Informe - Trabajo Práctico Integrador

OFERTA HOTELERA

Materia:
Programación con Objetos II

Integrantes:
Alvarenga Marcos
Garcia Smith Agustín
Gil, Maricruz

Fecha entrega: Viernes 19 Diciembre de 2015

Índice

Finalidad

El presente documento sintetiza los resultados del trabajo práctico integrador de la materia POII. Presentaremos explicaciones sobre las decisiones de diseño que se tomaron, herramienta, librerías y frameworks utilizados para llegar a los resultados obtenidos.

Alcance

Para el presente trabajo práctico se desarrollaron la mayoría de las solicitudes pedidas en el enunciado.

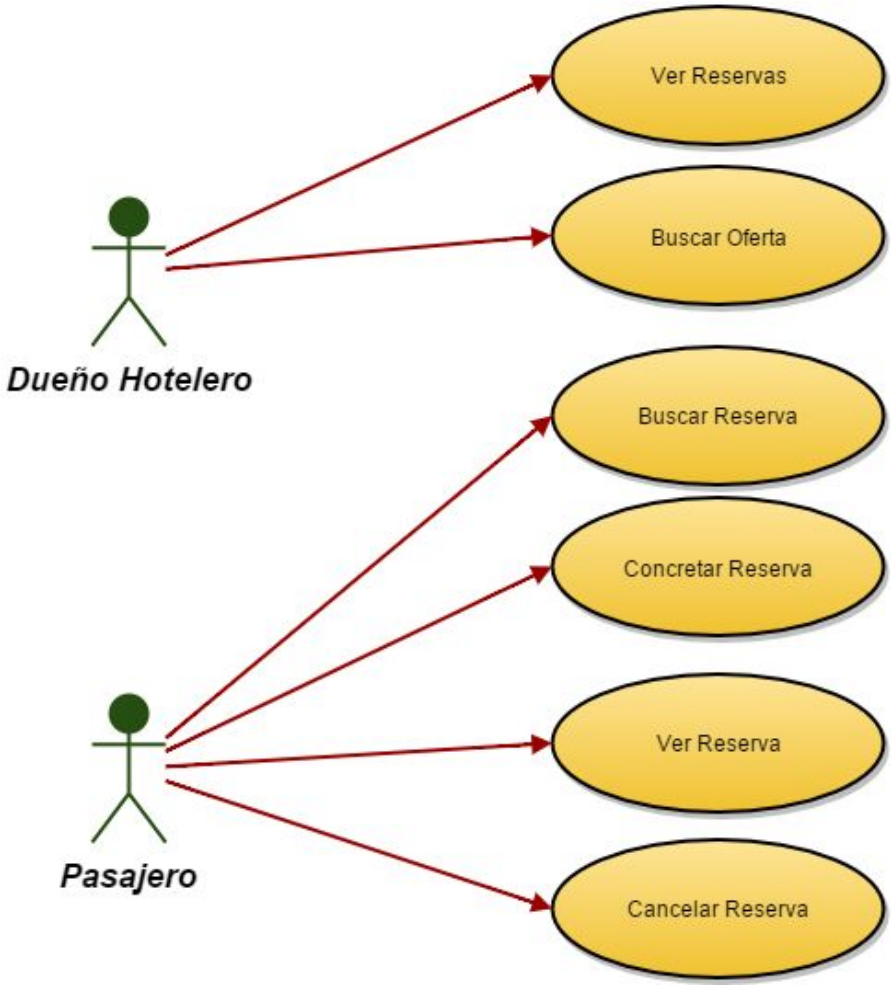
Por un lado, existen las clases ***Hotel*** y ***Habitacion*** las cuales cumplen en contener toda la información solicitada por el enunciado. Los hoteles contienen nombre, país, ciudad, dirección y demás atributos solicitados. Mientras que cada habitación cuenta con la capacidad máxima de huéspedes, servicios, precio y tipo de cama.

Además, el sistema entregado cumple con las funciones de búsqueda solicitada de acuerdo a un rango de fechas, el nombre de la ciudad / Hotel a buscar y/o la capacidad máxima de huéspedes. Esta funcionalidad es cubierta por las clases con nomenclatura ***Filtro*****.

Luego, el sistema es capaz de concretar y cancelar una reserva mediante el ***GestorConsulta***. Los motivos por los cuales se decidió implementar un gestor fue para cumplir con el Single Responsability Principle ya que nos parecía que el ***SistemaHotelero*** estaba cargando con demasiadas responsabilidades. De esta manera, el sistema recibe el pedido y lo deriva al gestor para que este se ocupe.

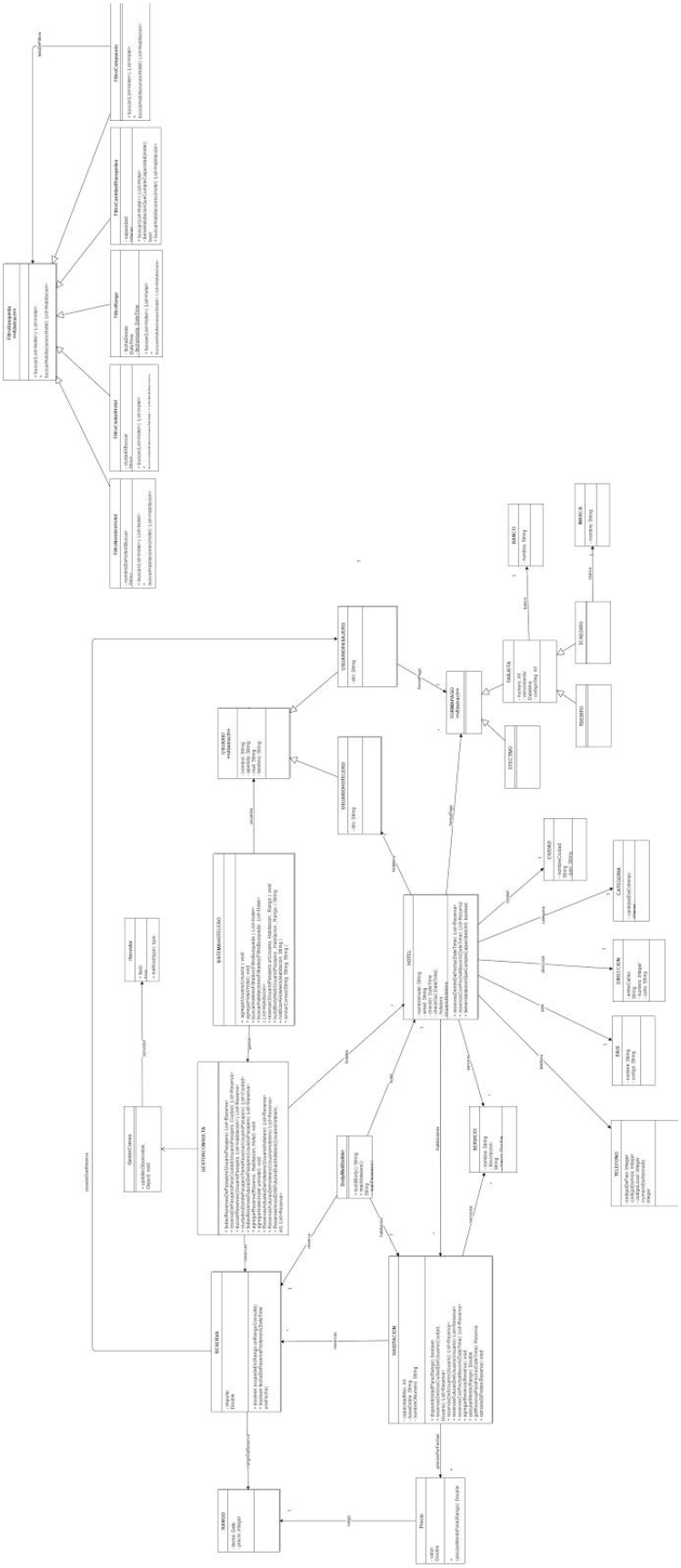
En cuanto a la administración de reservas, tanto las consultas de los pasajeros como de usuarios hoteleros, se encuentran cubiertas, las responsabilidades se encuentran distribuidas entre el hotel y la habitación.

Casos de Uso


Ver Reservas
Buscar Oferta
Dueño Hotelero
Buscar Reserva
Concretar Reserva
Ver Reserva
Pasajero
Cancelar Reserva

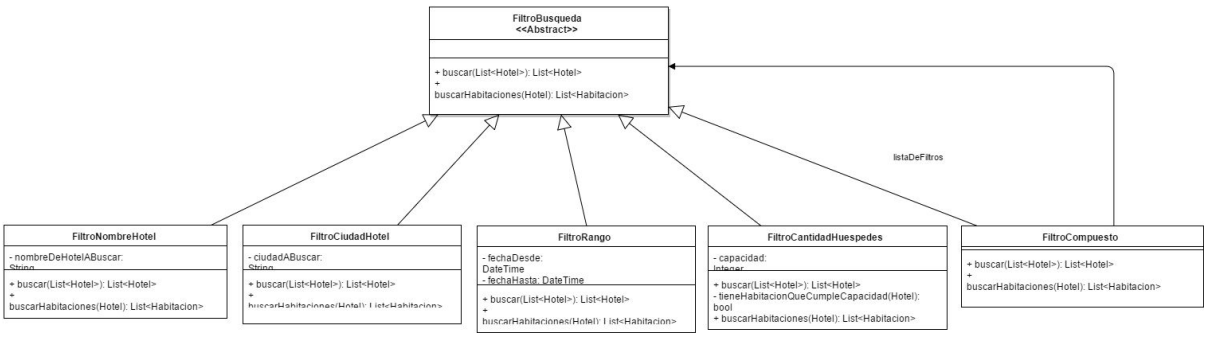
Modelo - Diagrama de Clases UML

Además se adjunta por separado a este documento para su mejor visualización.

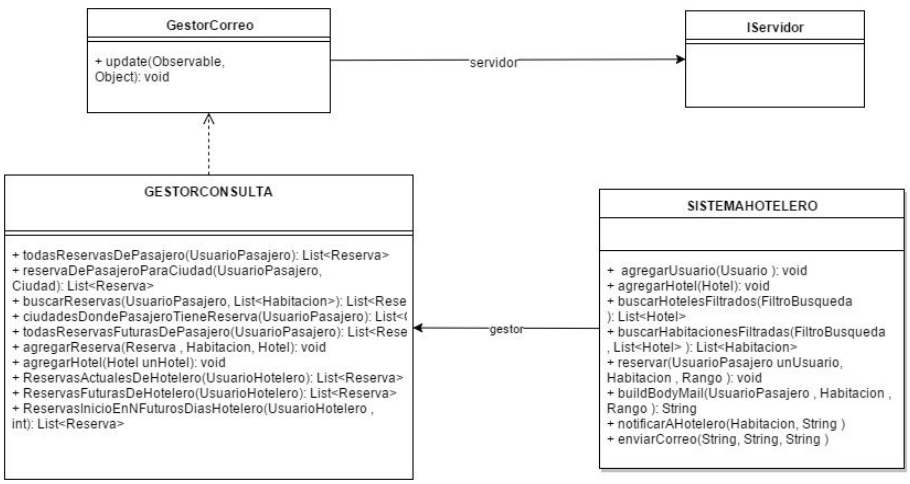


Patrones de diseño utilizados

Para la búsqueda de las reservas fue implementado un *COMPOSITE*. Al tener la pantalla de búsqueda varios campos como el nombre del hotel, cantidad de huéspedes, etc., los cuales deben tratarse de igual manera y además, se debía brindar la posibilidad de que la búsqueda sea por alguno o todos los campos solicitado teniendo así que pensar en una combinación de filtros, encontramos en el composite la mejor forma de representar una jerarquía parte-todo y además ganando la posibilidad de tratar tanto a los filtros individuales como los compuestos de la misma manera, obviando las diferencias.



Para el envío de mails de confirmación de reservas se utilizó el patrón *OBSERVER*. Si bien entendemos que no se llega a explotar del todo la utilización del patrón ya que solo existe un solo observador, nos pareció que este diseño refleja lo que necesitábamos implementar.
Existe en nuestro diseño un gestor de consulta (el cual se detalla en otra sección de este documento), el cual es observado por el gestor de correo. En el momento en que se realiza una reserva, el gestor de consulta envía una notificación para que el gestor de correo se encargue de enviar los mails correspondientes. De esta forma se quita la responsabilidad de la gestión de los mails al sistema buscando complir con el principio solid de Única Responsabilidad y además este solo acciona cuando se finaliza de manera satisfactoria una reserva.
Por otro lado, entendemos que este patrón se puede amoldar bien a futuros nuevo requerimientos ya que como cumple con la dependencia de uno-a-muchos entre objetos, si el dia de mañana se necesita agregar funcionalidad a la hora de realizar las reservas, esta implementación es fácilmente extensible.



Testing

En nuestro caso se testeo la mayor cantidad del trabajo práctico, teniendo un porcentaje de más del 90% del código probado. Decidimos dividir los test en dos paquetes, uno donde solo tiene las clases generadas sin comportamiento por completitud y otro donde se localizan las clases con mayor funcionalidad.

En el caso de la búsqueda se testearon los filtros de manera individual y luego la búsqueda compuesta, tratando de cumplir la mayor cantidad de casos posibles.

Para poder probar el envío de correo, utilizamos sysout por consola (system.out.print) para ver los resultados obtenidos.

Para dividir las tareas de test entre los miembros del equipo, se utilizó el framework MOCKITO, de esta manera se realizaron pruebas unitarias evitando solapamientos en la corrección de problemas del código y permitiendo el avance en paralelo del desarrollo del trabajo.

No fue utilizado para todos los test's, pero TDD fue parte importante en el desarrollo del sistema.

Otra técnica útil a la hora de testear, fue realizar pruebas sobre código que desarrolló otra persona. De esta manera se evita caer en el problema de los "test's felices" y de realizar casos de test solo de lo que se encuentra desarrollado hasta el momento o de los casos más básicos.

Dificultades

Creemos que lo más costoso fue llegar a un acuerdo sobre el diseño, el cual fue mutando en el transcurso del desarrollo. Aprendimos que en este tipo de trabajos grupales la herramienta más importante es la comunicación entre los miembros del equipo.

Conclusiones

Este TP fue de ayuda para consolidar los conocimientos adquiridos durante el período lectivo, de una manera más práctica y funcional. Los conocimientos sobre principios SOLID fueron de los más utilizados. Los patrones de diseño vistos fueron muy útiles para poder modelar el envío de mails y la búsqueda a nuestra necesidad. La implementación de TDD nos simplificó mucho la tarea a la hora de modelar lo justo y necesario.
Fueron adquirimos conocimientos sobre utilización de herramientas como Eclipse y Eclemma y conocimos de framework's como JodaTime y Mockito.