TRABAJO PRÁCTICO FINAL

TALLER DE PROGRAMACIÓN 1

Fecha: **26/11/2021**

Integrantes: **Ezama, María Camila**

Fioriti, Joaquín

García, Mariano Enrique

Pruis, Ramiro

UNIVERSIDAD NACIONAL DE MAR DEL PLATA

Índice

Introducción

El testing es una tarea esencial en el proceso de desarrollo de software. Gracias a esta actividad es posible detectar errores e incertezas durante el mismo, para luego realizar las respectivas correcciones. Es así que puede mantenerse la consistencia de un software, realizando testeos en forma continua para reconocer desaciertos en forma temprana, y no correr el riesgo de, al intentar solucionar uno de estos problemas, generar nuevas fallas que dificultan el desarrollo.

Por otra parte, para realizar pruebas es necesario tener una documentación robusta y estructurada, de modo que se pueda analizar correctamente para tener en cuenta todos los casos posibles que puedan generar problemas en la ejecución. Son parte de esta documentación la del código mismo y la Especificación de Requisitos de Software.

Es necesario tener en cuenta que no siempre se van a cubrir todos los errores con un testeo, pero si se encontrarán gran parte de estos.

En este trabajo, se propone realizar las pruebas relativas a Test Unitario de Caja Negra, Caja Blanca, Test de Persistencia de datos, Test de interfaces Gráficas mediante el uso de un robot, y por último, Test de Integración. Todos estos métodos se realizarán sobre un sistema basado en la Gestión de una Clínica.

Desarrollo

Caja Negra

El Test de Caja Negra es un tipo de prueba cuya principal característica es que no se tiene acceso al código en sí. El principal conocimiento parte de la documentación del programa, en la cual se establecen condiciones sobre las funciones, qué parámetros son aceptados, y cuáles son las salidas esperadas. A partir de estas es que podemos crear pruebas, para verificar que una cierta entrada produce la salida correcta.

La prueba de Caja Negra se basa en el testeo del elemento mínimo del lenguaje utilizado. En nuestro caso, las clases. De esta forma, se propone aislar los errores, de modo que se puedan hallar incongruencias fácilmente.

Para realizar las pruebas sobre el proyecto se procedió a utilizar las dependencias de JUnit, que nos permite generar los casos de prueba, los cuales contienen métodos de prueba. Además que nos da la posibilidad de generar métodos que se utilizan antes y después de cada prueba para, por ejemplo, setear el escenario.

Durante el estudio de la documentación se escogieron los métodos a probar cuya documentación permitía un correcto análisis de las entradas y salidas de los mismos. Es por esto que no fue posible abarcar la totalidad de aquellos para el testeo, ya que en muchos casos la documentación era confusa, o simplemente, no había forma de verificar que se cumplan los respectivos contratos.

Una vez seleccionados los métodos a probar, se procedió a la planificación y creación de los escenarios que servirían para, en cada caso, utilizar la entrada que mejor se adaptara al entorno de la función seleccionada.

Escenarios

Nro de escenario	Descripción
Escenario 1	Clínica Vacía. Es una instancia de la clínica que no posee pacientes, médicos ni facturas anteriores.
Escenario 2	Clínica Cargada. Instancia de la clínica con pacientes cargados al sistema y en lista de atención, así como habitaciones y también médicos.
Escenario 3	Base de Datos de pacientes cargada.
Escenario 4	Factura de un paciente con prestaciones cargadas.
Escenario 5	Factura de un paciente, pero esta vez sin prestaciones cargadas.
Escenario 6	Atributos correctos para la creación de un Médico.
Escenario 7	Atributos incorrectos para la creación de un Médico.

Todos estos fueron utilizados en distintas clases cuyo método “SetUp” delegaba la creación del escenario correspondiente a utilizar por esa clase. Es así que, dependiendo las circunstancias del método a testear, este se realizaba ya sea en una clase con datos previamente ingresados, o una clase sin datos, o ambos, para probar la robustez de la función y su comportamiento.

Se decidió realizar el análisis detallado del siguiente módulo del proyecto testeado:

Sistema de prioridades de la sala de espera de la clínica.

Cuando se ingresa un paciente a la clínica, el mismo puede ser derivado al patio o a la sala de espera. Esto último depende de un sistema de prioridades establecido en base al rango etario del paciente, es decir, si este es un Niño, un Joven, o una persona Mayor:

1) Entre un Niño y un Joven, la Sala de Espera queda para el Niño y el otro se va al Patio.

2) Entre un Joven y un Mayor, la Sala de Espera queda para el Joven y el otro se va al Patio.
3) Entre un Mayor y un Niño, la Sala de Espera queda para el Mayor y el otro se va al Patio

Esta gestión, tiene una importancia fundamental a la hora de establecer un orden en el sistema. Por lo tanto, debemos corroborar que todo funciona correctamente. Este método fue probado en el escenario 1 ya que, al estar vacía la clínica, nos permite controlar los ingresos a la misma, para así probar el correcto funcionamiento del método.

A continuación, se adjuntan las tablas de particiones y la batería de pruebas asociadas al módulo detallado anteriormente:

Método ingresoPaciente(Paciente p).

Tabla de Particiones.

Datos de Prueba y Casos de Prueba.

Escenario 1.

Condición de entrada	Clases Válidas	Clases Inválidas
Paciente que ingresa	El paciente que ingresa es un Nino **1.1**	
	El paciente que ingresa es un Joven **1.2**	
	El paciente que ingresa es un Mayor **1.3**	
Paciente en Sala de Espera	El paciente en sala de Espera es un Nino **2.1**	
	El paciente en sala de Espera es un Joven **2.2**	
	El paciente en sala de Espera es un Mayor **2.3**	
	Paciente en sala de Espera = null **2.4**	

Tipo de clase (correcta o incorrecta)	Valores de entrada	Salida esperada	Clases de prueba cubiertas	Salida Obtenida
Correcta	Paciente ingresa = Mayor Paciente en SaladeEspera=null	Paciente en SaladeEspera = Mayor	1.3,2.4	Paciente en SaladeEspera = Mayor
Correcta	Paciente ingresa = Joven Paciente en SaladeEspera = Mayor	Paciente en SaladeEspera = Joven	1.2,2.3	Paciente en SaladeEspera = Mayor
Correcta	Paciente ingresa = Nino Paciente en SaladeEspera = Joven	Paciente en SaladeEspera = Nino	1.2,2.1	Paciente en SaladeEspera = Joven
Correcta	Paciente ingresa = Nino Paciente en SaladeEspera = Mayor	Paciente en SaladeEspera = Mayor	1.1,2.3	Paciente en SaladeEspera = Nino

Respecto a las tablas anteriores, se podría decir que no hay clases inválidas ya que el contrato del método establece que no se puede ingresar un paciente nulo a la clínica. Sin embargo, puede existir un paciente nulo en la sala de espera, y el mismo será reemplazado por el que ingrese. Esto último es lo que sucede en la primera prueba.

Luego, se procedió a verificar que el sistema de prioridades funcione correctamente. Por esto, en la segunda prueba se ingresa en primer lugar un Mayor a la sala de espera, y luego un Joven. El resultado obtenido no es el esperado. El Joven es quien debería haber quedado en la Sala de Espera, pero, en su lugar, en la misma estaba el paciente Mayor ingresado anteriormente. Como consecuencia, se decidió seguir con las pruebas que deberían provocar rotaciones entre los pacientes, para descartar problemas. Es así que en la tercera prueba se hace ingresar a un Niño en la Sala de Espera ocupada por un Joven. El resultado debería ser que el Niño ingrese en la Sala de Espera, pero esto no es así. Por último, una situación parecida se da en la cuarta prueba.

En resumen, los testeos nos llevan a la conclusión de que hay un problema en la implementación del módulo, ya que, como se mencionó anteriormente, las “rotaciones” están invertidas, pues, a la sala no ingresa quién sí debería, y permanece quien no. Sin embargo, dado que al código lo vemos como una caja negra, no es posible determinar con exactitud dónde se encuentra el fallo que ocasiona este error, ya que para realizar la prueba actuamos sobre la clase clínica, que en realidad es un “intermediario” de la acción de derivar el paciente a la Sala de Espera.

En base a estos resultados, se continuó con el testeo del método *prioridad(Paciente p)* de la clase Paciente. Este es el encargado de retornar la prioridad que tiene el paciente que invoca al método sobre el pasado por parámetro, devolviendo un booleano

Se realizó un test orientado a resultados incorrectos, para comprobar que la falla mencionada anteriormente ocurre gracias a este método. Es por eso que los métodos son de la forma:

- *public void prioridad**Nino**Sobre**Mayor**IncorrectoTest()*.

En este caso, el texto resaltado describe que se va a probar que el paciente Niño tiene prioridad de la Sala de Espera por sobre el paciente Mayor. Esto en realidad es incorrecto, pues debería ocurrir al revés. Es por esto que el test ejecutó sin problemas, evidenciando que hay una falla en la implementación de estas prioridades. Se repitió el procedimiento para las relaciones Nino-Joven y Joven-Mayor y se obtuvieron los resultados esperados, confirmando que el resultado del booleano está invertido para todas las correspondencias. Para poder resolver este problema y lograr que el programa actúe como debe, se pueden modificar los retornos de los respectivos métodos para que el proceso sea exitoso.

Caja Blanca

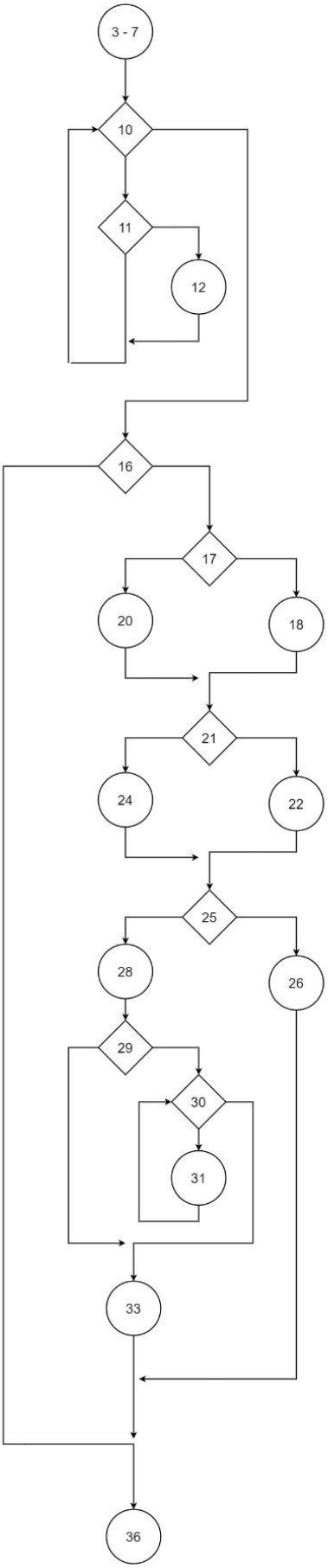
Las pruebas de caja blanca son un proceso en el que uno encuentra un conjunto de datos para pasar por parámetro, con el objetivo de generar una cobertura lo más totalitaria posible en un método dado.

En nuestro caso, el método a testear con caja blanca es *calculoImporteAdicionales*, donde básicamente, como su nombre lo indica, calcula un importe adicional a una factura, dependiendo de unas constantes (A,B,C,D) y de una lista de insumos utilizados durante la actividad sobre la que se factura. Antes de realizar el test de caja blanca, se propuso crear una nueva clase de testing: “CajaBlancaPrevio” en la que se carga la clínica desde un escenario previamente creado, y luego se llama al método a testear, pasando por parámetro una factura, un array de doubles con valores cualesquiera, y la fecha actual. La cobertura de

la misma está en el repositorio. En esta vemos que no se logra el 100% de cobertura en el método deseado(abrir el “index.html” desde un navegador).

Para comenzar el proceso de caja blanca, es necesario analizar el método y, a partir de este, generar un grafo, cuya complejidad ciclomática está dada por los “nodos decisión” indicando la cantidad máxima de caminos necesarios para generar una cobertura total.

A continuación se adjunta el grafo ciclomático:


3 - 7
10
11
12
16
17
20
18
21
24
22
25
28
26
29
30
31
33
36

Para visualizar el código, haga click aqui

En el grafo vemos que tiene 8 nodos decisión, entonces la complejidad ciclomática del algoritmo es de 9. Sin embargo, basta con cuatro caminos para generar la totalidad de la cobertura. Estos son:

- **C1:** (3-7) - (10) - (11) - (12) - (10) - (16) - (17) - (18) - (21) - (22) - (25) - (26) - (36)
- **C2:** (3-7) - (10) - (11) - (12) - (16) - (17) - (20) - (21) - (24) - (25) - (28) - (29) - (30) - (31) - (30) - (33) - (36)
- **C3:** (3-7) - (10) - (11) - (12) - (16) - (17) - (20) - (21) - (24) - (25) - (28) - (29) - (33) - (34)
- **C4:** (3-7) - (10) - (16) - (36)

Luego, una vez encontrados los caminos, propusimos crear otra clase como la que se mencionó al comienzo de esta sección, pero con valores de entrada adecuados para cubrir todos los caminos. Para ello serán necesarios 4 métodos de tipo @Test , en el que cada uno complete un camino dentro del método.

Tomamos dos escenarios:

- Escenario 1 = ArrayList de facturas dentro de la clínica vacío
- Escenario 2 = ArrayList de facturas dentro de la clínica con al menos 2 facturas. Una de ellas tiene el número de factura 1, la fecha 24/11/2021 y el rango etario de su paciente es “mayor”. La otra tiene el número de factura 2, la fecha 6/11/2021 y el rango etario de su paciente es “joven”

  Los casos de prueba nos quedaron de esta forma:

Camino	Escenario	Caso	Salida Esperada
C1	2	numeroDeFactura = 1 fechaDeSolicitud = 26/11/2021 aleatorio = 24 listaDeInsumos = null	respuesta = 962
C2	2	numeroDeFactura = 2 fechaDeSolicitud = 26/11/2021 aleatorio = 3 listaDeInsumos = { 10.0 , 20.0 }	respuesta = 477
C3	2	numeroDeFactura = 2 fechaDeSolicitud = 26/11/2021 aleatorio = 3 listaDeInsumos = null	respuesta = 447
C4	1	numeroDeFactura = 2 fechaDeSolicitud = 26/11/2021 aleatorio = 3 listaDeInsumos = null	respuesta = 0

El caso en el que listaDeInsumos es vacía se ve cubierto por el caso en que la lista es { 10.0 , 20.0 } y retorna el mismo valor que el caso en que la lista es nula.

Un inconveniente que hubo fue que un camino depende de un valor aleatorio dentro del scope local del método, entonces optamos por la utilización de un mock. Como este número se generaba, en un principio, dentro del método, no podíamos utilizar la librería *mockito* para darle un valor desde fuera. Factorizamos el método para modularizar la creación del número aleatorio y poder hacer un mock de este. Entonces cuando el método es llamado, retorna el valor que se le adjudica a propósito, con el fin de lograr el camino indicado.

Luego de aplicar las técnicas de caja blanca, la cobertura del código fue la siguiente (abrir el “index.html” desde un navegador). A grandes rasgos esta fue la mejora:

```
345     public double calculoImporteAdicionales(int numeroDeFactura, GregorianCalendar fechaDeSolicitud, ArrayList<Double> listaDeInsumos) {
346
347         Factura factura = null;
348          double importeParcial = 0;
349          double importeTotal = 0;
350          double respuesta = 0;
351          int aleatorio = Util.createRandom();
352
353          //busco la factura
354          for (Factura facturaact : this.facturas) {
355              if (facturaact.getNroFactura() == numeroDeFactura) {
356                  factura = facturaact;
357              }
358          }
359
360          if (factura != null) {
361              if ((factura.getFecha().get(Calendar.DAY_OF_YEAR) - fechaDeSolicitud.get(Calendar.DAY_OF_YEAR) < 10)) {
362                  importeParcial = factura.getImporteTotal() - (factura.getSubTotalImpar() * 0.7);
363              } else
364                  importeParcial = factura.getImporteTotal() * 0.3;
365              if (factura.getPaciente().getRangoEtario().equalsIgnoreCase("mayor")) {
366                  importeTotal = importeParcial * 1.4;
367              } else
368                  importeTotal = importeParcial * 0.85;
369              if (aleatorio == factura.getFecha().get(Calendar.DAY_OF_MONTH)) {
370                  respuesta = importeTotal;
371              } else {
372                  double sumavalores = 0;
373                  if (listaDeInsumos != null) {
374                        for (Double valor : listaDeInsumos)
375                         sumavalores += valor;
376                  }
377                  respuesta = importeTotal + sumavalores;
378              }
379          }
380          return respuesta;
381      }
```

Cobertura de código antes de aplicar Test Caja Blanca

```
343     public double calculoImporteAdicionales(int numeroDeFactura, GregorianCalendar fechaDeSolicitud, ArrayList<Double> listaDeInsum
344
345          Factura factura = null;
346          double importeParcial = 0;
347          double importeTotal = 0;
348          double respuesta = 0;
349          int aleatorio = Util.createRandom();
350          System.out.println(aleatorio);
351          //busco la factura
352          for (Factura facturaact : this.facturas) {
353              if (facturaact.getNroFactura() == numeroDeFactura) {
354                  factura = facturaact;
355              }
356          }
357
358          if (factura != null) {
359              if (Math.abs(factura.getFecha().get(Calendar.DAY_OF_YEAR) - fechaDeSolicitud.get(Calendar.DAY_OF_YEAR)) < 10) {
360                  importeParcial = factura.getImporteTotal() - (factura.getSubTotalImpar() * 0.7);
361              } else
362                  importeParcial = factura.getImporteTotal() * 0.3;
363              if (factura.getPaciente().getRangoEtario().equalsIgnoreCase("mayor")) {
364                  importeTotal = importeParcial * 1.4;
365              } else
366                  importeTotal = importeParcial * 0.85;
367              if (aleatorio == factura.getFecha().get(Calendar.DAY_OF_MONTH)) {
368                  respuesta = importeTotal;
369              } else {
370                  double sumavalores = 0;
371                  if (listaDeInsumos!=null) {
372                        for (Double valor : listaDeInsumos)
373                         sumavalores += valor;
374                  }
375                  respuesta = importeTotal + sumavalores;
376              }
377          }
378          return respuesta;
379      }
```

Cobertura de código luego de aplicar Test Caja Blanca

Test de Persistencia

En el caso de estudio, la persistencia utilizada es de tipo binaria y se realiza sobre la totalidad de la Clínica, ya que esta es la clase que contiene toda la información dentro del hilo de ejecución del programa.

Al momento de generar la clase "*PersistenciaTest*" se tuvo en cuenta que debía cumplir con ciertos requisitos.

- El sistema debe ser capaz de exportar información a un archivo
- El sistema debe ser capaz de importar la información a partir de un archivo

Además, la persistencia puede arrojar ciertas excepciones con las que también se debe lidiar, como por ejemplo "IOException", que se genera cuando ocurren errores a la hora de creación o lectura de archivos.

Se realizaron tres métodos de tipo @Test, de los cuales dos prueban que la persistencia de la clínica se lleve a cabo correctamente, pero en escenarios diferentes.

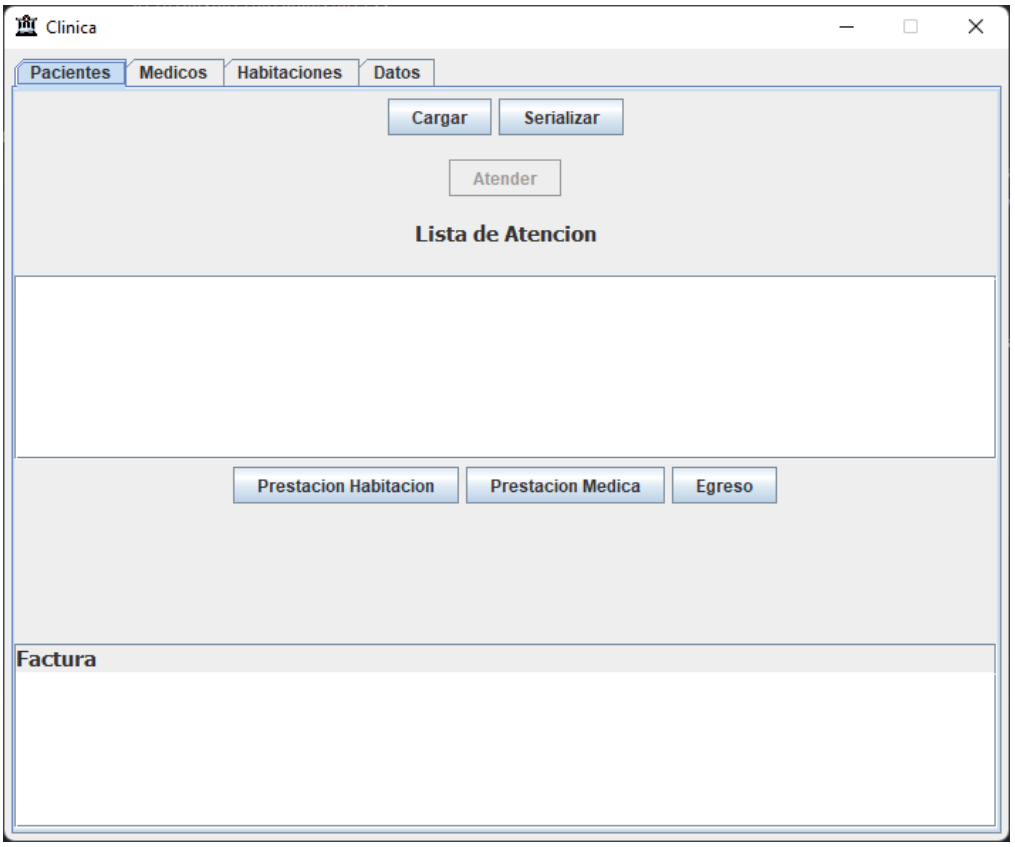
En primera instancia, los tests fallaron. Pero esto fue a causa de la falta de métodos Equals() específicos en varias clases, por lo que algunas comparaciones eran erróneas. Se procedió a realizar las modificaciones necesarias sobre las clases DTO para que estas tengan su forma de verificar la igualdad entre dos objetos.

Luego de llevar a cabo dichas implementaciones, los test arrojaron buenos resultados, y se concluye que la persistencia fue llevada a cabo de manera correcta.

Test de GUI para las interfaces gráficas

En esta instancia, buscamos verificar que las funciones de la interfaz gráfica sean correctas y funcionen de manera adecuada. Para esto es necesario revisar la documentación de la misma. En este caso no contamos con una documentación sobre la interfaz pero interpretamos su funcionamiento en base a las pruebas. Sin embargo, es posible haber realizado una interpretación errónea sobre alguna funcionalidad de la misma debido a la falta de información.

En este caso se testea la siguiente interfaz gráfica:



De esta ventana se prueban las pestañas de Pacientes, Médicos y Habitaciones ya que están vinculadas entre sí. Para realizar estos test se programa un robot que realiza distintas acciones sobre la interfaz.

Durante el test se verificó que cada botón funcione de manera correcta, verificando los mensajes emitidos o las acciones ejecutadas. El robot realizó la siguientes funciones:

- *cargaMedico()*: Selecciona un paciente, un médico y simula la prestación.
- *cargaHabitacion()*: Selecciona un paciente, una habitación y simula la prestación de habitación.
- *intentaAtenderSinCargar()*: Verifica que no se pueda clickear el botón atender cuando no hay pacientes en atención.
- *todosAtendidos()*: Verifica que, al momento de no haber más pacientes para atender y se apriete el botón *Atender*, se muestre una ventana emergente.

En la ejecución de este test se detectó que la interfaz permite solicitar una internación con una cantidad de días negativa y luego esto no se verifica en la capa de negocio ni en la de modelo, por lo que vemos que faltó la verificación sobre la interfaz. Sin embargo, este fue el único error encontrado a nivel de interfaz. Todas las pruebas restantes dieron los resultados esperados.

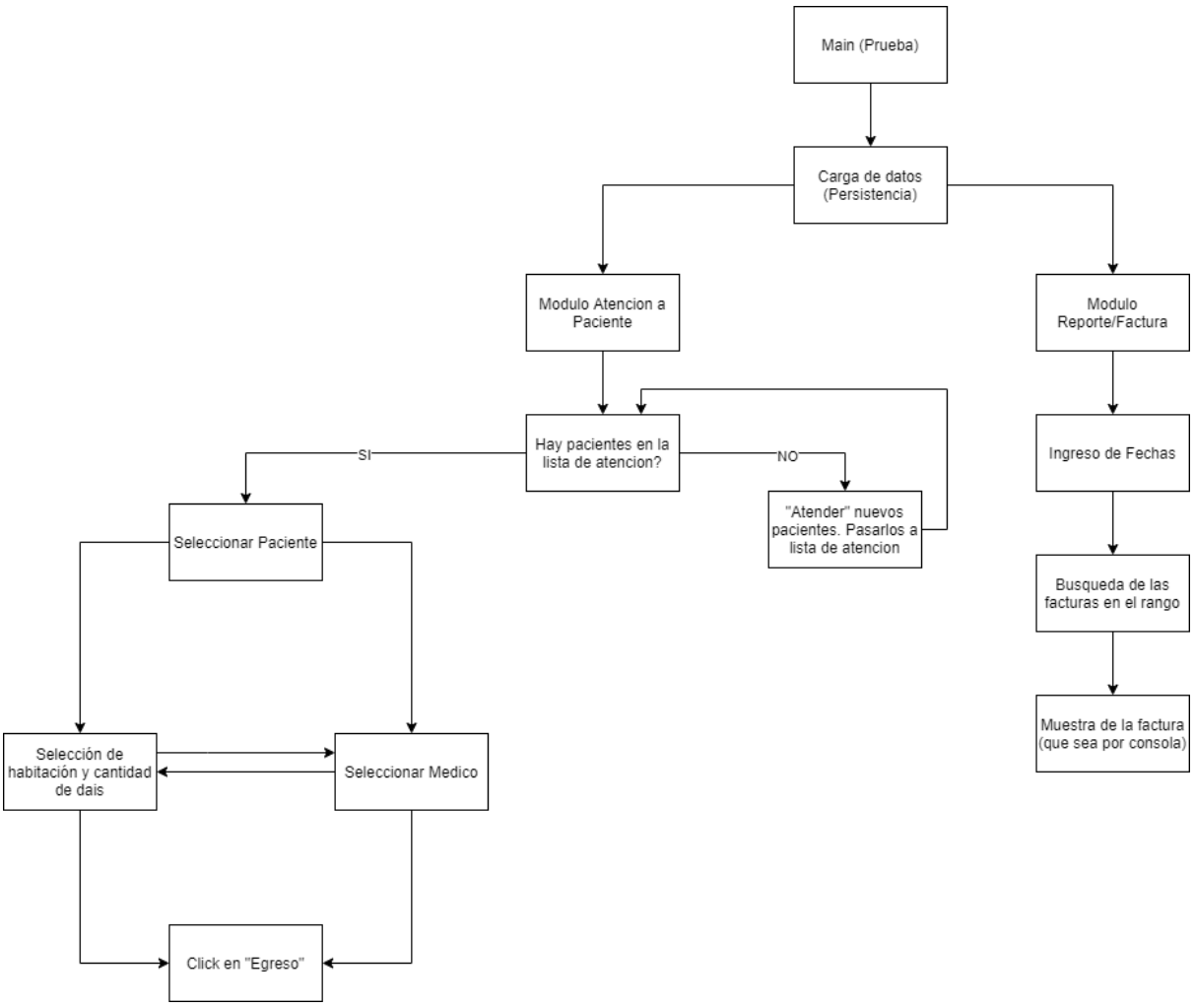
Test de integración

El propósito principal de las pruebas de integración es probar la interacción entre los distintos componentes de un sistema con el fin de encontrar errores de programación entre componentes o errores de interoperabilidad.

Existen distintos modelos para estas pruebas. Para este caso se decidió utilizar pruebas de tipo incrementales con una estrategia de integración descendente (“Top-Down”), y en profundidad (“Depth-first”).

Utilizamos este modelo debido a la estrategia que optamos para conseguir el caso de uso principal dentro de un flujo típico en el programa. Esta fue ejecutar el programa una y otra vez y entender cuál era el orden correcto de ejecución del mismo, es decir, qué botones apretar y en qué orden. Siguiendo esa dinámica logramos llegar al siguiente diagrama de actividad, del cual partimos a la hora de hacer el test de integración. Es decir, fuimos creando “indirectamente” las ramas que significan funcionalidades dentro de un caso de uso.

A continuación, el diagrama:



En él, vemos tres principales ramas:

- La rama de la izquierda, encargada de la funcionalidad de seleccionar un paciente, asignarle un médico o una habitación, y posteriormente realizar el egreso del mismo de la Clínica. Esto significó la creación de un método dentro de la clase *IntegracionDescendenteTes*t llamado *ramaHabitacionesTest()* en el que se crea un paciente y una habitación, los cuales se añaden en la clínica. Para este procedimiento, son utilizadas funcionalidades para modificar el estado de la habitación y el de la clínica.
- La Rama central cumple la función de transferir los pacientes de la lista de espera a la lista de atención, en caso de que los haya.
- La rama de la derecha es la encargada de la búsqueda de las facturas.

Realizamos pruebas sobre la rama izquierda y la rama derecha ya que vemos que son casos de uso esenciales en el uso normal del sistema. Los resultados de los tests de ambas fueron exitosos, mostrando así el funcionamiento en contexto de los módulos implementados.

En cuanto a la rama de la factura volvimos a usar mocks. En este caso se creó un mock de la factura para poder hacer una búsqueda rápida de las facturas sin la necesidad de cargar prestaciones y realizar funciones que no pertenecen a la rama a testear.

Conclusión

Durante el transcurso del testing, se tuvieron en cuenta las distintas metodologías vistas en la materia. Sin quitar mérito sobre el trabajo testeado, cada técnica aportó información congruente, dando a luz algunos errores de implementación que, de no ser por estas herramientas, no se hubieran detectado.

Se realizó el test de Caja Negra sobre los métodos más importantes del proyecto, utilizando su documentación para encontrar casos de prueba correspondientes a estos. Luego, se determinaron clases válidas e inválidas para varios de los métodos testeados. Gracias a las pruebas de unidad se pudieron determinar errores generales, para luego poder profundizar los testeos y descubrir el origen de aquellos. Es un método útil para reconocer grandes fallos, pero que se ve afectado cuando la documentación del programa es escasa, ya que dificulta la determinación de entradas, salidas, y posibles errores en la ejecución. Sin embargo, como se mencionó anteriormente, en nuestro caso fue de gran importancia para detectar el error del sistema de prioridades de la sala de espera, ya que, de otra forma, no hubiésemos podido conocer la existencia del mismo.

Aplicamos el test de Caja Blanca sobre el método agregado al sistema. Se determinó su complejidad ciclomática en base a su Grafo ciclomatico. Al momento de realizar estas pruebas se utilizó un mock para poder forzar las distintas coberturas que los test deben realizar. Una vez realizados estos pudimos observar una mejoría en cuanto a la cobertura del código.

En el test de GUI se probaron las distintas alternativas que podían surgir a partir de la interfaz dada. Se verificó que los componentes cumplieran su función principal aún teniendo algún fallo menor.

El test de persistencia requirió de realizar algunas modificaciones menores a la capa de modelo para que este funcionara correctamente, pero una vez aplicados los cambios mencionados previamente, los test resultaron exitosos.

Por último, en el test de Integración, se utilizaron pruebas incrementales por profundidad, dando así lugar a casos en los que se integran múltiples módulos del sistema para verificar las funcionalidades en conjunto. En esta etapa se volvió a hacer uso de mocks, creando uno sobre la Factura para poder testear de manera directa la búsqueda de estas.

En conclusión, vemos que la etapa de testing en el desarrollo de Software es un pilar fundamental para asegurar calidad en los productos. Se pudo comprobar también la necesidad de contar con una buena documentación a la hora de la participación en un proceso de desarrollo de Software, pues, de otra manera, las tareas de testeo pueden dar lugar a resultados difusos.

También, es evidente que la etapa de pruebas no debe ser la última del proceso, ya que, si esto sucede, las tareas de testing se vuelven tediosas y corregir estos errores puede incurrir en nuevos problemas que, de encontrarse en períodos avanzados, quizás no tengan solución. Es por esto que el testing debe acompañar en manera periódica al desarrollo del software, para así mejorar la eficacia del manejo de un proyecto.