Informe del desafío 2

# Hugo Esteban Barrero García – Juan Angel Omaña Montañez

# Universidad de Antioquia

# Informática 2

# Aníbal José Guerra Soler – Augusto Salazar

# Medellín, Colombia

Mayo, 2025

**1. Introducción**

El presente informe detalla el desarrollo y la implementación del sistema **UdeAStay**, una plataforma integral para la gestión de reservas de alojamiento, concebida como parte del Desafío II de la asignatura Informática II. Este sistema emula la funcionalidad de plataformas de alquiler de propiedades, facilitando la interacción entre anfitriones que ofrecen espacios y huéspedes que buscan realizar reservas. UdeAStay fue diseñado para modelar de manera precisa las entidades involucradas, sus atributos, métodos y las relaciones que definen su comportamiento dentro del ecosistema de reservas.

El proyecto se centró en la creación de una arquitectura robusta y modular, con énfasis en el uso exclusivo de **memoria dinámica** y la prohibición de librerías STL o el concepto de herencia, tal como lo estipulaban los requisitos del desafío. Además, se implementaron mecanismos para la gestión de archivos, el control de la disponibilidad de alojamientos y la medición del rendimiento interno del sistema.

**2. Objetivo del Diseño**

El objetivo principal de UdeAStay fue construir un modelo de clases que permitiera:

* **Gestionar alojamientos** ofrecidos por anfitriones, incluyendo información detallada y amenidades.
* **Facilitar la realización de reservas** por parte de los huéspedes, con validaciones de fechas y disponibilidad.
* **Registrar y mantener un histórico** de todas las reservas realizadas y finalizadas.
* **Validar la disponibilidad y fechas**, implementando un control de solapamiento de reservas para evitar conflictos.
* **Controlar estructuras dinámicas de datos** sin depender de las librerías STL ni del concepto de herencia.
* **Medir métricas internas de eficiencia** del sistema, como el total de iteraciones, memoria utilizada, archivos abiertos y líneas leídas.
* Asegurar una **buena modularidad**, **manejo eficiente de la memoria** mediante asignaciones dinámicas y **encapsulamiento** de datos.

Se buscó optimizar la cohesión de cada clase y minimizar el acoplamiento entre ellas, adhiriéndose a las mejores prácticas de la programación estructurada y orientada a objetos.

**3. Clases Implementadas y Justificación**

Se definieron e implementaron seis clases principales, cada una con un rol específico y justificado en la arquitectura del sistema:

**3.1 Clase Fecha**

* **Descripción:** Diseñada para representar fechas (día, mes, año), fundamental para campos como fechas de entrada, pago, salida y la fecha de corte del sistema.
* **Funcionalidades Clave:** Incluye validación de fechas en el constructor y setters para prevenir valores inválidos. Implementa operadores de comparación (==, <, <=, >=), el operador de suma (+) para calcular fechas futuras, y el operador de asignación (=) para copias correctas. El método toString() genera una representación textual dinámica.
* **Justificación:** Centraliza el manejo de la lógica de fechas, evitando duplicidad de código y errores comunes en la manipulación temporal.

**3.2 Clase Alojamiento**

* **Descripción:** Modela los espacios ofertados por los anfitriones. Contiene información textual (ubicación, tipo, dirección), numérica (precio) y un sistema de amenidades codificadas mediante **bitmask** (se usan los bits de un entero para representar múltiples opciones, ej: si amenidades = 5 (0101 en binario), significa que tiene activadas las amenidades 0 y 2), lo que permite una gestión eficiente de múltiples características. Cada alojamiento mantiene su propio historial de fechas reservadas.
* **Funcionalidades Clave:** Permite verificar la disponibilidad, registrar nuevas reservas, y mostrar las reservas activas dentro de un intervalo. Incluye métodos privados como copiarTexto y redimensionarReservas para mantener el encapsulamiento, y se mejoró la seguridad al reemplazar funciones como strcpy por strncpy.
* **Justificación:** Encapsula toda la información y lógica relacionada con una propiedad, desde sus características hasta la gestión de su disponibilidad. El uso de bitmask optimiza el almacenamiento y la manipulación de amenidades.

**3.3 Clase Anfitrion**

* **Descripción:** Representa al usuario que ofrece y administra alojamientos. Maneja un **arreglo dinámico de punteros a alojamientos**.
* **Funcionalidades Clave:** Permite agregar nuevos alojamientos, consultar reservas dentro de un rango de fechas específico, y anularlas de forma simulada. La memoria dinámica se gestiona con redimensionamiento progresivo para mantener la eficiencia, y el uso de punteros favorece la referencia cruzada sin duplicación de datos.
* **Justificación:** Separa las responsabilidades del anfitrión del resto del sistema, permitiendo una gestión clara de sus propiedades.

**3.4 Clase Huesped**

* **Descripción:** Modela a los usuarios que realizan reservas. Almacena el nombre completo, documento, antigüedad, puntuación y una **lista dinámica de reservas**. El código del huésped se genera automáticamente (ej. HSP-0001).
* **Funcionalidades Clave:** Incluye métodos para agregar reservas (validando que no se solapen), anular reservas (notificando al alojamiento), imprimir comprobantes de todas las reservas y mostrar un resumen del estado del huésped.
* **Justificación:** Centraliza la lógica asociada al huésped, incluyendo la crucial detección de solapamientos en las fechas de reserva.

**3.5 Clase Reservacion**

* **Descripción:** Encapsula toda la información de una reserva específica, incluyendo un código único generado automáticamente (ej. RSV-0001), fechas de entrada y pago, duración, alojamiento reservado, huésped responsable, medio de pago, monto pagado y anotaciones opcionales.
* **Funcionalidades Clave:** Implementa dos constructores (uno automático y otro manual para carga desde archivos), copia profunda, operador de asignación y validaciones internas. Además, cuenta con métodos para obtener la fecha de salida, imprimir comprobantes con formato detallado y acceder de forma segura a sus atributos.
* **Justificación:** Permite centralizar la lógica de validación, visualización y cálculo de los datos inherentes a una reserva.

**3.6 Clase UdeAStay**

* **Descripción:** La clase principal y controladora del sistema. Administra las listas dinámicas de todas las demás entidades (anfitriones, huéspedes, alojamientos, reservas vigentes e históricas). También controla la lógica general del programa, incluyendo la gestión de archivos, sesiones, filtros y métricas de rendimiento.
* **Funcionalidades Clave:** Centraliza la carga y guardado de datos, el inicio y cierre de sesión, la creación y anulación de reservas, consultas por rango de fechas, la actualización del histórico de reservas y la medición de rendimiento.
* **Justificación:** Actúa como el orquestador principal del sistema, manteniendo el archivo main.cpp limpio y modular al centralizar la lógica de negocio y la interacción global.

**4. Relaciones entre Clases**

El diseño de UdeAStay se basa en relaciones funcionales y necesarias entre las clases, siguiendo principios de **alta cohesión y bajo acoplamiento**.

* **UdeAStay y todas las demás clases:** UdeAStay se relaciona con todas las clases, ya que **gestiona directamente listas dinámicas** de Anfitrion, Huesped, Alojamiento, y tanto Reservacion vigentes como históricas. Esta es una **relación de agregación múltiple**, centralizando el flujo de datos y operaciones. Adicionalmente, UdeAStay contiene un objeto de tipo Fecha para la fecha de corte, lo que representa una **composición**.
* **Huesped y Reservacion:** Cada Huesped tiene una **relación de agregación** con múltiples objetos Reservacion, gestionándolos a través de un arreglo dinámico de punteros. Un huésped puede tener varias reservas activas.
* **Reservacion y Alojamiento:** Cada Reservacion está asociada a un único Alojamiento, manteniendo un **puntero** a este, lo que representa una **asociación simple**. Esto permite saber en qué inmueble se realizó la reserva.
* **Reservacion y Huesped:** Similarmente, cada Reservacion está vinculada a un único Huesped, quien la creó. La clase Reservacion almacena un **puntero** a Huesped, siendo una **asociación directa**.
* **Alojamiento y Anfitrion:** Cada Alojamiento pertenece a un Anfitrion, por lo que la clase Alojamiento tiene un **puntero** a su Anfitrion responsable. Esta es una **relación de agregación**, ya que un anfitrión puede tener varios alojamientos.
* **Anfitrion y Alojamiento:** Un objeto Anfitrion contiene un **arreglo dinámico de punteros** a Alojamiento. Esta **relación de uno a muchos** permite al anfitrión consultar, gestionar y anular reservas en sus propiedades.
* **Reservacion y Fecha:** La clase Reservacion utiliza **composición** con la clase Fecha, conteniendo objetos Fecha para representar la fecha de entrada y la fecha de pago. La relación es fuerte, ya que estas fechas nacen y mueren con la reserva.

**5. Implementación de Clases y Decisiones Técnicas**

A continuación, se detallan las decisiones clave tomadas durante la implementación:

* **Manejo de Memoria Dinámica:** Una de las restricciones fundamentales del proyecto. Todas las estructuras de datos que requieren crecimiento flexible se implementaron con **arreglos dinámicos** que se redimensionan progresivamente. Se utilizó una plantilla genérica redimensionarArreglo<T> para la gestión eficiente del tamaño de los arreglos, evitando así la necesidad de librerías STL.
* **Punteros entre Clases:** Para modelar las relaciones entre objetos (ej. Reservacion a Alojamiento y Huesped), se emplearon **punteros**. Esta estrategia evita copias innecesarias de objetos y permite referencias cruzadas eficientes entre las entidades.
* **Funciones Auxiliares Centralizadas:** Para evitar la duplicación de código en funciones comunes como copiarTexto, sonIguales, iniciaCon, extraerFecha, longitudTexto, entre otras, se creó un archivo central Funciones.h. Todas las clases se adaptaron para utilizar esta fuente unificada, lo que mejora la consistencia del código, facilita el mantenimiento y la depuración.
* **Uso de const y static:**
  + La palabra clave const se empleó para asegurar que ciertos métodos no modificaran el estado interno de los objetos, lo que aumenta la seguridad del código y mejora la legibilidad.
  + La palabra clave static se utilizó para mantener **contadores globales por clase** (ej. totalReservasCreadas, totalHuespedesCreados) y para gestionar la **generación automática de códigos** únicos (ej. RSV-0001, HSP-0001), proporcionando un mecanismo eficiente para el seguimiento de recursos y la identificación de entidades.
* **Formato de Archivos:** La persistencia de los datos se implementó mediante **archivos de texto (.txt) con formato delimitado por punto y coma (;)**. La carga de datos se realiza campo por campo utilizando funciones auxiliares propias, sin dependencias externas. Se maneja un archivo especial para la fecha de corte del sistema.
* **Manejo de Errores en Entrada de Usuario:** Para garantizar la robustez del sistema, se implementaron validaciones en las entradas de usuario, especialmente para campos numéricos o de fecha. Se utilizó cin.fail(), cin.clear() y cin.ignore() para evitar bloqueos del programa ante entradas inválidas, ofreciendo mensajes explicativos que mejoran la experiencia del usuario.

**6. Funcionalidades Implementadas**

El sistema UdeAStay ofrece las siguientes funcionalidades principales:

* **Carga y Guardado Automático de Datos:** Al iniciar el sistema, se cargan automáticamente todos los datos desde los archivos .txt. Al cerrar sesión o salir del programa, todos los datos se guardan para asegurar la persistencia.
* **Gestión de Sesiones:** Permite el inicio y cierre de sesión tanto para huéspedes como para anfitriones, controlando el acceso a funcionalidades específicas.
* **Reserva de Alojamientos:** Incluye un proceso de reserva con **validación de cruce de fechas** y la capacidad de aplicar filtros avanzados para la búsqueda de alojamientos.
* **Anulación de Reservas:** Permite a huéspedes y anfitriones anular reservas, con la correcta liberación de memoria dinámica y la notificación a las entidades involucradas.
* **Consulta de Reservas:** Ofrece la funcionalidad de consultar reservas vigentes por un rango de fechas determinado.
* **Impresión de Comprobantes:** Genera comprobantes detallados para cada reserva, con un formato claro y completo.
* **Visualización de Historial:** Permite al huésped activo visualizar todas sus reservas, tanto vigentes como pasadas.
* **Actualización del Histórico:** Las reservas cuya fecha de entrada es anterior a la fecha de corte del sistema se trasladan automáticamente al archivo de histórico.
* **Medición de Rendimiento:** El sistema incorpora métricas internas para monitorear su eficiencia, incluyendo el total de iteraciones, la memoria dinámica utilizada en bytes, el número de archivos abiertos y las líneas leídas.

**7. Pruebas Realizadas y Resultados**

Se llevaron a cabo pruebas exhaustivas para cada clase y para la integración del sistema, verificando su comportamiento esperado y la robustez frente a diversos escenarios:

* **Pruebas Unitarias por Clase:**
  + **Fecha:** Se probaron comparaciones, copias, conversiones a texto y operadores personalizados, confirmando su correcto funcionamiento.
  + **Alojamiento:** Se verificó la gestión de amenidades (bitmask), el registro de reservas, la disponibilidad por fechas y la impresión de datos.
  + **Anfitrion:** Se comprobó la correcta asociación con alojamientos y la capacidad de consultar y anular reservas.
  + **Reservacion:** Se validó la creación de reservas mediante ambos constructores, copia, asignación, generación de códigos, validación de fechas y la impresión del comprobante completo.
  + **Huesped:** Se comprobó la gestión dinámica de reservas, la prevención de solapamientos, la anulación correcta con referencia al alojamiento, la generación automática de código y el resumen de estado del huésped.
* **Pruebas de Integración:**
  + La integración entre Huesped y Reservacion fue validada, incluyendo la creación automática de códigos, almacenamiento y recuperación de reservas, **detección de conflictos de fechas** al agregar nuevas reservas, y la correcta anulación, con la consecuente liberación de memoria dinámica.
  + Se validaron los **contadores de recursos internos** (totalHuespedesCreados, totalReservasCreadas, totalIteraciones), lo que permite un análisis de eficiencia sin herramientas externas. Las impresiones de comprobantes y los resúmenes de información se mostraron correctamente, sin errores en tiempo de ejecución ni fugas de memoria observadas.
* **Validación de Robustez:**
  + **Detección de Solapamientos:** El sistema impidió correctamente reservar cuando ya existía una reserva vigente en el rango de fechas, tanto en el alojamiento como en el historial del huésped.
  + **Actualización del Histórico:** Las reservas con fecha anterior a la nueva fecha de corte se trasladaron correctamente al archivo de histórico.
  + **Manejo de Errores de Entrada:** Se comprobó que el sistema no se bloqueaba ante entradas numéricas inválidas, gracias a las validaciones con cin.fail(), cin.clear() y cin.ignore(), y que mostraba mensajes explicativos.
  + **Pruebas con Archivos:** Se cargaron y guardaron archivos .txt con estructuras complejas sin errores, demostrando la fiabilidad del manejo de datos.
  + **Métricas de Rendimiento:** Las métricas arrojaron resultados consistentes, como el ejemplo de una prueba exitosa:
  + --- Metricas UdeAStay ---
  + Total de iteraciones: 19193
  + Memoria dinamica (bytes): 8606
  + Archivos abiertos: 24
  + Lineas leidas: 27
* **Ejemplo de Reserva Exitosa:** Un caso de prueba típico de reserva exitosa generó el siguiente comprobante:
* Codigo: RSV-0010
* Fecha de entrada: lunes, 2 de junio de 2025
* Fecha de salida: jueves, 5 de junio de 2025
* Metodo de pago: PSE
* Fecha de pago: 1/6/2025
* Monto pagado: 255000
* Anotaciones: Hola

**8. Retos Enfrentados y Soluciones**

Durante el desarrollo de UdeAStay, se presentaron diversos retos técnicos, principalmente asociados al manejo de memoria dinámica, las validaciones de entrada y la consistencia entre clases.

* **Duplicación de Funciones Auxiliares:** Inicialmente, funciones como copiarTexto, sonIguales, iniciaCon y longitudTexto estaban repetidas en varias clases (Anfitrion, Huesped, Alojamiento, Reservacion, UdeAStay), lo que generaba código extenso, difícil de mantener y propenso a errores.
  + **Solución:** Se creó un archivo central Funciones.h donde se unificaron estas funciones. Todas las clases se adaptaron para utilizar esta única fuente, asegurando consistencia y facilitando la depuración.
* **Errores en el Manejo de Memoria Dinámica:** Se identificaron errores en constructores de copia y operadores de asignación, donde se realizaban copias superficiales o se omitía la liberación de memoria anterior, lo que podía provocar fugas de memoria y comportamientos impredecibles.
  + **Solución:** Se corrigieron todos estos puntos implementando **copias profundas** correctamente y asegurando que los destructores eliminaran todas las referencias dinámicas asignadas.
* **Validación de Interacción con el Usuario:** Al principio, el programa se bloqueaba si el usuario ingresaba texto en lugar de números, especialmente en campos de fechas o cantidades, debido a la falta de validación.
  + **Solución:** Se mejoró main.cpp incorporando validaciones robustas mediante cin.fail(), cin.clear() y cin.ignore(), garantizando que el sistema nunca se congele ni permita entradas corruptas. Además, se añadieron mensajes explicativos para mejorar la experiencia del usuario.
* **Lógica de Negocio y Refactorización:**
  + Se implementó un sistema de **detección de conflictos de fechas en las reservas**: si un huésped intentaba reservar un alojamiento en una fecha ya ocupada, el sistema detectaba el solapamiento y lo impedía.
  + Se refactorizaron métodos como anularReservacion para que funcionaran por fecha y duración en lugar de por código, haciendo el sistema más flexible.
* **Independencia de Librerías Estándar:** Esta versión se diseñó evitando completamente dependencias de librerías como <cstring> o <iostream>, reemplazándolas por funciones auxiliares propias para copiar, comparar y gestionar texto. Esto refuerza la independencia del sistema, permitiendo mayor control sobre la memoria, ciclos y recursos.

**9. Conclusiones**

El sistema UdeAStay ha logrado implementar de forma **completa y funcional** todos los requisitos del Desafío II, adhiriéndose estrictamente a las restricciones técnicas de la asignatura (sin STL, sin herencia y uso exclusivo de memoria dinámica).

El diseño basado en **clases bien encapsuladas**, el uso estratégico de **punteros**, la centralización de **funciones auxiliares** y la implementación de **contadores de rendimiento internos** han dado como resultado un sistema **estable, eficiente y de código claro**. Todas las funcionalidades solicitadas están accesibles a través de un menú amigable, y los resultados de las pruebas confirman un comportamiento correcto incluso ante entradas erróneas o casos límite.

El código desarrollado demuestra ser **fácil de mantener y escalar**, lo que lo hace apto para futuras ampliaciones o una posible migración a arquitecturas más complejas sin requerir cambios significativos. Se ha cumplido cabalmente con la validación de requisitos, el diseño orientado a objetos, el manejo eficiente de recursos y la entrega de un producto funcional y de calidad.

### **10. Bibliografía y Referencias**

* Documentación oficial de Qt. Disponible en: <https://doc.qt.io>
* Apuntes de clase de la asignatura Informática II.
* Archivo entregado “Desafio Info II 2025-1 v1.01.pdf” con instrucciones del reto.
* Explicaciones técnicas desarrolladas durante el trabajo práctico.
* Discusiones y ayudas recibidas a través del canal del curso.
* Asistencia generada con ChatGPT (OpenAI) y Gemini.
* Repositorio del proyecto: [https://github.com/Hugo-24/Desafio-2](https://github.com/Hugo-24/Desafio-21).