Memoria Técnica: Práctica 4 SDIS

Sistema de Gestión de Taller con Concurrencia en Go

Práctica 4 - Sistemas Distribuidos

8 de enero de 2026

Índice

1. Introducción

Este documento presenta la memoria técnica de la Práctica 4 de Sistemas Distribuidos, en la que se ha desarrollado un sistema concurrente en Go para la gestión de un taller mecánico. El sistema modela el flujo completo de reparación de vehículos, desde su entrada hasta su entrega, utilizando mecanismos avanzados de concurrencia como goroutines, canales y primitivas de sincronización.

El proyecto implementa un modelo cliente-servidor donde:

- **Servidor**: Gestiona las conexiones y actúa como broadcaster de información.
- **Cliente (Taller)**: Implementa la lógica del taller con múltiples fases de procesamiento.
- **Mutua**: Programa externo que envía comandos de cambio de estado al sistema.

2. Arquitectura del Sistema

2.1. Estructura General

El sistema está organizado en tres módulos principales:

- **servidor.go**: Servidor TCP que gestiona conexiones y broadcasting.
- **taller.go**: Cliente principal que implementa la lógica del taller.
- **mutua.go**: Cliente simulador que envía comandos de cambio de estado.

Adicionalmente, el módulo `cliente` contiene los siguientes archivos de soporte:

- **types.go**: Definiciones de tipos de datos y constantes.
- **utility.go**: Funciones auxiliares y lógica de gestión de prioridades.
- **goroutines.go**: Funciones que implementan las goroutines del sistema.
- **mutex.go**: Métodos protegidos por RWMutex para acceso concurrente.
- **taller_test.go**: Suite de tests para validación del sistema.

2.2. Diagrama de la Arquitectura

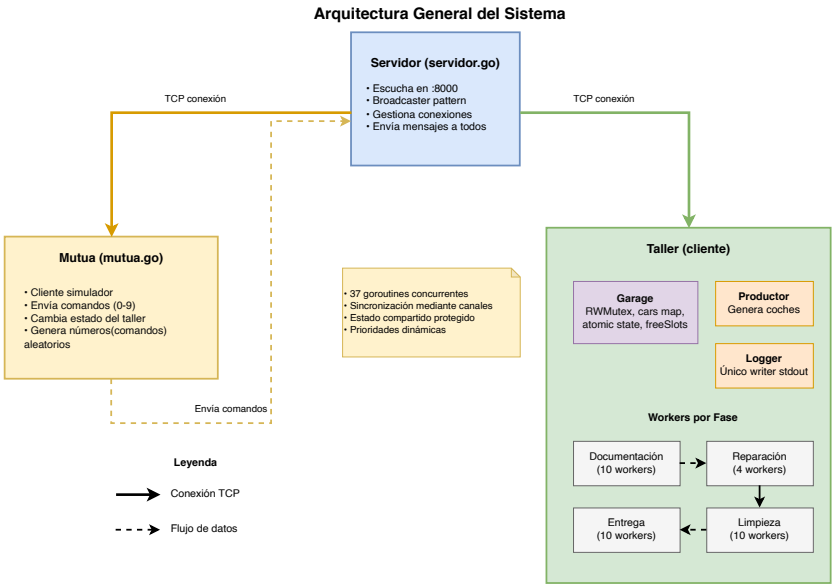

Figura 1: Diagrama de arquitectura del sistema

3. Descripción Detallada de Componentes

3.1. Servidor (servidor.go)

El servidor implementa un patrón broadcaster simple pero efectivo:

- Escucha en `localhost:8000`
- Mantiene un mapa de clientes conectados
- Redistribuye mensajes recibidos a todos los clientes
- Gestiona entrada y salida de clientes mediante canales

El servidor actúa como intermediario de comunicación entre la mutua y el taller, asegurando que todos los cambios de estado sean transmitidos correctamente.

3.2. Cliente Taller (taller.go)

Es el componente principal del sistema. Su función `main()` realiza las siguientes tareas:

1. Inicializa el garage con capacidad definida (10 plazas).
2. Genera un pool de 30 coches con distribución aleatoria de categorías.

3. Configura 4 trabajadores por tipo de fase (documentación, reparación, limpieza, entrega).
4. Establece conexión TCP con el servidor.
5. Lanza goroutines para cada fase del proceso.
6. Inicia el productor de coches.
7. Lee continuamente mensajes del servidor para actualizar el estado.

3.3. Gestión de Concurrencia

3.3.1. Estructura Garage

La estructura `Garage` centraliza el estado compartido:

```
type Garage struct {
    mu        sync.RWMutex
    cars      map[int]*Car
    sts       atomic.Int32
    freeSlots chan struct{}
    wg        sync.WaitGroup
}
```

- `mu`: Protege el acceso al mapa de coches.
- `cars`: Mapa de coches actualmente en el taller.
- `sts`: Estado atómico del taller (modo de operación).
- `freeSlots`: Canal bufferizado que implementa semáforo para plazas.
- `wg`: WaitGroup para sincronización de finalización.

3.3.2. Canales de Comunicación

El sistema utiliza canales para comunicación entre fases:

- **Canal de entrada**: Cada fase tiene 3 canales (uno por prioridad).
- **Canal de salida**: Conecta con la siguiente fase.
- **Canal de eventos**: Centraliza logs en una única goroutine.
- **Canal de stop**: Señaliza terminación de goroutines.

3.4. Fases del Proceso

El sistema implementa 4 fases secuenciales:

1. **Documentación (Fase 1)**: Preparación de papeles. Limitada por plazas disponibles.
2. **Reparación (Fase 2)**: Arreglo del vehículo. Limitada por número de mecánicos (4).
3. **Limpieza (Fase 3)**: Preparación para entrega. Limitada por plazas.
4. **Entrega (Fase 4)**: Entrega final al cliente. Al terminar, libera la plaza.

3.5. Sistema de Prioridades

3.5.1. Categorías de Coches

Categoría	Tipo	Prioridad	Duración
A	Mecánica	Alta	5s + variación
B	Eléctrica	Media	3s + variación
C	Carrocería	Baja	1s + variación

Cuadro 1: Categorías de vehículos

3.5.2. Modos de Operación

Modo	Estado	Descripción
0	Taller Inactivo	No entran coches nuevos, se gestionan los existentes
1	Solo Categoría A	Solo entran coches de categoría A
2	Solo Categoría B	Solo entran coches de categoría B
3	Solo Categoría C	Solo entran coches de categoría C
4	Prioridad A	Se priorizan coches A en todas las fases
5	Prioridad B	Se priorizan coches B en todas las fases
6	Prioridad C	Se priorizan coches C en todas las fases
7	No definido	Mantiene el modo anterior
8	No definido	Mantiene el modo anterior
9	Taller Cerrado	No entran coches nuevos, se gestionan los existentes

Cuadro 2: Modos de operación del taller

Comportamiento por tipo de modo:

- **Modos 1-3 (restrictivos)**: Solo afectan a la entrada de coches nuevos. Los coches que ya están dentro continúan siendo procesados normalmente.
- **Modos 4-6 (prioridades)**: Afectan a todos los coches, incluidos los que ya están dentro. Cambian dinámicamente el orden de atención en todas las fases.

- **Modos 0 y 9**: No permiten entrada de coches nuevos, pero los coches dentro siguen siendo gestionados.
- **Modos 7 y 8**: No modifican el estado, mantienen el modo activo previamente.

4. Diagramas UML

Esta sección presenta los diagramas UML que ilustran el comportamiento y la estructura del sistema.

4.1. Diagrama de Secuencia: Flujo Completo de un Coche

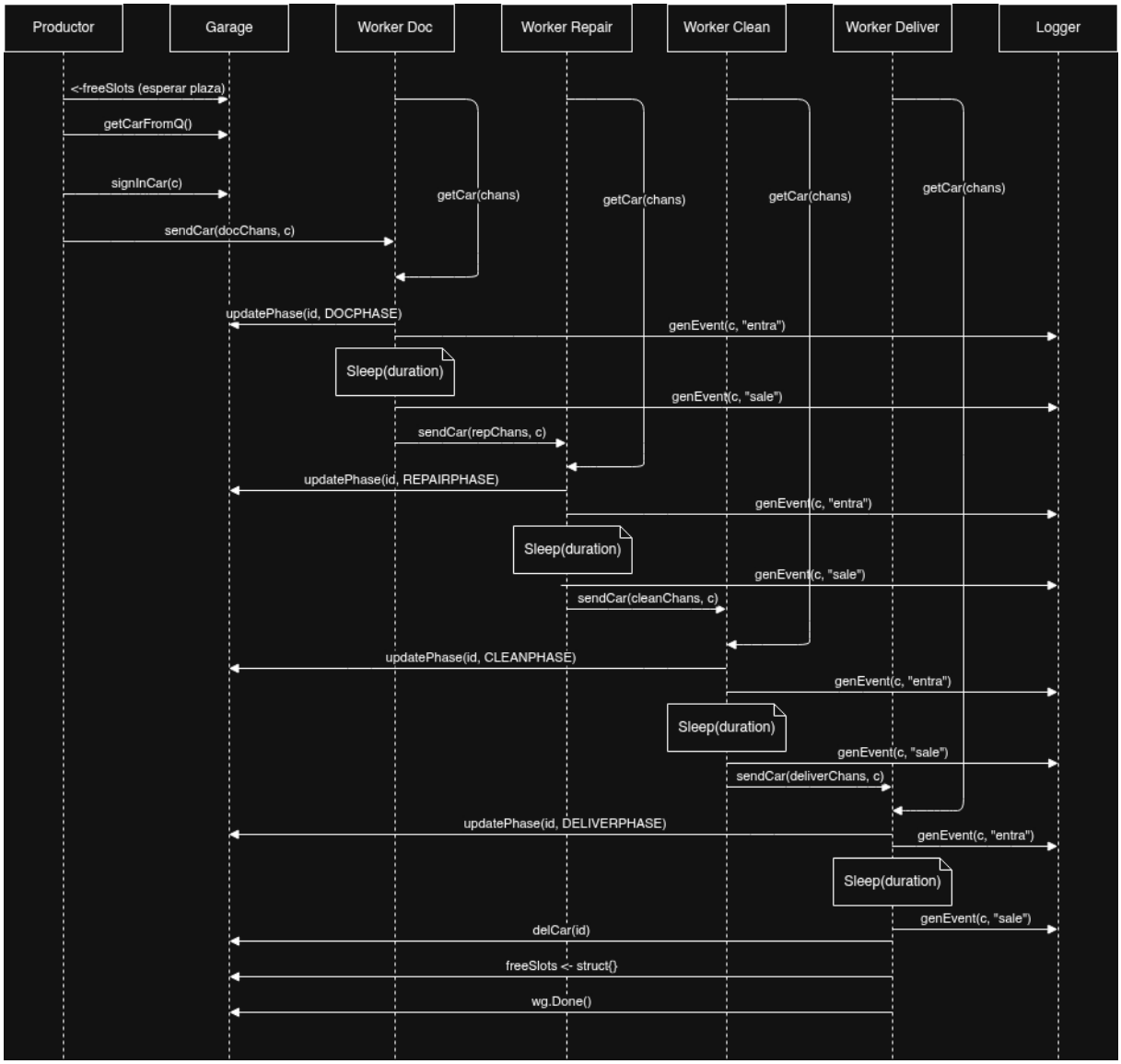

Figura 2: Diagrama de secuencia: Flujo completo de un coche por las 4 fases

Este diagrama muestra la interacción entre:

- Productor
- Garage (estructura compartida)
- Workers de cada fase (Documentación, Reparación, Limpieza, Entrega)
- Logger
- Canales de comunicación

4.2. Diagrama de Secuencia: Comunicación Cliente-Servidor

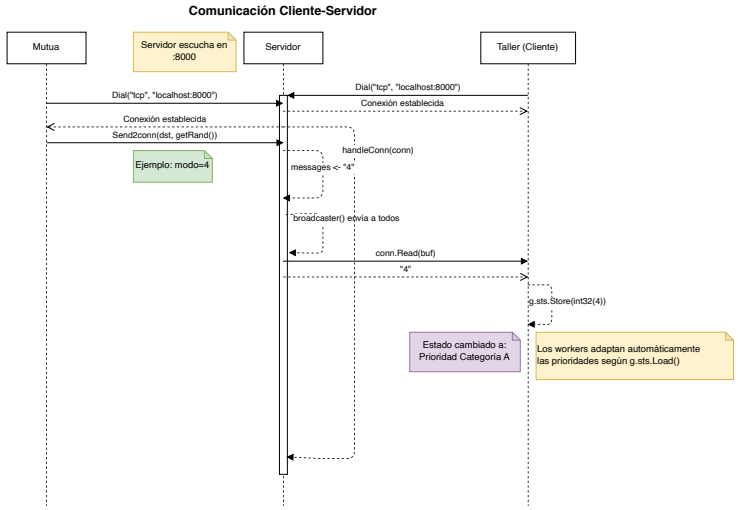

Figura 3: Diagrama de secuencia: Comunicación cliente-servidor

Este diagrama ilustra:

- Conexión inicial entre Mutua, Servidor y Taller
- Broadcasting de mensajes
- Actualización de estado en el taller

4.3. Diagrama de Flujo: Gestión de Prioridades

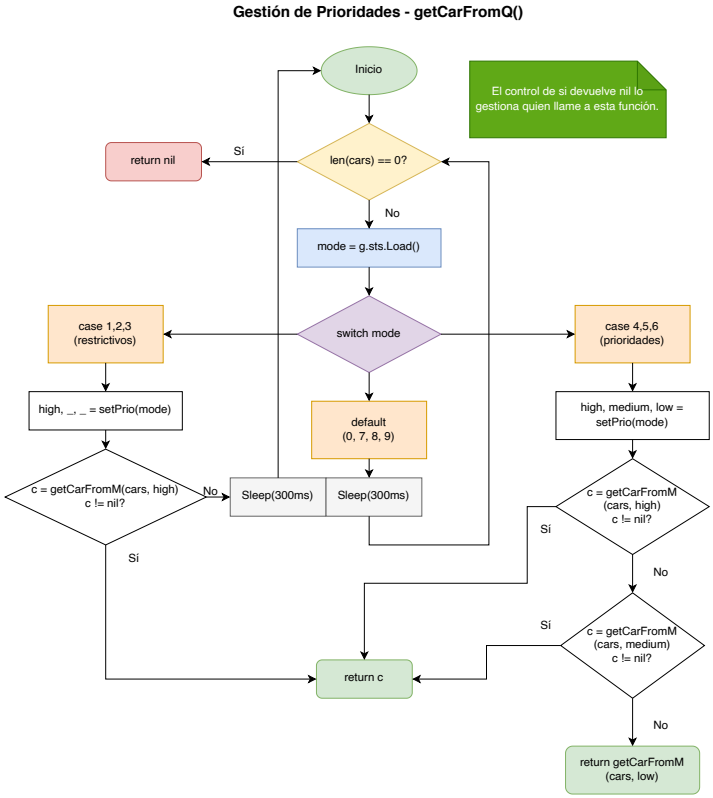

Figura 4: Diagrama de flujo: Lógica de gestión de prioridades

Este diagrama muestra el algoritmo de decisión para:

- Selección de coches según el modo activo
- Priorización en la cola de entrada
- Gestión de workers según prioridades

4.4. Diagrama de Flujo: Worker de Fase

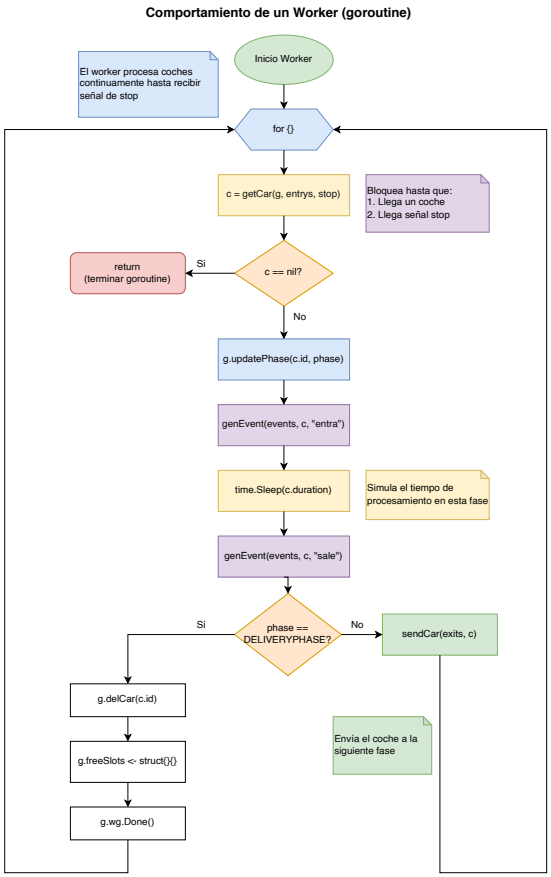

Figura 5: Diagrama de flujo: Comportamiento de un worker

5. Implementación de Concurrencia

5.1. Goroutines del Sistema

El sistema lanza múltiples goroutines concurrentes:

- **1 Logger**: Única goroutine con permiso de escritura en stdout.
- **1 Productor**: Gestiona la entrada de coches al sistema.
- **10 Workers de Documentación**: Uno por plaza disponible.
- **4 Workers de Reparación**: Uno por mecánico.
- **10 Workers de Limpieza**: Uno por plaza.
- **10 Workers de Entrega**: Uno por plaza.

- **1 Lector de Servidor**: Lee comandos del servidor.

Total: 37 goroutines concurrentes en el caso base.

5.2. Sincronización y Seguridad

5.2.1. RWMutex para Acceso al Mapa

Las operaciones sobre el mapa de coches están protegidas:

```
func (g *Garage) signInCar(c *Car) {
    g.mu.Lock()
    defer g.mu.Unlock()
    g.cars[c.id] = c
}

func (g *Garage) updatePhase(id int, phase int) {
    g.mu.Lock()
    defer g.mu.Unlock()
    if c, ok := g.cars[id]; ok {
        c.curphase = phase
    }
}

func (g *Garage) delCar(id int) {
    g.mu.Lock()
    defer g.mu.Unlock()
    delete(g.cars, id)
}
```

5.2.2. Atomic Int32 para Estado

El estado del taller se gestiona mediante operaciones atómicas:

```
// Lectura atómica
mode := g.sts.Load()

// Escritura atómica
g.sts.Store(int32(sts))
```

5.2.3. Canal Bufferizado como Semáforo

Las plazas libres se gestionan mediante un canal bufferizado:

```
// Inicialización
freeSlots: make(chan struct{}, numSlots)

// Ocupar plaza (bloquea si no hay disponibles)
<-g.freeSlots

// Liberar plaza
g.freeSlots <- struct{}{}
```

5.3. Patrón Select para Priorización

La función `getCar()` implementa priorización mediante select anidados:

```
select {
case c = <-chans[high]:  // Prioridad alta
default:
    select {
    case c = <-chans[medium]:  // Prioridad media
    default:
        select {
        case c = <-chans[high]:
        case c = <-chans[medium]:
        case c = <-chans[low]:   // Prioridad baja
        case <-stop:             // Señal de terminación
            return nil
        }
    }
}
```

6. Tests y Resultados

6.1. Estrategia de Testing

Los tests implementados validan:

1. **Corrección funcional**: El sistema procesa correctamente todos los coches.
2. **Ausencia de race conditions**: Ejecutados con `-race`.
3. **Cobertura de código**: Medida con `-cover`.
4. **Rendimiento**: Comparación de tiempos entre configuraciones.

Importante: Los tests **no** evalúan los archivos `mutua.go` ni `servidor.go`, solo el módulo `cliente`. Los tests miden la robustez del sistema en modo 4 (prioridad categoría A) y realizan comparativas a partir de ahí.

6.2. Configuración de Tests

Se ejecutaron 6 tests combinando:

Test	A	B	C	Plazas	Mecánicos
T1	10	10	10	6	3
T2	20	5	5	6	3
T3	5	5	20	6	3
T4	10	10	10	4	4
T5	20	5	5	4	4
T6	5	5	20	4	4

Cuadro 3: Configuraciones de tests

Cada test se ejecutó 10 iteraciones con semilla aleatoria variable. Los tiempos están reescalados (1 segundo real = 100ms en test).

6.3. Resultados de Tests

6.3.1. Tiempos de Ejecución

Test	Tiempo Medio (s)	Min (s)	Max (s)	Por Coche (s)
T1	107.31	99.19	113.41	3.57
T2	133.55	127.68	137.66	4.45
T3	82.97	76.17	87.74	2.76
T4	153.47	148.35	159.85	5.11
T5	191.88	185.35	200.31	6.39
T6	116.05	106.33	123.28	3.86

Cuadro 4: Resultados de tiempos de ejecución (tiempos reales × 10)

6.4. Análisis de Resultados

6.4.1. Impacto de la Distribución de Categorías

Comparando T1, T2 y T3 (misma configuración 6 plazas, 3 mecánicos):

- **T2 (mayoría A)**: 133.55s - El más lento debido a que los coches A requieren 5s por fase.
- **T1 (equilibrado)**: 107.31s - Tiempo medio con distribución equilibrada.
- **T3 (mayoría C)**: 82.97s - El más rápido, los coches C solo requieren 1s por fase.

Conclusión: La distribución de categorías tiene un impacto significativo en el tiempo total. Una mayor proporción de coches categoría A incrementa notablemente el tiempo de procesamiento.

6.4.2. Impacto de Plazas vs Mecánicos

Comparando configuraciones:

- **6 plazas, 3 mecánicos**: Permite más coches en espera, pero limita la reparación.
- **4 plazas, 4 mecánicos**: Reduce espera pero acelera fase crítica (reparación).

Observación: Con 4 plazas y 4 mecánicos, los tiempos aumentan (T4 vs T1: 153.47s vs 107.31s) porque el cuello de botella se traslada a las plazas disponibles. A pesar de tener más mecánicos, menos coches pueden estar en el sistema simultáneamente.

6.4.3. Cobertura de Código

```
Cobertura total: 47.4%
```

Funciones con 100 % de cobertura:

- worker
- startPhase
- signInCar, updatePhase, delCar
- newGarage, getCarFromM
- closeChans, sendCar, initPhaseChans, genEvent

Funciones con 0 % de cobertura:

- main (no se ejecuta en tests)
- logManager (se silencia en tests)
- genCars, randDecimal, genCar (se usan versiones de test)

Nota: La cobertura del 47.4 % es razonable considerando que:

1. La función `main()` no se ejecuta en tests.
2. El logger se silencia en entorno de test.
3. Se utilizan generadores de datos específicos para tests.

6.4.4. Race Conditions

Todos los tests pasaron con `-race` sin detectar condiciones de carrera:

```
PASS
ok      taller/cliente  787.884s
```

Conclusión: El sistema es thread-safe y no presenta race conditions detectables.

7. Repositorio y Código Fuente

El código fuente completo está disponible en:

`https://github.com/alacoste96/Practica-4-SDIS`

7.1. Estructura del Repositorio

```
Practica4
   doc
       4_practica_ssdd_dist.pdf
   README.md
   src
       cliente
           goroutines.go
           mutex.go
           taller.go
           taller_test.go
           types.go
           utility.go
       go.mod
       mutua
           mutua.go
       servidor
            servidor.go
   tests
        cover.out
        test_cover.txt
        test_race.txt
        test_report.txt
```

8. Conclusiones

8.1. Decisiones de Diseño Destacables

- **Canal bufferizado como semáforo**: Solución para gestionar plazas sin locks explícitos.
- **Select anidado para priorización**: Implementa prioridad sin polling activo.
- **Logger centralizado**: Evita race conditions en stdout.
- **Atomic Int32 para estado**: Acceso rápido sin locks pesados.
- **Separación de modos restrictivos y de prioridad**: Permite diferenciar entre entrada y gestión interna.

8.2. Lecciones Aprendidas

1. El diseño de sistemas concurrentes requiere planificación cuidadosa de sincronización.
2. Los canales de Go son herramientas poderosas pero deben usarse correctamente.
3. La gestión de prioridades dinámicas es compleja pero implementable con select.
4. Los tests con `-race` son esenciales para validar seguridad concurrente.

5. La reescalación temporal permite tests rápidos sin perder validez.

9. Referencias

- Donovan, A. A. A., & Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley.
- Documentación oficial de Go: `https://golang.org/doc/`
- Effective Go: `https://golang.org/doc/effective_go`
- Go Concurrency Patterns: `https://go.dev/blog/pipelines`