono()		

TestGuiConDatos

Método	Descripción
testAgregarCompleto()	Verifica que se agregue un médico cuando todos los datos han sido completados correctamente
testBorraMedico()	Verifica que se elimine un médico cuando se lo selecciona y se pulsa eliminar
testCantidad()	Verifica que la cantidad de médicos al inicio sea correcta
testCompletoErrorDni()	Verifica que no se agregue un médico si no se ingresa un dni correcto (> 7 caracteres)
testCompletoErrorTelefono()	Verifica que no se agregue un médico si no se ingresa un telefono correcto (> 8 caracteres)
testErrorMatricula()	Verifica que si se ingresa una matrícula incorrecta (caracteres) el mensaje de error sea correcto y que no se agregue el médico

TestGuiSinDatos

Método	Descripción
testAgregarCompleto()	Verifica que se agregue un médico cuando todos los datos han sido completados correctamente
testCantidad()	Verifica que la cantidad de médicos al inicio sea correcta
testCompletoErrorDni()	Verifica que no se agregue un médico si no se ingresa un dni correcto (> 7 caracteres)
testCompletoErrorTelefono()	Verifica que no se agregue un médico si no se ingresa un telefono correcto (> 8 caracteres)
testErrorMatricula()	Verifica que si se ingresa una matrícula incorrecta (caracteres) el mensaje de error sea correcto y que no se agregue el médico

En los únicos casos que presentaron fallos los test de GUI fueron para los métodos de testCompletoErrorTelefono en todas las clases. Esto se debe a que nuestros pares no tuvieron en cuenta que no se pueda agregar un médico con un teléfono incorrecto. Por lo tanto, cuando se completan todos los datos con un teléfono erróneo, igualmente se activa el botón de agregar y, haya datos o no, se agrega el médico a la clínica.

Test de Integración:

En el caso de los Test de Integración se buscó hacer una integración ascendente haciendo pruebas incrementales de los principales métodos del proyecto. Como son el caso de las clase Clínica y Factura del paquete Modelo, aunque varias otras clases tienen métodos en los que fueron necesarios testearlos de esta manera.
Nuestro enfoque fue probar primero las clases más básicas y a partir de esas ir avanzando a otras con mayor complejidad.

Para poder realizar estos test decidimos usar el framework Mockito que nos permitió usar mocks, objetos que imitan el comportamiento de objetos reales pero que podemos manipular fácilmente, para testear en nuestro caso objetos como los médicos, los pacientes o las prestaciones.

Conclusión

Al finalizar el trabajo logramos ver e implementar prácticamente todo lo aprendido durante la cursada de la materia, logrando aplicar todos los diferentes métodos de testing que aprendimos en un proyecto acotado pero con mucha variedad de casos para aplicar los mismos. Se logró implementar módulos de testing que serían fácilmente replicables para el resto del proyecto.