

Documentación Técnica: Sistema de Gestión de Taller en Go

Alexandre Lacoste Rodríguez

Noviembre 2025

Índice

1. Resumen	3
1.1. Características principales	3
2. Diseño del Sistema	3
2.1. Arquitectura General	3
2.2. Estructuras de Datos	3
2.2.1. Tipos Enumerados	4
2.2.2. Entidades Principales	4
2.3. Diagramas de Secuencia UML	6
2.3.1. Flujo de Asignación de Vehículo a Plaza	6
2.3.2. Flujo de Creación de Incidencia	8
2.3.3. Sistema de Contratación Concurrente	9
2.4. Funciones Principales	10
2.4.1. Inicialización y Configuración	10
2.4.2. Gestión de Concurrencia	11
2.4.3. Validación y Consultas	12
2.4.4. CRUD de Entidades	12
2.5. Algoritmos Clave	13
2.5.1. Cálculo de ETA	13
2.5.2. Asignación de Mecánicos a Incidencias	14
2.5.3. Gestión de Cola de Espera	14
3. Resultados de Tests y Análisis	15
3.1. Suite de Tests	15
3.2. Resultados de Ejecución	15
3.3. Cobertura de Código	15
3.4. Análisis de Tests Específicos	16
3.4.1. TestDupCarsWithSameIssue	16
3.4.2. TestDupMechs	17
3.4.3. Test3MechsVS1others	17
3.4.4. Test1MechVS3others	17
3.5. Métricas de Calidad	18
4. Código Fuente Completo	18
4.1. types.go - Definición de Estructuras	18
4.2. main.go - Punto de Entrada	21
4.3. new.go - Funciones de Creación (Extracto)	22
4.4. aux.go - Funciones Auxiliares (Extracto)	23
4.5. modify.go - Asignación Concurrente (Extracto Clave)	25
5. Conclusiones	28
5.1. Objetivos Cumplidos	28
5.2. Características Destacadas	28
5.3. Limitaciones Actuales	29
5.4. Lecciones Aprendidas	29
5.5. Trabajo Futuro	29

6. Referencias	30
7. Apéndices	30
7.1. Apéndice A: Estructura del Proyecto	30
7.2. Apéndice B: Comandos de Ejecución	31
7.3. Apéndice C: Configuración de LyX	31

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

```
1 type IssueType string
2 const (
3     MECHTYPE      IssueType = "Mecanica"
4     ELECTRICTYPE IssueType = "Electrica"
5     BODYTYPE      IssueType = "Carroceria"
6 )
7
8 type Priority string
9 const (
10    LOW      Priority = "Baja"
11    MEDIUM   Priority = "Media"
12    HIGH     Priority = "Alta"
13 )
14
15 type IssueStatus string
16 const (
17     OPEN      IssueStatus = "Abierta"
18     INPROCESS IssueStatus = "En proceso"
19     CLOSED    IssueStatus = "Cerrada"
20 )
21
22 type SkillType string
23 const (
24     MECHSKILL   SkillType = "Mecanica"
25     ELECTRICSKILL SkillType = "Electrica"
26     BODYSKILL    SkillType = "Carroceria"
27 )
28
29 type MechStatus string
30 const (
31     ACTIVE     MechStatus = "Activo"
32     INACTIVE   MechStatus = "De baja"
33 )
```

2.2.2. Entidades Principales

Cliente (Client):

```
1 type Client struct {
2     id        ClientID      // Identificador unico
3     name      string        // Nombre del cliente
4     phone     string        // Telefono de contacto
5     email     string        // Correo electronico
6     vehicles []VehicleID   // Vehiculos asociados (relacion 1:N)
7 }
```

Vehículo (Vehicle):

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

```

3   brand      string           // Marca del vehiculo
4   model      string           // Modelo del vehiculo
5   ownerID    ClientID        // Propietario (FK a Client)
6   checkInAt  time.Time       // Fecha de entrada al taller
7   eta        time.Duration   // Tiempo estimado de reparacion
8   issues     [] IncidenceID // Incidencias asociadas
9 }
```

Mecánico (Mechanic):

```

1 type Mechanic struct {
2   id        MechanicID      // Identificador unico
3   name      string          // Nombre del mecanico
4   skill     SkillType       // Especialidad
5   experience int            // Anios de experiencia
6   status    MechStatus      // Activo o de baja
7   issues    [] IncidenceID // Incidencias asignadas
8 }
```

Incidencia (Incidence):

```

1 type Incidence struct {
2   id        IncidenceID    // Identificador unico
3   vehicleID VehicleID     // Vehiculo afectado
4   mechanics [] MechanicID // Mecanicos asignados (N:M)
5   kind      IssueType      // Tipo de incidencia
6   prio     Priority        // Prioridad
7   description string        // Descripcion detallada
8   status    IssueStatus    // Estado actual
9 }
```

Plaza (Slot):

```

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

Taller (Garage):

```

1 type Garage struct {
2   clients    []*Client
3   vehicles   []*Vehicle
4   mechanics  []*Mechanic
5   issues     []*Incidence
6   slots      []*Slot
7   vpool      []VehicleID   // Cola de vehiculos esperando
8   hirereqs   chan HireRequest // Canal para solicitudes de
9   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.

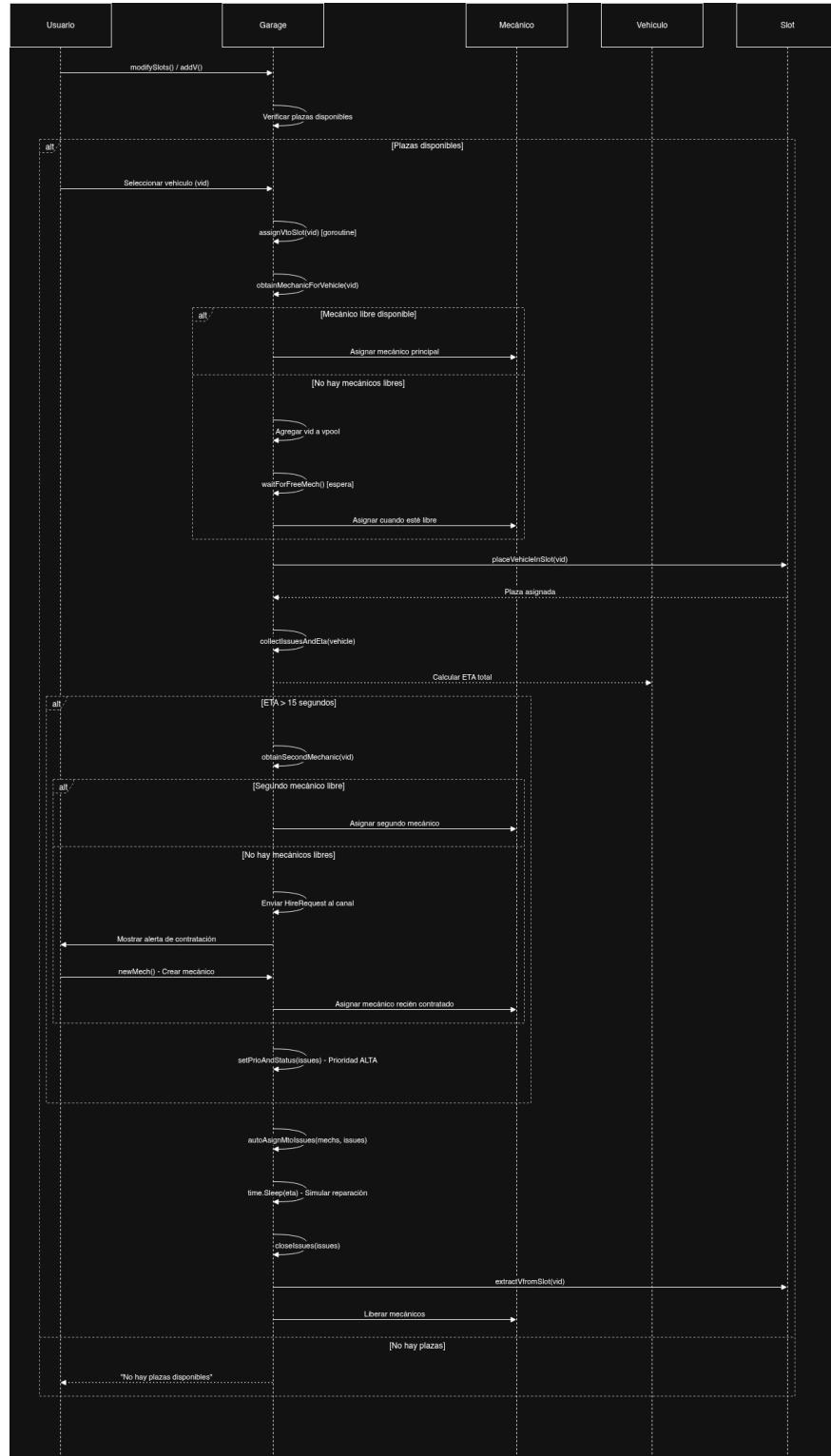


Figura 1: Diagrama de Flujo de Asignación de un Vehículo a una Plaza

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 `autoAssignMtoIssues()`
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.

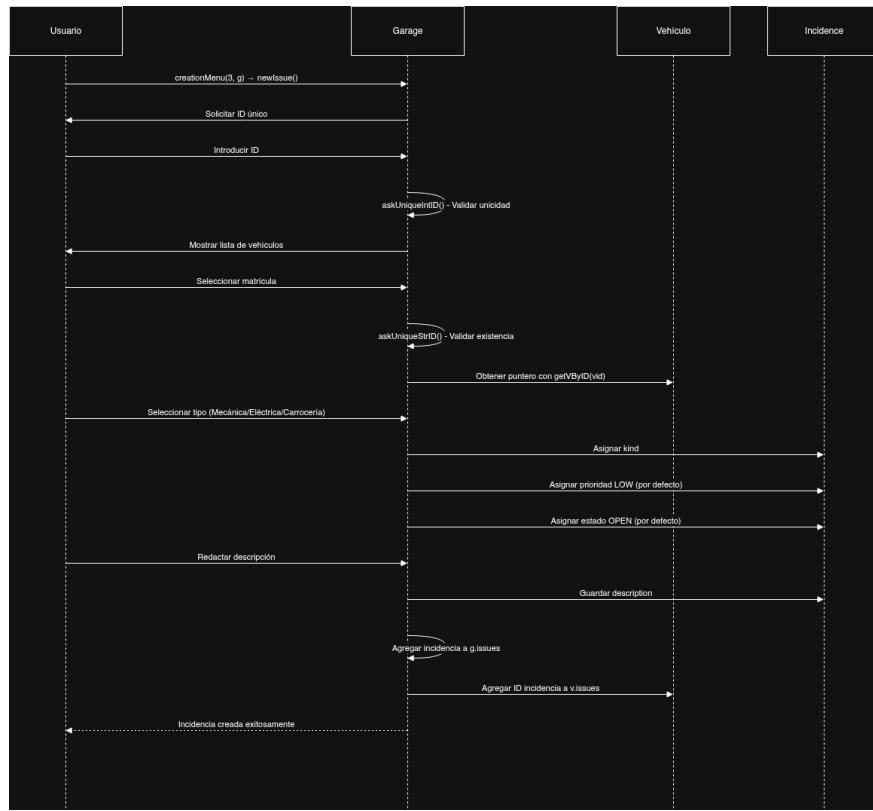


Figura 2: Diagrama de Flujo de Creación de una Incidencia

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 `getVByD(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.

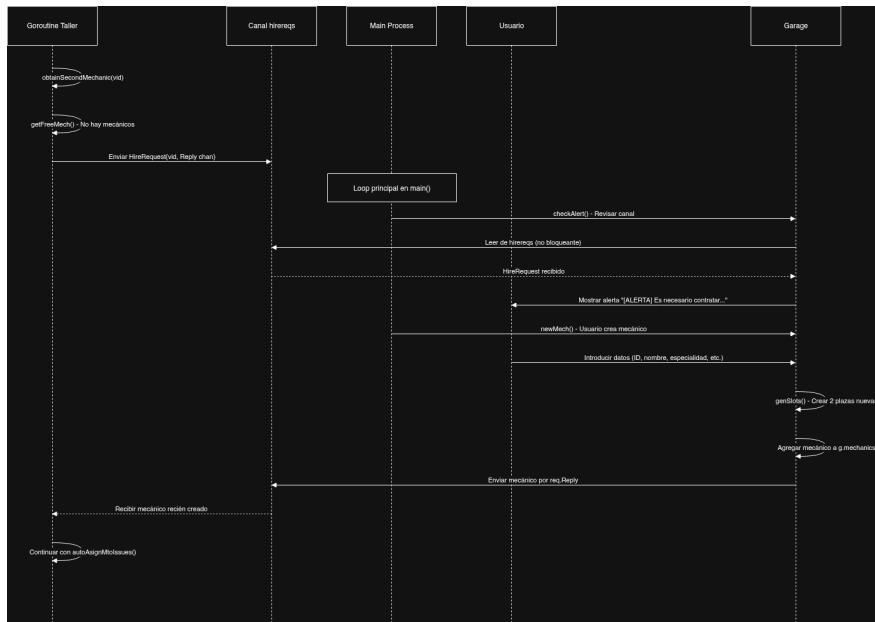


Figura 3: Diagrama de Flujo de la obtención de un nuevo mecánico concurrente cuando un hilo lo necesita

Flujo:

1. Durante `assignVtoSlot()`, 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 responsive
- 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:

```

1 func newGarage() *Garage {
2     return &Garage{
3         clients:    nil,
4         vehicles:   nil,
5         mechanics:  nil,
6         issues:     nil,
7         slots:      nil,
8         vpool:      nil,
9         hirereqs:   make(chan HireRequest),
10    }
11 }
```

`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:

```
1 func (g *Garage) waitForFreeMech() *Mechanic {
2     var m *Mechanic
3     for {
4         m = g.getFreeMech()
5         if m != nil {
6             return m
7         }
8         time.Sleep(1 * time.Second)
9     }
10 }
```

checkAlert() (aux.go:262)

Revisa el canal de contratación de forma no bloqueante. Se ejecuta en cada iteración del menú principal:

```
1 func (g *Garage) checkAlert() {
2     for {
3         select {
4             case req := <-g.hirereqs:
5                 fmt.Printf("[ALERTA] Es necesario contratar un
6                     mec nico nuevo para %s.\n", req.vid)
7                 mech := g.newMech()
8                 req.Reply <- mech
9                 continue
10            default:
11                return
12            }
13        }
```

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:

```
1 func askUniqueIntID(prompt string, exists func(int64) bool) int64
2 {
3     var id int64
4     for {
5         fmt.Print(prompt)
6         fmt.Scanf("%d", &id)
7         if id <= 0 {
8             fmt.Println("Debe ser > 0.")
9             continue
10        }
11    }
12 }
```

```

9     }
10    if exists(id) {
11      fmt.Println("Incorrecto, pruebe otro.")
12      continue
13    }
14    return id
15  }
16 }
```

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

- askUniqueIntID(íD cliente: "", func(n int64) bool { return g.ownerIDExists(ClientID) })
- askUniqueIntID(íD mecánico: "", func(n int64) bool { return g.mechIDExists(MechanicID) })

Funciones de existencia:

```

1 func (g *Garage) ownerIDExists(id ClientID) bool
2 func (g *Garage) mechIDExists(id MechanicID) bool
3 func (g *Garage) issueIDExists(id IncidenceID) bool
4 func (g *Garage) vidExists(vid VehicleID) bool
```

Todas iteran sobre sus respectivos slices comparando IDs.

Funciones de obtención:

```

1 func (g *Garage) getClientByID(id ClientID) *Client
2 func (g *Garage) getMechByID(id MechanicID) *Mechanic
3 func (g *Garage) getIssueByID(id IncidenceID) *Incidence
4 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(): Sigue la misma lógica que askUniqueIntID
- 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,
2   time.Duration) {
3   var issues []*Incidence
4   var eta time.Duration
5
6   for _, issueID := range v.issues {
7     inc := g.getIssueByID(issueID)
8     if inc == nil {
9       continue
10    }
11    issues = append(issues, inc)
12
13    switch inc.kind {
14      case MECHTYPE:
15        eta += 5 * time.Second
16      case ELECTRICTYPE:
17        eta += 7 * time.Second
18      default: // BODYTYPE
19        eta += 11 * time.Second
20    }
21
22  }
23
24  return issues, eta
25 }
```

```

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:

```

1 // Agregar a cola
2 g.vpool = append(g.vpool, vid)
3
4 // Esperar activamente
5 m := g.waitForFreeMech()
6
7 // Remover de cola
8 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:

- 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

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`):

```

1 func TestDupCarsWithSameIssue(t *testing.T) {
2     const baseN = 3
3     g1 := genGarage(t, baseN, MECHTYPE)
4     got1 := g1.CountVehiclesWithSingleIssue(MECHTYPE)
5     if got1 != baseN {
6         t.Fatalf("para %d coches esperabamos contador=%d; got=%d
7             ",
8                 baseN, baseN, got1)
9 }
10
11    g2 := genGarage(t, 2*baseN, MECHTYPE)
12    got2 := g2.CountVehiclesWithSingleIssue(MECHTYPE)
13    if got2 != 2*baseN {
14        t.Fatalf("para %d coches esperabamos contador=%d; got=%d
15             ",
16                 2*baseN, 2*baseN, got2)
17 }
18
19    if got2 != 2*got1 {
20        t.Errorf("se esperaba que al duplicar coches, el contador
21             se duplicase")
22 }
23 }
```

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 (`polyAskMenuItem`): 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. Código Fuente Completo

4.1. types.go - Definición de Estructuras

```

1 package main
2
3 import "time"
4
5 // Tipos enumerados para incidencias
6 type IssueType string
7 const (
8     MECHTYPE           IssueType = "Mecánica"
9     ELECTRICTYPE       IssueType = "Eléctrica"
10    BODYTYPE           IssueType = "Carrocería"
11)
12
13 type Priority string
14 const (
15     LOW      Priority = "Baja"
16     MEDIUM   Priority = "Media"
17     HIGH     Priority = "Alta"
18)
19
20 type IssueStatus string
21 const (
22     OPEN      IssueStatus = "Abierta"
23     INPROCESS IssueStatus = "En proceso"
24     CLOSED    IssueStatus = "Cerrada"
25)
26
27 // Tipos para mecánicos
28 type SkillType string
29 const (
30     MECHSKILL        SkillType = "Mecánica"
31     ELECTRICKILL     SkillType = "Eléctrica"
32     BODYSKILL        SkillType = "Carrocería"
33)
34
35 type MechStatus string
36 const (
37     ACTIVE      MechStatus = "Activo"

```

```

38     INACTIVE MechStatus = "De baja"
39 }
40
41 // Tipos de identificadores (para mayor claridad semántica)
42 type ClientID int64
43 type MechanicID int64
44 type IncidenceID int64
45 type VehicleID string
46
47 // -----
48 // ----- ESTRUCTURAS DE DATOS -----
49 // -----
50
51 // Client representa a un cliente del taller
52 type Client struct {
53     id        ClientID
54     name      string
55     phone     string
56     email     string
57     vehicles  []VehicleID // Relación 1:N con vehículos
58 }
59
60 // Mechanic representa un mecánico del taller
61 type Mechanic struct {
62     id        MechanicID
63     name      string
64     skill     SkillType
65     experience int
66     status    MechStatus
67     issues   []IncidenceID // Incidencias asignadas
68 }
69
70 // Vehicle representa un vehículo en el taller
71 // Mantiene relación con cliente (ownerID)
72 type Vehicle struct {
73     id        VehicleID      // Matrícula
74     brand     string
75     model    string
76     ownerID  ClientID
77     checkInAt time.Time
78     eta       time.Duration // Tiempo estimado de reparación
79     issues   []IncidenceID
80 }
81
82 // Incidence representa una incidencia/reparación
83 // Asignación N:M con mecánicos mediante slice de IDs
84 type Incidence struct {
85     id        IncidenceID
86     vehicleID VehicleID
87     mechanics []MechanicID // Multiples mecánicos pueden
                                trabajar

```

```

88     kind      IssueType
89     prio      Priority
90     description string
91     status     IssueStatus
92 }
93
94 // Slot representa una plaza física del taller
95 // Si vehicleID == nil, la plaza está libre
96 type Slot struct {
97     number    int
98     vehicleID *VehicleID
99 }
100
101 // HireRequest representa una solicitud de contratación
102 type HireRequest struct {
103     vid       VehicleID          // Vehículo que necesita mecánico
104     Reply chan *Mechanic        // Canal para devolver mecánico creado
105 }
106
107 // Garage es la estructura principal del sistema
108 type Garage struct {
109     clients   []*Client
110     vehicles  []*Vehicle
111     mechanics []*Mechanic
112     issues    []*Incidence
113     slots     []*Slot
114     vpool     []VehicleID        // Cola de vehículos esperando
115     hirereqs  chan HireRequest // Canal para contratación
116 }
```

4.2. main.go - Punto de Entrada

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     var option int
7     var g *Garage
8
9     // Inicializar taller
10    g = newGarage()
11
12    fmt.Printf("-----\n")
13    fmt.Printf("- BIENVENIDO AL TALLER LACOSTE.CO -\n")
14    fmt.Printf("-----\n")
15
16    // Loop principal del menú
17    for {
18        // Revisar alertas de contratación
19    }
20 }
```

```

19     g.checkAlert()
20
21     fmt.Printf("==> Menú Principal ==>\n\n")
22     option = polyAskMenuInt("Seleccione una opción:", 7)
23     g.checkAlert()
24
25     switch option {
26     case 1: // Crear
27         fmt.Printf("==> Menú de Creación ==>\n\n")
28         option = polyAskMenuInt("Seleccione qué crear: ", 8)
29         creationMenu(option, g)
30
31     case 2: // Visualizar
32         fmt.Printf("==> Menú de Visualización ==>\n\n")
33         option = polyAskMenuInt("Seleccione qué visualizar: ",
34                                 9)
34         fmt.Println()
35         visualitationMenu(option, g)
36
37     case 3: // Modificar
38         fmt.Printf("==> Menú de Modificación ==>\n\n")
39         option = polyAskMenuInt("Seleccione qué modificar: ",
40                                 10)
40         fmt.Println()
41         modificationMenu(option, g)
42
43     case 4: // Eliminar
44         fmt.Printf("==> Menú de Eliminación ==>\n\n")
45         option = polyAskMenuInt("Seleccione qué eliminar: ",
46                                 11)
46         fmt.Println()
47         deletionMenu(option, g)
48
49     case 5: // Salir
50         return
51     }
52 }
53 }
```

4.3. new.go - Funciones de Creación (Extracto)

```

1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "time"
8 )
9
```

```

10 // Constructor del taller
11 func newGarage() *Garage {
12     return &Garage{
13         clients:    nil,
14         vehicles:   nil,
15         mechanics:  nil,
16         issues:     nil,
17         slots:      nil,
18         vpool:      nil,
19         hirereqs:   make(chan HireRequest),
20     }
21 }
22
23 // CREAR un CLIENTE
24 func (g *Garage) newClient() {
25     var c Client
26     c.id = ClientID(askUniqueIntID("ID cliente (>0): ",
27         func(n int64) bool { return g.ownerIDExists(ClientID(n)) })
28     )
29     fmt.Printf("Nombre: ")
30     fmt.Scanf("%s", &c.name)
31     fmt.Printf("Tel fono: ")
32     fmt.Scanf("%s", &c.phone)
33     fmt.Printf("Email: ")
34     fmt.Scanf("%s", &c.email)
35     g.clients = append(g.clients, &c)
36 }
37
38 // Genera 2 plazas adicionales
39 func (g *Garage) genSlots() {
40     for i := 0; i < 2; i++ {
41         g.slots = append(g.slots, &Slot{
42             number:    len(g.slots) + 1,
43             vehicleID: nil,
44         })
45     }
46 }
47
48 // CREAR MEC NICO
49 func (g *Garage) newMech() *Mechanic {
50     var m Mechanic
51
52     // Ampliar plazas del taller (2 por mecnico)
53     g.genSlots()
54
55     m.id = MechanicID(askUniqueIntID("ID del mecnico: ",
56         func(n int64) bool { return g.mechIDExists(MechanicID(n)) })
57     )
58     fmt.Printf("Nombre: ")
59     fmt.Scanf("%s", &m.name)
60     fmt.Printf("Experiencia(a os): ")

```

```

59     fmt.Scanf("%d", &m.experience)
60     m.skill = SkillType(polyAskMenuStr("Especialidad:", 1))
61     m.status = MechStatus(polyAskMenuStr("Estado:", 2))
62
63     g.mechanics = append(g.mechanics, &m)
64     return g.mechanics[len(g.mechanics)-1]
65 }
```

4.4. aux.go - Funciones Auxiliares (Extracto)

```

1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7 )
8
9 // Contar plazas libres
10 func (g *Garage) availableSlots() int {
11     var count int = 0
12     for _, s := range g.slots {
13         if s.vehicleID == nil {
14             count++
15         }
16     }
17     return count
18 }
19
20 // Verificar si un mecanico est libre
21 func isFree(m *Mechanic) bool {
22     return len(m.issues) == 0 && m.status == ACTIVE
23 }
24
25 // Pedir un ID nico de tipo entero
26 func askUniqueIntID(prompt string, exists func(int64) bool) int64 {
27     var id int64
28     for {
29         fmt.Print(prompt)
30         fmt.Scanf("%d", &id)
31         if id <= 0 {
32             fmt.Println("Debe ser > 0.")
33             continue
34         }
35         if exists(id) {
36             fmt.Println("Incorrecto, pruebe otro.")
37             continue
38         }
39     }
40     return id
41 }
```

```

40     }
41 }
42
43 // Revisar canal de contratación (no bloqueante)
44 func (g *Garage) checkAlert() {
45     for {
46         select {
47             case req := <-g.hirereqs:
48                 fmt.Printf("-----\n")
49                 fmt.Printf("[ALERTA] Contratar mec nico para %s.\n",
50                           req.vid)
51                 fmt.Printf("-----\n\n")
52                 mech := g.newMech()
53                 req.Reply <- mech
54                 continue
55             default:
56                 return
57         }
58     }
59
60 // Función de testing: cuenta vehículos con una incidencia
61 func (g *Garage) CountVehiclesWithSingleIssue(kind IssueType) int
62 {
63     count := 0
64     for _, v := range g.vehicles {
65         if len(v.issues) != 1 {
66             continue
67         }
68         inc := g.getIssueByID(v.issues[0])
69         if inc != nil && inc.kind == kind {
70             count++
71         }
72     }
73     return count
74 }
75
76 // Función de testing: cuenta mecnicos por especialidad
77 func (g *Garage) countSkill(skill SkillType) int {
78     var count int = 0
79     for _, m := range g.mechanics {
80         if m.skill == skill {
81             count++
82         }
83     }
84     return count
}

```

4.5. modify.go - Asignación Concurrente (Extracto Clave)

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 // Meter un veh culo a una plaza del taller
9 func (g *Garage) addV() {
10     var vid VehicleID
11
12     if g.availableSlots() == 0 {
13         fmt.Printf("No hay plazas disponibles.\n\n")
14         return
15     }
16
17     fmt.Printf("Veh culos no asignados a una plaza:\n")
18     for _, v := range g.vehicles {
19         if !g.vIsInSlot(v.id) {
20             fmt.Printf(" - %s\n", v.id)
21         }
22     }
23     fmt.Println()
24
25     vid = VehicleID(askUniqueStrID("Matr cula del veh culo: ",
26         func(s string) bool { return !g.vidExists(VehicleID(s)) }
27     ))
28
29     // Lanzar goroutine para asignaci n concurrente
30     go g.assignVtoSlot(vid)
31 }
32
33 // Funci n principal de asignaci n (goroutine)
34 func (g *Garage) assignVtoSlot(vid VehicleID) {
35     vehicle := g.getVByID(vid)
36     if vehicle == nil || len(vehicle.issues) == 0 {
37         return
38     }
39
40     // 1) Obtener mecnico principal
41     mainMech := g.obtainMechanicForVehicle(vid)
42
43     // 2) Meter veh culo en plaza
44     if !g.placeVehicleInSlot(vid) {
45         return
46     }
47
48     // 3) Recoger incidencias y calcular ETA
49     issues, eta := g.collectIssuesAndEta(vehicle)
```

```

49     vehicle.eta = eta
50
51     // 4) Asignar mecnico principal
52     g.autoAssignMtoIssues(mainMech, issues)
53
54     // 5) Posible segundo mecnico
55     var extraMech *Mechanic
56     if eta > 15*time.Second {
57         extraMech = g.obtainSecondMechanic(vid)
58         setPrioAndStatus(issues)
59         g.autoAssignMtoIssues(extraMech, issues)
60     }
61
62     // 6) Simular reparacion
63     time.Sleep(eta)
64
65     // 7) Cerrar incidencias y liberar
66     closeIssues(issues)
67     g.extractVfromSlot(vid)
68     for _, inc := range issues {
69         delIssueFromMech(mainMech, inc)
70         if extraMech != nil {
71             delIssueFromMech(extraMech, inc)
72         }
73         delIssueFromV(vehicle, inc)
74     }
75 }
76
77 // Espera activa hasta mecnico libre
78 func (g *Garage) waitForFreeMech() *Mechanic {
79     var m *Mechanic
80     for {
81         m = g.getFreeMech()
82         if m != nil {
83             return m
84         }
85         time.Sleep(1 * time.Second)
86     }
87 }
88
89 // Construir slice de incidencias y calcular ETA
90 func (g *Garage) collectIssuesAndEta(v *Vehicle) ([]*Incidence,
91                                         time.Duration) {
92     var (
93         issues []*Incidence
94         eta    time.Duration
95     )
96
97     for _, issueID := range v.issues {
98         inc := g.getIssueByID(issueID)
99         if inc == nil {

```

```

99         continue
100    }
101    issues = append(issues, inc)
102
103    switch inc.kind {
104        case MECHTYPE:
105            eta += 5 * time.Second
106        case ELECTRICTYPE:
107            eta += 7 * time.Second
108        default:
109            eta += 11 * time.Second
110    }
111 }
112 return issues, eta
113 }

114 // Obtener segundo mecánico (o contratar)
115 func (g *Garage) obtainSecondMechanic(vid VehicleID) *Mechanic {
116     if m := g.getFreeMech(); m != nil {
117         return m
118     }

119     // Enviar solicitud de contratación
120     req := HireRequest{
121         vid:    vid,
122         Reply: make(chan *Mechanic),
123     }
124     g.hirereqs <- req
125     return <-req.Reply
126 }
127 }
128 }
```

5. Conclusiones

5.1. 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

5.2. 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

5.3. 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

5.4. Lecciones Aprendidas

1. Diseño de Tipos Personalizados:

El uso de tipos como ClientID, MechanicID, VehicleID mejoró significativamente la legibilidad del código y previno errores de mezcla de identificadores.

2. 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.

3. Separación de Responsabilidades:

La organización del código en archivos por funcionalidad facilitó el mantenimiento y la comprensión del sistema.

4. 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.

5.5. Trabajo Futuro

Mejoras de Corto Plazo:

- Implementar persistencia con codificación JSON
- Añadir mutexes para acceso seguro a estructuras compartidas
- Ampliar suite de tests con tests de integración
- Validación avanzada de entrada con expresiones regulares

Mejoras de Medio Plazo:

- Interfaz web con servidor HTTP
- API REST para integración con otros sistemas
- Base de datos relacional (SQLite o PostgreSQL)
- Sistema de autenticación y roles de usuario

Mejoras de Largo Plazo:

- Dashboard en tiempo real con WebSockets
- Notificaciones push a clientes
- Sistema de facturación integrado
- Análisis de datos y métricas de rendimiento

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.

7. Apéndices

7.1. Apéndice A: 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
coverage.txt
resultados.txt
```

7.2. 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
```

7.3. Apéndice C: Configuración de LyX

Para importar este documento en LyX:

1. Guardar el archivo .tex generado
2. Abrir LyX
3. **Archivo → Importar → LaTeX (Simple)**
4. Seleccionar el archivo .tex
5. Verificar que se importó correctamente
6. **Archivo → Exportar → PDF (pdflatex)**

Paquetes necesarios en LaTeX:

- inputenc, babel (español)
- amsmath, amsfonts, amssymb
- listings, xcolor
- graphicx, hyperref
- geometry