Facultad de Ingeniería

Universidad Nacional de Mar del Plata

Informe: Testing del software "Subí que te llevo"

Integrantes:

Esteban, Octavio
Frickmann, Tomás
Goñi Pattiño, Lautaro
Julachis, Luciano Agustín

Fecha de entrega: 19/11/2025

Introducción:

El presente trabajo práctico grupal tiene como objetivo aplicar los conocimientos adquiridos en la materia Taller de Programación I mediante la realización de pruebas sobre un sistema denominado “Subí que te llevo”. Dicho sistema consiste en una aplicación de gestión para una empresa de transporte de pasajeros, que permite administrar clientes, choferes, vehículos y viajes.

A partir de este proyecto, se llevaron a cabo diferentes tipos de pruebas de software con el propósito de evaluar su correcto funcionamiento, detectar fallas y verificar el cumplimiento de los requerimientos definidos en el documento SRS (Especificación de requerimientos de software)

El proceso incluyó pruebas de Caja Negra, Integración, Persistencia y GUI, utilizando herramientas de desarrollo y testing en Java, principalmente JUnit y la clase Robot para la automatización de la interfaz gráfica.

Este informe presenta la metodología de pruebas empleada, los resultados obtenidos y el análisis de los errores detectados, destacando la importancia del testing como una etapa esencial dentro del ciclo de vida del software para garantizar la calidad, confiabilidad y usabilidad del sistema.

Objetivos:

- Verificar la correcta funcionalidad del sistema “Subí que te llevo” mediante distintos tipos de pruebas de software.
- Detectar errores, excepciones y comportamientos inesperados en los diferentes módulos del sistema.
- Comprobar el cumplimiento de los requerimientos funcionales y no funcionales definidos en el SRS.
- Aplicar las técnicas de testing vistas en clase, especialmente las pruebas de Caja Negra, Integración, Persistencia y GUI.
- Analizar el impacto de los errores encontrados sobre la lógica del sistema.

Test Caja Negra:

Se utilizó la herramienta JUnit en Java para realizar pruebas unitarias, tomando como base el Javadoc del programa y los requerimientos funcionales definidos en el documento SRS.

Cada prueba se diseñó considerando los posibles casos válidos e inválidos que un usuario podría ejecutar, organizados mediante tablas de particiones de equivalencia y casos de prueba específicos para los métodos más relevantes del sistema.

En el siguiente apartado se presentan las respectivas tablas de particiones y la batería de pruebas diseñadas:

Modelo de Datos- Prueba de Caja Negra

Modelo de Negocios - Prueba de Caja Negra

Estas pruebas permitieron comprobar cómo el sistema respondía ante diferentes entradas y condiciones, y detectar errores relacionados con la lógica de negocio, el manejo de excepciones y la validación de datos.

A continuación, se detallan los principales escenarios evaluados y los errores detectados durante la ejecución de los tests.

Test Empresa:

Las pruebas realizadas sobre la clase Empresa corresponden no solo a pruebas unitarias, sino a pruebas de integración, ya que ejercen simultáneamente múltiples componentes del sistema. Cada método de Empresa depende de varios objetos del dominio (Cliente, Chofer, Pedido, Vehículo, Viaje, etc.) y sus interrelaciones. Por este motivo, al ejecutar un test sobre Empresa se está verificando el funcionamiento conjunto de estas clases y no únicamente la lógica aislada de un método.

Este tipo de pruebas permite detectar errores que surgen de la interacción entre las distintas entidades del modelo, por ejemplo, secuencias incorrectas de validaciones, manejo indebido de estados o propagación errónea de excepciones, ofreciendo una visión más realista del comportamiento del sistema completo. Sin embargo, debido a su alcance más amplio, los tests de integración también tienden a exponer bugs que las pruebas unitarias no muestran, como los casos identificados en este trabajo.

- **Escenario 1:** No presenta errores.
- **Escenario 2:**
  - Al intentar iniciar sesión con una contraseña incorrecta, el sistema debería lanzar la excepción PasswordErroneaException. Sin embargo, en su lugar se arroja la excepción UsuarioNoExisteException, indicando un manejo incorrecto de la validación de credenciales.
  - Al agregarle un pedido a un cliente que cuenta con uno pendiente, al lanzar la excepción ClienteConPedidoPendienteException no muestra el mensaje esperado.

- Al agregar un vehículo que ya fue registrado previamente, se lanza la excepción correcta VehiculoRepetidoException. Sin embargo, el vehículo pasado como parámetro dentro de la excepción es null, lo que indica un error en su almacenamiento.
- Al intentar registrar un chofer que ya existe en el sistema, debería lanzarse la excepción ChoferRepetidoException. En cambio, el sistema permite que el chofer sea agregado nuevamente al HashMap de choferes.

- **Escenario 3:**
  - Al intentar crear un viaje con un vehículo que no pertenece al sistema, el test esperaba que se lanzara la excepción VehiculoNoDisponibleException. Sin embargo, se arrojó una excepción distinta (Cliente sin Viaje Exception), lo que provocó el fallo del test en la línea fail(e.getMessage()). Esto indica que el método crearViaje() no está manejando correctamente la validación del vehículo utilizado.
  - Al crear un viaje para un cliente que ya posee uno en curso, el sistema debería lanzar ClienteConViajeException. En cambio, el método permite la creación del viaje y la posterior carga de un nuevo pedido, sin validar la condición del cliente. Este error es heredado de ClienteViajePendiente.
  - No se devuelve la lista de vehículos correctamente ordenada según el puntaje obtenido para un pedido determinado. Lo que indica que el primer vehículo de la lista no posee un puntaje mayor que el segundo, es decir, la lista no está ordenada correctamente. Además, se detectó que el cálculo del puntaje de los vehículos también es incorrecto, lo que contribuye al fallo del test.
  - Al intentar agregar un nuevo pedido a un cliente que ya tiene un viaje en curso, el sistema debería lanzar la excepción ClienteConViajePendienteException lo que indica que el método agregarPedido() no está validando correctamente el estado del cliente antes de permitir la creación del nuevo pedido.

- **Escenario 4:**
  - Al intentar obtener la calificación de un chofer que no posee viajes registrados, el sistema debería lanzar la excepción

SinViajesException. Como resultado, el sistema permite evaluar a un chofer sin historial de viajes, lo cual es incorrecto.
- Al intentar crear un viaje con un chofer que no está disponible, el sistema debería lanzar la excepción ChoferNoDisponibleException. Se lanzó una correctamente pero la misma está mal construida.
- Al agregar un pedido a un cliente con uno ya pendiente, se lanza la excepción ClienteConPedidoPendienteException con el error de mostrar un mensaje distinto al esperado.

Test Datos:

- **TestAuto:**
  - Al calcular el puntaje de un vehículo tipo Auto cuando el pedido no solicita baúl, el valor retornado por el método getPuntajePedido() no coincide con el puntaje esperado. Esto indica que el cálculo dentro del método es incorrecto, posiblemente debido a un error en la fórmula o en las condiciones que determinan el valor base del puntaje.

- **TestChoferPermanente:**
  - Al calcular el sueldo bruto de un chofer permanente con antigüedad mayor a 20 años, el cálculo es erróneo. Es decir, el método getSueldoBruto() no está limitando correctamente la antigüedad a un máximo de 20 años.

- **TestCombi:**
  - No debería calcular el puntaje del pedido al tener un pedido con una cantidad de pasajeros menor a cuatro. El método devuelve un valor distinto de null, por lo que se realiza el cálculo del puntaje incluso cuando se incumple la cantidad de pasajeros mínimos determinados para la combi. Es decir, hay un error de validación en getPuntajePedido(), ya que este debería devolver null.

- Lo mismo ocurre cuando la cantidad de personas es mayor a la permitida, sigue calculando el puntaje del pedido, lo cúal es incorrecto.
- Al calcular el puntaje de un vehículo tipo Combi cuando el pedido solicita baúl, el valor retornado por el método getPuntajePedido() no coincide con el puntaje esperado. Esto indica que el cálculo dentro del método es incorrecto.

- **TestViaje:**
  - La prueba intenta validar el cálculo del costo de un viaje a zona estándar partiendo de un valor base 1000 . El cálculo del costo de un viaje estándar evidencia que el método getValor() devuelve un monto distinto al previsto, lo que sugiere que la fórmula no está correctamente aplicada.
  - El cálculo del valor del viaje con baúl también se realiza de manera incorrecta, otorgando un resultado distinto al esperado mediante el método getValor().

Test GUI:

En esta etapa del proyecto se testea el comportamiento de las ventanas de login, registro, administrador y cliente. Se estudia su comportamiento al ingresar datos, crear y eliminar objetos, la actualización de las listas y de los campos de texto.

- **test_GUI.TestAdminGestionPedidos:**

  En esta clase se implementa la gestión de los pedidos, es decir, la creación de pedidos y viajes.

  - **testListaChoferesLibresCorrecta:**
    - Se encontró un fallo en la ejecución de este método. El mismo verifica que la lista de choferes libres sea igual a la lista de choferes libres del singleton Empresa, el fallo se debe a que la JList tiene un modelo que no está correctamente ordenado.

- **test_GUI.TestAdminAltasChoferes:**

  En este documento se implementa la creación de choferes.

- **testButtonNuevoChofer_DNIexistente:**
  - En este método se encontró que se pueden crear dos choferes con el mismo DNI, ya que no lanza el mensaje de error correspondiente.
- **testListaChoferesDesocupados_correctaExistente:**
  - Se encontró que al crear un chofer con DNI repetido, el original es suplantado de la lista de choferes totales pero clonado en la lista de choferes desocupados, generando más choferes en la lista de desocupados que en la de totales, siendo que todos están desocupados.
- **testButtonNuevoChofer_hijosEmpty:**
  - El JTextField de hijos en el registro del chofer no se vacía al crear el chofer.
- **testButtonNuevoChofer_dniEmpty:**
  - El JTextField de DNI en el registro del chofer no se vacía al crear el chofer.
- **testListaChoferesDesocupados_correcta:**
  - La lista de choferes desocupados no se ordena correctamente al crear un nuevo chofer, por ende, es distinta a la lista de choferes desocupados de la empresa.
- **testButtonNuevoChofer_anioEmpty:**
  - El JTextField de año en el registro del chofer no se vacía al crear el chofer.
- **testButtonNuevoChofer_nameEmpty:**
  - El JTextField de nombre en el registro del chofer no se vacía al crear el chofer.

- **test_GUI.TestAdminAltasVehiculos:**

  En este documento se implementa la creación de vehículos.

  - **testAltaVehiculo_borraJTextPatente:**

        - El JTextField de patente en el registro del vehículo no se vacía al crear el vehículo.

    - **testAltaVehiculo_borraJTextPax:**

        - El JTextField de pasajeros en el registro del vehículo no se vacía al crear el vehículo.

- **test_GUI.TestCliente:**

  En este documento se implementan las acciones de la ventana cliente.

    - **testPagoExitoso_actualizacionTextCosto:**

        - El JTextField de costo en el pago del viaje no se vacía al pagar el viaje.

Test Controlador:

Se realizaron pruebas unitarias sobre la clase Controlador con el objetivo de verificar su lógica interna y su correcto manejo de excepciones de manera aislada del resto del sistema. Para garantizar este aislamiento, se empleó la biblioteca Mockito, simulando todas las dependencias externas (IVista, IPersistencia) y también la clase Empresa, creando una instancia falsa (mock) denominada empresaMock. Mediante el uso de reflexión, esta instancia simulada fue inyectada dentro del Singleton de Empresa, asegurando que el Controlador interactúe únicamente con objetos controlados por los tests.

Resultados y fallos detectados:

- **testLeerExcepcion y testEscribirExcepcion:**

Ambos tests fallaron debido a que el Controlador no contiene bloques try-catch para manejar las excepciones provenientes de la capa de persistencia.

Los tests simulaban que se arrojara una RuntimeException desde Persistencia, pero el Controlador dejaba que la excepción se propagara sin tratamiento. Esto contradice lo indicado en el Javadoc, donde se establece que este tipo de errores deben ser capturados y comunicados al usuario mediante la vista.

- **testNuevoChoferTemporarioRepetido:**

El test falló indicando que no hubo interacción con la vista. El problema se debe a que el bloque catch del Controlador para la excepción ChoferRepetidoException está vacío, por lo que, aunque la excepción es arrojada por Empresa y capturada por el Controlador, éste nunca muestra el mensaje correspondiente en pantalla. La falta de esta llamada provoca el fallo del verify del test.

- **testNuevoPedidoConExcepcion:**

El test esperaba que el Controlador mostrará un mensaje relacionado con un pedido pendiente, pero el mensaje emitido correspondía a un viaje pendiente.

Esto revela un error en el orden o contenido de la cadena de bloques try-catch del método nuevoPedido(), donde el Controlador está atrapando una excepción incorrecta, o bien está ordenando mal los catch, provocando que se capture antes una excepción más general.

- **testNuevoViajeConExcepcion:**

El test simula la excepción ChoferNoDisponibleException, pero el Controlador mostró el mensaje correspondiente a “Pedido inexistente”.

Esto indica que el Controlador está capturando una excepción equivocada, o que su estructura de manejo de errores no respeta el orden correcto entre las excepciones posibles.

Test Controlador – Integración:

Se ejecutó una batería de pruebas de integración para validar el funcionamiento conjunto del Controlador con una instancia real de Empresa.
Antes de cada test, se utilizó reflexión para resetear el Singleton de Empresa, asegurando un entorno limpio e independiente entre pruebas.

Resultados y fallos detectados:

- **testIntegracionLoginPasswordErronea:**

El Controlador mostró el mensaje “Usuario inexistente” cuando la contraseña era incorrecta. Esto confirma que la excepción PasswordErroneaException está siendo

tratada como UsuarioNoExisteException, lo cual coincide con los errores detectados en el Escenario 2 del Informe.

- **testIntegracionCrearPedidoFallaConPedidoPendiente:**

El Controlador emitió el mensaje “Cliente con viaje pendiente” cuando debía informar sobre un pedido pendiente. Esto revela un error en el método nuevoPedido(), donde la excepción ClienteConViajePendienteException está siendo atrapada antes o en lugar de la excepción ClienteConPedidoPendienteException.

- **testIntegracionNuevoChoferRepetido:**

El test indicó que no hubo interacciones con la vista, lo que demuestra dos fallas encadenadas:

1. Empresa no lanzó la excepción ChoferRepetidoException (bug del Escenario 2, donde se permite duplicar choferes).
2. El Controlador posee un bloque catch vacío para dicha excepción, por lo que tampoco muestra mensaje alguno.

Test Action Performed:

Al testear las acciones realizadas por el usuario, no encontramos fallos en la invocación de los métodos, pero al hacer pruebas de integración encontramos un total de 4:

- **testActionPerformed_NuevoChoferFallaRepetido:**

El test falló indicando que se esperaba una invocación al método ShowMessage conteniendo la palabra "Chofer", pero en realidad hubo cero interacciones con la vista (el objeto mock). Esto evidencia que, aunque el Modelo lanzó correctamente la excepción ChoferRepetidoException, el Controlador no comunicó el error a la Vista. La causa raíz es un bloque catch vacío o mal implementado en el método nuevoChofer del Controlador, que silencia la excepción e impide que el usuario reciba la notificación del error.

- **testActionPerformedPedidoFallaConPedidoPendiente:**

  Se obtuvo un error de aserción (AssertionError) con el texto "Error inesperado en el test: null". Esto demuestra que el test capturó una excepción dentro del flujo esperado, pero dicha excepción tenía un mensaje vacío o nulo. La causa es que la excepción ClienteConPedidoPendienteException se está instanciando sin un mensaje descriptivo en el Modelo, o bien el Controlador está relanzando una excepción genérica sin texto, haciendo imposible que la Vista muestre un mensaje de error válido al usuario.

- **testActionPerformed_NuevoViajeFallaChoferOcupado:**

  El test arrojó una discrepancia de argumentos: se esperaba que la vista mostrara un mensaje conteniendo la palabra "Chofer" (correspondiente a la excepción de chofer no disponible), pero el sistema devolvió el mensaje "El Pedido no figura en la lista". Esto indica que, al intentar crear un viaje con un chofer ocupado, el sistema falla prematuramente en la validación del pedido. La causa es un bug de lógica en el método crearViaje (posiblemente en la comparación de igualdad del objeto Pedido o en la búsqueda dentro de la colección de pedidos), que impide llegar a la validación de disponibilidad del chofer.

- **testActionPerformedNuevoViajeFallaClienteConViaje:**

  De manera similar al caso anterior, el test esperaba un mensaje conteniendo la palabra "viaje" (correspondiente a la excepción de cliente con viaje pendiente), pero el sistema recibió nuevamente el mensaje "El Pedido no figura en la lista". Esto confirma que el error de validación del pedido es bloqueante y tiene una prioridad errónea sobre otras reglas de negocio. La causa es el mismo defecto de lógica en la validación de existencia del pedido dentro del Modelo o Controlador, que está enmascarando el control de estado del cliente.

Test Persistencia:

- **TestPersistenciaBin:** Evalúa la capacidad del sistema para leer y escribir objetos en archivos binarios mediante la clase PersistenciaBIN.
  Se realizaron pruebas de escritura y posterior lectura de un objeto simple, comprobando que los datos recuperados coincidan exactamente con los almacenados. Además, se incluyeron casos de error, como la apertura de

archivos inexistentes o nulos, y el intento de cerrar flujos sin haberlos abierto previamente.

Las fallas que se encontraron fueron:

- Al intentar cerrar un flujo de salida (testCerrarSinAbrir_Output) sin haberlo abierto previamente, el sistema debe lanzar una IOException, lo cual no sucede. Es decir, no se está validando correctamente si el flujo de salida fue abierto antes de intentar cerrarlo.
- Se intenta cerrar un flujo de entrada (testCerrarSinAbrir_Input())sin haberlo abierto previamente, con el objetivo de que el sistema lance una IOException. El método cerrarInput() no genera ninguna excepción al ejecutarse. Esto significa que la clase PersistenciaBIN permite cerrar el flujo de lectura incluso cuando nunca fue inicializado.
- Se debe lanzar una IOException cuando se intenta abrir un archivo nulo. El método no lanza la excepción esperada, lanza una excepción diferente. No valida correctamente si la ruta del archivo es nula antes de intentar abrir el flujo de lectura.

- **TestEmpresaDTO:** La clase TestPersistenciaEmpresaDTO se centró en comprobar la correcta inicialización y manipulación del objeto EmpresaDTO, el cual actúa como intermediario para la persistencia de la información del sistema. Se verificó que, al instanciar un EmpresaDTO, todas sus colecciones (choferes, clientes, vehículos, pedidos y viajes) se inicializan correctamente y comiencen vacías.
  Luego, se realizaron pruebas sobre los métodos getters y setters, garantizando que cada estructura de datos mantenga la integridad de la información asignada, como los choferes desocupados, los vehículos disponibles, los pedidos activos y los viajes en curso o finalizados.
  De esta forma, se confirma que el DTO conserva correctamente el estado de la empresa previo a su almacenamiento.

- **TestUtilPersistencia:** Finalmente, la clase TestUtilPersistencia evalúa las funciones de transformación entre los objetos de negocio (Empresa) y su correspondiente representación en EmpresaDTO.
  En el primer caso, se comprobó que el método EmpresaDtoFromEmpresa() copie correctamente todas las estructuras de la empresa (choferes, clientes, vehículos, pedidos y viajes) al DTO.
  En el segundo, se validó que empresaFromEmpresaDTO() permita

reconstruir una instancia funcional de Empresa a partir de los datos contenidos en el DTO.

- testDeEmpresaDTOAEmpresa: Se valida que el método de conversión entre la clase Empresa y su correspondiente EmpresaDTO mantenga la información del usuario logueado. El test falla porque el valor del usuario activo en la instancia de Empresa no coincide con el almacenado en el EmpresaDTO luego de la conversión. Esto indica que durante el proceso de transferencia de datos entre ambos objetos no se está copiando correctamente el atributo del usuario logueado.

Test de excepciones:

- ChoferNoDisponibleException: Se lanza otra excepción (Escenario 4).
- ChoferRepetidoException: En el escenario 2 debería lanzarse la excepción, pero el sistema permite agregar choferes ya existentes.
- ClienteConPedidoPendienteException: En el escenario 2 y 4, el sistema lanza la excepción, pero con mensaje incorrecto.
- ClienteConViajePendienteException: No se lanza la excepción, el sistema crea el viaje igualmente (Escenario 3).
- ClienteNoExisteException: No presenta fallas.
- ClienteSinViajePendienteException: No presenta fallas.
- PasswordErroneaException: Se lanza UsuarioNoExisteException en su lugar (Escenario 2).
- PedidoInexistenteException: No presenta fallas.
- SinVehiculoParaPedidoException: No presenta fallas.
- SinViajesException: En el escenario 4 no se lanza la excepción; se permite calcular calificación sin viajes.
- UsuarioNoExisteException: No presenta fallas.
- UsuarioYaExisteException: No presenta fallas.
- VehiculoNoDisponibleException: En el escenario 3 se lanza una excepción distinta (no coincide con la esperada).
- VehiculoNoValidoException: No presenta fallas.
- VehiculoRepetidoException: Se lanza la excepción correcta, pero el objeto vehiculoExistente llega en null dentro de la excepción (Escenario 2).
- IOException: en la clase de prueba TestPersistenciaBIN no se lanza la excepción esperada. En particular, no se genera al intentar cerrar flujos

sin haberlos abierto, como en los métodos cerrarInput() y cerrarOutput(), ni al abrir un archivo nulo mediante abrirInput(null).

Conclusión:

El desarrollo y ejecución de las distintas baterías de pruebas permitieron evaluar de manera integral el funcionamiento del sistema “Subí que te llevo”. A través de pruebas de Caja Negra, Integración, Controlador, Persistencia y excepciones, se logró identificar una serie de errores importantes que afectan la lógica del negocio, la validación de datos y el manejo adecuado de excepciones.

Las pruebas sobre el modelo de datos permitieron identificar errores de cálculo y condiciones mal validadas, como puntajes incorrectos, antigüedades mal limitadas y verificaciones incompletas en las clases Auto, Combi, ChoferPermanente y Viaje.

Las pruebas de integración sobre la clase Empresa mostraron que muchos métodos dependen de cómo interactúan entre sí los distintos objetos del sistema. Esto hizo aparecer errores que no se ven en pruebas más simples, como validaciones que no se hacen correctamente, excepciones que no se lanzan cuando deberían o que se lanzan mal, y situaciones que no tendrían que pasar, como permitir choferes duplicados o crear pedidos y viajes en momentos en los que no corresponde.

El análisis del Controlador evidenció errores en el manejo de excepciones, bloques catch vacíos y mensajes incorrectos devueltos a la interfaz, lo que confirma que su lógica no refleja adecuadamente los escenarios planteados en el dominio del problema. Estos fallos fueron detectados tanto en las pruebas unitarias con mocks como en las pruebas de integración con la Empresa real.

En relación a la persistencia, se observó un funcionamiento parcial: aunque los procesos de lectura y escritura funcionan en condiciones normales, las excepciones de E/S no se manejan correctamente. Métodos como cerrarInput(), cerrarOutput() o abrirInput(null) no lanzan las IOException esperadas, lo que indica que la clase PersistenciaBIN carece de validaciones previas esenciales para evitar operaciones inválidas.

Por último, las pruebas de GUI permitieron verificar cómo responden las distintas ventanas ante las acciones del usuario. Mediante la simulación con la clase Robot se detectaron problemas propios de la interfaz, como campos que no se vaciaban y listas que no se actualizaban o se ordenaban de forma incorrecta. Estas pruebas complementaron al testing interno, mostrando cómo los errores en la lógica afectan directamente la interacción visual del sistema.

En general, este trabajo nos permitió comprender cómo interactúan las distintas capas del sistema, cómo un error interno puede propagarse hacia el controlador o la interfaz, y por qué el testing es fundamental para detectar estos problemas antes de que lleguen al usuario final.La experiencia nos mostró lo importante que es hacer pruebas completas, revisar tanto los casos correctos como los incorrectos y analizar bien la lógica del programa para que el sistema funcione de forma segura y cumpla realmente con lo que se espera.