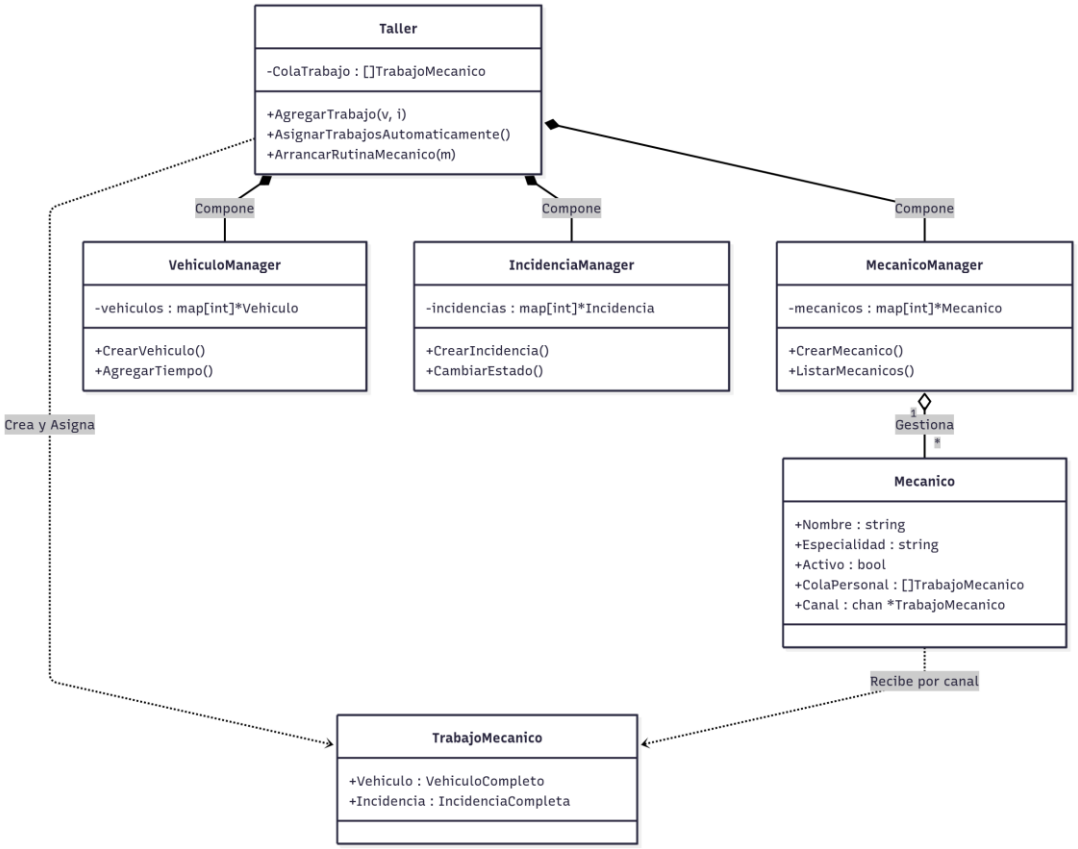
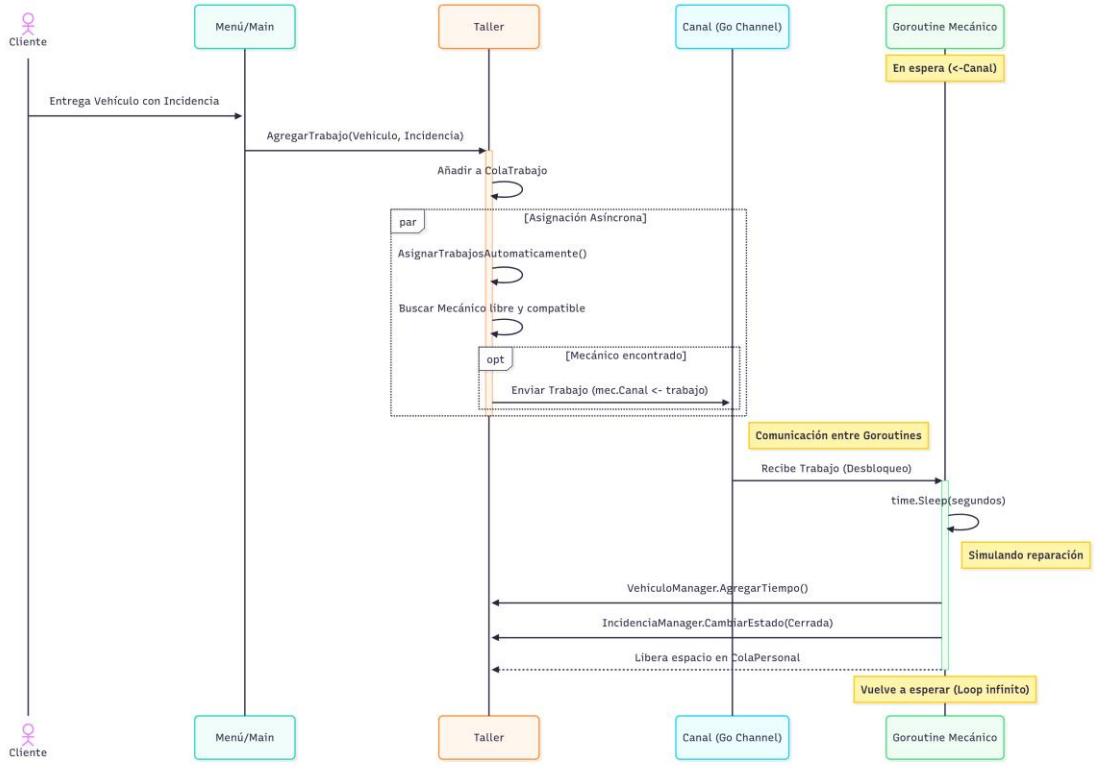
Práctica 3 – El Taller del pueblo

1. Diagrama de Clases



Este diagrama muestra cómo está organizado el taller para que todo funcione correctamente. La clase **Taller** actúa como el "jefe" principal que reparte el trabajo, pero se apoya en unos “ayudantes” llamados **Managers** para tener bien organizadas las listas de vehículos, incidencias y empleados. Lo más importante es la parte del **Mecánico**: el diagrama enseña que cada mecánico tiene un canal especial para recibir tareas. Cuando hay trabajo, el taller empaqueta el coche y el problema en una ficha llamada *TrabajoMecanico* y se la envía al mecánico por ese canal, permitiendo que cada uno trabaje a su ritmo sin liarse con los demás.

2. Diagrama de Secuencia

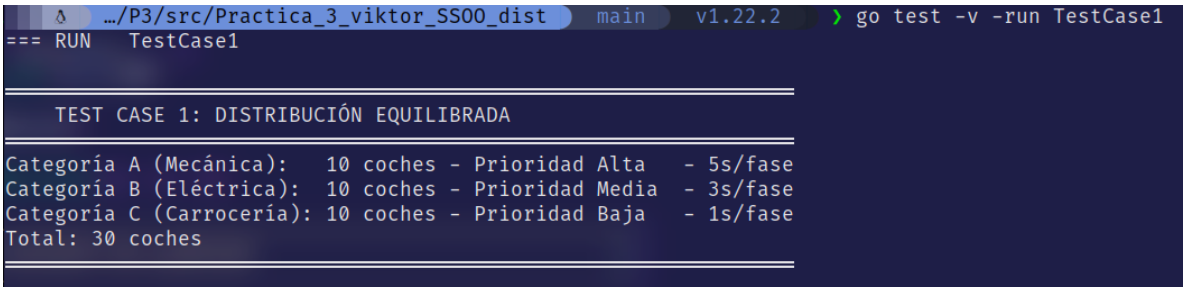


En este diagram vemos como funciona el flujo de trabajo, todo empieza cuando el **Cliente** entrega su coche. El **Taller** toma nota y busca un mecánico libre. Para pasar la tarea se usa un **Canal** (que funciona como un buzón directo). El Taller deja ahí el trabajo y el **Mecánico**, que estaba esperando, lo recoge al instante. Entonces, el mecánico simula que trabaja durante un tiempo y, al terminar, actualiza el estado del vehículo y queda libre para recibir el siguiente coche.

3. Tests

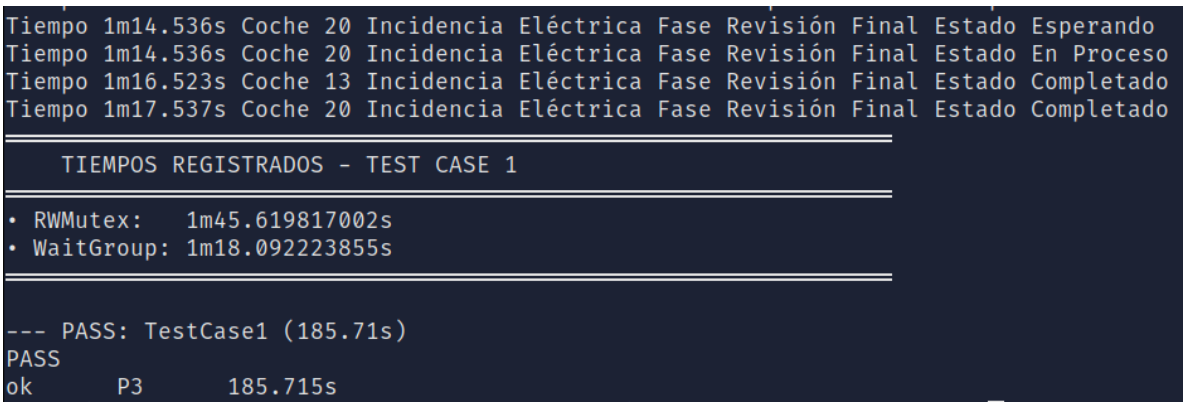
Test 1 → *Distribución equilibrada*

Test_1_Distribucion_Equilibrada



- Este escenario evalúa el rendimiento del sistema bajo una carga de trabajo uniforme. Se procesaron **30 vehículos** divididos equitativamente entre las tres categorías (10 de prioridad Alta, 10 Media y 10 Baja), lo que permite observar el comportamiento de la concurrencia sin sesgos por tipos de reparación

Test_1_Comparacion



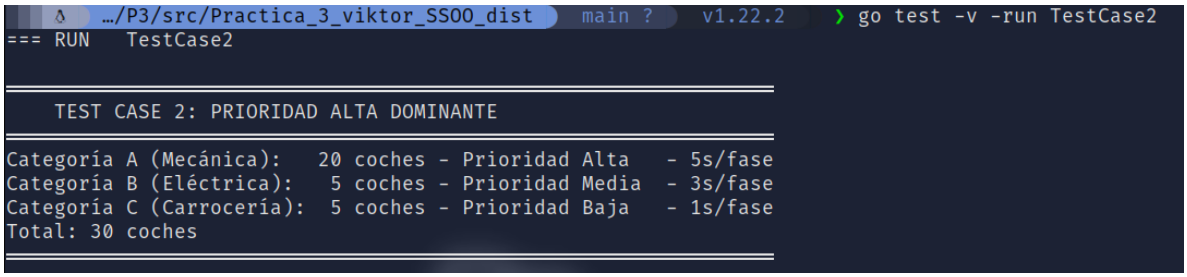
- Una vez acabado el test, vemos que este reveló una diferencia notable en los tiempos de ejecución. La implementación basada en ***WaitGroup*** completó el ciclo en **1m 18s**, siendo considerablemente más eficiente que la versión con ***RWMutex***, que tardó **1m 45s**.

  Estos datos sugieren que, en situaciones de carga balanceada donde el flujo de vehículos entre fases es constante, el uso de ***WaitGroup*** genera menos tiempos de espera internos (menor sobrecarga) que la

gestión de bloqueos de lectura/escritura (***RWMutex***). El sistema coordinó las goroutines de manera más fluida con la sincronización simple.

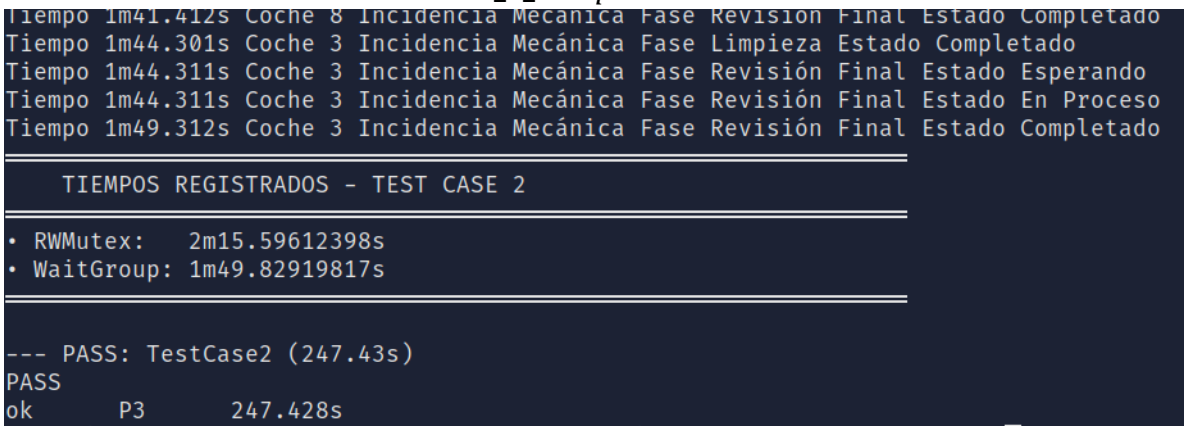
Test 2 → *Mayor número de vehiculos con prioridad alta*

Test_2_Prio_Alta



- Este escenario simula una situación de **alta carga computacional**. Se introdujeron 30 vehículos, pero con una distribución sesgada: **20 vehículos de prioridad Alta** (Mecánica, 5s/fase), frente a solo 5 de Media y 5 de Baja. Esto implica que el 66% de las tareas tienen la duración máxima posible dentro de la simulación

Test_2_Comparación



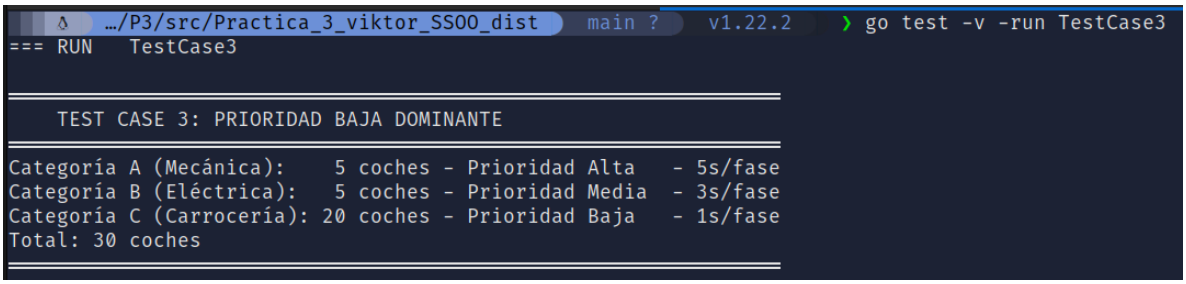
- La brecha de rendimiento se amplió significativamente en comparación con el caso equilibrado. La implementación con

RWMutex registró un tiempo total de **2m 15s**, mientras que la versión con ***WaitGroup*** finalizó en **1m 49s**, logrando una mejora de más de 25 segundos.

Al predominar las tareas de larga duración (reparaciones de 5 segundos), los recursos del taller permanecen ocupados durante periodos más extensos. En este contexto, la sobrecarga (overhead) que introduce la gestión de bloqueos ***RWMutex*** se hace más evidente, ya que las goroutines pasan más tiempo compitiendo por el acceso a las estructuras de datos compartidas. El modelo ***WaitGroup*** demostró ser más ágil gestionando la concurrencia cuando el sistema está saturado por procesos lentos.

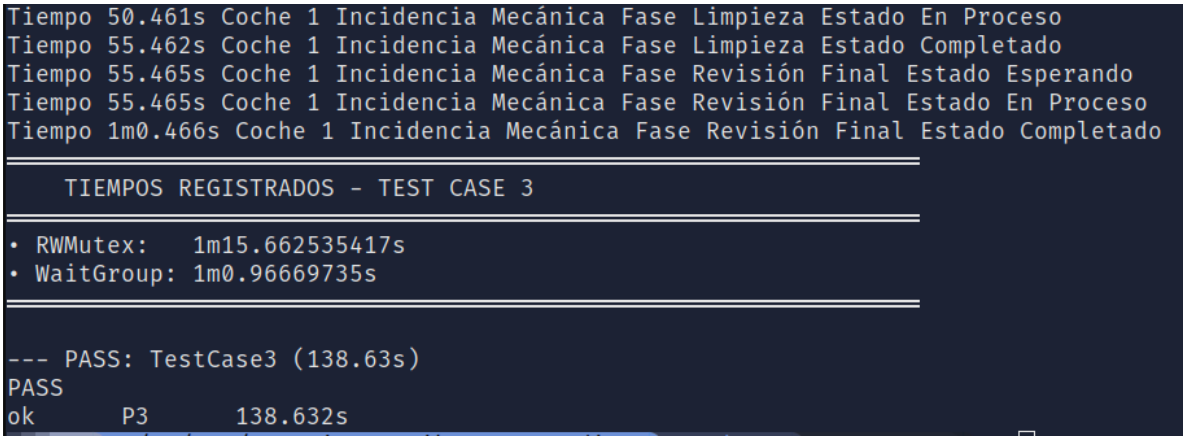
Test 3 → *Mayor número de vehiculos con prioridad baja*

Test_3_Prio_Baja



- Este último escenario plantea una carga de trabajo inversa a la anterior, caracterizada por un alto volumen de tareas breves. De los 30 vehículos, **20 pertenecen a la categoría de Carrocería** (Prioridad Baja, 1s/fase), mientras que las reparaciones complejas (Mecánica y Eléctrica) se reducen al mínimo.

Test_3_Comparación



- Vemos que incluso con tareas de corta duración, la implementación con ***WaitGroup*** mantuvo su superioridad, registrando un tiempo de **1m 00s**. Por su parte, la versión con ***RWMutex*** necesitó **1m 15s** para finalizar el procesamiento de todos los vehículos.

  Este caso demuestra que el coste de gestión de los semáforos (***RWMutex***) es constante y acumulativo. Al tener muchas tareas rápidas (1 segundo), la frecuencia con la que las goroutines intentan adquirir y liberar los bloqueos aumenta drásticamente. Esta "contención" genera un cuello de botella administrativo que no existe en la implementación con ***WaitGroup***, la cual gestiona el alto flujo de tareas cortas con mucha mayor agilidad.

Puedes encontrar el repositorio del proyecto en GitHub

(pulsar "GitHub" para acceder al repositorio)