Trabajo Práctico Grupal

Integrantes:
Kollman, Deborah
Gonzalez Betti, Francisco
Stimmler, Francisco
Veitch, Matias

Cátedra: Taller de programación I

- Caja negra para la totalidad de los métodos. No testear getters y setters.
- **Caja blanca para un método anexo – tener en cuenta este agregado.**
- Test de persistencia. *Elegir solo uno de los módulos, por ejemplo Pacientes, para realizar el test.*
- Test de GUI para una interfaz gráfica que cumpla con las consideraciones descritas anteriormente . *Elegir solo una de las interfaces gráficas, (por ejemplo la que controla los pacientes). Si todo el manejo del sistema estuviese en la misma ventana, solo testear uno de los paneles*
- Test de Integración.

  Testear: Modelo, Persona, infraestructura, Persistencia

Informe Caja Negra:

Se creó un paquete testCajaNegra conteniendo las clases de JUnit para testear las clases del modelo. Se testearon las clases principales Clinica, Factura, PacienteFactory, MedicoFactory y Habitacion (junto con las clases que la heredan).

- Clase Clinica:
  - Los métodos de la clase Clínica no lanzan excepciones. Los métodos realizaron correctamente los reportes que debían hacer.
  - Se sugiere que el contrato del método derivarPaciente especifique qué pacientes según prioridad se derivan a patio o sala de espera para poder testear que sea derivado correctamente. También se sugiere que se aclare si el paciente debe estar en la lista de pacientes de la clínica o no.
  - Se sugiere que el método reporteMedico lance una excepcion si la primera fecha es mayor a la segunda que informe que el periodo de fechas es incorrecto.
- Clase Factura:
  - El constructor inicializa correctamente el objeto sin producir excepciones
  - En asignarMédico se encontraron varios salidas que no coinciden con la salida esperada cuando se asigna un médico con especialidad en cirugía y el posgrado.
  - En asignarHabitacion también se encontraron salidas que no cumplen con el contrato. Esto se da cuando se asigna una habitación privada con días mayores a iguales a 2 en ambos escenarios.
  - En mostrarFacturas no se encontró ninguna anormalidad. Se muestra respetando el formato definido.
- Clase PacienteFactory:

    - Se sugiere que el contrato del método getPaciente indique que rangoEtario son considerados válidos. El método instancia correctamente al paciente según su rango etario y lanza NoExisteRangoEtarioException correctamente cuando se ingresa otro rango etario.

- Clase MedicoFactory:
    - No se pueden generar instancias de MedicoCirujano a partir de MedicoFactory. Se lanza la excepción NoExisteEspecialidadException al indicar “cirugía” como especialidad. El método instancia correctamente al médico con Especialidad ‘Clínica’ y ‘Pediatría’. De estos se instancia correctamente si se indica ‘Permanente’ y ‘Residente’ como Contratación, y ‘Magíster’ y ‘Doctor’ como Posgrado.
    - Se lanzan correctamente NoExisteEspecialidadException si se indica otra especialidad, NoExisteContratacionException si se indica otra contratación y NoExistePosgradoException si se indica otro posgrado
    - Se sugiere que el contrato del método getMedico indique que Especialidad, Contratación y Posgrado son considerados válidos.
- Clase Habitación:
    - La abstracta clase Habitación y las clases HabitacionCompartida, HabitacionPrivada y TerapiaIntensiva que la heredan no lanzan excepciones.
    - Como la clase Habitacion es abstracta y las clases que heredan de esta no agregan métodos propios, se decidió testear a las clases HabitacionCompartida, HabitacionPrivada y TerapiaIntensiva como escenarios de la clase Habitacion.
    - Los constructores y los métodos no incluyen los contratos
    - Los constructores de las tres clases instancian correctamente las habitaciones correspondientes con los valores indicados.
    - El método costoDeHabitacion de la clase HabitacionCompartida y HabitacionPrivada no devuelve los valores correctos.

https://docs.google.com/spreadsheets/d/1VUVsuSyRSxRi3do3k2sHkui66M0FeEMW/edit?usp=sharing&ouid=109995058829083048047&rtpof=true&sd=true

Informe Caja Blanca:

Se creó el grafo del método calculoImporteAdicionales() y se sacaron las siguientes conclusiones.

Al calcular la complejidad ciclomática se llegó a 14, por lo que basado en el trabajo de McCabe, es un método medianamente complejo con riesgo moderado.

Se presentaron 4 escenarios:

- Facturas vacías y con paciente con rango etario diferente a mayor
- Facturas vacías y con paciente con rango etario igual a mayor
- Facturas con importe y con paciente con rango etario diferente a mayor

- Facturas con importe y con paciente con rango etario igual a mayor

A partir de los casos de prueba y el testeo se comprobó que no hay caminos inaccesibles.

Los datos de entrada variaron en el número de factura para testear cómo reacciona el código al buscar facturas que no se encontraban almacenadas, la fecha de solicitud para verificar el importe y la lista de insumos.

Dada la complejidad ciclomática sería correcto delegar ciertas tareas para reducir el número de nodos condición. Por ejemplo, el cálculo del SubTotalImpar hacerlo dentro de la factura, ya que la misma tiene la lista de prestaciones. El cálculo de diferencia entre dos fechas hacerla en otro objeto aparte, ya que además se puede reutilizar en otras partes del programa.

Por último, se encontraron dos errores. El primero es que el programa deja de funcionar cuando no hay facturas. El segundo es cuando se calcula la diferencia de año (linea 233), donde se confunde mes con año y esto hace que si se ingresa una fecha de factura 11 años más antigua se toma como si estuviese en el mismo mes

Informe Test de Persistencia:

Se creó en el paquete 'prueba' la clase TestPersistencia de JUnit para testear la clase Persistencia y su aplicación al módulo de pacientes. En esta se crearon dos clases de setUp para iniciar la prueba, una que crea una instancia de la clase Persistencia y de la clase PacienteDTO (la clase encargada de contener los pacientes registrados en el sistema), llamada setUpVacio, y otra que además de esto registra dos pacientes, llamada setUpLleno.
Luego se crearon las clases para testear la persistencia:

- TestCrearArchivo: Comprueba que la persistencia sea capaz de crear un archivo con el nombre suministrado. La clase de persistencia fue capaz de crear correctamente un archivo con el nombre y extensión que le fue indicado.
- TestPacientesVacioArchivo: Prueba si la persistencia podía crear un archivo, escribir en él una lista de pacientes vacía y leerla sin que se lancen excepciones. La clase de persistencia fue capaz de crear el archivo, escribir y leer sin que se lancen excepciones.
- TestBDPacientes: Prueba si la persistencia puede crear un archivo, escribir en este una lista de pacientes no vacía y leerla sin que lancen excepciones o que hubiera pérdida de información. Se comprobó que la clase de persistencia pudo crear el archivo, escribirlo con la lista de pacientes en el sistema y leer correctamente la lista sin que haya pérdida de información ni se lancen excepciones.
- despercistirArchivoInexistente: Comprueba que la persistencia lance una excepción si se intenta leer un archivo inexistente. Se comprobó que al intentar leer un archivo no existente se lanzó una excepción de FileNotFoundException correctamente

En conclusión: La persistencia puede crear un archivo, escribirlo y leerlo correctamente, sin que se lancen excepciones o que haya pérdida de información. También lanza correctamente una excepción si se intenta leer un archivo inexistente.

Informe Test de GUI:

Se ha tenido que cambiar algunos aspectos del código para que se pueda hacer el test de GUI correctamente, entre ellos se agregó un getter de la vista dentro del controlador, ya que de otra forma no era posible acceder.
También se creó la clase MiOptionPane la cual muestra los mensajes así por el test a partir de un FalsoOptionPane se puede ver el mensaje a mostrar y hacer el asserto de si está bien.

También se creó un enum de Mensajes para hacer más fácil la comparación entre mensajes.

El test de GUI verifica si se puede añadir un médico exitosamente a través de la GUI también de que se muestre correctamente el mensaje en caso de los tres errores posibles, todos esos test tanto dentro de un conjunto con médicos ya ingresados y otro vacío.

La vista lanza 3 tipos distinto de Excepciones las cuales las captura el controlador y muestra un mensaje cuál es la excepción y el dato erróneo.

Informe Test de integración:

Para probar la interacción entre los los módulos definidos en el contrato:

- Facturación al Paciente
- Reportes de Actividad de lo Médicos (con suma de honorarios)
- Resolución de Conflictos de Sala de Espera.

Se comenzó con las pruebas no incrementales. Para esto se tomaron todos los métodos más importantes.
Para las pruebas incrementales, tanto la integración descendente como ascendente, se definió el diagrama en profundidad y en anchura. Para el primer caso se encontró un error en la prioridad cuando dos pacientes están esperando a ser atendidos. Según el contrato los Jóvenes tienen prioridad sobre los Mayores.
Dado que el programa esta completo, cubriendo todas los requerimientos planteados, y que no presentan comportamientos imposible o impracticables, no hubo necesidad de usar mock

Para el test de integracion orientado a objetos, se diseñaron los diagramas *Diagrama de secuencia,* Estados identificables, Valores adecuados, Casos de prueba, para los casos de:

- *agregar médico*
- *agregar un nuevo Paciente*
- *derivar un Paciente*
- *atender un paciente*
- *egreso factura*
- *mostrar reporte de médico*

En los casos donde más errores se pueden dar es en el ingreso de médicos y pacientes, debido a la cantidad de datos que se debe agregar. Sin embargo hemos detectado errores en “reporteMedico” donde no se controla si las fechas son

correctas, aunque en este caso no mostrará ninguna debido a que no encontrará facturas que se adecuen a ese intervalo.