# Taller de Programación I

# Informe Trabajo Práctico


## Profesores:

- **Adolfo Tomas Spinelli**
- **Guillermo Lazzurri**
- **Bonifazzi Aquino Paula**

## Integrantes:

- **Agurto, Gonzalo**
- **Finocchio, Fausto**
- **Gaillour, Juan**
- **Ratigan, Iván**

# Introducción

El testing es un proceso fundamental que busca verificar y validar la funcionalidad de una aplicación de software. Su objetivo principal es asegurar que el producto final se entregue con la menor cantidad posible de defectos. Este proceso implica la verificación dinámica del comportamiento del programa, la cual se realiza ejecutando un conjunto finito de casos de prueba. Estos casos deben ser cuidadosamente seleccionados del dominio de ejecución para contrastar los resultados obtenidos con el comportamiento esperado. Contrario a una visión tradicional, el testing no es una fase final, sino una actividad que se ejecuta en paralelo al ciclo de desarrollo del software.

El presente informe se centra en la aplicación de diversas pruebas funcionales, abarcando Pruebas Unitarias de Caja Negra, Test de Persistencia, Test de Interfaces Gráficas de Usuario (GUI) y Test de Integración. Todas estas técnicas se demostrarán sobre un sistema de gestión de viajes.

# Desarrollo

## Test Unitario de Caja Negra

### Package Modelo.Datos

A continuación se detalla el alcance y los resultados de las pruebas realizadas sobre las entidades del modelo de negocio. Es fundamental resaltar que se detectaron errores funcionales y discrepancias en los resultados esperados únicamente en las clases Auto, Combi, Chofer Permanente y Viaje. El resto de las entidades superó las validaciones satisfactoriamente.

#### Administrador

Las pruebas se concentraron en garantizar la integridad del patrón Singleton. Se verificó que el sistema mantenga una única instancia global compartida durante todo el ciclo de vida de la aplicación y se validó la imposibilidad de violar este patrón mediante intentos de instanciación directa.
➕ Baterías de Prueba

#### Cliente

El foco estuvo en asegurar que los atributos críticos para la autenticación y perfil (usuario, contraseña y nombre real) cumplan con los criterios de no nulidad y contenido no vacío.
➕ Baterías de Prueba

#### Chofer Permanente

Esta clase requirió un análisis detallado debido a la complejidad de su fórmula salarial. Se realizaron pruebas para validar el cálculo del sueldo bruto integrando variables dinámicas: el sueldo básico, el plus por antigüedad (verificando el cálculo correcto de años transcurridos desde la fecha de ingreso) y el adicional por cantidad de hijos. Además, se incluyeron controles

de límites para evitar la consistencia de datos, como la asignación de una cantidad negativa de hijos.

Errores
- El sistema no respeta el máximo de antigüedad. Sigue aumentado el Sueldo Bruto pese a sobrepasar los años máximos de antigüedad establecidos.

➕ Baterías de Prueba

**Chofer Temporario**

Al tener una lógica de negocio simplificada, las pruebas se limitaron a validar la corrección de los datos personales en el constructor y a certificar que el método de cálculo de sueldo devuelve consistentemente el valor fijo estipulado, sin verse afectado por variables externas como la antigüedad.
➕ Baterías de Prueba

**Auto**

El trabajo de prueba en esta entidad se enfocó en la capacidad de transporte y la fórmula de puntuación. Se verificó estrictamente que el vehículo solo acepte entre 1 y 4 pasajeros. Respecto a la lógica de negocio, se validó cómo varía el algoritmo de puntuación dependiendo de si el viaje requiere uso de baúl o no, asegurando que se apliquen los multiplicadores correctos y que el vehículo rechace solicitudes que excedan su capacidad física.
Errores
- El sistema está sobreestimando el puntaje para viajes simples (sin uso de baúl)

➕ Baterías de Prueba

**Combi**

Se validaron los rangos de capacidad intermedios (de 5 a 10 pasajeros) para diferenciarla de otros vehículos. Las pruebas sobre el método de puntuación se diseñaron para confirmar la aplicación de un costo fijo adicional cuando se solicita el uso del baúl, validando que la fórmula matemática responda adecuadamente a esta variable booleana específica.
Errores
- El sistema aceptó un pedido que excedía la cantidad de plazas disponibles del vehículo
- El cálculo del puntaje cuando se utiliza el baúl es significativamente menor al esperado

➕ Baterías de Prueba

**Moto**

Dadas las limitaciones físicas de este vehículo, las pruebas fueron restrictivas y orientadas a escenarios de rechazo. Se verificó exhaustivamente que el método de puntuación devuelva null (indicando inviabilidad) ante cualquier solicitud que incluya uso de baúl, transporte de mascotas o más de un pasajero, garantizando que solo se asignen viajes unipersonales simples.
➕ Baterías de Prueba

**Pedido**

Se probó que la solicitud esté vinculada a un cliente existente, que la cantidad de pasajeros sea lógica (mayor a uno, salvo excepciones de moto gestionadas aparte) y, fundamentalmente, que la zona del viaje corresponda estrictamente a las constantes definidas (Standard, Sin Asfaltar o Peligrosa), ya que este dato es crucial para la facturación posterior.
Baterías de Prueba

**Viaje**

Esta fue la clase con mayor carga de trabajo en términos de diseño de pruebas debido a su alta complejidad combinatoria. Se implementó una estrategia de tablas de decisión para validar el método de cálculo del valor final (getValor). Las pruebas cruzaron múltiples variables simultáneamente: el tipo de zona, la presencia de mascotas y uso de baúl, y el tipo de vehículo asignado. El objetivo fue asegurar que la fórmula financiera integre y pondere correctamente todos estos factores para arrojar el importe exacto a cobrar.
Errores
- El Sistema no aumenta el valor del viaje si la Zona es Standard.
- El Sistema no aumenta el valor del viaje cuando el vehículo tiene Baul.

Baterías de Prueba

**Package Modelo.Negocio**
*Clase Empresa*
Para los test unitarios del package modelos implementamos todas las pruebas en el test de la clase Empresa, ya que es la única presente en este package. Planteamos 5 escenarios, que, como iban a ser reutilizados en el futuro test de ventanas, los convertimos en clases.

Escenarios:
En el siguiente recuadro ilustraremos en qué estado se encuentran las listas de elementos en cada escenario:

| Nro Escenario | Clientes | Choferes | Vehículos | Pedidos | Viajes en curso | Viajes finalizados |
|---|---|---|---|---|---|---|
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |

Como vemos estos están diseñados de tal manera que cada uno plantea un “paso” más

en la utilización del sistema respecto al escenario anterior. Cabe aclarar también que contamos con el escenario 0 el cual es un escenario vacío y todos los demás se extienden de este, pero también cuenta con un método el cual rellena las listas de elementos básicos del sistema (clientes, choferes y vehículos).

**Paquete Modelo de Negocio**

- Baterías de Prueba

**Los errores encontrados en los distintos escenarios son los siguientes:**

Escenario 0:

- El total de salarios calculado con 1 chofer en la lista difiere del esperado.
- El total de salarios calculado con varios choferes en la lista difiere del esperado.
- Al intentar agregar un nuevo chofer con parámetros ya ingresados registrados en otro chofer se espera que se lance la excepción ChoferRepetidoException, pero esta no se lanza.

Escenario 1:

- No se encontraron errores.

Escenario 2:

- El metodo vehiculosOrdenadosPorPedido() devuelve los vehículos disponibles, pero estos no están ordenados correctamente ya que el puntaje que se calcula no es el debido.
- Se creó un pedido de 10 pasajeros teniendo únicamente una combi de 9 plazas pero el arrayList que devolvió vehiculosOrdenadosPorPedido() no estaba vacío, como debería.
- No se lanza la excepción SinVehiculoDisponible al crear un pedido de 10 pasajeros en una combi de capacidad 9. Deducimos que se debe al error en vehiculosOrdenadosPorPedido().

Escenario 3:

- Cuando no hay choferes disponibles se lanza la excepción esperada, pero en vez de contener el mensaje “El chofer no está disponible”, la excepción contenía “El pedido no figura en la lista”.
- Cuando un cliente que tiene un pedido pendiente realiza otro pedido, se lanza la excepción esperada, ClienteConPedidoPendienteException, pero el mensaje obtenido no corresponde a dicha excepción: “Cliente con VIAJE pendiente”

Escenario 4:

- Se crea un viaje con un cliente con un viaje ya en curso, se esperaba el lanzamiento de ClienteConViajePendienteException al crear un viaje con un cliente que ya tiene un viaje en curso pero no se lanzó excepción alguna.

- Se crea un pedido con un cliente con un viaje ya en curso, se esperaba el lanzamiento de ClienteConViajePendienteException al agregar otro pedido con el mismo cliente, pero esta no se lanza.

Escenario 5:

- Después de finalizar 9 viajes se solicita la calificación de un chofer el cual no había realizado ninguno, se esperaba un SinViajesException al intentar obtener la calificación de un chofer sin viajes pero no se lanzó ninguna excepción.
- El promedio de calificación de un chofer difiere del esperado.

## Test de Excepciones

Durante el proceso de prueba se evaluó el correcto lanzamiento de las excepciones definidas por el sistema, verificando que cada una se active en los escenarios que corresponden según la lógica de negocio.

A continuación se detallan los resultados obtenidos.

**Excepciones que funcionan correctamente**

Las siguientes excepciones se lanzan en los escenarios adecuados, cumpliendo con las precondiciones establecidas y deteniendo correctamente el flujo cuando ocurre una situación inválida:

ChoferNoDisponibleException.
ClienteNoExisteException
ClienteSinViajePendienteException
PedidoInexistenteException
PasswordErroneaException
UsuarioNoExisteException
UsuarioYaExisteException
SinVehiculoParaPedidoException
VehiculoRepetidoException
VehiculoNoDisponibleException
VehiculoNoValidoException

**Excepciones que NO funcionan**

Las siguientes excepciones no se lanzan correctamente o no cubren los escenarios para los cuales fueron diseñadas:

ChoferRepetidoException:

- Problema detectado:
  La excepción solo se lanza cuando se intenta registrar *exactamente el mismo objeto*

*Chofer*.

- Situación esperada:
  Debe lanzarse cuando ya existe otro chofer con el mismo DNI, aunque sea una instancia distinta.

- Impacto:
  El sistema permite registrar múltiples choferes duplicados por DNI, comprometiendo la integridad de los datos.

ClienteConViajePendienteException:

- Problema detectado:
  La excepción nunca se lanza.

- Situación esperada:
  Debe lanzarse cuando un cliente intenta iniciar un nuevo viaje teniendo aún un viaje pendiente sin finalizar.

- Impacto:
  El sistema permite acumular viajes simultáneos para un mismo cliente, lo cual viola las reglas de negocio previstas.

SinViajesException

- Problema detectado:

  La excepción no se lanza cuando se solicita la calificación de un chofer que no ha realizado ningún viaje.

- Situación esperada:

  Debe lanzarse cuando un chofer no tiene viajes finalizados y se intenta obtener su calificación, ya que no existen valores válidos para calcular un promedio.

- Impacto:

  El sistema devuelve una calificación incorrecta o un valor por defecto, permitiendo realizar operaciones sobre un chofer sin historial de viajes.
  Esto genera inconsistencias en el cálculo de calificaciones, oculta errores lógicos y viola las reglas de negocio que requieren al menos un viaje para poder evaluar al chofer.

ClienteConPedidoPendienteException

- Problema detectado:

La excepción se lanza exitosamente, pero en lugar de contener el mensaje "El chofer no está disponible", la excepción contiene "El pedido no figura en la lista", lo cual no representa correctamente la situación..

- Situación esperada:

  La excepción debe lanzarse cuando un cliente que ya tiene un pedido pendiente intenta generar uno nuevo.

  En ese caso, el mensaje correcto debe ser: "El chofer no está disponible".

- Impacto:

  El sistema presenta un comportamiento erróneo al comunicar un mensaje incorrecto, lo que puede confundir al desarrollador o al usuario, dificultando la identificación del problema real.
  Además, el mensaje inapropiado puede llevar a errores lógicos o de validación, ya que no refleja el motivo real por el cual se rechaza el nuevo pedido.

## Test de Persistencia

El test de persistencia consiste en validar que los datos almacenados en un medio de persistencia, ya sea una base de datos o un archivo binario, puedan ser recuperados de manera correcta y consistente.
El objetivo principal es asegurar que las operaciones de escritura y lectura funcionen adecuadamente, garantizando que la información no se pierda, no se corrompa y mantenga su integridad a lo largo del ciclo de vida del sistema.

Las pruebas se realizaron contemplando escenarios específicos del archivo "empresa.bin":

1. Archivo no existente
2. Archivo existente pero vacío
3. Archivo existente con elementos en sus colecciones

Los métodos testados fueron:

- EmpresaDTO EmpresaDtoFromEmpresa(Empresa empresa)
- void empresaFromEmpresaDTO(EmpresaDTO dto)
- abrirInput(String nombre)
- abrirOutput(String nombre)

- cerrarInput()
- cerrarOutput()
- escribir(Serializable serializable)
- Serializable leer()

**Paquete Persistencia**

- UtilPersistencia**:** **Baterías de Prueba - Hojas de cálculo de Google**
- EmpresaDTO y PersistenciaBin: **Baterías de Prueba - Hojas de cálculo de Google**

Los resultados fueron consistentes con lo esperado, cubriendo la totalidad de las clases de equivalencia definidas.

Las pruebas realizadas permiten concluir que:

- El módulo de persistencia funciona correctamente bajo condiciones normales.
- El sistema maneja adecuadamente los errores, lanzando las excepciones correspondientes en cada caso.
- La técnica de partición en clases de equivalencia permitió cubrir todos los escenarios relevantes sin redundancia.
- La conversión entre objetos de dominio y DTO se realiza de forma completa y sin pérdidas de información.
- El sistema garantiza que los datos escritos en el archivo binario puedan ser recuperados de manera íntegra.

En consecuencia, el módulo de persistencia se considera validado para su uso dentro del proyecto.

**Tests de Integración del Controlador**

El objetivo principal de los tests de integración fue validar la correcta interacción entre la capa de Controlador y la capa de Modelo (específicamente, la clase Empresa).
Se buscó verificar que el Controlador respondiera correctamente a los eventos simulados de la vista, invocando los métodos de negocio adecuados en Empresa y manejando las respuestas (tanto exitosas como de excepción).

El foco de las pruebas fue la clase controlador.Controlador. Para aislar esta capa y probarla de forma efectiva, se implementó la siguiente estrategia:

**Simulación de la Vista (Mocking):** Se utilizó el framework Mockito para crear un mock (un objeto simulado) de la IVista. Esto nos permitió simular las acciones del usuario (ej. clics en botones, ingreso de texto) sin depender de la GUI real.

**Captura de Excepciones:** Para validar los casos de error (ej. "usuario repetido", "chofer no disponible"), se creó una clase FalseOptionPanel. Esta clase implementa la interfaz IOptionPane y nos permitió interceptar los mensajes de error que el controlador intentaba mostrar al usuario. De esta forma, pudimos usar aserciones (Assert.assertEquals) para verificar que se estaba reportando el mensaje de excepción correcto.

Se definieron dos escenarios base para probar el comportamiento del Controlador bajo diferentes condiciones del sistema:

**Escenario 1:** Estado Vacío

Un escenario de "caja blanca" inicial donde todas las listas del modelo (Empresa) están vacías.

- listaChoferes: Vacía
- listaVehiculos: Vacía
- listaClientes: Vacía
- listaPedidos: Vacía
- listaViajesIniciados: Vacía
- listaViajesTerminados: Vacía

**Escenario 2:** Estado con Datos

Un escenario complejo pre-cargado, diseñado para probar la lógica de negocio, colisiones y excepciones:

- **Clientes (3):**
  - Cliente 1: Con un pedido pendiente.
  - Cliente 2: Sin pedidos ni viajes.
  - Cliente 3: Con un viaje iniciado.
- **Choferes (2):**
  - Chofer 1: Permanente, disponible.
  - Chofer 2: Temporario, con un viaje iniciado (no disponible).
- **Vehículos (3):**
  - Vehículo 1: Auto (4 plazas, admite mascota), disponible.
  - Vehículo 2: Combi (6 plazas, no admite mascota), disponible.
  - Vehículo 3: Moto, con un viaje iniciado (no disponible).
- **Pedidos (1):**
  - 1 pedido pendiente (asociado al Cliente 1).
- **Viajes (1):**

    - 1 viaje iniciado (asociado al Cliente 3, Chofer 2 y Moto).
- **Viajes Terminados:**
    - Lista vacía.

## Test de GUI

### Tests de Interfaz Gráfica (GUI)

El objetivo de las pruebas de GUI es validar la funcionalidad de la aplicación desde la perspectiva del usuario final. Este enfoque de "caja negra" verifica el flujo completo del sistema, asegurando que los elementos visuales (paneles, botones, campos de texto) y las interacciones del usuario funcionen como se espera.

Para lograr esto, se utilizó la clase Robot (del paquete de Java AWT). Esta herramienta nos permitió crear tests automatizados y repetibles al simular programáticamente las interacciones del usuario, como clics de mouse y entradas de teclado. De esta forma, se pudo testear la comunicación completa del sistema:

1. Vista (GUI): El Robot simula un clic en un botón.
2. Controlador: Recibe el evento de la vista.
3. Modelo (Empresa): El controlador invoca la lógica de negocio.
4. Vista (GUI): El test verifica que la interfaz se actualice correctamente como resultado de esa lógica (mostrando un panel nuevo, un mensaje de error, o deshabilitando un botón).

### Resultados y Análisis de Fallas en Tests de GUI

Para asegurar la robustez del sistema, todas las pruebas de la lógica de negocio se ejecutaron contra un conjunto de seis escenarios de prueba predefinidos. Estos escenarios fueron diseñados en el testeo de modelo de negocio, abarcando desde un estado inicial del sistema completamente vacío hasta configuraciones complejas con múltiples usuarios, vehículos y viajes en progreso, permitiendo así validar el comportamiento de la aplicación bajo condiciones controladas y realistas.

Se ejecutó la suite de 128 pruebas de GUI, reportando un total de 16 fallas (Failures) y 0 errores (Errors).

El análisis de las 16 fallas revela patrones consistentes que pueden agruparse en las siguientes categorías:

### Fallas de Mensajes de Error (ComparisonFailure)

Varios tests fallaron porque el mensaje de error mostrado al usuario no coincidía con el texto esperado por la prueba. En la mayoría de estos casos, se esperaba un mensaje de error específico (ej. "Usuario repetido"), pero el valor que se obtuvo fue null.

- testPanelRegistro_SegundoAdm: expected:<Usuario repetido> but was:<null>
- testPanelCliente_NuevoPedido_SinVehiculoParaPedido: expected:<Ningun vehiculo puede satisfacer el pedido> but was:<null>
- testPanelAdm_AltaChofer_YaRegistrado: expected:<Chofer Ya Registrado> but was:<null>
- testPanelLogin_AdmPasswordIncorrecta: expected:<[Password incorrecto]> but was:<[Usuario inexistente]>
- testPanelLogin_Titulo: expected:<[user1]> but was:<[]>

Análisis: Estas fallas sugieren que el FalseOptionPanel (usado para capturar los mensajes) no está recibiendo el mensaje del controlador, o que el controlador está lanzando una excepción diferente a la esperada (como en el caso de "Password incorrecto").

**Fallas en el Estado de Componentes (Habilitado/Deshabilitado)**

Estos tests verifican que ciertos componentes se habiliten o deshabiliten según la lógica de la aplicación (ej. un botón "Aceptar" debe estar deshabilitado si el formulario está incompleto).

- testPanelCliente_PagarViaje_TextFieldCalificacion: Falló la aserción "El campo Calificacion deberia estar deshabilitado".
- testPanelCliente_PagarViaje_TextFieldValor: Falló la aserción "El campo Valor deberia estar deshabilitado".
- testPanelAdm_AltaVehiculo_AutoCantPlazasFueraDeRango: Falló la aserción "El boton de Aceptar Vehiculo deberia estar deshablitado".

Análisis: Indican un error en la lógica de la vista, que no está actualizando correctamente el estado setEnabled(false) de los componentes.

**Fallas en el Reseteo de Formularios**

Un grupo de tests falló porque, después de una operación de "Alta" (como registrar un chofer o vehículo), los campos de texto del formulario no se limpiaron.

- testPanelAdm_AltaChofer_TexFieldDNI: "El campo DNI del Chofer deberia vaciarse".
- testPanelAdm_AltaChofer_TexFieldNombreChofer: "El campo del Nombre del Chofer deberia vaciarse".
- testPanelAdm_AltaVehiculo_TextFieldPatente: "El campo Patente deberia estar vacio".
- *(...y 5 tests similares para CantPlazas, CantHijos, AnioDeIngreso, y Valor).*

Análisis: La funcionalidad de "Alta" parece estar funcionando, pero la vista no está reseteando los JTextField a "" (vacío) después de una inserción exitosa.

**Falla en Lógica de Negocio (Estado de Colección)**

Se detectó una falla crítica que expone un error en la lógica de negocio (Modelo).

- testPanelAdm_AltaChofer_YaRegistradoListaChoferesLibres: "La Lista de Choferes Libres en el JList ha aumentado, se ha registrado un Chofer REPETIDO expected:<7> but was:<8>"

## Resultados

Los resultados de los Testeos se encuentran en:
**subiquetellevoTest-main\AllTest\AllTest.html**

Fue utilizado el comando:
**mvn clean site -Dtest=AllTest**
En la caja de comandos ubicada en la carpeta **subiquetellevoTest-main**

## Conclusión

El trabajo de testing realizado sobre el sistema "Subí que te llevo" nos permitió evaluar el estado real de la aplicación, aplicando estrategias que fueron desde pruebas unitarias hasta la automatización de la interfaz gráfica.

Los resultados mostraron dos realidades muy distintas. Por un lado, la Persistencia funcionó muy bien: el guardado y la recuperación de datos son sólidos y no presentaron fallas. Sin embargo, encontramos problemas importantes en la Lógica de Negocio. Detectamos errores de cálculo en los sueldos de los choferes y en los precios de los viajes, además de varias excepciones que no se lanzan cuando deberían, lo que podría dejar al sistema en un estado inconsistente.

En cuanto a la Interfaz Gráfica (GUI), las pruebas automatizadas revelaron que al usuario muchas veces le falta información: hay mensajes de error que no aparecen y campos que no se limpian o deshabilitan correctamente después de una acción.

En resumen, aunque la estructura para guardar datos es correcta, la aplicación necesita varias correcciones funcionales y visuales antes de poder considerarse lista para su uso real.