Taller Práctico 01: Tipos de datos Abstractos y Estructuras Básicas

Curso: Estructuras de Datos y Algoritmos

Período: 2026-10

Duración: 2 semanas

Facultad: TIC - UPB

Modalidad: Equipos de máximo 3 personas

Entrega: 2026-02-25

Descripción General

En este taller práctico integrador, se implementará un **Sistema de Gestión de Tickets de Soporte Técnico** (Help Desk System) que permita a una empresa de software gestionar eficientemente las solicitudes de soporte de sus clientes.

Los sistemas de soporte técnico son fundamentales en cualquier organización de tecnología. Cuando un cliente reporta un problema, se crea un "ticket" que debe ser asignado a un técnico especializado y resuelto de manera oportuna. Este taller te permitirá aplicar conceptos fundamentales de estructuras de datos y algoritmos a un problema real del mundo empresarial.

Objetivos de Aprendizaje

Al completar este taller, serás capaz de:

1. **Diseñar e implementar ADTs (Abstract Data Types)** con encapsulación apropiada y contratos bien definidos
2. **Aplicar estructuras de datos básicas** (Queue, List, Bag) para resolver problemas del mundo real
3. **Implementar iteradores personalizados** que permitan recorrer colecciones con criterios específicos
4. **Escribir pruebas unitarias exhaustivas** que validen la corrección de tu implementación
5. **Tomar decisiones de diseño fundamentadas** sobre qué estructuras de datos usar y por qué
6. **Documentar código de manera profesional** siguiendo convenciones del lenguaje elegido

---

El Problema

Tu equipo de desarrollo ha sido contratado por **TechSupport Inc.**, una empresa de software que necesita modernizar su sistema de gestión de tickets. Actualmente, manejan los tickets de manera manual y desorganizada, lo que resulta en tiempos de respuesta lentos y clientes insatisfechos.

Contexto del Negocio

- **Clientes** reportan problemas técnicos a través del sistema
- Cada problema se registra como un **Ticket** con información relevante
- Los tickets tienen **Estados** que reflejan su progreso: Nuevo, Asignado, En Progreso, Resuelto, Cerrado
- La empresa cuenta con un equipo de **Técnicos**, cada uno con especialidades diferentes (Redes, Bases de Datos, Aplicaciones, Seguridad, etc.)
- Los tickets deben asignarse al técnico apropiado según su especialidad y disponibilidad

Requisitos de Negocio

1. Los tickets se procesan en orden de llegada (FIFO - First In, First Out)
2. Los técnicos solo pueden trabajar en **un ticket a la vez**
3. El sistema debe permitir **reasignar tickets** si un técnico no está disponible
4. Se debe poder generar **reportes** filtrando tickets por diferentes criterios

---

ADTs a Implementar

Deberás diseñar e implementar los siguientes Abstract Data Types. Para cada uno, se proporciona una guía de los atributos y operaciones sugeridos, pero **tienes libertad de agregar o modificar** según tu diseño.

3.1. ADT Ticket

Propósito: Representa una solicitud de soporte técnico.

Atributos sugeridos:

- `id`: Identificador único del ticket
- `descripcion`: Descripción del problema reportado
- `estado`: Estado actual (usar `enum` con valores: Nuevo, Asignado, EnProgreso, Resuelto, Cerrado)
- `categoria`: Categoría del problema (Red, BD, Aplicación, Seguridad, etc.)
- `cliente`: Cliente que reportó el ticket
- `tecnicoAsignado`: Técnico encargado de resolver el ticket (puede ser null si no está asignado)
- `fechaCreacion`: Timestamp de cuándo se creó el ticket

- `fechaResolucion`: Timestamp de cuándo se resolvió (null si aún no se resuelve)

Operaciones requeridas:

- Constructor para crear un nuevo ticket
- `asignarTecnico(tecnico)`: Asigna un técnico al ticket
- `cambiarEstado(nuevoEstado)`: Cambia el estado del ticket
- `resolver()`: Marca el ticket como resuelto
- `cerrar()`: Marca el ticket como cerrado
- Getters para todos los atributos
- `tiempoTranscurrido()`: Calcula el tiempo desde creación (o hasta resolución)

Invariantes:

- El ID debe ser único y no modificable una vez creado
- Un ticket no puede ser resuelto sin estar asignado a un técnico
- Un ticket solo puede cerrarse si está en estado Resuelto
- Las fechas deben ser coherentes (resolución posterior a creación)

3.2. ADT Cliente

Propósito: Representa un cliente que reporta tickets.

Atributos sugeridos:

- `id`: Identificador único
- `nombre`: Nombre completo del cliente
- `email`: Correo electrónico
- `empresa`: Nombre de la empresa (opcional)
- `telefono`: Número de contacto (opcional)

Operaciones requeridas:

- Constructor para crear un cliente
- Getters y setters apropiados
- `toString()` o equivalente para representación textual

Invariantes:

- El ID debe ser único
- El email debe tener formato válido
- El nombre no puede estar vacío

3.3. ADT Técnico

Propósito: Representa un técnico de soporte.

Atributos sugeridos:

- `id`: Identificador único

- `nombre`: Nombre completo del técnico
- `especialidad`: Área de especialización (Red, BD, Aplicación, Seguridad, etc.)
- `ticketAsignado`: El ticket actualmente asignado (solo uno, puede ser null)
- `disponible`: Indica si el técnico está disponible para nuevas asignaciones

Operaciones requeridas:

- Constructor para crear un técnico
- `asignarTicket(ticket)`: Asigna el ticket a trabajar
- `completarTicket()`: Libera al técnico al completar un ticket
- `estaDisponible()`: Verifica si puede recibir un ticket
- `marcarDisponibilidad(disponible)`: Cambia su estado de disponibilidad
- `getTicketAsignado()`: Retorna el ticket actualmente asignado
- Getters para los demás atributos

Invariantes:

- El técnico solo puede tener un ticket asignado a la vez

3.4. ADT TicketSystem (Sistema Principal)

Propósito: Coordina toda la gestión de tickets, técnicos y operaciones del sistema.

Atributos sugeridos:

- `ticketsPendientes`: Cola FIFO de tickets esperando asignación
- `todosLosTickets`: Colección de todos los tickets en el sistema
- `tecnicos`: Colección de técnicos disponibles en el sistema
- `categorias`: Bag con las categorías válidas en el sistema
- `proximoTicketId`: Contador para generar IDs únicos

Operaciones requeridas:

- `crearTicket(descripcion, categoria, cliente)`: Crea y registra un nuevo ticket
- `asignarTicketAutomatico()`: Toma el próximo ticket de la cola y lo asigna al técnico apropiado
- `asignarTicketManual(ticket, tecnico)`: Asigna un ticket a un técnico específico
- `cambiarEstadoTicket(ticketId, nuevoEstado)`: Cambia el estado de un ticket
- `resolverTicket(ticketId)`: Marca un ticket como resuelto
- `cerrarTicket(ticketId)`: Cierra un ticket resuelto
- `agregarTecnico(tecnico)`: Registra un nuevo técnico en el sistema
- `agregarCategoria(categoria)`: Agrega una categoría válida al sistema
- `obtenerTicketsPorEstado(estado)`: Retorna tickets con cierto estado

- `obtenerTicketsPorCategoria(categoria)`: Retorna tickets de una categoría
- `obtenerTicketDeTecnico(tecnico)`: Retorna el ticket asignado a un técnico
- `generarEstadisticas()`: Calcula métricas del sistema

Invariantes:

- Cada ticket debe tener un ID único en el sistema
- Los tickets pendientes se procesan en orden FIFO
- Solo se pueden crear tickets con categorías válidas

Estructuras de Datos Requeridas

Tu implementación **debe utilizar** las siguientes estructuras de datos. Se proporciona la justificación de por qué cada una es apropiada para este problema.

4.1. List (Lista Simple)

Uso: Gestionar tickets pendientes de asignación.

4.2. Bag (Bolsa/Multiconjunto)

Uso: Mantener colección de todos los tickets y colección de técnicos.

Requisitos Funcionales

Tu sistema **debe implementar** las siguientes funcionalidades. Cada operación debe funcionar correctamente y ser demostrable.

5.1. Gestión de Tickets

Crear Nuevo Ticket

- **Input:** Descripción del problema, categoría, información del cliente
- **Output:** Ticket creado con ID único, estado inicial "Nuevo"
- **Validación:** La categoría debe existir en el Bag de categorías válidas
- **Efecto:** Ticket agregado al sistema y a la cola de pendientes

Asignar Ticket a Técnico (Automático)

- **Input:** Ninguno (toma el siguiente ticket de la cola)
- **Proceso:**
  1. Obtener siguiente ticket de la cola FIFO
  2. Buscar técnicos con la especialidad requerida
  3. Entre ellos, seleccionar uno disponible (sin ticket asignado)

4. Asignar el ticket y cambiar estado a “Asignado”

- **Output:** Ticket asignado al técnico seleccionado
- **Manejo de errores:** Si no hay técnicos disponibles, el ticket permanece en la cola

Asignar Ticket a Técnico (Manual - Opcional)

- **Input:** Ticket y técnico específico
- **Validaciones:** Verificar que el técnico esté disponible (sin ticket asignado)
- **Output:** Ticket asignado al técnico especificado

Cambiar Estado de Ticket

- **Input:** ID del ticket, nuevo estado
- **Validaciones:**
  - Transiciones válidas (ej: no se puede ir de “Nuevo” a “Resuelto” directamente)
  - Ticket debe estar asignado para cambiar a “En Progreso”
- **Efecto:** Actualiza el estado del ticket

Resolver Ticket

- **Input:** ID del ticket
- **Validaciones:** Ticket debe estar en estado “En Progreso”
- **Efecto:**
  - Cambiar estado a “Resuelto”
  - Registrar fecha de resolución
  - Liberar al técnico (ahora está disponible para otro ticket)

Cerrar Ticket

- **Input:** ID del ticket
- **Validaciones:** Ticket debe estar en estado “Resuelto”
- **Efecto:** Cambiar estado a “Cerrado”

Reabrir Ticket (Opcional - no es necesario implementar)

- **Input:** ID del ticket cerrado
- **Efecto:**
  - Cambiar estado a “Nuevo”
  - Agregar nuevamente a la cola de pendientes
  - Liberar técnico si estaba asignado

5.2. Gestión de Técnicos

Para simplificar el ejercicio los técnicos se crearan al iniciar el programa usando valores predefinidos. No es necesario implementar operaciones de creación/modificación/borrado de técnicos.

5.3. Gestión de Categorías

Para simplificar el ejercicio las categorias se crearan al iniciar el programa usando valores predefinidos. No es necesario implementar operaciones de creación/modificación/borrado de categorias.

5.4. Consultas y Reportes (Uso de Iteradores)

- Listar Tickets Pendientes
- Listar todos los Tickets resueltos.

Requisitos Técnicos

Tu implementación debe cumplir los siguientes estándares técnicos.

6.1. Encapsulación

Tu implementación debe seguir los principios de encapsulación vistos en clase:

6.2. Uso de Enums

Los estados y las categorias deben implementarse usando **enum** (o su equivalente según el lenguaje): - Python: usar `enum.Enum` - Java: usar `enum` - C#: usar `enum`

6.3. Iteradores

Implementar los iteradores requeridos sobre las colecciones implementadas en el proyecto.

Los iteradores deben seguir el patrón Iterator del lenguaje elegido (Python: `__iter__()` y `__next__()`, Java: `Iterator<T>`, C#: `IEnumerator<T>`).

6.4. Pruebas

Requerimiento: Mínimo **10 casos de prueba** implementados en métodos `main` de las clases usando sentencias `assert`.

Categorías de pruebas requeridas:

- Creación de objetos ADT
- Operaciones básicas (getters/setters)
- Cambios de estado
- Asignación de tickets
- Validaciones (casos que deben fallar)
- Iteradores
- Operaciones del sistema completo

6.5. Manejo de Excepciones

Validar entradas y lanzar excepciones cuando sea apropiado, por ejemplo:

- Intentar operar con ID de ticket inexistente
- Intentar asignar ticket a técnico ocupado
- Intentar transición de estado inválida
- Crear ticket con categoría inválida

No es necesario crear clases de excepción personalizadas complejas; puedes usar las excepciones estándar del lenguaje con mensajes descriptivos.

Entregables

Deberás entregar los siguientes componentes a través del sistema de **Aula Digital** de la universidad.

7.1. Código Fuente Completo

Contenido:

- Implementación de todos los ADTs: Ticket, Cliente, Técnico, TicketSystem
- Implementación de estructuras de datos requeridas (o uso apropiado de las de la biblioteca estándar según indique el instructor)
- Implementación de al menos 2 iteradores personalizados
- Enums para Estados y Categorias.
- Métodos main con pruebas (mínimo 10 asserts en total)

IMPORTANTE: Todos los archivos fuente deben incluir en comentarios al inicio:

```
/*
 * Taller Práctico 01 - Sistema de Gestión de Tickets
 * Estructuras de Datos y Algoritmos - 2026-10
 *
 * Integrantes:
 * - Nombre Completo 1 - ID: 123456
 * - Nombre Completo 2 - ID: 234567
 * - Nombre Completo 3 - ID: 345678
 */
```

Calidad:

- Código indentado correctamente
- Variables y métodos con nombres descriptivos
- Comentarios para lógica compleja
- Sin código comentado o no utilizado

7.2. Archivo README.md

Contenido requerido:

Sección 1: Información del Equipo

- Nombres completos e IDs de los integrantes
- Lenguaje y versión utilizados

Sección 2: Descripción del Proyecto

- Breve explicación de qué hace el sistema

Sección 3: Estructura del Proyecto

- Lista de archivos principales y qué contiene cada uno
- Diagrama de estructura (árbol de directorios)

Sección 4: Instrucciones de Compilación/Ejecución

- Comandos exactos para compilar (si aplica)
- Comandos para ejecutar el programa principal
- Cómo ejecutar las pruebas (main de cada clase)

Sección 5: Decisiones de Diseño

- Breve explicación de decisiones importantes que tomaron
- Por qué eligieron ciertas implementaciones

Formato: Markdown (`.md`)

7.3. Documento prompts.md

Documentación del uso de asistentes de IA.

Propósito: Registrar cómo utilizaron herramientas de IA (ChatGPT, Claude, Copilot, etc.) durante el desarrollo.

Contenido para cada prompt usado:

```
## Prompt [Número]

**Fecha:** YYYY-MM-DD

**Herramienta:** [ChatGPT / Claude / Copilot / Otra]

**Objetivo:** ¿Qué intentabas lograr?

**Prompt usado:**
```

[Texto exacto del prompt que usaste]

````
**Resultado:**
- [ ] Correcto en el primer intento
- [ ] Incorrecto, pero se corrigió con realimentación (indicar cuántas iteraciones)
- [ ] Requirió desarrollo manual posterior

**Descripción del resultado:**
[Breve explicación de qué tan útil fue la respuesta y qué ajustes hiciste]

**Archivo afectado:** Nombre del archivo fuente.
````

Ejemplo:

````
## Prompt 1

**Fecha:** 2026-10-15

**Herramienta:** ChatGPT

**Objetivo:** Implementar el iterador por estado en Python

**Prompt usado:**
```md
Necesito implementar un iterador en Python que filtre una lista de
tickets por estado. Los tickets tienen un método getEstado() que
retorna un enum Estado.

**Resultado:**
- [x] Incorrecto, pero se corrigió con realimentación (2 iteraciones)

**Descripción del resultado:**
La primera versión no manejaba correctamente el StopIteration.
Tuve que pedir que corrigiera eso y funcionó en la segunda iteración.

**Código generado:**
```python
class TicketsByStateIterator:
    def __init__(self, tickets, estado):
        self.tickets = [t for t in tickets if t.getEstado() == estado]
        self.index = 0
    # ... resto del código


**Nota:** Si no usaron asistentes de IA, incluir un archivo `prompts.md` que simplemente dig
```markdown
# Uso de Asistentes de IA
````
````

`No se utilizaron asistentes de inteligencia artificial en el desarrollo de este proyecto.`

7.4. Programa de Demostración (Opcional pero Recomendado)

Propósito: Demostrar que tu sistema funciona end-to-end.

Contenido:

- Archivo `main` o `demo` que ejecute escenarios completos
- Debe mostrar:
  1. Crear tickets
  2. Asignar tickets automáticamente
  3. Cambiar estados de los tickets: Resolver y cerrar tickets.
  4. Generar reportes usando iteradores

Salida: Texto en consola mostrando cada operación y sus resultados.

Criterios de Éxito

Usa esta lista para verificar que tu taller está completo antes de entregar.

Implementación de ADTs

- ☐ ADT Ticket implementado con enum Estado
- ☐ ADT Cliente implementado correctamente
- ☐ ADT Técnico implementado (un ticket a la vez)
- ☐ ADT TicketSystem implementado como coordinador central
- ☐ Todos los ADTs usan encapsulación apropiada

Estructuras de Datos

- ☐ List usada para colección de tickets
- ☐ Bags para las tategorias de técnicos

Requisitos Funcionales

- ☐ Sistema puede crear tickets con categorías válidas
- ☐ Asignación automática FIFO funciona
- ☐ Técnico solo tiene un ticket a la vez
- ☐ Cambio de estado con validaciones funciona
- ☐ Resolver ticket libera al técnico

Iteradores

- ☐ Iterator por estado implementado y funcional
- ☐ Iteradores siguen el patrón del lenguaje elegido

Pruebas

- ☐ Mínimo 10 casos de prueba (asserts) en métodos main
- ☐ Pruebas cubren casos normales y casos límite
- ☐ Todas las pruebas pasan

Documentación

- ☐ README completo con todas las secciones
- ☐ Todos los archivos fuente tienen nombres e IDs de integrantes
- ☐ Documento prompts.md incluido (con contenido o indicando que no se usó IA)

Calidad del Código

- ☐ Código indentado y formateado correctamente
- ☐ Variables y métodos con nombres descriptivos
- ☐ Sin código comentado o no utilizado
- ☐ Validaciones en lugares necesarios

Formato de Entrega

Sistema: Aula Digital de la Universidad

Archivo: Comprimido (.zip)

Nombre del archivo: `Taller01_ApellidoID1_ApellidoID2_ApellidoID3.zip`

Ejemplo: `Taller01_Perez123456_Lopez234567_Garcia345678.zip`

Contenido del .zip:

- Carpeta con código fuente (todos los archivos .py, .java, .cs)
- README.md
- prompts.md

NO incluir:

- Archivos binarios compilados (.class, .pyc, .exe)
- Carpetas de IDEs (.idea, .vscode, ___pycache___)
- Archivos temporales