TRABAJO PRÁCTICO FINAL

TALLER DE PROGRAMACIÓN I

Grupo N°3:

- **Cármenes, Santiago**
- **García, Federico**
- **Orte, Bautista**

Introducción

El testing es una tarea esencial en el proceso de desarrollo de software. Gracias a esta actividad es posible la detección de errores durante el mismo, para luego realizar las respectivas correcciones. Es importante realizar testeos en forma continua para reconocer los errores en forma temprana, y no correr el riesgo de, al intentar solucionar uno de estos errores, generar nuevos que dificultan el desarrollo. Por otra parte, para realizar pruebas es necesario tener una buena documentación, de modo que así surjan los casos posibles que puedan generar problemas en la ejecución. 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, realizaremos las pruebas 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 agencia de contratación.

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.
Para realizar las pruebas sobre el proyecto se utilizó el programa 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 setear los escenarios correspondientes. 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.
Para los test unitarios se van a considerar los siguientes escenarios. Vamos a enumerarlos con un identificador único porque se harán referencia a ellos a lo largo de todo el documento. De todas maneras, para cada método vamos a aclarar con qué escenarios fueron testeados, ya que estos no aplican para todos los casos.

Escenario	Descripción
1	La agencia tiene clientes
2	La agencia no tiene clientes
3	Agencia tiene clientes y en estado de búsqueda Laboral, se pueden crear y eliminar Tickets
4	Agencia tiene clientes y se encuentra en estado de contratación, se generaron listas de postulantes
5	Empleador tiene ticket1
6	Empleador tiene ticket2
7	Empleador tiene ticket3

Ticket1 = {"ticket.estudios=Primario", "ticket.experiencia=Exp_nada", "ticket.Jornada=Jornada_media", "ticket.Locacion=Home_office", "ticket.Puesto=Junior", 0<ticket.Remuneracion<V1}

Ticket2= {"ticket.estudios=Secundario", "ticket.experiencia=Exp_media", "ticket.Jornada=Jornada_completa", "ticket.Locacion=Presencial", "ticket.Puesto=Senior", V1<ticket.Remuneracion<V2}

Ticket3= {"ticket.estudios=Terciario", "ticket.experiencia=Exp_mucha", "ticket.Jornada=Jornada_extendida", "ticket.Locacion=Indistinto", "ticket.Puesto=Management", ticket.Remuneracion<V2}

A continuación mostraremos un ejemplo de cómo hicimos las clases de equivalencia y las baterías de prueba de cada método. *Los demás métodos se encuentran en el documento anexo.*

void crearTicketEmpleador(String locacion, int remuneracion, String jornada, String puesto, String experiencia, String estudios, Cliente cliente)

Numero escenario	Descripción
1	*Agencia tiene clientes y en estado de búsqueda Laboral, se pueden crear y eliminar Tickets*
2	*Agencia tiene clientes y se encuentra en estado de contratación, se generaron listas de postulantes*

Tabla de Particiones en Clases de Equivalencia

Dato de entrada	Clases de equivalencia	Aplica?	Motivo	Identificador de clase de equivalencia
locación	*=“Home_Office” | =“Presencial” | =“Cualquiera”*	*si*	*cumple contrato*	*1*
remuneración	*remuneración>0*	*si*	*cumple contrato*	*2*
jornada	*=“Jornada_Completa” | “Jornada_Extendida” | “Jornada_Media”*	*si*	*cumple contrato*	*3*
puesto	*=“Junior” | “Senior” | “managment”*	*si*	*cumple contrato*	*4*
experiencia	*=“nada” | “media” | “mucha”*	*si*	*cumple contrato*	*5*
estudios	*=“primario” | “secundario” | ”terciario”*	*si*	*cumple contrato*	*6*
cliente	*Cliente es un empleador*	*si*	*cumple contrato*	*7*
cliente	*Cliente no es empleador*	*no*	*no cumple contrato*	*8*

Batería de pruebas (Esc. 1)

Número de prueba	Datos de entrada	Valor	Salida Esperada	Clases que abarca
1	*locacion*	*“presencial”*	*Crea un nuevo ticket para el empleador*	*1,2,3,4,5,6,7*
	remuneracion	*70000*		
	jornada	*“completa”*		
	puesto	*“junior”*		
	experiencia	*“nada”*		
	estudios	*“secundario”*		
	cliente	*Empleador1*		
2	*locacion*	*“presencial”*	*Elimina el ticket y crea uno nuevo para el empleador*	*1,2,3,4,5,6,7*
	remuneracion	*70000*		
	jornada	*“completa”*		
	puesto	*“junior”*		
	experiencia	*“nada”*		
	estudios	*“secundario”*		
	cliente	*Empleador2*		

Aclaración: Empleador1 no posee ticket y Empleador2 si posee.

Batería de pruebas (Esc.2)

Número de prueba	Datos de entrada	Valor	Salida esperada	Clases que abarca
1	*locacion*	*“presencial”*	*ImposibleModificarTicketsException*	*1,2,3,4,5,6,7*
	remuneracion	*70000*		
	jornada	*“completa”*		
	puesto	*“junior”*		
	experiencia	*“nada”*		
	estudios	*“secundario”*		
	cliente	*Empleador1*		

Aclaración: Para el esc. 2 no se toman más casos de prueba ya que la salida esperada en este escenario no va a depender del cliente.

Informe de errores en caja negra

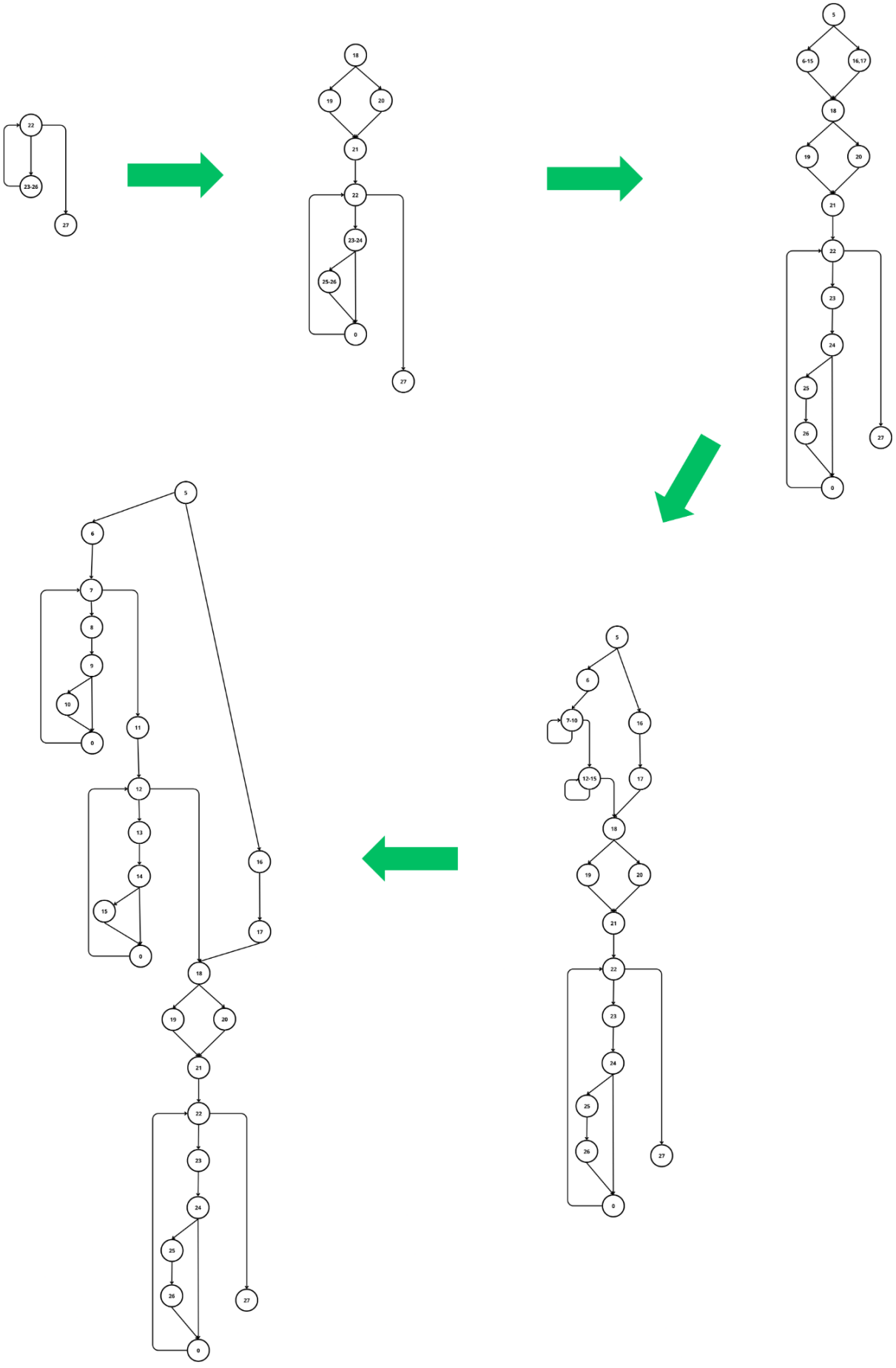
N° de error	Método	N° de escenario	Descripción del error
1	login	2	Lanza la excepción NombreUsuario en vez de ContraException.
2	GatillarRonda	1 - 2	No coinciden los puntajes con los esperados.
3	RegistroEmpleado	1 - 2	Guarda el apellido en el lugar del teléfono y viceversa.
4	Empleador	1 - 2	Guarda el apellido en el lugar del teléfono y viceversa.
5	RegistroEmpleador	1	Cuando se quiere crear un empleador con el mismo nombre de usuario, no lanza la excepción NewRegisterException.
6	RegistroEmpleador	1 - 2	Cuando el rubro es algo diferente de los rubros válidos debería haber lanzado ImposibleCrearEmpleadoException.
7	setLimitesRemuneracion	1 - 2	Cuando el límite superior es menor que el inferior, no lanza la excepción LimiteSuperiorRemuneracionInvalidaException.
8	getComparacionRemuneracion	5	El resultado de la comparación no es el esperado cuando el empleador tiene una remuneración menor que V1 y el empleado una remuneración mayor que V1 pero menor que V2.
9	getComparacionTotal	6	No calcula bien el resultado.
10	getComparacionPuesto	7	El resultado de la comparación no es el esperado cuando el empleador pide un Senior y el empleado quiere ser management.
11	getComparacionJornada	7	El resultado de la comparación no es el esperado cuando el empleador pide jornada completa y el empleado quiere jornada extendida.

Caja Blanca y test de cobertura

Las pruebas de caja blanca son un proceso en el que el tester utiliza distintos conjuntos de datos que son pasados por parámetro, con el objetivo de lograr una cobertura del 100% siempre que sea posible en un método dado.
En nuestro caso, el método a testear con caja blanca es aplicaPromo de la clase UtilPromo. Al método se le pasan por parámetro un boolean: “promoPorListaDePostulantes” y dos Hashmap: empleados y empleadores. El método se encarga de devolver el empleado/empleador beneficiado (será el que posea más puntos del iterator clientes, creado a partir del hashmap que posea un contador mayor) dicho contador crecerá a partir de la cantidad de postulantes que posean en total los empleadores y empleados en caso de ser true el boolean. En el caso de que este último sea falso, el contador será el tamaño del hashmap de cada uno. Si ningún cliente puede ser beneficiado por este método, el mismo devolverá null.

1.

Para obtener el grafo de forma ascendente como se nos pidió, partimos de instrucciones individuales y las fuimos agrupando formando las estructuras. Comenzamos por las instrucciones que estaban dentro de las iteraciones y avanzamos hacia el exterior.


22
23-26
27
18
19
20
21
22
23-24
25-26
0
27
5
6-15
16,17
18
19
20
21
22
23
24
25
26
27
0
5
6
7-10
16
12-15
17
18
19
20
21
22
23
24
25
26
27
0
5
6
7
8
9
10
0
11
12
13
14
15
0
16
17
18
19
20
21
22
23
24
25
26
27
0

2.

La complejidad ciclomática define el número de caminos independientes dentro de un fragmento de código y determina la cota superior de la cantidad de pruebas que se deben realizar para asegurar que todas las sentencias se ejecutan al menos una vez. Para obtener la complejidad ciclomática del grafo de flujo utilizamos los tres métodos posibles para obtener la misma:

- La suma de los arcos menos la cantidad de nudos + 2.
- El número de regiones más el espacio exterior.
- El número de nodos condición + 1.

Arcos - nodos + 2 = 34 - 27 + 2 = 9
N°Regiones = 9
Nodos condición + 1 = 8 + 1 = 9

La complejidad ciclomática del grafo es 9.

3.

Para obtener el conjunto básico de caminos independientes por el método simplificado como se nos pidió, comenzamos seleccionando el camino más corto y luego fuimos buscando segmentos no recorridos hasta completar el número de caminos necesarios.

C1: 5 - 16 - 17 - 18 - 20 - 21 - 22 - 27
C2: 5 - 6 - 7 - 11 - 12 - 18 - 20 - 21 - 22 - 27
C3: 5 - 16 - 17 - 18 - 19 - 21 - 22 - 23 - 24 - 25 - 26 - 0 - 22 - 23 - 24 - 0 - 22 - 27
C4: 5 - 6 - 7 - 8 - 9 - 0 - 7 - 11 - 12 - 18 - 20 - 21 - 22 - 27
C5: 5 - 6 - 7 - 11 - 12 - 13 - 14 - 0 - 12 - 18 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 22 - 27
C6: 5 - 6 - 7 - 8 - 9 - 10 - 0 - 7 - 11 - 12 - 18 - 19 - 21 - 22 - 23 - 24- 25 - 26 - 27
C7: 5 - 6 - 7 - 11 - 12 - 13 - 14 - 15 - 0 - 12 - 18 - 20 - 21 - 22 - 23 - 24- 25 - 26 - 27

4.

Camino indep. N°	Parámetros de entrada	Resultado esperado
1	promoPorListaDePostulantes= false empleados= vacío empleadores= vacío	Null
2	promoPorListaDePostulantes= true empleados= vacío empleadores= vacío	Null
3	promoPorListaDePostulantes= false empleados= vacío empleadores= no vacío (mínimo 2 elementos en el que el puntaje del 2do sea menor que el del primero)	El primer elemento de empleadores
4	promoPorListaDePostulantes=true empleados= vacío empleadores= no vacío (donde lista de postulantes == null)	Null
5	promoPorListaDePostulantes= true empleados= no vacío (donde lista de postulantes == null) empleadores= vacío	Null
6	promoPorListaDePostulantes= true empleados= vacío empleadores= no vacío (donde lista de postulantes tiene elementos)	El elemento de empleadores
7	promoPorListaDePostulantes= true empleados= no vacío (donde lista de postulantes tiene elementos) empleadores= vacío	El elemento de empleados

Para realizar el test de cobertura se siguieron los caminos mencionados anteriormente con el objetivo de lograr el 100% de cobertura del método aplicaPromo pasando por parámetro los distintos conjuntos de datos mencionados en cada camino.

Camino independiente	Porcentaje de cobertura
C1	35.8%
C2	53.7%
C3	72.6%
C4	80.0%
C5	87.4%
C6	93.7%
C7	100%

```
    public Cliente aplicaPromo(boolean promoPorListaDePostulantes,
            HashMap<String, EmpleadoPretenso> empleados, HashMap<String, Empleador> empleadores)
    {
        Iterator clientes = null;
        int contadorEmpleador = 0;
        int contadorEmpleadoPretenso = 0;
        Cliente clienteBeneficiado = null;

        if (promoPorListaDePostulantes)
        {
            Iterator<Empleador> itEmpleadores = empleadores.values().iterator();
            while (itEmpleadores.hasNext())
            {
                Empleador empleador = itEmpleadores.next();
                if (empleador.getListaDePostulantes() != null)
                    contadorEmpleador += empleador.getListaDePostulantes().size();
            }
            Iterator<EmpleadoPretenso> itEmpleados = empleados.values().iterator();
            while (itEmpleados.hasNext())
            {
                EmpleadoPretenso empleadoPretenso = itEmpleados.next();
                if (empleadoPretenso.getListaDePostulantes() != null)
                    contadorEmpleadoPretenso += empleadoPretenso.getListaDePostulantes().size();
            }
        } else
        {
            contadorEmpleador = empleadores.size();
            contadorEmpleadoPretenso = empleados.size();
        }
        if (contadorEmpleador > contadorEmpleadoPretenso)
            clientes = empleadores.values().iterator();
        else
            clientes = empleados.values().iterator();

        int puntajeMaximo = Integer.MIN_VALUE;
        while (clientes.hasNext())
        {
            Cliente cl = (Cliente) clientes.next();
            if (cl.getPuntaje() > puntajeMaximo)
            {
                puntajeMaximo = cl.getPuntaje();
                clienteBeneficiado = cl;
            }
        }
        return clienteBeneficiado;
    }
}
```

Cobertura del 100%

Test de Persistencia

Para realizar este test, tuvimos en cuenta que acciones de persistencia realizaba el programa. El programa, al cerrar sesión o cerrarlo, persiste los datos necesarios en un archivo en formato .xml. Entonces, las preguntas que nos hicimos fueron:
¿Guarda el archivo correctamente?
¿Carga los archivos correctamente?
Los datos después de cargarlos, ¿coinciden con los que se deberían haber guardado?
¿Lanza las excepciones que debería lanzar?

Para responder estas preguntas creamos una nueva instancia de la Agencia, le asignamos los valores que esta debería persistir y la persistimos. Luego de esto le sobreescribimos los datos a la agencia y la cargamos desde el archivo.

Informe de errores persistencia

N° de error	Descripción del error
12	El único error existente en el persistencia es que al persistir la agencia los límites de remuneración se persisten de la manera incorrecta, el límite inferior nunca persiste.

Test de GUI

Para el test de GUI, nuestro objetivo fue verificar que los comportamientos de la interfaz gráfica sean los establecidos en el contrato.
En el caso de esta interfaz de usuario, tuvimos que realizar el test sobre 4 paneles, el panel de inicio de sesión, el de registro, el de administrador y el de cliente.

Hacer el test de todas las combinaciones y permutaciones de los campos de entrada se hace muy extenso, por lo que tuvimos que hacer un recorte. Asumimos que todos los paneles son independientes y que no van a alterar el comportamiento uno del otro y, además, seguimos un orden “lógico” en las entradas de datos.

El test se dividió en dos partes. La primera fue el testeo del comportamiento esperado de los elementos, es decir, si los elementos se habilitaban y si se deshabilitaban correctamente de acuerdo a lo establecido en el contrato. Para este caso, no era necesario conocer nada sobre la lógica de negocio del sistema ni del acceso a datos.

En la segunda parte nos encargamos de comprobar si los botones cumplen su función, es decir, invocan al método correcto del contralor y posteriormente, en caso de que haya una excepción o se espera algún comportamiento específico, se cumple. Por ejemplo, si al ingresar una contraseña incorrecta se muestra el cartel correspondiente. En esta parte tuvimos que hacer un acceso a datos, a diferencia de la primera.

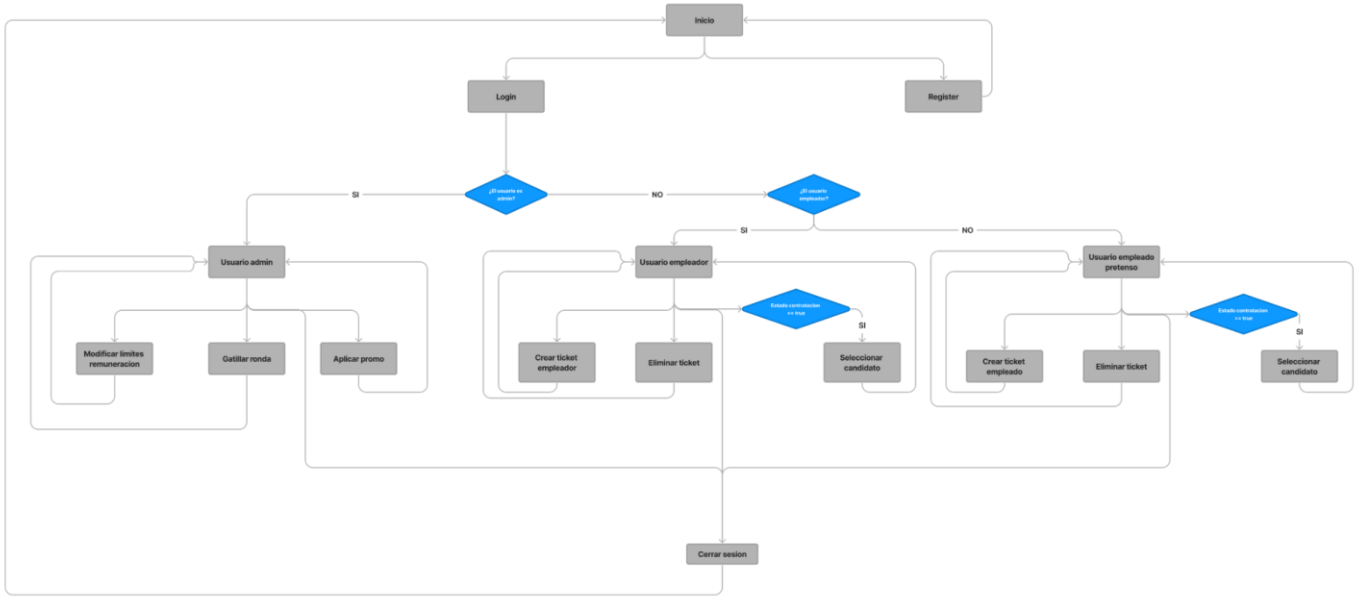
En la parte de accesos a datos, en algunos casos, utilizamos mocks para evitar entrar excesivamente en la lógica de negocio desde el test de GUI. Un ejemplo puede ser el caso de cuando se aprieta el botón de seleccionar candidato desde el panel cliente. Al apretar este botón debería mostrar en una ventana emergente los datos del cliente que se seleccionó como candidato, es por eso que hicimos un mock que devuelva un cliente cuando se llamaba al método vista.getCandidato().

Informe de errores en GUI

N° de error	Descripción del error
13	No se lanza la excepción de que las contraseñas no coinciden cuando se registra un empleado o un empleador, por lo tanto no muestra la ventana correspondiente.
14	No se lanza la excepción de empleador existente cuando se registra un empleador con el mismo nombre de usuario que uno que ya existía, por lo tanto no se muestra la ventana correspondiente.
15	En el panel de login cuando se loguea el admin con una contraseña incorrecta, se muestra la ventana de usuario inexistente en lugar de contraseña incorrecta, como debería ser.

Test de integración

Realizamos pruebas de integración para probar la interacción de los distintos componentes del sistema. Para esto, partimos de la base de crear un diagrama de flujo del programa, para luego, determinar los casos de uso posibles de este.



Se adjunta en mayor resolución en el documento "Anexo"

Con este diagrama utilizamos la estrategia de integración descendente y en profundidad, con el que determinamos 5 casos de prueba para hacer los respectivos test.

Caso de prueba 1: El administrador inicia sesión, establece los límites de remuneración, gatilla ronda y cierra sesión.

Caso de prueba 2: Un empleado se registra, el estado de contratación es falso, por lo tanto crea un ticket y luego cierra sesión.

Caso de prueba 3: Un empleador inicia sesión, el estado de contratación es falso, elimina un ticket y cierra sesión.

Caso de prueba 4: Un empleador inicia sesión, el estado de contratación está en true, selecciona candidato y cierra sesión.

Caso de prueba 5: El admin iniciamos sesión, gatilla ronda y se hacen matches.

En el anexo se adjunta para cada caso de prueba con sus respectiva rama recorrida.

Hicimos un recorte y no probamos todos los casos de uso posibles, ya que eso derivan en múltiples casos de prueba y es muy tedioso de llevar a cabo. Con estos casos de prueba consideramos que logramos una buena “cobertura” de las funcionalidades del sistema.

Conclusión

Durante el transcurso del testing, se tuvieron en cuenta las distintas metodologías vistas en la materia. Cada técnica aportó información, mostrando algunos errores de implementación que, de otra forma, 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 los mismos. Gracias a las pruebas de unidad se pudieron determinar errores generales, para luego poder profundizar los testeos y descubrir el origen de cada uno de ellos.

Aplicamos el test de Caja Blanca sobre el método aplicarPromo. Se determinó su complejidad ciclomática en base a su Grafo ciclomatico. Luego se realizó el test de cobertura sobre el método testeando con los distintos parámetros de entrada para así lograr una cobertura del 100%.

En el test de GUI logramos verificar si los componentes cumplían sus funciones y comportamientos en la interfaz dada, además de detectar los errores. Además hicimos un test con acceso a datos para probar el manejo de excepciones.

En el test de persistencia verificamos que los componentes referidos a la agencia se persistían correctamente en el archivo “agencia.xml” y descubrimos los errores correspondientes en dicha persistencia.

Por último, en el test de Integración, se probaron distintos casos de uso que dieron lugar a poder testear cómo se integran múltiples módulos para verificar su funcionalidad en conjunto.

En conclusión, vemos que la etapa de testing en el desarrollo de Software es fundamental para asegurar calidad en los productos. Además, se pudo comprobar también la necesidad de contar con una buena documentación para que el testeo sea más sencillo y preciso.