Programación II - Trabajo Práctico Integrador 1er Cuatrimestre 2023

SEGUNDA PARTE

Universidad Nacional de General Sarmiento

Integrantes del grupo:

Moreno, Lautaro Emmanuel
DNI: 44112501
lemoreno2002@gmail.com
Schaab, Julieta Daiana
DNI: 43776754
julietaschaab13@gmail.com

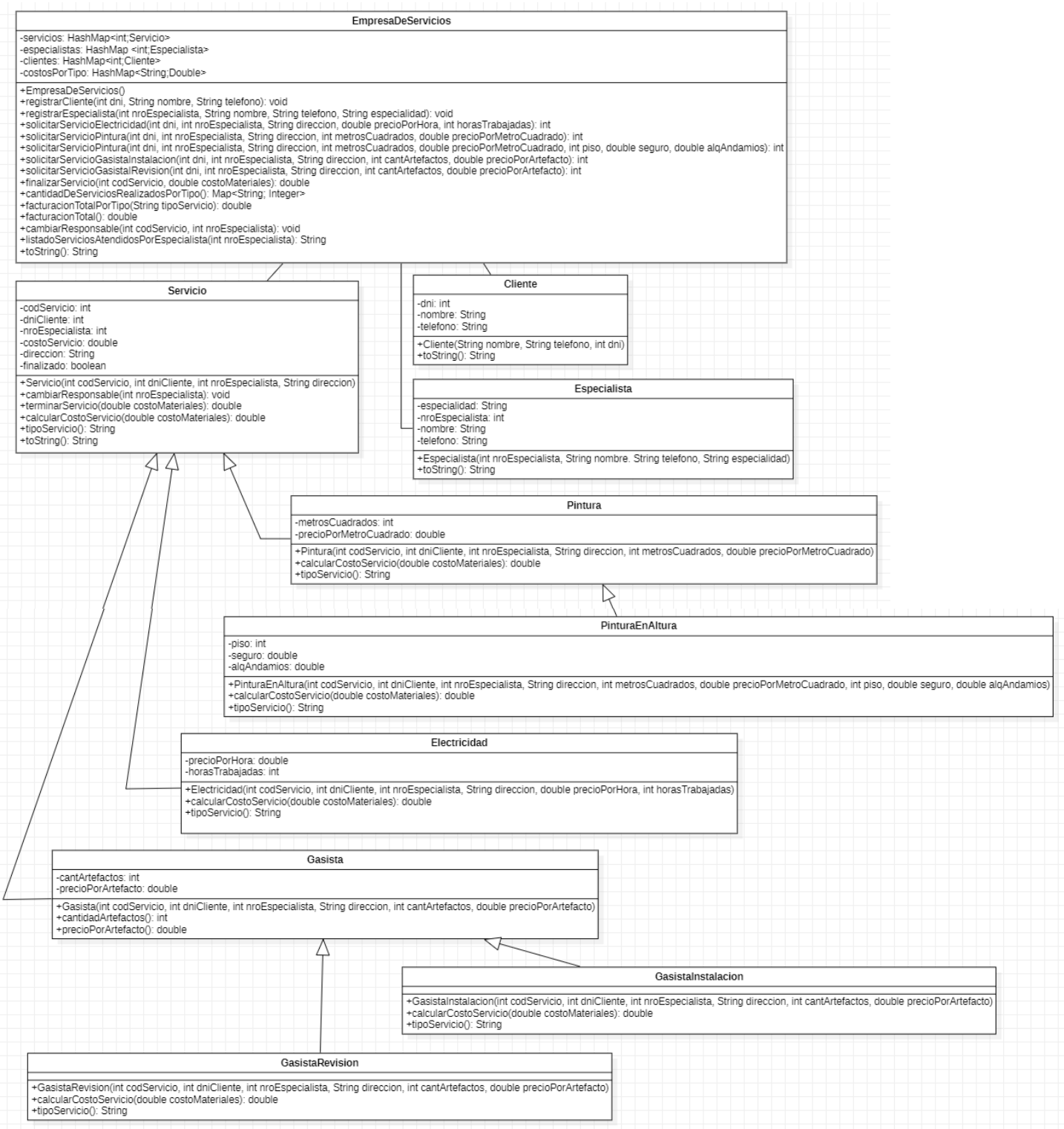
Docentes:

Adriana Gaudiani
Daniel Carrillo

Introducción

En este informe, se buscará exponer la implementación que realizamos en esta segunda etapa del trabajo práctico con el fin de cumplir con los puntos señalados en el enunciado. Incluyendo el diagrama de clases actualizado de la primera parte del trabajo, el análisis de complejidad para las funciones solicitadas y el invariante de representación para cada TAD desarrollado en nuestra solución. También, expondremos dónde y de qué manera utilizamos las Tecnologías Java solicitadas y los conceptos de abstracción, herencia, polimorfismo, sobreescritura, sobrecarga e interfaces.

Diagrama de clases actualizado



Conceptos utilizados en la implementación

Herencia

Mediante la herencia, implementamos una superclase `Servicio` de la cual derivan tres subclases, que heredan sus atributos y métodos, las cuales definen los distintos tipos de servicios: `ServicioGasista`, `ServicioPintura` y `ServicioElectricidad`.
De la misma manera, en las clases `ServicioGasista` y `ServicioPintura` utilizamos el concepto de herencia. De la clase `ServicioPintura` deriva la subclase `ServicioPinturaEnAltura` y de la clase `ServicioGasista` derivan dos subclases: `ServicioGasistaInstalacion` y `ServicioGasistaRevision`.

Polimorfismo

Aplicamos el concepto de polimorfismo cuando alguna instancia del TAD `Servicio`, o de sus subclases, utilizan los métodos: `terminarServicio()` - `calcularCostoServicio()` - `tipoServicio()` - `toString()`. Ya que, cada instancia de tipo `Servicio` resuelve esos métodos de forma distinta , principalmente el método de `calcularCostoServicio()`, ya que cada tipo de servicio calcula su costo utilizando datos muy diferentes.

Abstracción

La superclase `Servicio` es abstracta, ya que no se debe poder instanciar un servicio sin saber su tipo. Además, su método `calcularCostoServicio()` es abstracto también ya que en principio cada tipo de servicio calcula su costo de una forma distinta al resto.
Otra clase que decidimos implementar como clase abstracta es `ServicioGasista`, ya que no se puede instanciar un servicio gasista sin especificar si es para instalación o revisión.

Sobreescritura

Sobreescribimos el método `tipoServicio()` del TAD `Servicio` en cada una de las subclases derivadas. De esta forma, al llamar a este método se obtiene el tipo de servicio distinto de cada instancia.

Sobrecarga

Sobrecargamos el método auxiliar `validarDatos()` del TAD `EmpresaDeServicios` (el cual se encarga de verificar si los datos pasados por parámetro cumplen las condiciones necesarias para usarse, en caso contrario se lanza una excepción) para poder validar mayor cantidad de parámetros. Se usa al momento de solicitar un servicio de algún tipo con los siguientes parámetros: `validarDatos(int dni,int nroEspecialista,String especialista,double valor1,double valor2)`. En el caso particular de tener que solicitar un servicio de tipo `PinturaEnAltura`, se validan más datos. En este caso, el método se define de la siguiente forma: `validarDatos(int dni,int nroEspecialista,String`

`especialista,double valor1,double valor2,double valor,double valor4,double valor5).`

Implementación de Tecnologías Java

Iteradores

Utilizamos Iteradores en los métodos `cantidadDeServiciosRealizadosPorTipo()` y `listadoServiciosAtendidosPorEspecialista(int nroEspecialista)`, con el fin de recorrer fácilmente la colección de valores de los servicios registrados por la empresa .

StringBuilder

Hicimos uso del objeto StringBuilder dentro del TAD `EmpresaDeServicios` en los métodos `listadoServiciosAtendidosPorEspecialista()` - `toString()`. Ya que en estos métodos se tiene que recorrer estructuras y almacenar los datos constantemente en un String, lo cual no es óptimo si usamos un String y concatenamos los datos. Por esta razón, optamos por usar un StringBuilder que guarde los datos y al final se retorna con el método toString de dicho objeto.

ForEach

Utilizamos ForEach en el método toString de la empresa, el cuál necesita recorrer la colección de valores de los servicios, de los clientes y de los especialistas registrados en la empresa. Por esta razón, nos pareció más eficiente utilizar ForEach en lugar de inicializar tres Iteradores.

Análisis de complejidad

Complejidad del método listadoServiciosAtendidosPorEspecialista()

A continuación, por medio de álgebra de órdenes, calculamos la complejidad del método `listadoServiciosAtendidosPorEspecialista()` que implementamos junto con el conteo de instrucciones, el cual utiliza el método auxiliar `especialistaEstaRegistrado(int nroEspecialista)`.

```
public String listadoServiciosAtendidosPorEspecialista(int nroEspecialista) {
     StringBuilder historialEspecialista = new StringBuilder(); → 1
     if (! especialistaEstaRegistrado(nroEspecialista)) { → 3
          throw new RuntimeException("El especialista no se encuentra registrado");
     }
     Iterator <Servicio> iterador = servicios.values().iterator(); → 3
     while (iterador.hasNext()) { → n+1
          Servicio servicio = iterador.next(); → 2*n
          if (servicio.getEspecialista() == nroEspecialista) { → 2*n
               historialEspecialista.append(servicio.toString()).append("\n"); → 3*n
          }
     }
     return historialEspecialista.toString(); → 2
}

private boolean especialistaEstaRegistrado(int nroEspecialista) {
     return this.especialistas.containsKey(nroEspecialista); → 2
}
```

f(n)= 1 + 3 + 3 + (n +1) + n (2 + 2 + 3) + 2 = 10 + 8n

Por álgebra de órdenes :O(10 + 8n)

O(10) + O(8n)	**(Por propiedad 2)**
O(10) + O(8) * O(n)	**(Por propiedad 3)**
O(1) + O(1) * O(n)	**(Por propiedad 4)**
O(1) + O(n)	**(Por propiedad 3)**
O(max{1,n})	**(Por propiedad 2)**
O(n)	**(Por propiedad 1)**

Teniendo en cuenta que n = servicios.values().size() (n representa la cantidad de servicios registrados) y que el peor caso es cuando todos los servicios registrados son del mismo especialista, queda demostrado que la complejidad del método `listadoServiciosAtendidosPorEspecialista()` es O(n).

Historial de servicios de un cliente particular (coloquialmente)

Suponiendo que se quiera conocer el historial de servicios de un cliente en particular, se tendría que pedir el DNI de ese cliente y recorrer el registro con los servicios registrados. Por cada servicio debe preguntar si el DNI que tiene registrado es el mismo DNI del cliente solicitado. De esa forma, puedo armarme un listado con todos los servicios que tengan el DNI del cliente particular. Este método tiene una complejidad de O(n) ya que

debe recorrer una estructura de servicios con un largo de n entradas, hace operaciones constantes y luego retorna el listado con los servicios.

Invariante de Representación

TAD EmpresaDeServicios

El TAD `EmpresaDeServicios` lleva un registro de los clientes y especialistas que se registran, y los servicios que se van solicitando. No pueden estar repetidos los DNI de los clientes, los números de especialista y los códigos de los servicios.

TAD Cliente

Debe respetar que el DNI instanciado sea válido, es decir, que tenga entre 7 y 8 dígitos.

TAD Especialista

El especialista puede tener solo un tipo de especialidad y todos los servicios que atienda deben ser de dicho tipo.

TAD Servicio

Al solicitar el servicio, se debe ingresar un número de dni del cliente y un número de especialista, ambos registrados en la empresa, dicho especialista debe estar especializado en el tipo de servicio solicitado.

IREP general para todos los tipos de servicio

Los datos numéricos, que se registran para calcular el costo del servicio, deben ser mayores a 0 estrictamente. Como los tipos de servicios son subclases del TAD Servicio, heredan su mismo tipo de IREP.