**Actividad Práctica Momento 3**

**Análisis y Aplicación de las Propiedades de POO**

**Julian David Laserna Moreno**

**Edgar Diaz Cabiativa**

**Tutor**

**Universidad Cooperativa de Colombia**

**Programación Orientada a Objetos**

**Agosto de 2025**

Contenido

[**Paso 1. Selección del problema, objetivos y justificación** 3](#_Toc206000967)

[**Problema real:** 3](#_Toc206000968)

[**Objetivos del proyecto** 3](#_Toc206000969)

[**Justificación** 3](#_Toc206000970)

[**Diagramas: Diagrama UML con Clases, atributos y métodos**. 4](#_Toc206000971)

[**Diagrama con patrones arquitectónicos incluidos** 4](#_Toc206000972)

[**Diagrama, con arquitectura modular** 5](#_Toc206000973)

[**Principios POO en el nuevo diagrama UML** 6](#_Toc206000974)

[**Paso 2. Resumen del Desarrollo del Proyecto: Reservas en un Hotel** 8](#_Toc206000975)

[**1. Punto de partida: Diagrama de clases** 8](#_Toc206000976)

[**2. Mejoras con principios POO** 8](#_Toc206000977)

[**3. Patrones de diseño integrados** 8](#_Toc206000978)

[**4. Diseño modular** 8](#_Toc206000979)

[**5. Implementación en Python + SQLite** 8](#_Toc206000980)

[**Solución implementada en el desarrollo de esta actividad:** 9](#_Toc206000981)

[**Análisis de eficiencia** 9](#_Toc206000982)

[**6. Mejoras propuestas** 9](#_Toc206000983)

[**Referencias:** 10](#_Toc206000984)

# **Paso 1. Selección del problema, objetivos y justificación**

## **Problema real:**

En un hotel con capacidad de crecimiento, la administración enfrenta dificultades para manejar múltiples procesos manuales, como: Registro de clientes y empleados, asignación de habitaciones, control de reservas, gestión de notificaciones internas, integración de datos desde diferentes fuentes (archivos, base de datos). Esto genera errores, pérdida de información y baja eficiencia operativa.

## **Objetivos del proyecto**

**Objetivo general:**

Desarrollar un sistema de software modular para gestionar un hotel, aplicando principios de la Programación Orientada a Objetos (POO) y patrones de diseño que favorezcan la escalabilidad, mantenimiento y reutilización del código.

**Objetivos específicos:**

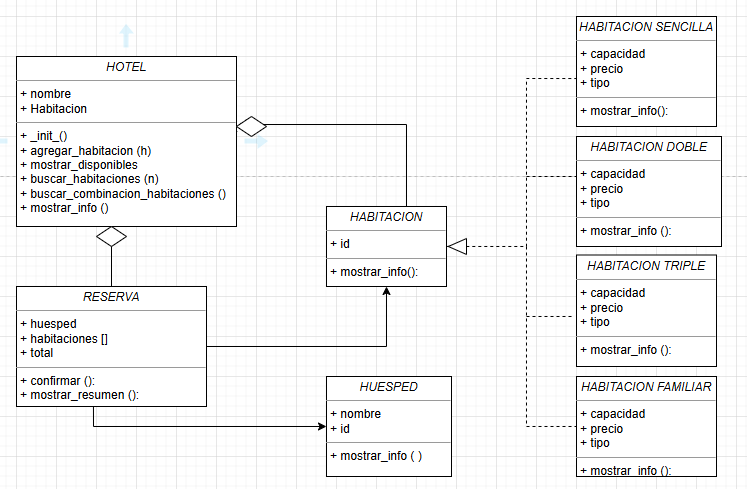
* Implementar un sistema que permita registrar, consultar y modificar clientes, empleados, habitaciones y reservas.
* Aplicar principios de POO como encapsulamiento, herencia, abstracción y polimorfismo.
* Utilizar patrones de diseño para mejorar la arquitectura del software: Singleton para asegurar una única instancia de la clase principal que coordina el sistema. Observer para notificar automáticamente cambios en reservas o disponibilidad de habitaciones. Factory Method para la creación flexible de objetos como empleados y habitaciones, según el tipo. Adapter para integrar distintas fuentes de datos (como archivos CSV o bases de datos SQLite) sin alterar la lógica central del sistema.

## **Justificación**

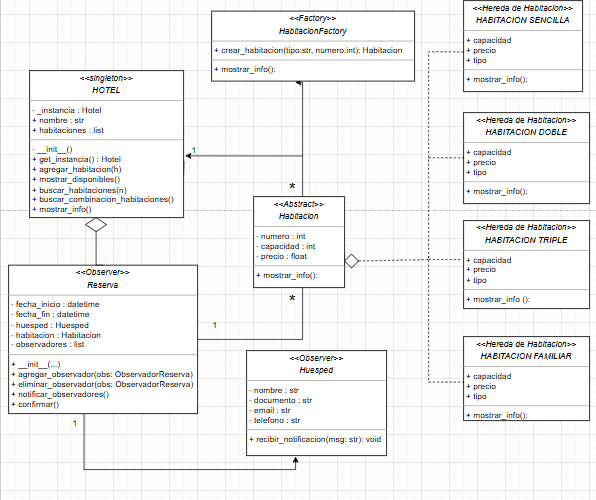
Este proyecto se enfoca en un caso de uso cotidiano y escalable: la gestión de un hotel. Se eligió porque: Permite aplicar múltiples conceptos de POO y patrones de diseño de forma concreta. Es ejecutable con una base de datos pequeña (incluso archivos CSV o SQLite). Puede ampliarse fácilmente (por ejemplo, para agregar facturación, pagos en línea o conexión con APIs de reservas).

La arquitectura modular y el uso de patrones garantizan: Bajo acoplamiento, facilitando mantenimiento y expansión, alta cohesión, con clases especializadas para cada funcionalidad, reutilización del código en diferentes escenarios, como hoteles de otras categorías o sistemas afines (hostales, residencias, etc.).

## **Diagramas: Diagrama UML con Clases, atributos y métodos**.



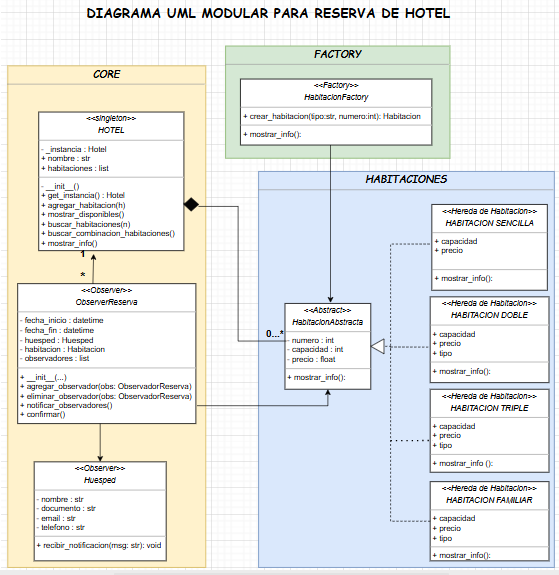
## **Diagrama con patrones arquitectónicos incluidos**



La **arquitectura modular** implica dividir el sistema en **módulos bien definidos y separados**, donde cada módulo se encarga de una responsabilidad específica. Los módulos están implícitos por grupos de clases relacionadas.

1. <<Singleton>> en **Hotel**; con el propósito de que se asegure que haya una sede central (esta será Hotel) El hotel es el **centro del sistema**, así que todas las operaciones (crear reserva, asignar habitación, etc.) deben gestionarse desde un **único punto de control.**
2. <<Observer>>, en Reserva y en Huésped; permitirá que cuando algo cambie en una reserva (por ejemplo, se confirma la reserva), los huéspedes recibirán una notificación automática. Puede ser útil para un Hotel enviar alertas o confirmaciones de cambios automáticamente al huésped.
3. <<Factory Method>> en Habitación Factory (Que se creó), centralizará y abstrae los objetos (las habitaciones: sencilla, doble, triple, familiar)

## **Diagrama, con arquitectura modular**



En este nuevo diagrama las clases se agruparon en paquetes según su función dando la forma de arquitectura modular bien definida

En el contexto de **arquitectura modular de software**, el término significa literalmente **"núcleo".** Se refiere al **módulo central del sistema,** donde está la **lógica principal** que hace funcionar todo.

En UML, cuando una clase contiene a otra como parte de su estructura interna y es responsable de su ciclo de vida, se utiliza una composición, y eso se representa con: Hotel – Habitación Abstracta, **Un círculo negro (rombo sólido)** en el extremo de la clase "contenedora". (el Hotel contiene (0…\*) cero o muchas habitaciones; Si el Hotel se destruye, también sus habitaciones desaparecen (composición fuerte).

**en la clase Habitación**, Habitación ahora es una **clase abstracta** porque: **No se crean objetos de tipo** Habitación **directamente.** Sirve como una **plantilla base** **(padre)** con atributos y métodos comunes (por ejemplo, numero, precio, etc.) que las clases concretas .) heredan de ella y **sobrescriben los métodos si es necesario.** Por seguridad se evita que alguien cree una habitación que no sea válida dentro de las existentes. Se evidencia el **Polimorfismo**, porque tratan: diferentes tipos (.) como si fueran , y usan métodos comunes como sin importar el tipo real.

Las habitaciones (sencilla, doble, triple y familiar) se unen a la habitación abstracta con una flecha que en la punta tiene **un triángulo blanco vacío, esto significa herencia o Generalización**

**Hotel- Habitación Factory**, hay una relación de dependencia o asociación

Porque usar fabrica ()?; porque permite agregar más tipos de habitaciones sin modificar el código del hotel. Hace el sistema más modular y flexible. **Desacopla** el proceso de creación de objetos. (Significa que **una clase no necesita saber los detalles exactos de cómo se construyen (o qué tipo específico son) los objetos que usa.)** Así el hotel no necesita conocer ni cambiar si luego agregas

**Reserva a Hotel** es una relación de asociación simple

## **Principios POO en el nuevo diagrama UML**

|  |  |
| --- | --- |
| **Principio POO** | **Ejemplo en el diagrama** |
| **Encapsulamiento** | Atributos privados como: nombre, habitaciones, capacidad, etc. |
| **Herencia**  permite que una clase hija obtenga atributos y métodos de una clase padre. | * Habitación (simple, doble…) heredan de Habitación Abstracta * Habitación Abstracta, sirve como clase base con atributos comunes: numero, capacidad, precio y el método [*mostrar info ()*]   Esto promueve la reutilización del código y evita duplicación. |
| **Polimorfismo**  permite que se puedan usar clases derivadas como si fueran del tipo base. | * [*crear Habitación ()*] retorna diferentes subtipos tratándolos como Habitación * La clase Habitación Factory, tiene un método , que devuelve un objeto del tipo Habitación, pero en realidad puede ser cualquiera de las subclases (Sencilla, Doble, etc.). * Cuando Hotel recibe una habitación, no necesita saber su tipo exacto, solo que puede tratarse como [Habitación Abstracta]   Esto permite que el código sea más flexible y extensible. |
| **Abstracción**  consiste en **modelar las entidades del sistema** con clases que representan su comportamiento esencial, omitiendo detalles irrelevantes. | * La clase [Habitación Abstracta] es una clase abstracta (marcada con <<abstract>>), lo que indica que no se puede instanciar directamente, sino que solo sirve como modelo base. * El método [*mostrar\_info ())* está definido en la clase abstracta, pero puede ser implementado o sobrescrito en las clases hijas.   Esto permite centrarse en las funcionalidades esenciales y construir modelos generales para casos específicos. |

Se Implementa modularidad basada en el nuevo diseño UML:

* Usa Singleton para la clase Hotel.
* Aplica el patrón Factory para crear habitaciones.
* Integra el patrón Observer con notificaciones a huéspedes.
* Guarda las reservas en una base de datos SQLite (hotel.db).
* Mantiene la interfaz interactiva por consola como en tu versión anterior.

La integración a base de datos (SQLite) significa que tu sistema de gestión del hotel ahora almacena, consulta y actualiza información (como reservas y habitaciones) en una base de datos real, en lugar de manejarlo todo solo en memoria.

# **Paso 2. Resumen del Desarrollo del Proyecto: Reservas en un Hotel**

## **1. Punto de partida: Diagrama de clases**

Se inicio con un diagrama UML clásico que representaba las entidades del sistema de gestión del hotel: Hotel, Habitación, Reserva, Cliente, etc.

Se identificaron relaciones como:

* Composición entre Hotel y Habitación.
* Asociación entre Reserva y Cliente

## **2. Mejoras con principios POO**

Se aplicaron los cuatro principios fundamentales de POO:

Abstracción: Creando la clase abstracta Habitación con atributos y métodos comunes, que fue heredada por clases como Habitación Sencilla, Habitación Doble, etc.

Encapsulamiento: Atributos privados y uso de métodos getters y setters.

Herencia: Las habitaciones heredan de la clase base Habitación.

Polimorfismo: Permiten tratar todas las habitaciones de forma uniforme (usando métodos como

## **3. Patrones de diseño integrados**

Se rediseñó el sistema incorporando patrones de diseño que aumentan la escalabilidad y modularidad:

|  |  |  |
| --- | --- | --- |
| **Patrón** | **Aplicación** | **Propósito** |
| Singleton | Para la conexión a la base de datos | Garantiza una sola instancia compartida |
| Factory Method | Para crear objetos tipo Habitación | Desacopla la creación del tipo concreto |
| Adapter (futuro) | Adaptar datos de entrada (por ejemplo, formularios) a objetos del sistema | Interoperabilidad y conversión |
| Observer | Para notificar cambios (por ejemplo, en disponibilidad) | Comunicación automática entre objetos |

## **4. Diseño modular**

Se reorganizó todo en arquitectura modular: Cada módulo cumple una única responsabilidad, aplicando el principio SRP (Single Responsibility Principle). “establece que una clase debe proporcionar solo una funcionalidad, no más de una”[[1]](#footnote-1)

## **5. Implementación en Python + SQLite**

Se codificó todo el sistema:

* Python con estructuras limpias, siguiendo los principios SOLID.
* Conexión a base de datos SQLite para persistencia.
* Métodos básicos: crear habitaciones, clientes, reservas, consultar disponibilidad.

## **Solución implementada en el desarrollo de esta actividad:**

Se desarrolló un sistema de gestión hotelera para un hotel pequeño, partiendo de un diseño orientado a objetos y evolucionando hacia una arquitectura modular con patrones de diseño.

La solución incluye:

* Modelo de clases flexible con principios POO.
* Uso de Factory Method para crear habitaciones según tipo.
* Conexión centralizada a base de datos con Singleton.
* Estructura modular organizada en capas (modelo, lógica, datos).
* Persistencia de datos con SQLite, apta para sistemas pequeños y portables.

## **Análisis de eficiencia**

* Escalabilidad: el sistema puede crecer sin romper la lógica interna gracias al uso de herencia y modularidad.
* Reutilización: la fábrica de habitaciones permite añadir más tipos sin modificar código existente.
* Bajo acoplamiento: cada módulo está desacoplado del resto; es fácil hacer pruebas y reemplazos.
* Facilidad de mantenimiento: la arquitectura clara reduce el riesgo de errores al modificar el código.

## **6. Mejoras propuestas**

- Notificaciones (Observer) en cambios de disponibilidad.

- Pruebas unitarias para asegurar la estabilidad del sistema.

- Almacenamiento en la nube o base de datos remota si el sistema escala.

- Historial de cambios y control de versiones en GitHub para rastrear mejoras.

El programa final en Python que corre incluye:

* Arquitectura modular: separación conceptual de responsabilidades.
* Patrón Singleton en Hotel: solo hay una instancia.
* Patrón Factory en centraliza la creación de habitaciones.
* Patrón Observer: Reserva notifica al .
* Sistema ejecutable desde un único archivo.
* Interfaz por consola simple.

# **Referencias:**

Moreno Pérez, J. C. (2015). *Programación orientada a objetos* (1.ª ed.). RA-MA Editorial. [**Enlace**](https://elibro-net.bbibliograficas.ucc.edu.co/es/lc/ucc/titulos/106461)

Programación.net (2025) *Principio de Responsabilidad Única* (SRP- SOLID explicado con ejemplos. [Principio de Responsabilidad Única (SRP) – SOLID explicado con ejemplos](https://programaciondotnetdb.blogspot.com/2025/03/SOLID-principio-de-responsabilidad-unica-srp.html)

1. [Principio de Responsabilidad Única (SRP) – SOLID explicado con ejemplos](https://programaciondotnetdb.blogspot.com/2025/03/SOLID-principio-de-responsabilidad-unica-srp.html) [↑](#footnote-ref-1)