Linnéuniversitetet
Kalmar Växjö

2DV603

DESIGN DOCUMENT

Authors: - Alaa Alwan
- Mohammed Basel Nasrini
Team: 13
Supervisor : Mauro Caporuscio
Semester: Autumn 2018
Subject: Re-engineering Legacy Software
Course code: 2DV603

Content

1. Purpose

1.1 This design document describes

The purpose of this document is to present and describe the designs used to implement the hotel management system. The design follows the requirements that are specified in the Requirements Specification Document. The software designed to be used by Linnaeus Hotel to manage the front-desk activities. The hotel clerk will use it to enter reservations as well as to check guests in and out of both hotel's building in Växjö and Kalmar.

1.2 Requirements Traceability

This section provides a cross references show the association between the software requirements and the Requirements Specification Document.

Requirement	*Description*	*Design Reference*
R1	Choose location	UC1
R2.1	List all rooms	UC2.1
R2.2	View room's details	UC2.2
R2.3	Search for a room with specific details	UC2.3
R3.1	Create reservation	UC3.1
R3.2	Search a reservation and view its details	UC3.2
R3.3	Edit a pending reservation	UC3.3
R3.4	Cancel a pending reservation	UC3.4
R3.5	Delete a pending or checked out reservation	UC3.5
R3.6	Add service to a reservation	UC3.6
R4.1	Search for an existing customer	UC4.1
R4.2	Edit an existing customer	UC4.2
R4.3	Delete an existing customer	UC4.3
R5.1	Search for a bill	UC5.1
R5.2	View a bill as PDF	UC5.2
R5.3	Print a bill	UC5.3
R5.4	Mark bill as paid	UC5.4
R6	Check-in a pending reservation	UC6
R7	Check-out a checked-in reservation	UC7

2. General Priority

This section defines the software requirements prioritization on which requirements will be implemented first. A 3-level scale will prioritize the requirements where 1 is most critical and 3 least critical.

Requirement	*Description*	*Priority*
R1	Choose location	1
R2.1	List all rooms	2
R2.2	View room's details	2
R2.3	Search for a room with specific details	1
R3.1	Create reservation	1
R3.2	Search a reservation and view its details	2
R3.3	Edit a pending reservation	1
R3.4	Cancel a pending reservation	2
R3.5	Delete a pending or checked out reservation	2
R3.6	Add service to a reservation	2
R4.1	Search for an existing customer	2
R4.2	Edit an existing customer	1
R4.3	Delete an existing customer	2
R5.1	Search for a bill	2
R5.2	View a bill as PDF	2
R5.3	Print a bill	3
R5.4	Mark bill as paid	1
R6	Check-in a pending reservation	1
R7	Check-out a checked-in reservation	1

3. Outline of the design

3.1 Re-engineering by Re-writing

Re-engineering decision:

- Old: The legacy software is about 14 years old. It's implemented using the java version of that time, so some of the command are not work anymore.
    - The implementation is old, some of the methods are deprecated like (setNextFocusableComponent) in JHotel_Translator class. We don't know what this method was doing in that version of java and with which method we can replace it.
    - Some of the methods are not compatible with the new operating systems. The method getType() in the class RoomSelectWindow.java is not allowed in JDK 8 anymore since it is not compatible with the new operating systems.
- The software we need to maintain is inherited from previous developers, so it needs us to guess how the code works.
- The the legacy system is not documented so it's hard to understand it.
- The new features that we are contracted to add need to make changes. So, it's difficult and more likely to cause a regression. As the legacy system is using a local data base but we need to have a shared database that we have two branches of the hotel in two different cities.
- The legacy system is untested. So, it's hard to test if all features work correctly.
- The legacy software is not reusable or maintainable. All classes are gathered in one package, and there is no design pattern used (such as Model-View or another pattern)
- Inflexible code:
    - Some classes are responsible for more than one feature and while we need to edit some of the features, it could affect unexpectedly other features.
    - Almost all the existing features are implemented without following any pattern. For example, the searching feature does not follow any search pattern (as the Filter pattern). So, it is difficult to add a new search category.
- The legacy system does not follow any architecture pattern and we decided to use the client-server pattern as an architecture pattern to support the multiple locations of the hotel. So, it is time consuming to refactor the legacy system to follow this pattern.

After we collect all the previous results by analyzing the legacy system. We decided to re-write the software instead of re-factor it. While refactoring is more time consuming and less reliable.

3.2 High-level description of the design

Architectural pattern: Centralized Client/Server

- One server provides the service to many client. The client does not concern with how the server works while processing the request and send the response.
- Clients and server exchange messages in a request–response pattern, i.e. the client sends a request the server responds to the request.
- Clients and the server should know the format of the data for the request and the response based on an already defined.

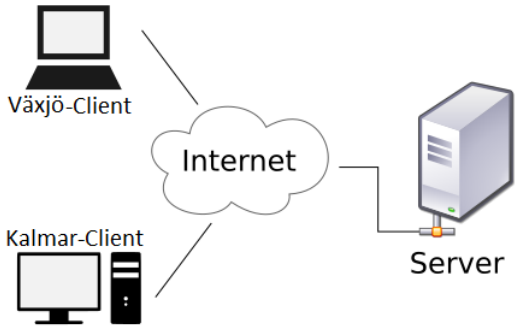

FIGURE 1 - A REPRESENTATION OF ARCHITECTURAL PATTERN (CLIENT/SERVER)

4. Major design issues

- **Issue 1:** Choosing the programming language

  Option 1.1: Java

  Option 1.2: C#

  Decision: Regarding the interview with the hotel manager, the program should be compatible with Windows and IOS. So, we decided to use Java.

- **Issue 2:** Choosing the architecture pattern

Regarding the software requirement, the software will be used from two different locations and each location should have access to the room information of the other location. Therefore, we have two optional patterns.

Option 2.1: Peer-to-Peer

Option 2.2: Server-Client

Decision: To prevent have a conflict or overbooking, the database and the process of each request should be analyzed in a central resource and the peers do not need to connect each other. Therefore, we chose the Server-Client pattern.

- **Issue 3:** Choosing the database type

  Option 3.1: SQL

  Option 3.2: XML

  Decision: Since we chose the Server-Client pattern, we already have a central server that can save the database locally as an XML file. Instead of making a new connection to the SQL server. So, we let the client avoid having two queues.

  - Queue in the case of save the database locally as an XML file:

FIGURE 2 - A PICTURE SHOW QUEUE OF THE HOTELSERVER

  - Queue in the case of save database in a SQL server:

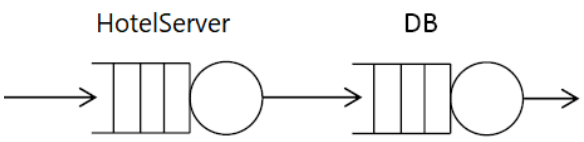

FIGURE 3 - A PICTURE SHOW QUEUES OF THE HOTELSERVER AND THE DATABASE SERVER

- **Issue 4:** How the sever perform saving new data without affect the response speed and be able to deal with two saving requests at the same time?
  Decision: To avoid a slow response, we decide that the save process will be done by a separate thread. And to solve the problem of having two saving requests at the same time, we decide to use message-queue pattern to provide an asynchronous saving. Therefore, each saving request will be saved in the defined list and the thread will continuously check the list and perform the saving while there is a new request in the list.

- **Issue 5:** How the program will convert a bill to PDF file.
  Decision: We decided to use third party (iText library) since it is reliable and this library is familiar for us. We use iText at no charge since we distributed the source code on github under the AGPLv3.

5. Design details

5.1 Software Architecture

Architectural pattern: Centralized Client/Server

- One server provides the service to many client. The client does not concern with how the server works while processing the request and send the response.
- Clients and server exchange messages in a request–response pattern, i.e. the client sends a request the server responds to the request.
- Clients and the server should know the format of the data for the request and the response based on an already defined.

Example:

When a client wants to view the list of all rooms in the hotel, the client initiates a request to the hotel's server. An application server interprets the received request, and provides the output to the client for display.

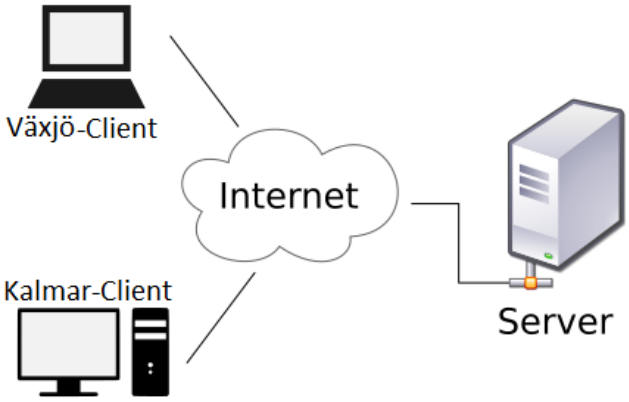

FIGURE 4 - A REPRESENTATION OF ARCHITECTURAL PATTERN (CLIENT/SERVER)

Client/Server Communication:

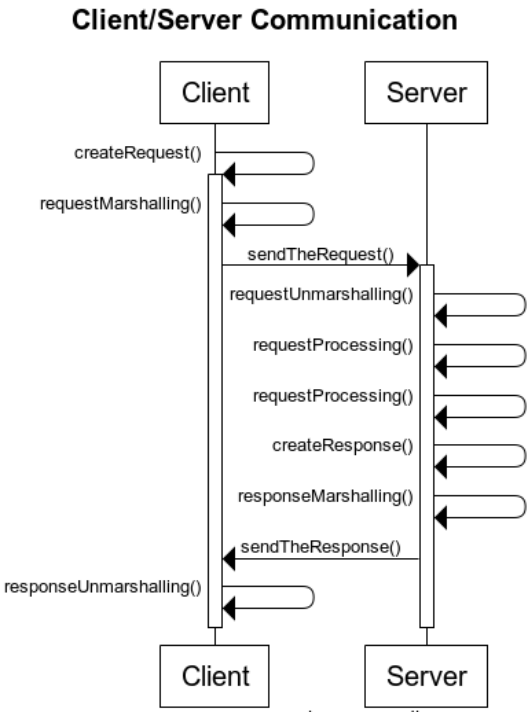

FIGURE 5 -A SEQUENCE DIAGRAM OF THE CLIENT/SERVER COMMUNICATION

UML Component Diagram of the architecture

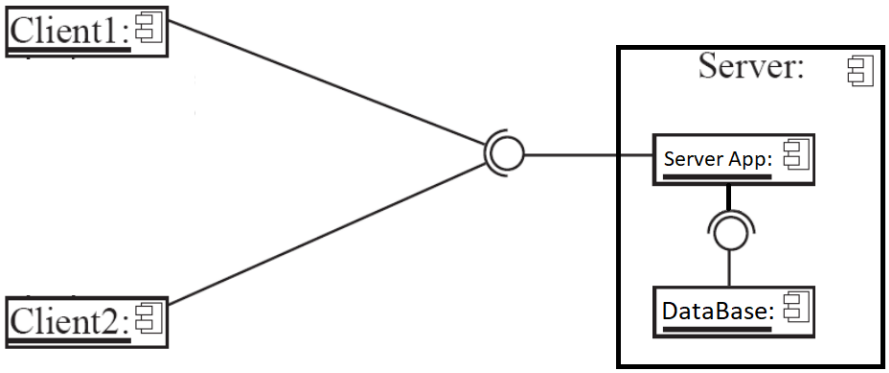

FIGURE 6 - AN UML COMPONENT DIAGRAM OF THE SOFTWARE ARCHITECTURE

Database:

In this version (v1) of the server app, the server will save the database locally as an XML file.

The server app has 3 threads:

1- Thread stay in listening state and wait for new connection. When new request arrives, the server create new thread to serve the request and continues waiting for new request.

2- Thread is responsible for updating the status of each reservation and room. This thread updates the status every day at 00:00:00 o'clock. Then it sleep until the next day unless another process interrupts it.

3- Thread is responsible for saving the new information to the database after each modify request. This thread excutes all saving request in its message-queue and then it sleeps until the server performs a new modify response.

The server state machine

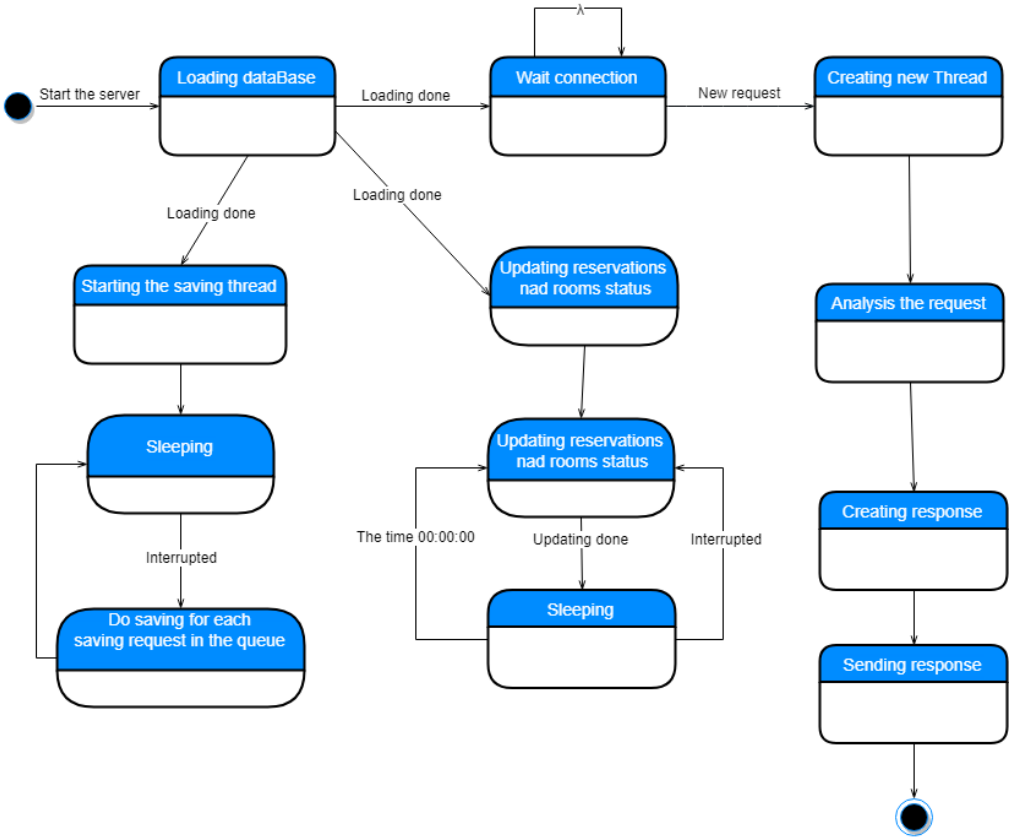

FIGURE 7 - STATE MACHINE DIAGRAM OF THE SERVER

The server has many states at the same time. After loading the database, the server will go to two states at same time (wait connection and update the status).
λ means the server will stay in waiting states even if it go to another state.

Design Patterns:

Generating response (Factory pattern):

When the server get request, it will generate corresponding response. To generate a response, the server follow the factory pattern

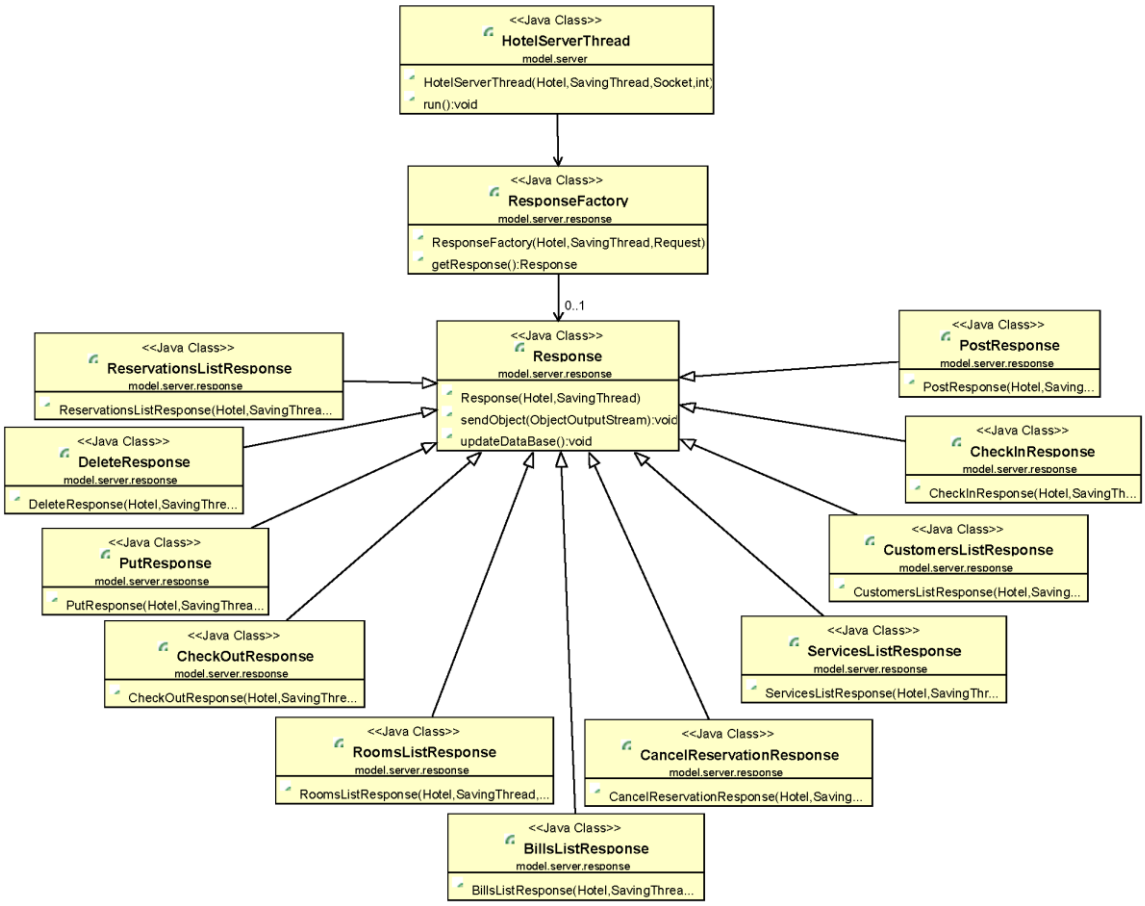

FIGURE 8 - A CLASS DIAGRAM OF THE SOFTWARE FACTORY RESPONSE (CLICK TO VIEW FULL SIZE IMAGE)

Search pattern: (Filter pattern)

The server or the client follow the filter pattern to search for customers, reservations, rooms or bills. For example, for customers search:

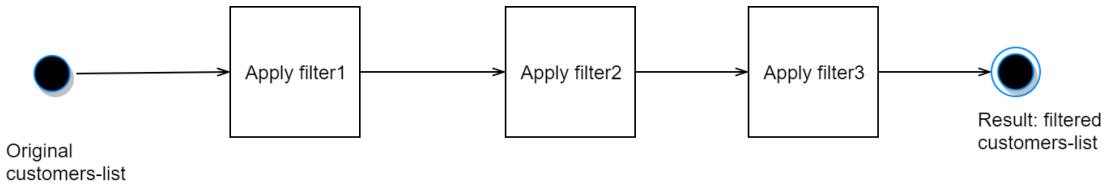

FIGURE 9 - A DIAGRAM SHOWING THE PROCESS OF FILTER PATTERN

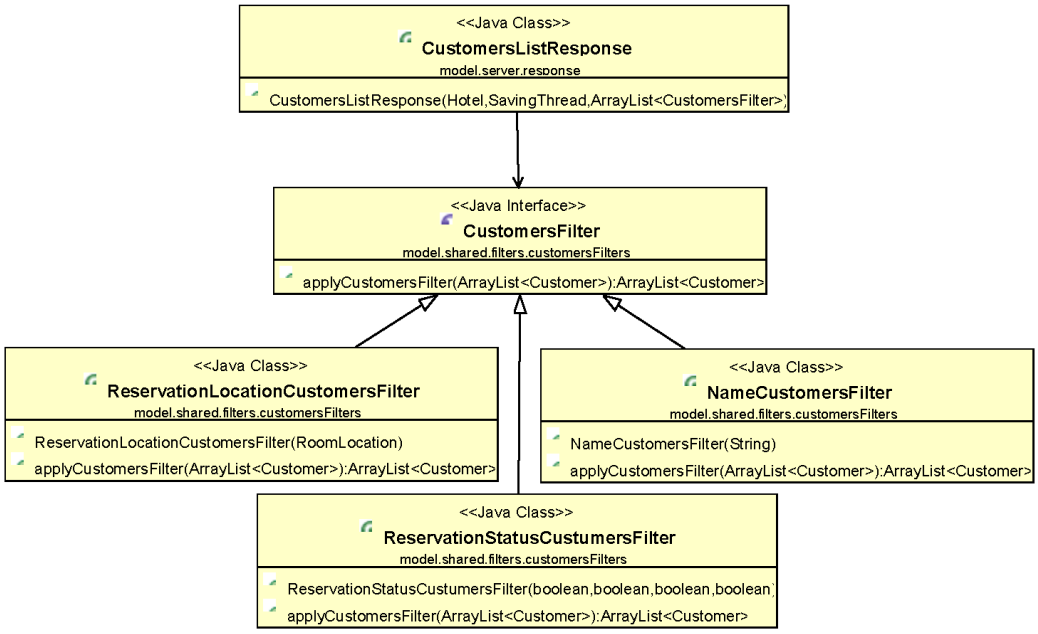

FIGURE 10 - A CLASS DIAGRAM OF THE SEARCHING PROCESS USING FILTER PATTERN (CLICK TO VIEW FULL SIZE IMAGE)

Saving Pattern: (message-queue pattern)

Each time the server receive a request to add/edit/remove an object, the server will put a saving request in a buffer. The responsible object for the saving will continues for all elements in the buffer. The saving is performed by separate thread which uses message-queue pattern to provide an asynchronous saving (This means the saving can be performed after a while from receiving the request and send the response).

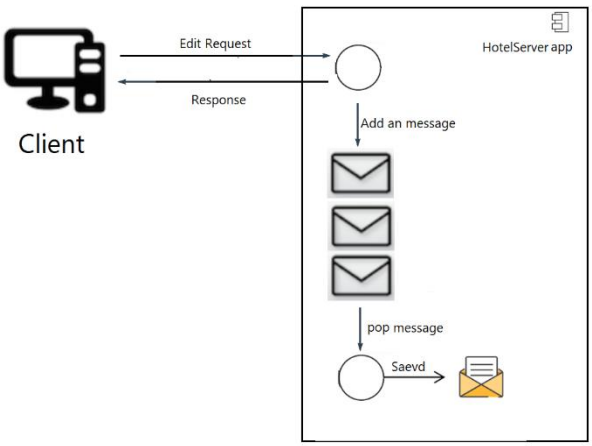

FIGURE 11 - A PICTURE SHOWING THE SAVING PROCESS USING MESSAGE-QUEUE PATTERN

API:

- **Server API:**
  ServerAPI.html

- **DataBase API:**
  HotelDAO.html

5.2 The UML Class Diagram

- The model Package:
  - The client
    -

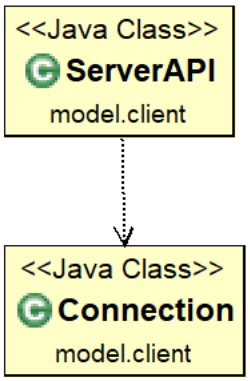

FIGURE 12 - A CLASS DIAGRAM OF THE MODEL.CLIENT PACKAGE

  - The server
    - Associations and Generalizations

FIGURE 13- A CLASS DIAGRAM SHOWING THE ASSOCIATIONS AND GENERALIZATIONS OF MODEL.SERVER PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- Dependencies and Realizations

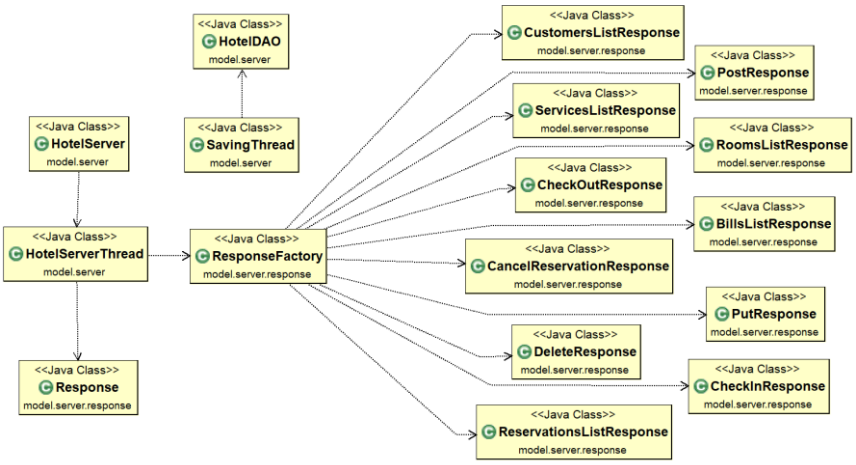

FIGURE 14 - A CLASS DIAGRAM SHOWING THE DEPENDENCIES AND REALIZATIONS OF MODEL.SERVER PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- The shared package (Shared classes between Server and Client)
  - Associations and Generalizations

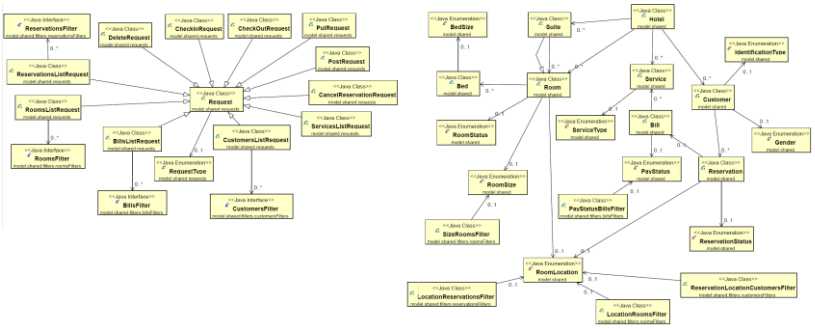

FIGURE 15 - A CLASS DIAGRAM SHOWING THE ASSOCIATIONS AND GENERALIZATIONS OF SHARED PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- Dependencies and Realizations

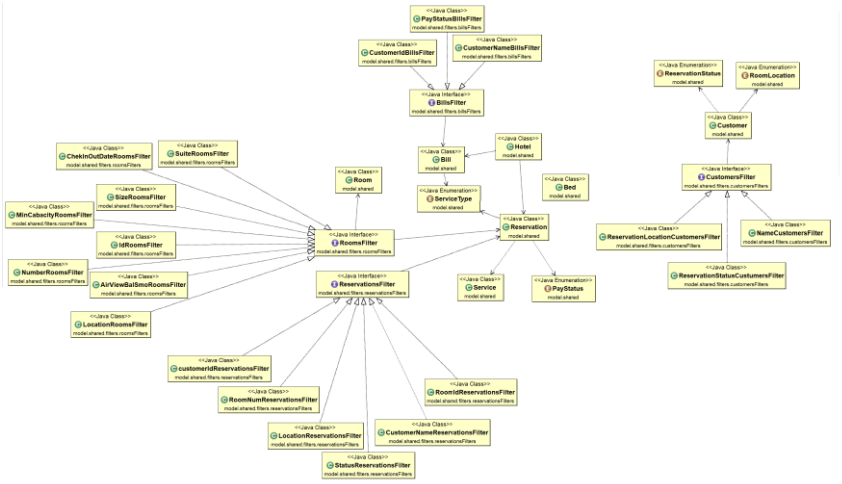

FIGURE 16 - A CLASS DIAGRAM SHOWING THE DEPENDENCIES AND REALIZATIONS OF SHARED PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- The relations between the server package and the shared package
  - All relations

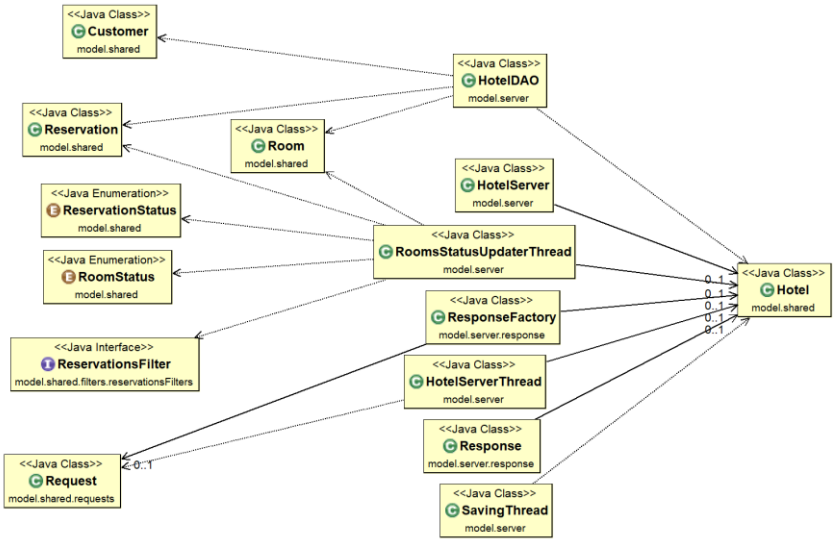

FIGURE 17 - A CLASS DIAGRAM SHOWING THE RELATIONS BETWEEN THE SERVER AND SHARED PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- The relations between the client package and the shared package
  - All relations

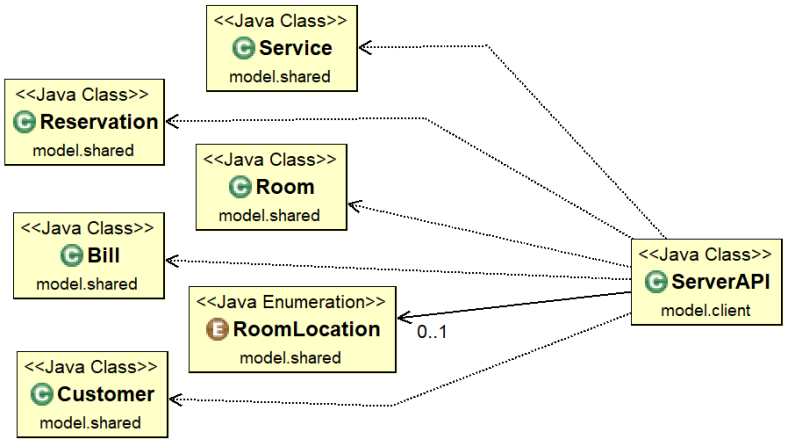

FIGURE 18 - A CLASS DIAGRAM SHOWING THE RELATIONS BETWEEN THE CLIENT AND SHARED PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- The controller package:
  - The client
    - Associations and Generalizations

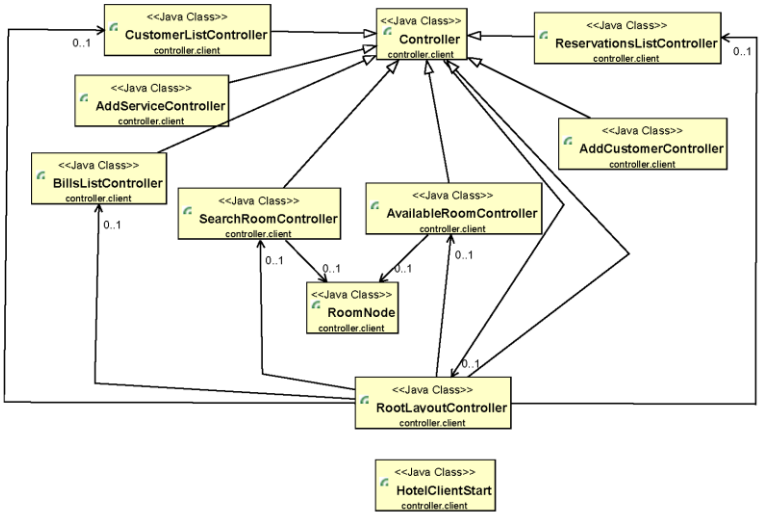

FIGURE 19 - A CLASS DIAGRAM SHOWING THE ASSOCIATIONS AND GENERALIZATIONS OF CONTROLLER.CLIENT PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)

- Dependencies and Realizations

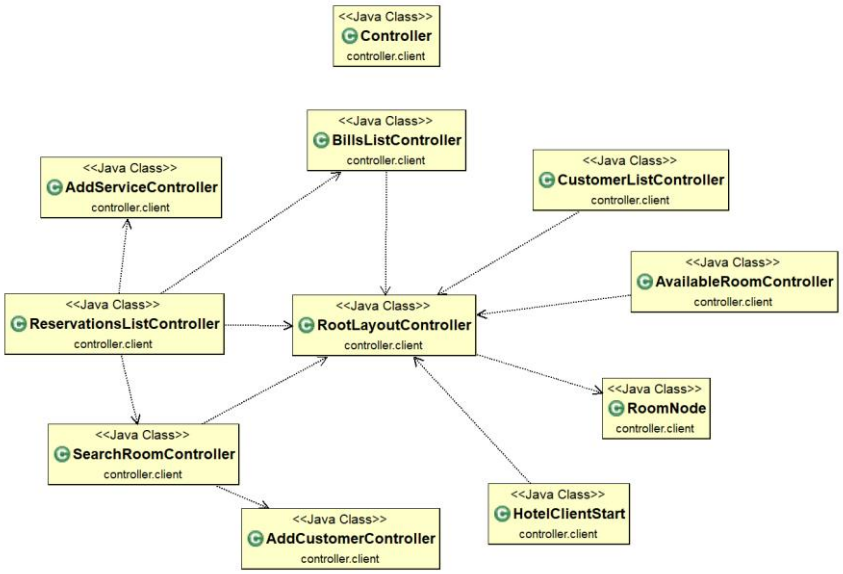

FIGURE 20 - A CLASS DIAGRAM SHOWING THE DEPENDENCIES AND REALIZATIONS OF CONTROLLER.CLIENT PACKAGE (CLICK TO VIEW FULL SIZE IMAGE)