Informe proceso de análisis y diseño de la solución

Desafío II

Samuel Zuleta Zapata
Juan Fernando Henao Ledesma

Informática II

Universidad de Antioquia

2025

Tras haber analizado los requerimientos para la plataforma UdeAStay y teniendo en cuenta el paradigma de POO, consideramos que el paso inicial para nuestra solución es realizar un proceso de abstracción de la realidad y determinar cuántas clases van a ser necesarias. Como solución inicial se nos ocurren 5 clases, clase anfitrión, clase alojamiento, clase reservación, clase huésped y finalmente clase fecha. Como dice en el enunciado un anfitrión puede tener numerosos alojamientos, un alojamiento puede tener numerosas reservaciones y un huésped puede tener numerosas reservaciones. Fecha sería una clase que nos brindaría utilidad a la hora de verificar disponibilidades, fechas válidas, cálculos de estadías, entre otras funciones.

Los atributos de las clases están prácticamente explícitos en el documento, lo verdaderamente importante es analizar cuál va a ser la estructura de datos que vamos a usar y cómo se van a relacionar entre ellos, la primera idea es que anfitrión tenga un arreglo de objetos de la clase alojamientos, alojamientos tenga un arreglo de la clase reservaciones y huéspedes tenga un puntero a objetos tipo reservación, la clase fecha estaría incluida como atributo en la clase reservación. Todo esto al ser un análisis inicial está sujeto a cambios claramente, sin embargo, en este momento nos parece una ruta viable.

Otro punto de suma importancia es el almacenamiento permanente, se define que se van a usar 5 archivos txt, uno para los huéspedes, uno para los anfitriones, uno para los alojamientos y 2 para las reservaciones, siendo uno el histórico y otro para las reservas vigentes, el formato de estos se sigue evaluando y estará evidenciado en el informe final para la legibilidad del código. La parte principal de la funcionalidad de carga es el orden en el que se va a realizar y cómo se van a ir instanciando los objetos en cuestión, esta funcionalidad debe ir abriendo archivo por archivo, crear el objeto y agregarlo a la estructura de datos correspondiente.

Otra funcionalidad clave es la medición de los recursos, las iteraciones por funcionalidad se pueden resolver simplemente a través de contadores estáticos en los bucles más internos de los métodos que se usen, la medición de la memoria es algo más complejo, ya que el reto va a estar en los datos estructurados de nuestros objetos, hay algunas clases que tienen arreglos de otros objetos, teniendo que calcular no solo datos primitivos sino datos altamente estructurados. Para ser lo más eficientes posibles en términos de memoria es esencial el correcto uso de las referencias para evitar hacer copias innecesarias de datos pesados, aparte de claramente el uso de la memoria dinámica y su correcta liberación.

El problema también plantea varias restricciones que deben ser verificadas en el código, tanto las más banales como que la fecha sea válida, hasta que el huésped no

tenga reservas para la fecha en la que quiere reservar o que el alojamiento esté disponible durante toda la totalidad de la reserva, es esencial manejar estos casos correctamente para conservar la lógica interna del problema. La funcionalidad 3 es lo más esencial del programa y por ello debe ser implementado con mucho cuidado.

Tras este análisis inicial, consideramos que tenemos una base para la posterior implementación, detectamos antes de empezar a codificar posibles dificultades como la correcta vinculación de los huéspedes a las reservaciones, sin embargo, esto es algo que se seguirá analizando.

IMPLEMENTACION.

La cantidad de archivos txt que se definieron en el análisis incial se conservó durante la implementación, estos siguen un formato de la siguiente manera:

- anfitriones.txt:
  documento;antigüedad;puntuación

- huespedes.txt:
  documento;antigüedad;puntuación
- alojamientos.txt:
  código;documentoAnfitrion;departamento;municipio;tipo;dirección;precioNoche;amenidadesSeparadasPorComas

- reservas.txt y reservas_historicas.txt:
  codigoReserva;codigoAlojamiento;documentoHuesped;fechaInicio;noches;metodoPago;fechaPago;monto;nota

La solución posee algoritmos que leen estos archivos, construyen los respectivos objetos y los asocian a las estructuras de datos a los que pertenecen. Estas estructuras están determinadas por tamaños constantes que se definieron arbitrariamente, en el caso que los casos que se nos brinden excedan la capacidad, simplemente hay que cambiar el valor de estas constantes.

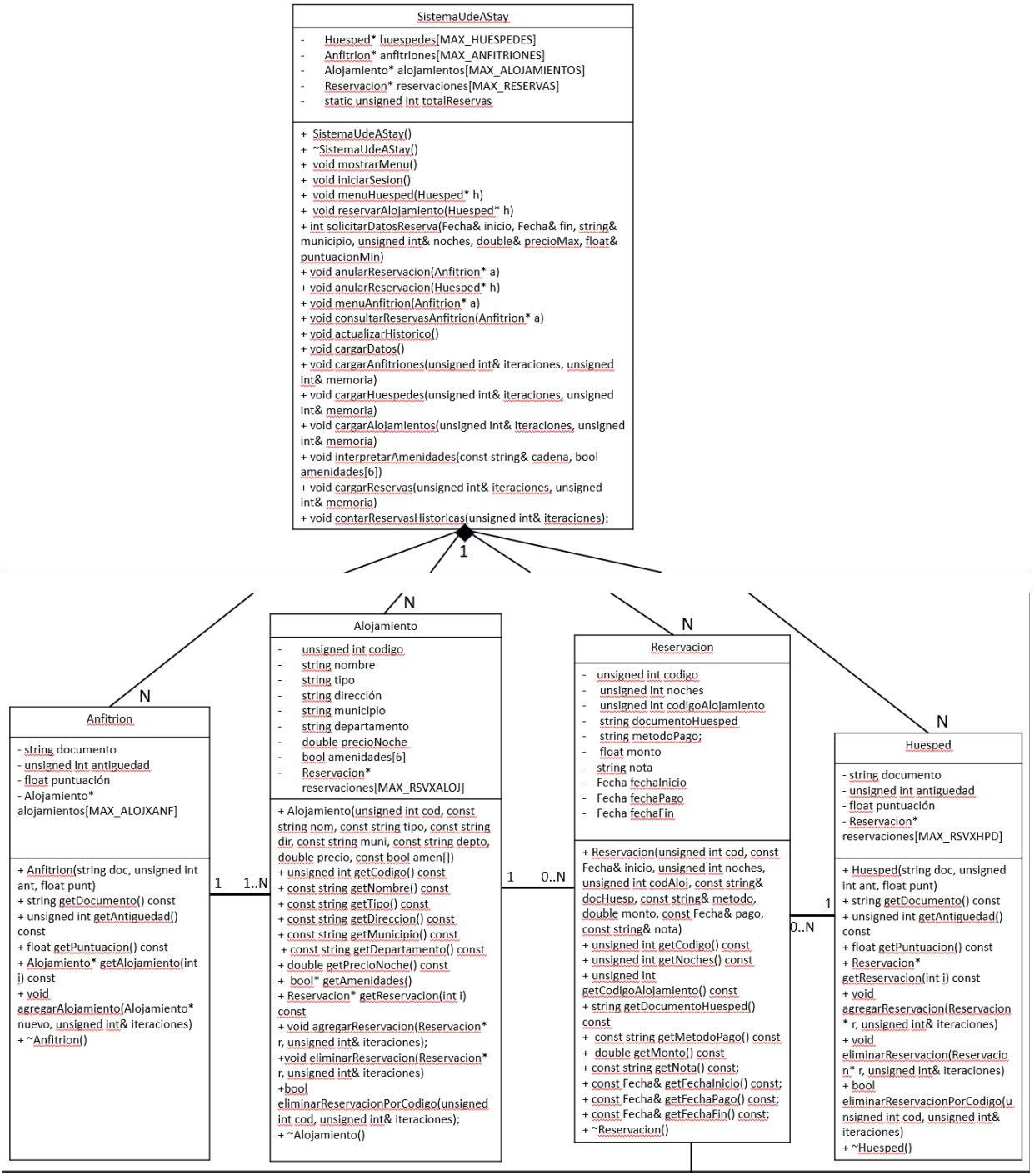
Durante la implementación decidimos agregar una clase llamada SistemaUdeaStay, esta va a ser la encargada de manejar el flujo del programa, teniendo las funcionalidades esenciales dentro de sus métodos. Los atributos de la clase son arreglos dinámicos donde se van a construir todos los objetos, ya sean de la carga del almacenamiento permanente o nuevas reservaciones. Las clases que se evidenciaron en el análisis inicial persistieron y adaptaron como estructura de datos arreglos de punteros a sus clases relacionadas, (anfitrión tiene punteros a sus alojamientos asociados, alojamiento tiene punteros a sus reservas asociadas y huésped tiene punteros a sus reservas asociadas) todo esto para evitar a toda costa la copia innecesaria de objetos que requerirían una alta cantidad de recursos en memoria.

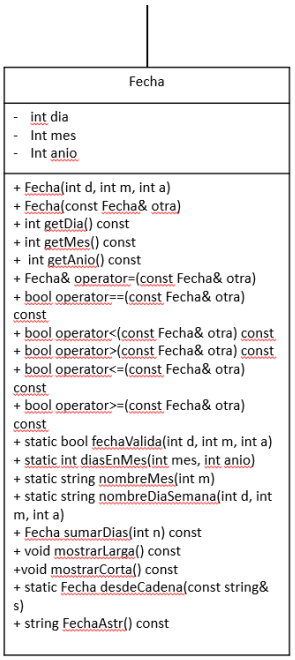
Una falencia de nuestro desarrollo fue subestimar la cantidad de tiempo requerida para cumplir al 100% con los requisitos de eficiencia, consideramos que en materia de eficiencia de memoria la solución es buena, sin embargo en materia de iteraciones es bastante mala, esto debido a que las búsquedas que realiza son lineales y por tanto son bastante ineficientes, teníamos muy presente que para mejorar esto lo ideal seria primeramente organizar los datos según x criterios y en base a ello implementar algoritmos de búsqueda que optimizarían mucho los recursos, nuestro error fue empezar el desarrollo muy tarde y estamos seguros que si hubiéramos empezado antes el resultado final sería mejor.

La clase fecha resultó muy útil para validaciones de fechas validas y unos atributos en la clase reservación, además que para las reservas es esencial en la verificación del traslape de fechas, para esta clase se tuvieron que sobrecargar los operadores de comparación booleanos (==, < , >, >=, <=), se sobrecargó el operador de asignación y se creó un constructor de copia.

Por último, somos consciente de varias falencias del código y ciertas funcionalidades que se encuentran incompletas, por ejemplo, a la hora de hacer la reserva no se implementó la opción de reservar por código y no se encuentra implementada la verificación de que las nuevas reservas solo puedan estar en el rango de un año posterior a la fecha de corte, todo esto lo atribuimos a la mala planeación del tiempo y para el desafío final lo tenemos que mejorar.

DIAGRAMA DE CLASES.




Fecha
- int dia
- Int mes
- Int anio
+ Fecha(int d, int m, int a)
+ Fecha(const Fecha& otra)
+ int getDia() const
+ int getMes() const
+ int getAnio() const
+ Fecha& operator=(const Fecha& otra)
+ bool operator==(const Fecha& otra) const
+ bool operator<(const Fecha& otra) const
+ bool operator>(const Fecha& otra) const
+ bool operator<=(const Fecha& otra) const
+ bool operator>=(const Fecha& otra) const
+ static bool fechaValida(int d, int m, int a)
+ static int diasEnMes(int mes, int anio)
+ static string nombreMes(int m)
+ static string nombreDiaSemana(int d, int m, int a)
+ Fecha sumarDias(int n) const
+ void mostrarLarga() const
+void mostrarCorta() const
+ static Fecha desdeCadena(const string& s)
+ string FechaAstr() const