PRÁCTICA 02 - Modelado y Programación

Equipo "Better Code Saul"

Integrantes

Nombre	No. de cuenta
Álcantara Estrada Kevin Isaac	319073799
Cureño Sánchez Misael	418002485
Hernández Páramo Elizabeth	319143209

Intrucciones de compilacion

Forma 1

1. Dirigirse al directorio `src` de la práctica.
2. Compilar usando `javac -sourcepath . -d ../target/ -cp .:../lib/* ./p02/Main.java`
3. Dirigirse al directorio generado `cd ../target`.
4. Ejecutar usando `java -cp .:../lib/* p02.Main`.

Forma 2 (Linux)

1. Ejecuta el script haciendo `./run.sh` o bien `bash run.sh` desde la terminal, dentro de la carpeta raiz de la práctica (en caso de no tener permiso de ejecución ejecutar `chmod +x run.sh`).

SECCIÓN TEÓRICA

Menciona los principios de diseño esenciales del patrón Template, State e Iterator.

TEMPLATE

Como sabemos, el patrón Template es de comportamiento ya que nos ayudan a resolver problemas de interacción entre clases y objetos. Visto en clase, este patrón define los pasos de un algoritmo y permite que las subclases proporcionen la implementación de uno o más pasos sin cambiar la estructura del

algoritmo. Cuenta con pasos abstractos y opcionales, pero también existen los pasos llamados hooks (ganchos), es un paso opcional con un cuerpo vacío.

ESTRUCTURA

- **Clase abstracta**: Declaran los métodos que actúan como pasos de un algoritmo. Esos pasos pueden declararse abstractos o contar con una implementación por defecto.
- **Clases concretas**: Pueden sobrescribir todos los pasos, pero no el propio método por defecto.

IMPLEMENTACIÓN

1. Analizamos el algoritmo para ver si se puede dividir en pasos. Consideramos qué pasos son comunes a todas las subclases y cuáles siempre serán únicos.
2. Creamos la clase abstracta y declaramos el método plantilla y un grupo de métodos abstractos que representen los pasos del algoritmo.
3. A algunos pasos les vendría bien tener una implementación por defecto.
4. Pensamos en añadir hooks entre los pasos del algoritmo.
5. Para cada variación del algoritmo, creamos una nueva subclase concreta. Debe implementar todos los pasos abstractos, pero también puede sobrescribir algunos de los opcionales.

DESVENTAJA

Algunos clientes pueden verse limitados por el esqueleto proporcionado de un algoritmo.

STATE

Como se vio en clase, el patrón state permite a un objeto alterar su comportamiento cuando su estado interno cambia, el objeto aparenta cambiar su clase. Se usa cuando una clase tiene distintas fases (estados) y en cada una de estas su comportamiento es distinto.

ESTRUCTURA

- **La clase contexto**: Almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la

interfaz de estado. El contexto expone un modificador (setter) para pasarle un nuevo objeto de estado.

- **La interfaz Estado**: Declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.
- **Estados concretos**: Proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

IMPLEMENTACIÓN

1. Decide qué clase actuará como contexto. Puede ser una clase existente que ya tiene el código dependiente del estado, o una nueva clase, si el código específico del estado está distribuido a lo largo de varias clases.
2. Declara la interfaz de estado. Aunque puede replicar todos los métodos declarados en el contexto, céntrate en los que pueden contener comportamientos específicos del estado.
3. Para cada estado actual, crea una clase derivada de la interfaz de estado. Después repasa los métodos del contexto y extrae todo el código relacionado con ese estado para meterlo en tu clase recién creada.
4. En la clase contexto, añade un campo de referencia del tipo de interfaz de estado y un modificador (setter) público que permita sobrescribir el valor de ese campo.
5. Vuelve a repasar el método del contexto y sustituye los condicionales de estado vacíos por llamadas a métodos correspondientes del objeto de estado.
6. Para cambiar el estado del contexto, crea una instancia de una de las clases de estado y pásala a la clase contexto. Puedes hacer esto dentro de la propia clase contexto, en distintos estados, o en el cliente. Se haga de una forma u otra, la clase se vuelve dependiente de la clase de estado concreto que instancia.

DESVENTAJA

Aplicar el patrón puede resultar excesivo si una máquina de estados sólo tiene unos pocos estados o raramente cambia.

ITERATOR

Como lo vimos en clase, el patrón Iterator provee una forma de acceder a los elementos de un objeto que los contiene de manera secuencial, sin exponer su representación interna. Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

ESTRUCTURA

- **La Interfaz Iteradora**: Declara las operaciones necesarias para recorrer una colección: extraer el siguiente elemento, recuperar la posición actual, reiniciar la iteración, etc.
- **Los Iteradores Concretos**: Implementan algoritmos específicos para recorrer una colección. El obejto iterador debe controlar el progreso del recorrido por su cuenta. Esto permite a varios iteradores recorrer la misma colección con independencia entre sí.
- **El Cliente**: Debe funcionar con colecciones e iteradores a través de sus interfaces. De este modo, el cliente no se acopla a clases concretas, permitiéndote utilizar varias colecciones e iteradores con el mismo código cliente.

IMPLEMENTACIÓN

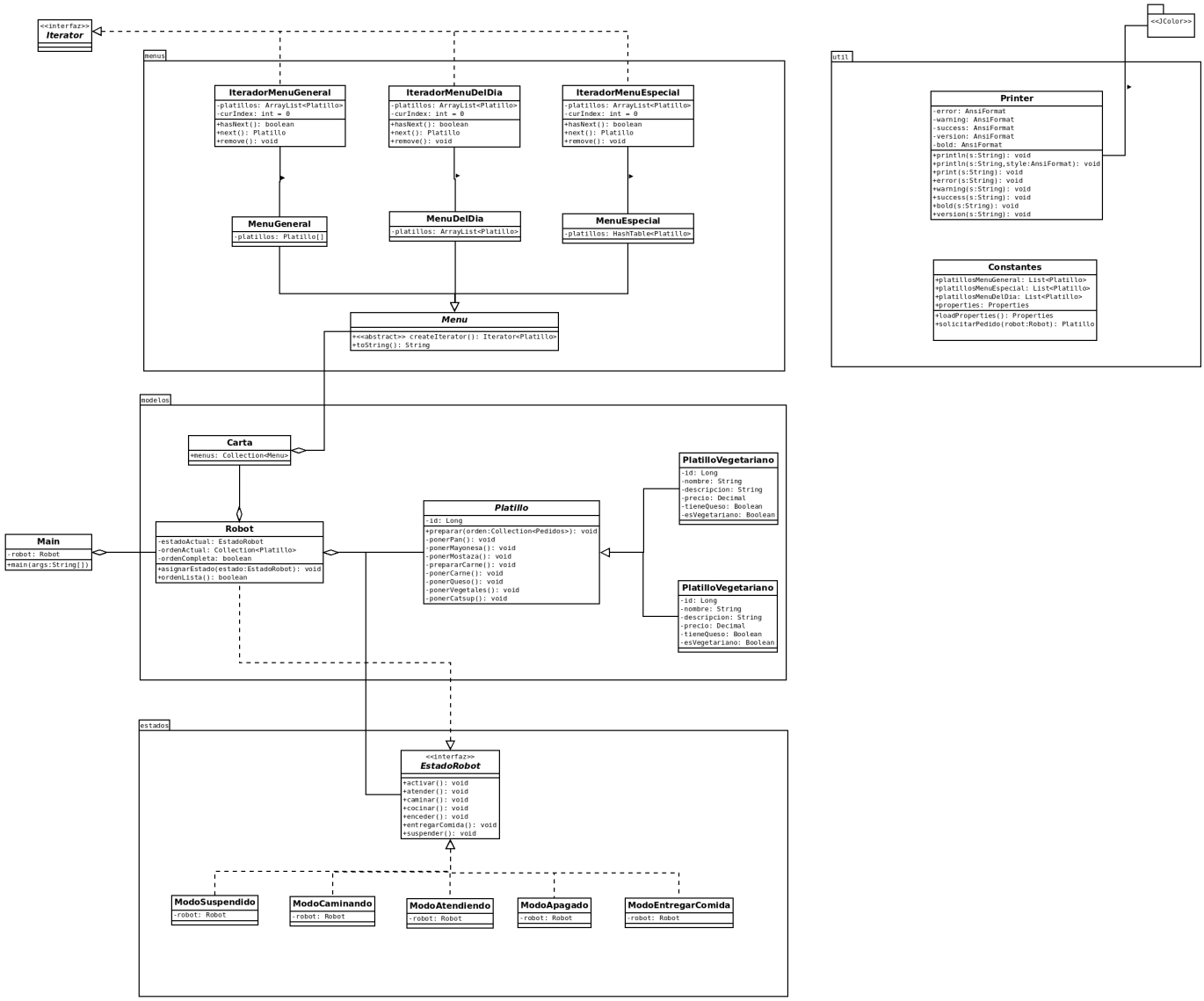
1. Declara la interfaz iteradora. Debe tener un método para extraer el siguiente elemento de una colección.
2. Declara la interaz de colección y describe un método para buscar iteradores. El tipo de retorno debe ser igual al de la interfaz iteradora.
3. Implementa clases iteradoras concretas para las colecciones que quieras que sean recorridas por iteradores.
4. Implementa la interfaz de colección en tus clases de colección. La idea principal es proporcionar al cliente un atajo para crear iteradores personalizados para una clase de colección particular.
5. Repasa el código cliente para sustituir todo el código de recorrido de la colección por el uso de iteradores.

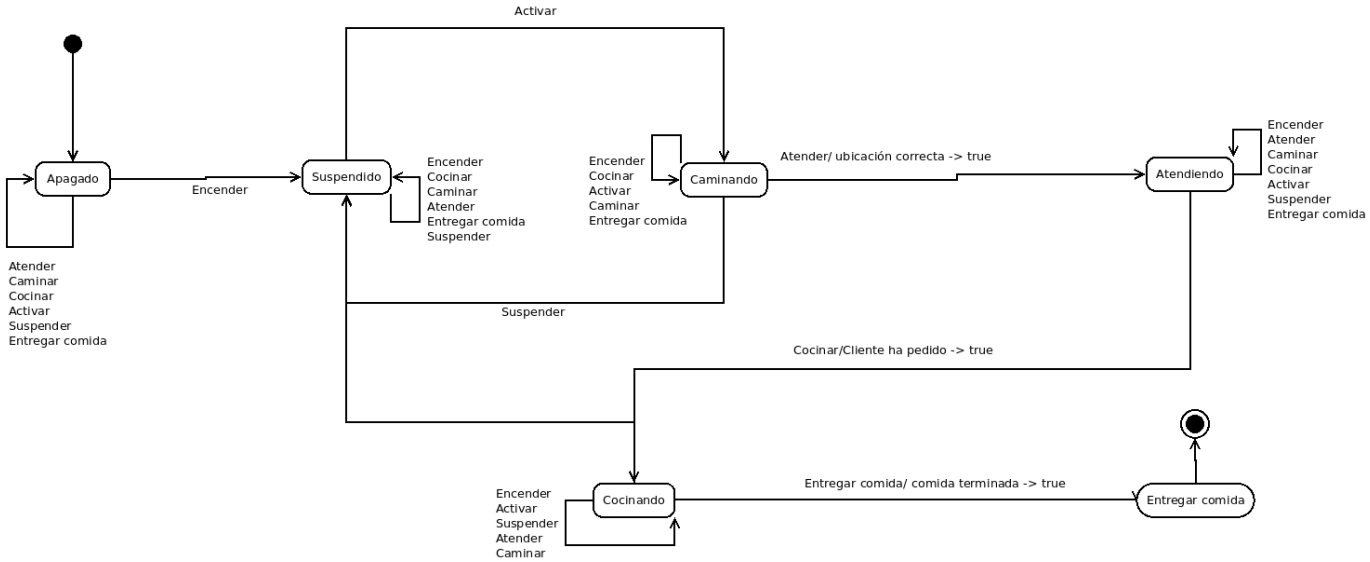
DESVENTAJA

Aplicar el patrón puede resultar excesivo si tu aplicación funciona únicamente con colecciones sencillas.

Utilizar un iterador puede ser menos eficiente que recorrer directamente los elementos de algunas colecciones especializadas.

Diagramas


<<interfaz>>
Iterator
menus
IteradorMenuGeneral
-platillos: ArrayList<Platillo>
-curIndex: int = 0
+hasNext(): boolean
+next(): Platillo
+remove(): void
IteradorMenuDelDia
-platillos: ArrayList<Platillo>
-curIndex: int = 0
+hasNext(): boolean
+next(): Platillo
+remove(): void
IteradorMenuEspecial
-platillos: ArrayList<Platillo>
-curIndex: int = 0
+hasNext(): boolean
+next(): Platillo
+remove(): void
MenuGeneral
-platillos: Platillo[]
MenuDelDia
-platillos: ArrayList<Platillo>
MenuEspecial
-platillos: HashTable<Platillo>
Menu
+<<abstract>> createIterator(): Iterator<Platillo>
+toString(): String
<<JColor>>
util
Printer
-error: AnsiFormat
-warning: AnsiFormat
-success: AnsiFormat
-version: AnsiFormat
-bold: AnsiFormat
+println(s:String): void
+println(s:String,style:AnsiFormat): void
+print(s:String): void
+error(s:String): void
+warning(s:String): void
+success(s:String): void
+bold(s:String): void
+version(s:String): void
Constantes
+platillosMenuGeneral: List<Platillo>
+platillosMenuEspecial: List<Platillo>
+platillosMenuDelDia: List<Platillo>
+properties: Properties
+loadProperties(): Properties
+solicitarPedido(robot:Robot): Platillo
modelos
Carta
+menus: Collection<Menus>
Main
-robot: Robot
+main(args:String[])
Robot
-estadoActual: EstadoRobot
-ordenActual: Collection<Platillo>
-ordenCompleta: boolean
+asignarEstado(estado:EstadoRobot): void
+ordenLista(): boolean
Platillo
-id: Long
+preparar(orden:Collection<Pedidos>): void
-ponerPan(): void
-ponerMayonesa(): void
-ponerMostaza(): void
-prepararCarne(): void
-ponerCarne(): void
-ponerQueso(): void
-ponerVegetales(): void
-ponerCatsup(): void
PlatilloVegetariano
-id: Long
-nombre: String
-descripcion: String
-precio: Decimal
-tieneQueso: Boolean
-esVegetariano: Boolean
PlatilloVegetariano
-id: Long
-nombre: String
-descripcion: String
-precio: Decimal
-tieneQueso: Boolean
-esVegetariano: Boolean
estados
<<interfaz>>
EstadoRobot
+activar(): void
+atender(): void
+caminar(): void
+cocinar(): void
+enceder(): void
+entregarComida(): void
+suspender(): void
ModoSuspendido
-robot: Robot
ModoCaminando
-robot: Robot
ModoAtendiendo
-robot: Robot
ModoApagado
-robot: Robot
ModoEntregarComida
-robot: Robot


Activar
Apagado
Encender
Suspendido
Encender
Cocinar
Caminar
Atender
Entregar comida
Suspender
Encender
Cocinar
Activar
Caminar
Entregar comida
Caminando
Atender/ ubicación correcta -> true
Atendiendo
Encender
Atender
Caminar
Cocinar
Activar
Suspender
Entregar comida
Atender
Caminar
Cocinar
Activar
Suspender
Entregar comida
Suspender
Cocinar/Cliente ha pedido -> true
Encender
Activar
Suspender
Atender
Caminar
Cocinando
Entregar comida/ comida terminada -> true
Entregar comida