Documentación Técnica:
Sistema de Gestión de Taller en Go

Alexandre Lacoste Rodríguez

Noviembre 2025

Índice

1. Resumen

Este documento describe el diseño, implementación y resultados de testing de un sistema de gestión de talleres mecánicos desarrollado en Go. El sistema permite gestionar clientes, vehículos, incidencias y mecánicos, implementando un modelo de concurrencia para la asignación automática de recursos y simulación de reparaciones.

1.1. Características principales

- Gestión completa CRUD para todas las entidades
- Sistema de plazas dinámico (2 plazas por mecánico activo)
- Asignación automática de mecánicos mediante goroutines
- Simulación de tiempos de reparación según tipo de incidencia
- Sistema de alertas para contratación de personal

2. Diseño del Sistema

2.1. Arquitectura General

El sistema sigue una arquitectura modular organizada en los siguientes componentes:

- **types.go**: Definición de estructuras de datos y tipos
- **main.go**: Punto de entrada y menú principal
- **new.go**: Funciones de creación de entidades
- **display.go**: Funciones de visualización
- **modify.go**: Funciones de modificación (incluye lógica de concurrencia)
- **delete.go**: Funciones de eliminación
- **aux.go**: Funciones auxiliares y de validación
- **menu.go**: Gestión de menús e interfaz de usuario

2.2. Estructuras de Datos

El sistema define las siguientes estructuras principales en `types.go`:

2.2.1. Tipos Enumerados

```
type IssueType string
const (
    MECHTYPE       IssueType = "Mecanica"
    ELECTRICTYPE   IssueType = "Electrica"
    BODYTYPE       IssueType = "Carroceria"
)

type Priority string
const (
    LOW    Priority = "Baja"
    MEDIUM Priority = "Media"
    HIGH   Priority = "Alta"
)

type IssueStatus string
const (
    OPEN      IssueStatus = "Abierta"
    INPROCESS IssueStatus = "En proceso"
    CLOSED    IssueStatus = "Cerrada"
)

type SkillType string
const (
    MECHSKILL     SkillType = "Mecanica"
    ELECTRICSKILL SkillType = "Electrica"
    BODYSKILL     SkillType = "Carroceria"
)

type MechStatus string
const (
    ACTIVE   MechStatus = "Activo"
    INACTIVE MechStatus = "De baja"
)
```

2.2.2. Entidades Principales

Cliente (Client):

```
type Client struct {
    id       ClientID      // Identificador unico
    name     string        // Nombre del cliente
    phone    string        // Telefono de contacto
    email    string        // Correo electronico
    vehicles []VehicleID   // Vehiculos asociados (relacion 1:N)
}
```

Vehículo (Vehicle):

```
type Vehicle struct {
    id         VehicleID      // Matricula (identificador unico)
```

```
    brand       string          // Marca del vehiculo
    model       string          // Modelo del vehiculo
    ownerID     ClientID        // Propietario (FK a Client)
    checkInAt   time.Time       // Fecha de entrada al taller
    eta         time.Duration   // Tiempo estimado de reparacion
    issues      []IncidenceID   // Incidencias asociadas
}
```

Mecánico (Mechanic):

```
type Mechanic struct {
    id          MechanicID      // Identificador unico
    name        string          // Nombre del mecanico
    skill       SkillType       // Especialidad
    experience  int             // Anios de experiencia
    status      MechStatus      // Activo o de baja
    issues      []IncidenceID   // Incidencias asignadas
}
```

Incidencia (Incidence):

```
type Incidence struct {
    id          IncidenceID     // Identificador unico
    vehicleID   VehicleID       // Vehiculo afectado
    mechanics   []MechanicID    // Mecanicos asignados (N:M)
    kind        IssueType       // Tipo de incidencia
    prio        Priority        // Prioridad
    description string          // Descripcion detallada
    status      IssueStatus     // Estado actual
}
```

Plaza (Slot):

```
type Slot struct {
    number    int          // Numero de plaza
    vehicleID *VehicleID   // Vehiculo ocupando la plaza (nil si libre
       )
}
```

Taller (Garage):

```
type Garage struct {
    clients   []*Client
    vehicles  []*Vehicle
    mechanics []*Mechanic
    issues    []*Incidence
    slots     []*Slot
    vpool     []VehicleID       // Cola de vehiculos esperando
    hirereqs  chan HireRequest // Canal para solicitudes de
       contratacion
}
```

2.3. Diagramas de Secuencia UML

2.3.1. Flujo de Asignación de Vehículo a Plaza

Este diagrama muestra el proceso completo de asignación de un vehículo a una plaza del taller, incluyendo la asignación de mecánicos y la simulación de reparación.

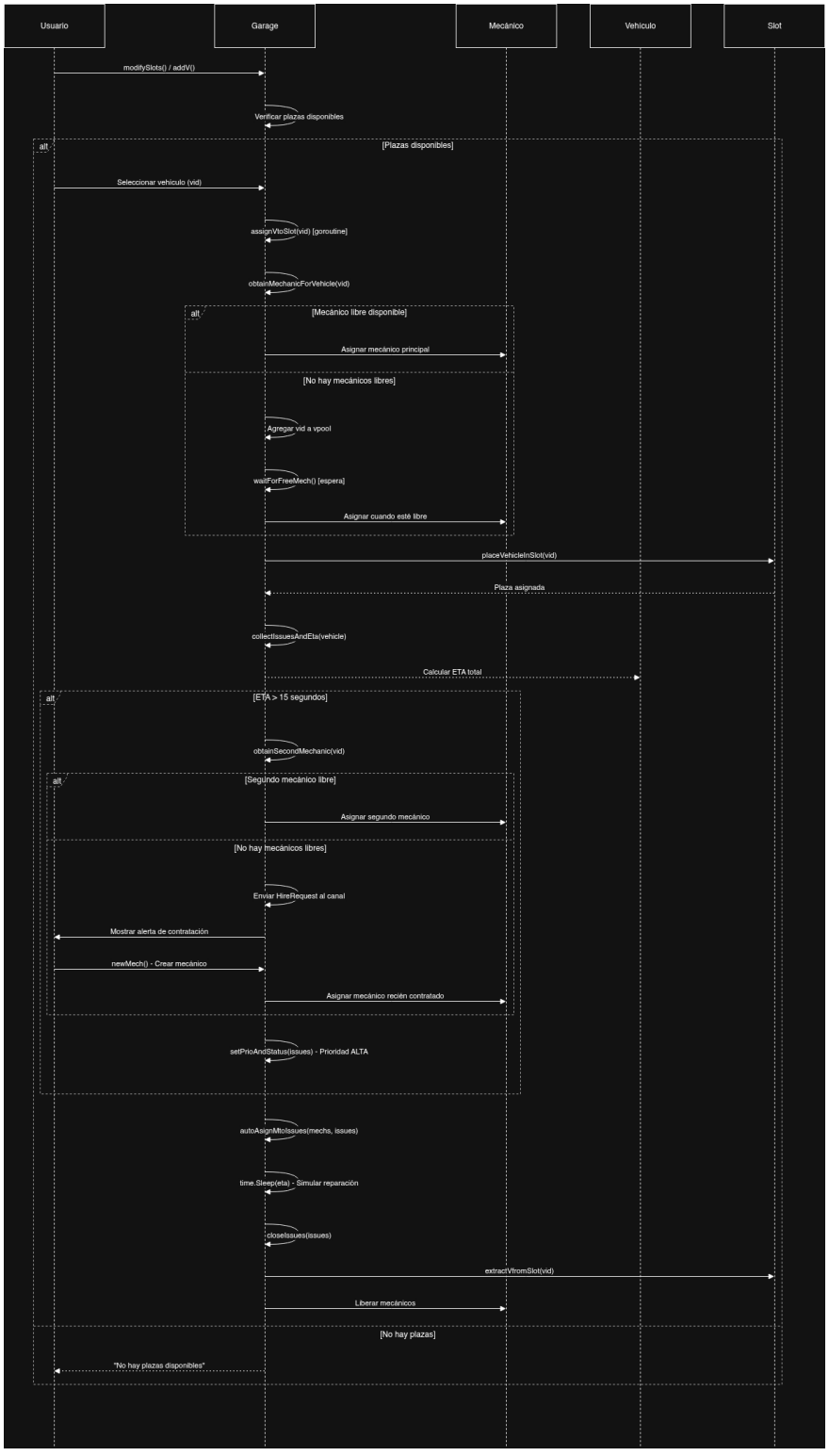

Figura 1: Diagrama de Secuencia de Asignación de un Vehículo a una Plaza

Flujo principal:

1. El usuario solicita meter un vehículo en el taller mediante `modifySlots()` / `addV()`
2. El sistema verifica si hay plazas disponibles
3. Si hay plazas, el usuario selecciona el vehículo por matrícula
4. Se lanza una goroutine con `assignVtoSlot(vid)`
5. Se obtiene un mecánico principal con `obtainMechanicForVehicle(vid)`:
   - Si hay mecánico libre, se asigna directamente
   - Si no, el vehículo se agrega a la cola `vpool` y se espera con `waitForFreeMech()`
6. Se coloca el vehículo en una plaza libre con `placeVehicleInSlot(vid)`
7. Se recopilan incidencias y se calcula el ETA total con `collectIssuesAndEta(vehicle)`:
   - Mecánica: 5 segundos
   - Eléctrica: 7 segundos
   - Carrocería: 11 segundos
8. Si el ETA supera 15 segundos, se asigna un segundo mecánico:
   - Se busca mecánico libre con `getFreeMech()`
   - Si no hay disponible, se envía `HireRequest` por el canal `hirereqs`
   - El menú principal detecta la solicitud con `checkAlert()`
   - Se muestra alerta al usuario
   - El usuario crea un nuevo mecánico con `newMech()`
   - El mecánico se devuelve por el canal de respuesta
9. Se asignan los mecánicos a las incidencias con `autoAsignMtoIssues()`
10. Se establece prioridad ALTA y estado EN PROCESO con `setPrioAndStatus()`
11. Se simula la reparación con `time.Sleep(eta)`
12. Se cierran las incidencias con `closeIssues()`
13. Se extrae el vehículo de la plaza con `extractVfromSlot()`
14. Se liberan los mecánicos eliminando las incidencias de sus listas

2.3.2. Flujo de Creación de Incidencia

Este diagrama muestra el proceso de creación de una nueva incidencia asociada a un vehículo existente.

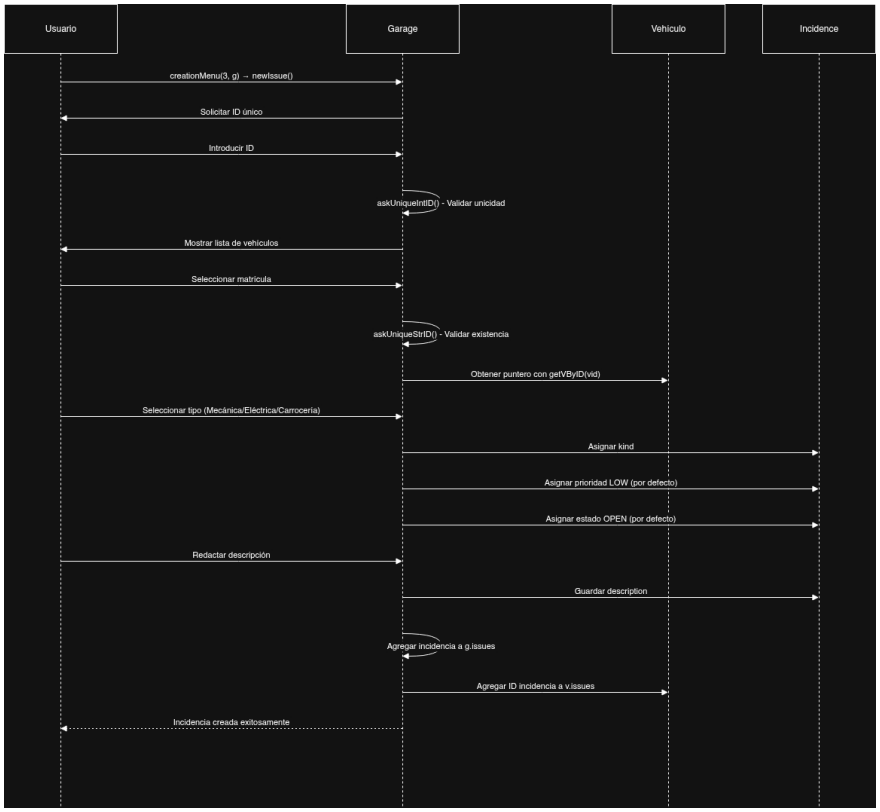

Figura 2: Diagrama de Secuencia de Creación de una Incidencia

Flujo:

1. El usuario selecciona la opción de crear incidencia desde el menú
2. El sistema solicita un ID único mediante `askUniqueIntID()`
3. Se valida que el ID no exista previamente
4. Se muestra la lista de vehículos registrados
5. El usuario selecciona la matrícula del vehículo afectado
6. Se valida que el vehículo exista con `vidExists()`
7. Se obtiene el puntero al vehículo con `getVByID(vid)`
8. El usuario selecciona el tipo de incidencia (Mecánica/Eléctrica/Carrocería)
9. Se asigna prioridad LOW por defecto
10. Se asigna estado OPEN por defecto
11. El usuario redacta una descripción detallada

12. La incidencia se agrega al slice `g.issues`

13. El ID de la incidencia se agrega al slice `v.issues` del vehículo

Nota: Los mecánicos NO se asignan en este momento. La asignación ocurre automáticamente cuando el vehículo se coloca en una plaza del taller.

2.3.3. Sistema de Contratación Concurrente

Este diagrama muestra cómo funciona el sistema de alertas y contratación mediante canales de Go en el caso en el que necesitemos contratar un mecánico para cubrir una incidencia de alta prioridad.

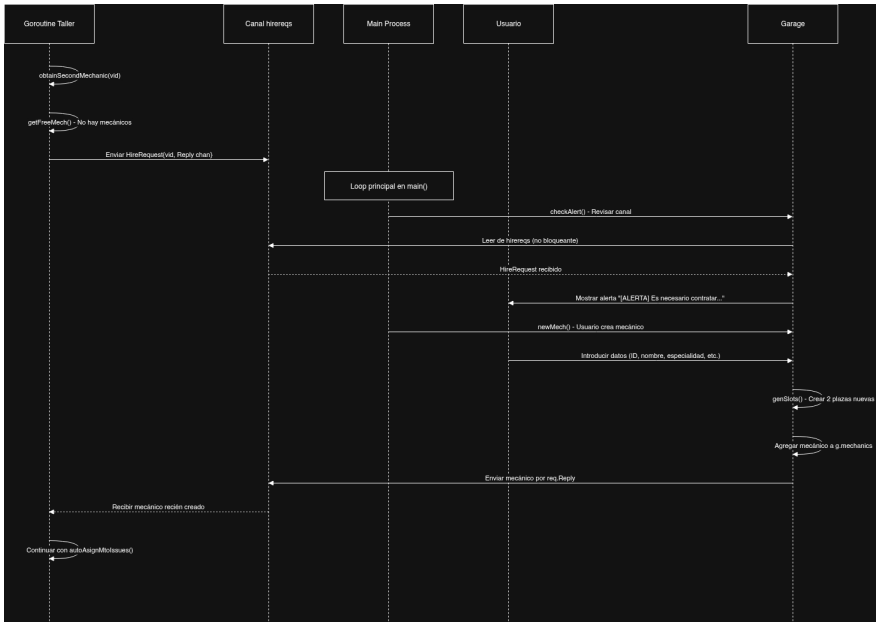

Figura 3: Diagrama de Secuencia de la obtención de un nuevo mecánico concurrentemente cuando un hilo lo necesita

Flujo:

1. Durante `assignVtoSlot()` y si ETA >15 segundos, se ejecuta `obtainSecondMechanic(vid)`
2. Se busca mecánico libre con `getFreeMech()` sin éxito
3. Se crea un `HireRequest` con:
   - `vid`: ID del vehículo que necesita mecánico
   - `Reply`: Canal para recibir el mecánico contratado
4. La solicitud se envía por el canal `hirereqs` (no bloqueante)
5. La goroutine queda esperando respuesta en `req.Reply`
6. En el loop principal de `main()`, se ejecuta `checkAlert()`
7. `checkAlert()` lee del canal `hirereqs` de forma no bloqueante usando `select` con `default`

8. Si hay solicitud pendiente, se muestra alerta al usuario
9. Se invoca `newMech()` para crear un mecánico interactivamente
10. Se generan automáticamente 2 plazas adicionales con `genSlots()`
11. El mecánico recién creado se envía por `req.Reply`
12. La goroutine del taller recibe el mecánico y continúa con `autoAsignMtoIssues()`

Ventajas de este diseño:

- No bloqueante: La interfaz de usuario permanece responsiva
- Desacoplado: La lógica de asignación no conoce los detalles de contratación
- Escalable: Múltiples goroutines pueden solicitar contrataciones simultáneamente

2.4. Funciones Principales

2.4.1. Inicialización y Configuración

newGarage() *Garage (new.go:16)
Constructor del taller. Inicializa todas las estructuras de datos y el canal de contratación:

```
func newGarage() *Garage {
    return &Garage{
        clients:   nil,
        vehicles:  nil,
        mechanics: nil,
        issues:    nil,
        slots:     nil,
        vpool:     nil,
        hirereqs:  make(chan HireRequest),
    }
}
```

genSlots() (new.go:66)
Genera 2 plazas adicionales por cada mecánico contratado. Se invoca automáticamente al crear un mecánico nuevo.

2.4.2. Gestión de Concurrencia

assignVtoSlot(vid VehicleID) (modify.go:237)
Función principal de asignación ejecutada como goroutine. Coordina todo el proceso de reparación:

1. Obtención de mecánico principal (espera si es necesario)
2. Colocación del vehículo en plaza
3. Cálculo de ETA
4. Asignación de segundo mecánico si es necesario

5. Simulación de reparación con `time.Sleep()`

6. Limpieza y liberación de recursos

waitForFreeMech() *Mechanic (modify.go:217)
Implementa una espera activa con polling cada segundo hasta encontrar un mecánico libre:

```go
func (g *Garage) waitForFreeMech() *Mechanic {
    var m *Mechanic
    for {
        m = g.getFreeMech()
        if m != nil {
            return m
        }
        time.Sleep(1 * time.Second)
    }
}
```

checkAlert() (aux.go:262)
Revisa el canal de contratación de forma no bloqueante. Se ejecuta en cada iteración del menú principal:

```go
func (g *Garage) checkAlert() {
    for {
        select {
        case req := <-g.hirereqs:
            fmt.Printf("[ALERTA] Es necesario contratar un mecanico
               nuevo para %s.\n", req.vid)
            mech := g.newMech()
            req.Reply <- mech
            continue
        default:
            return
        }
    }
}
```

2.4.3. Validación y Consultas

askUniqueIntID(prompt string, exists func(int64) bool) int64 (aux.go:135)
Solicita un ID entero único al usuario. Valida que sea positivo y que la función `exists` devuelva false:

```go
func askUniqueIntID(prompt string, exists func(int64) bool) int64 {
    var id int64
    for {
        fmt.Print(prompt)
        fmt.Scanf("%d", &id)
        if id <= 0 {
            fmt.Println("Debe ser > 0.")
            continue
        }
```

```
        if exists(id) {
            fmt.Println("Incorrecto, pruebe otro.")
            continue
        }
        return id
    }
}
```

Esta función se usa con closures para validar diferentes tipos de IDs:

```
askUniqueIntID("ID cliente: ", func(n int64) bool { return g.
    ownerIDexists(ClientID(n)) })
```

```
askUniqueIntID("ID mecánico: ", func(n int64) bool { return g.
    mechIDexists(MechanicID(n)) })
```

Funciones de existencia:

```
func (g *Garage) ownerIDexists(id ClientID) bool
func (g *Garage) mechIDexists(id MechanicID) bool
func (g *Garage) issueIDexists(id IncidenceID) bool
func (g *Garage) vidExists(vid VehicleID) bool
```

Todas iteran sobre sus respectivos slices comparando IDs.

Funciones de obtención:

```
func (g *Garage) getClientByID(id ClientID) *Client
func (g *Garage) getMechByID(id MechanicID) *Mechanic
func (g *Garage) getIssueByID(id IncidenceID) *Incidence
func (g *Garage) getVByID(vid VehicleID) *Vehicle
```

Devuelven punteros a las entidades buscadas o `nil` si no existen.

2.4.4. CRUD de Entidades

Creación (new.go):

- `newClient()`: Solicita ID, nombre, teléfono y email
- `newVehicle()`: Requiere cliente existente, solicita matrícula, marca y modelo
- `newIssue()`: Asocia a vehículo existente, solicita tipo y descripción
- `newMech()`: Crea mecánico y genera 2 plazas automáticamente

Visualización (display.go):

- `displayClients()`, `displayVehicles()`, `displayIssues()`, `displayMechs()`
- `displaySlots()`: Muestra estado de todas las plazas (libres/ocupadas)
- `listIncFromAvehicle()`: Lista incidencias de un vehículo específico
- `listVFromClient()`: Lista vehículos de un cliente
- `listDispMech()`: Mecánicos disponibles (sin incidencias asignadas)

- `dispIssuesOfMech()`: Incidencias asignadas a un mecánico
- `listClientsVInGarage()`: Clientes con vehículos actualmente en el taller

Modificación (modify.go):

- `modifyClient()`, `modifyVehicle()`, `modifyIssue()`, `modifyMech()`
- Cada función permite modificar campos específicos manteniendo integridad referencial
- `modifySlots()`: Permite meter o sacar vehículos de las plazas

Eliminación (delete.go):

- `delClient()`: Elimina cliente y en cascada todos sus vehículos
- `delVehicle()`: Elimina vehículo y todas sus incidencias
- `delIssue()`: Elimina incidencia y actualiza referencias en vehículos y mecánicos
- `delMech()`: Elimina mecánico y reduce plazas en 2

Todas las operaciones de eliminación mantienen la consistencia eliminando referencias en entidades relacionadas.

2.5. Algoritmos Clave

2.5.1. Cálculo de ETA

La función `collectIssuesAndEta()` recorre todas las incidencias de un vehículo y suma tiempos según el tipo:

```
1  func (g *Garage) collectIssuesAndEta(v *Vehicle) ([]*Incidence, time.
       Duration) {
2      var issues []*Incidence
3      var eta time.Duration
4
5      for _, issueID := range v.issues {
6          inc := g.getIssueByID(issueID)
7          if inc == nil {
8              continue
9          }
10         issues = append(issues, inc)
11
12         switch inc.kind {
13         case MECHTYPE:
14             eta += 5 * time.Second
15         case ELECTRICTYPE:
16             eta += 7 * time.Second
17         default: // BODYTYPE
18             eta += 11 * time.Second
19         }
20     }
21     return issues, eta
22 }
```

2.5.2. Asignación de Mecánicos a Incidencias

La función `assignMechsToIssue()` (aux.go:85) implementa un sistema interactivo de asignación:

1. Filtra mecánicos compatibles: misma especialidad que el tipo de incidencia, activos y no ya asignados
2. Muestra lista de candidatos
3. Permite asignar múltiples mecánicos de forma iterativa
4. Actualiza relación bidireccional: `mechanic.issues` e `incidence.mechanics`

2.5.3. Gestión de Cola de Espera

El slice `vpool []VehicleID` actúa como cola FIFO para vehículos esperando mecánico:

```
// Agregar a cola
g.vpool = append(g.vpool, vid)

// Esperar activamente
m := g.waitForFreeMech()

// Remover de cola
g.vpool = removeVID(g.vpool, vid)
```

La función `removeVID()` (aux.go:31) elimina un VehicleID de un slice preservando el orden.

3. Resultados de Tests y Análisis

3.1. Suite de Tests

El proyecto incluye 4 tests de comparación de escenarios (`*_test.go`):

- **TestDupCarsWithSameIssue**: Duplicar cantidad de vehículos con incidencias del mismo tipo
- **TestDupMechs**: Duplicar plantilla de mecánicos de 3 a 6
- **Test3MechsVS1others**: Distribución 3:1:1 (Mecánica:Eléctrica:Carrocería)
- **Test1MechVS3others**: Distribución 1:3:3 (Mecánica:Eléctrica:Carrocería)

3.2. Resultados de Ejecución

Todos los tests pasaron exitosamente:

```
=== RUN   TestDupCarsWithSameIssue
--- PASS: TestDupCarsWithSameIssue (0.00s)
=== RUN   TestDupMechs
--- PASS: TestDupMechs (0.00s)
=== RUN   Test3MechsVS1others
--- PASS: Test3MechsVS1others (0.00s)
=== RUN   Test1MechVS3others
--- PASS: Test1MechVS3others (0.00s)
PASS
ok      practica2/src   (cached)
```

Análisis:

- Tiempo de ejecución: 0.00s para todos los tests (instantáneo)
- Estado: Todos los tests pasan sin errores
- Cache: Resultados cacheados por Go, indicando que no hubo cambios en el código desde la última ejecución

3.3. Cobertura de Código

El archivo `coverage.txt` muestra la cobertura por archivo y función. Resumen por archivo:

Archivo	Funciones Cubiertas	Funciones Totales
aux.go	2	29
display.go	0	13
main.go	0	1
menu.go	0	6
new.go	1 (genSlots)	5
modify.go	0	15
delete.go	0	6
Total	**3**	**75**

Cuadro 1: Cobertura de código por archivo

Funciones cubiertas por tests:

1. `availableSlots()` - Cuenta plazas libres
2. `CountVehiclesWithSingleIssue()` - Cuenta vehículos con una incidencia de tipo específico
3. `genSlots()` - Genera 2 plazas por mecánico
4. `countSkill()` - Cuenta mecánicos por especialidad

Análisis de cobertura:

- Cobertura baja: Solo 4 % de las funciones están cubiertas por tests

- Enfoque selectivo: Los tests se centran en validar lógica de negocio específica (contadores y distribuciones)
- Funciones sin cobertura: Toda la interfaz de usuario (menús, entrada/salida) y operaciones CRUD no están cubiertas

Recomendaciones para prácticas futuras:

- Añadir tests unitarios para funciones auxiliares (validación, búsqueda)
- Implementar tests de integración para flujos completos (crear cliente → crear vehículo → crear incidencia)
- Mockear entrada/salida para testear funciones interactivas
- Añadir benchmarks para `assignVtoSlot()` y funciones concurrentes para medir rendimiento en casos de estrés computacional.

3.4. Análisis de Tests Específicos

3.4.1. TestDupCarsWithSameIssue

Objetivo: Verificar que al duplicar la cantidad de vehículos con una incidencia del mismo tipo, el contador también se duplica.

Implementación (dupmaxcar_test.go):

```
func TestDupCarsWithSameIssue(t *testing.T) {
    const baseN = 3
    g1 := genGarage(t, baseN, MECHTYPE)
    got1 := g1.CountVehiclesWithSingleIssue(MECHTYPE)
    if got1 != baseN {
        t.Fatalf("para %d coches esperabamos contador=%d; got=%d",
                 baseN, baseN, got1)
    }

    g2 := genGarage(t, 2*baseN, MECHTYPE)
    got2 := g2.CountVehiclesWithSingleIssue(MECHTYPE)
    if got2 != 2*baseN {
        t.Fatalf("para %d coches esperabamos contador=%d; got=%d",
                 2*baseN, 2*baseN, got2)
    }

    if got2 != 2*got1 {
        t.Errorf("se esperaba que al duplicar coches, el contador se
            duplicase")
    }
}
```

Resultado: PASS

Conclusión: La función `CountVehiclesWithSingleIssue()` cuenta correctamente vehículos con exactamente una incidencia del tipo especificado. La proporcionalidad lineal se mantiene al duplicar la población.

3.4.2. TestDupMechs

Objetivo: Verificar que al duplicar la plantilla de mecánicos de 3 a 6, las plazas también se duplican de 6 a 12.

Resultado: PASS

Conclusiones:

- La función auxiliar `gen3Mechs()` genera correctamente 3 mecánicos (uno de cada especialidad)
- La regla "2 plazas por mecánico" se cumple: 3 mecánicos → 6 plazas, 6 mecánicos → 12 plazas
- La distribución de especialidades es equitativa (1-1-1, luego 2-2-2)

3.4.3. Test3MechsVS1others

Objetivo: Verificar distribución 3:1:1 (3 mecánicos de Mecánica por cada 1 de Eléctrica y 1 de Carrocería).

Resultado: PASS

Conclusión: La función `countSkill()` cuenta correctamente mecánicos por especialidad. Las proporciones 3:1:1 se mantienen correctamente.

3.4.4. Test1MechVS3others

Objetivo: Verificar distribución 1:3:3 (1 mecánico de Mecánica por cada 3 de Eléctrica y 3 de Carrocería).

Resultado: PASS

Conclusión: El sistema mantiene correctamente proporciones inversas a las del test anterior.

3.5. Métricas de Calidad

Complejidad ciclomática estimada:

- Funciones sencillas (getters, setters): 1-2
- Funciones con validación (askUniqueIntID): 3-4
- Funciones con menús (polyAskMenuInt): 5-6
- Función principal `assignVtoSlot()`: aproximadamente 10

Acoplamiento:

- Alto acoplamiento entre `Garage` y todas las entidades (inevitable dado el diseño de repositorio central)
- Bajo acoplamiento entre entidades individuales

Cohesión:

- Alta cohesión funcional por archivo (new.go, delete.go, modify.go, display.go)

- Cada módulo tiene una responsabilidad clara

Mantenibilidad:

- Código bien estructurado y comentado
- Separación clara de responsabilidades
- Uso de tipos personalizados mejora legibilidad

4. Conclusiones

4.1. Objetivos Cumplidos

El sistema implementado cumple con todos los requisitos especificados:

- Gestión completa de Clientes (CRUD + listados)
- Gestión completa de Vehículos (CRUD + asociación a clientes)
- Gestión completa de Incidencias (CRUD + asignación de mecánicos)
- Gestión completa de Mecánicos (CRUD + alta/baja)
- Sistema de plazas dinámico (2 por mecánico)
- Visualización de estado del taller
- Listados específicos (incidencias por vehículo, vehículos por cliente, mecánicos disponibles, etc.)
- Menús y submenús intuitivos

4.2. Características Destacadas

1. Concurrencia con Goroutines:

- La asignación de vehículos a plazas se ejecuta en goroutines independientes
- Permite simular reparaciones sin bloquear la interfaz de usuario
- Sistema de canales para comunicación entre goroutines y el proceso principal

2. Sistema de Contratación Dinámica:

- Detección automática de necesidad de personal adicional
- Alertas al usuario cuando se requiere contratar
- Comunicación asíncrona mediante canales de Go

3. Simulación Realista:

- Cálculo de ETA basado en tipo y cantidad de incidencias

- Asignación de segundo mecánico para trabajos largos
- Priorización automática de incidencias urgentes

4. Integridad Referencial:

- Eliminación en cascada de entidades relacionadas
- Actualización bidireccional de relaciones
- Validación exhaustiva de IDs únicos

4.3. Limitaciones Actuales

1. Persistencia:

- Todos los datos se almacenan en memoria (se pierden al cerrar)
- Solución futura: Implementar serialización JSON o base de datos

2. Sincronización:

- No hay mutexes para proteger accesos concurrentes
- Posibles condiciones de carrera en entornos multi-goroutine intensivos
- Solución futura: Usar `sync.Mutex` o `sync.RWMutex`

3. Cobertura de Tests:

- Solo 4 % de funciones cubiertas
- Tests centrados en lógica de negocio
- Solución futura: Mocking de I/O, tests de integración

4.4. Lecciones Aprendidas

1. Canales como Mecanismo de Comunicación:
Los canales de Go demostraron ser una herramienta elegante para comunicación entre goroutines, especialmente para solicitudes asíncronas como la contratación de mecánicos.

2. Separación de Responsabilidades:
La organización del código en archivos por funcionalidad facilitó el mantenimiento y la comprensión del sistema.

3. Testing Basado en Comparaciones:
Los tests implementados verifican proporcionalidad y ratios en lugar de valores absolutos, lo que los hace más robustos ante cambios en configuración.

5. Apéndices

5.1. Apéndice A: Repositorio del proyecto y su estructura

Repositorio: https://github.com/alacoste96/Practica2
Estructura del proyecto:

```
practica2/
 src/
    main.go              # Punto de entrada
    types.go             # Estructuras de datos
    new.go               # Creación
    display.go           # Visualización
    modify.go            # Modificación
    delete.go            # Eliminación
    aux.go               # Auxiliares
    menu.go              # Menús
    dupmaxcar_test.go    # Tests
    dupmechs_test.go
    threevsone_test.go
 images/
    asignacionVplaza.png
    creacionIncidencia.png
    obtenerMconcurrencia.png
 Doc/
    Practica_2_alacoste_SSOO_dist.pdf
 test-results/
     coverage.txt
     coverage.html
     resultados.txt
```

5.2. Apéndice B: Comandos de Ejecución

Compilar y ejecutar:

```
go build -o taller *.go
./taller
```

Ejecutar tests:

```
go test -v
```

Generar cobertura:

```
go test -coverprofile=coverage.txt
go tool cover -html=coverage.txt -o coverage.html
```

6. Referencias

1. Donovan, A. A., & Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley Professional.

2. Cox-Buday, K. (2017). *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media.

3. Go Official Documentation. *Effective Go*. https://golang.org/doc/effective_go

4. Go Blog. *Go Concurrency Patterns*. https://go.dev/blog/pipelines

5. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.