<Hotel Booking System/Framework>

# **Technical Design Document**

Version 1.0

04/24/2017

<Hotel Booking System/Framework>

**Group Members:**

Van Hieu Ho – 985551

Le Hoang Nguyen - 985605

Huy Le – 985519

Tien Thanh Le - 985606

[**Technical Design Document**](#_xb8efen9ui7j) **0**

[**1. Introduction**](#_egpplgk4p54q) **2**

[**2. Goals, Objectives for New System**](#_ipyw78kyz0kv) **2**

[2.1 Project Purpose](#_v7rhmiuueq7i) 2

[2.2 System Goals and Objectives](#_qbfxq5ojfjce) 2

[2.2.1 High-Level Functional Requirements](#_fubr1fdoadml) 2

[**3. Proposed System**](#_gfqlyz7akhyl) **2**

[3.1 High-Level Operational Requirements](#_kdw3sub9rr9g) 2

[3.2 High-Level Architecture](#_g3a1ee1mmu29) 3

[3.2.1 Application Architecture](#_nq5nki75gf87) 4

[3.2.3 Interface Architecture](#_pw69wg15tq2e) 6

[**4. Design Patterns**](#_7uv8g7ulzrfx) **7**

[4.1 Singleton](#_p9kg1pnn8ipv) 7

[4.2 Builder](#_13b6mp1jomuj) 8

[4.3 Factory Method](#_16zg9ggq9jkz) 9

[4.4 Proxy](#_5ioq6lg4fxms) 10

[4.5 Prototype](#_g2ow5em1n4u2) 10

[4.6 Template method](#_tk0ojed838fs) 11

[4.7 Adapter](#_cuja6q4qowtb) 12

[4.8 Observer](#_7j0e6y6xgagf) 12

[4.9 Facade](#_bsi7whql4673) 13

[4.10 Strategy:](#_v99uy5gxlcsb) 13

[4.11 Visitor](#_1x6glw9fm992) 14

[**5. Framework**](#_x2ymo97wah5h) **15**

[5.1 Design Approach](#_6jue6xi1429q) 15

[5.2 Framework Architecture](#_qdrtzq8679dz) 16

[5.2.1 DaoAbstract:](#_f70ow0o9asna) 16

[5.2.2 Entity:](#_axeg3svqzhl) 17

[5.3 Usage](#_gaarf4djrgfw) 18

# **1.** **Introduction**

Hotel Booking System aims to build a multiplatform application for hotel booking and management system. Starting as a standalone app, then engineered to adapt Rest API architecture to better support the expandability.

# **2.** **Goals, Objectives for New System**

## **2.1** **Project Purpose**

Create a brand Hotel Booking System, which allow user to create and manage Hotel Room, Service, Booking as well as other common administrative tasks. Project aims to demonstrate the applicability of design patterns to solve design issue and come up with a partial framework as part of output.

## **2.2** **System Goals and Objectives**

### **2.2.1** **High-Level Functional Requirements**

* System should allow its user to create Room and Service.
* It should provide functionality for a customer to select and book Room.
* It should provide check in and check out operation.
* Reporting system needs to provide operational status of the hotel.

# **3.** **Proposed System**

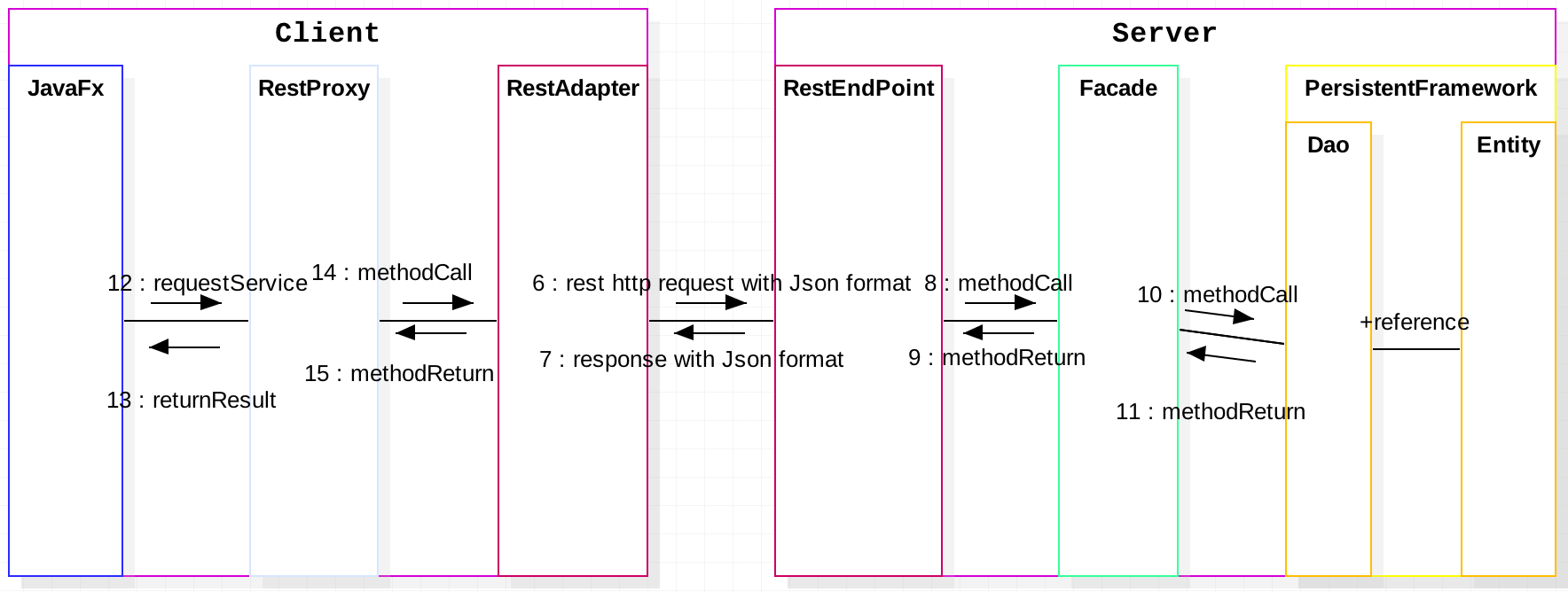
## **3.1** **High-Level Operational Requirements**

Table 1 - User Community Description

|  |  |  |
| --- | --- | --- |
| **ID** | **User Case** | **Description** |
| 1 | CRUD Customer | Customer:  - Full Name  - Day of birth  - ID/Passport  - Address  - Phone number |
| 2 | CRUD Room | Room:  - Room Number  - Room Type  - Room Class  - Price |
| 3 | CRUD Service | Services:  - Service Description  - Price |
| 4 | Checkin | Selecting room (class, type, number) and dates to check in. |
| 5 | Checkout | Room and service cost are calculated and add to Invoice once Checkout is selected. |
| 6 | Reporting | Reporting:  - Room Status Report  - Customer Status Report  - Operational (Revenue) Report |

## **3.2** **High-Level Architecture**

* Client:
  + JavaFX: GUI elements and Controller
  + RestAdapter: convert client request to RestFul request to server side
* Server:
  + RestEndPoint/API: In order to have better support performance, scalability, and modifiability, Restful Web service is introduced to handle request/response between clients and servers.
  + Query Facade and Maintenance Facade provides the interfaces and functionalities to multiple specific components.
  + DAO/Entity handle data back and forth between Java objects and database records.



### **3.2.1** **Application Architecture**

Table 2 - Description of Application Components

|  |  |  |
| --- | --- | --- |
| **ID** | **Application Component** | **Description** |
| 1 | Entity(Framework) | Customer  Room  Service  User  InvoiceRecord  Revenue |
| 2 | DAO (Framework) | move data back and forth between Java objects and database records:  CustomerDao  DaoFactory  InvoiceRecordDao  LoginDao  RoomDao  ServiceDao |
| 3 | Rest Adapter | RestAdapter, RestAdapterProxy, IRestAdapter |
| 4 | RestEndPoint/API | RestQueryInterface, RestUpdateInterface, RestUpdateImpl, RestQueryImpl |
| 4 | Controller | Main Controller  Checkin Controller  Checkout Controller  Room Controller  Service Controller  Customer Controller  Invoice Controller |
| 5 | Utility | Supporting utilities: SQLite, Email, Application Constants, General operational utilities. |

**3.2.2** **Information Architecture**

Table 3 - Description of Information Components

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Entity** | **Description** | |
| 1 | Customer | <Customer Entity>  Passport  Full Name  Phone  Address  Day of Birth | |
| 2 | Room | <Room Entity>  Room Number  Room Type  Room Class  Price | |
| 3 | Service | <Service Entity>  Service Description  Service Price | |
| 4 | User | <User Entity>  User Id  Firstname  Lastname  Age  Username  Password | |
| 5 | InvoiceRecord | <Invoice Record Entity>  Checkin Date  Checkout Date  Room Amount  Service Amount  Total Amount | |

### **3.2.3** **Interface Architecture**

Table 4 - Description of Required Interfaces

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Information Shared** | **Interfacing Application** | **Purpose** | **Data Stored Persistently** |
| 1 | Customer information  Room Information | Customer and Room Form | Enter customer information and selected room | Yes |
| 2 | Room information  Service information | Room Service Form | Select services for certain room | Yes |
| 3 | Customer information  Booked in information | Checkin Form | Enable customer to check in | Yes |
| 4 | Customer information  Checked in room information | Checkout Form | Enable customer to check out | Yes. Invoice generated & stored. |
| 5 | Room Information | Room Form | CRUD Room | Yes |
| 6 | Service Information | Service Form | CRUD Service | Yes |
| 7 | Room Information  Customer Information | Report: Room, Revenue | Enable to generate report for Room, Revenue or Customer | Yes |

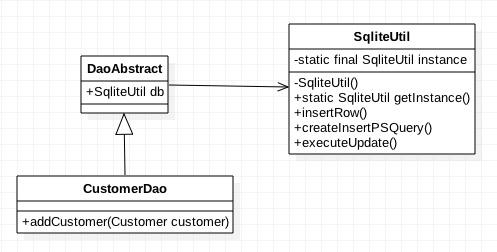
# **4. Design Patterns**

## **4.1 Singleton**

Motivation:

* Provide global point of access to application’s SQLUtil object. This object is used for all database activities
* Make sure there is only one instance of the SQLUtil initialized.

Implementation:



## **4.2 Builder**

Motivation: Invoice is a complex object. Instead of passing many parameters to its constructor InvoiceBuilder allows us to construct step by step from independent part of Customer, Room and Service..

Implementation:

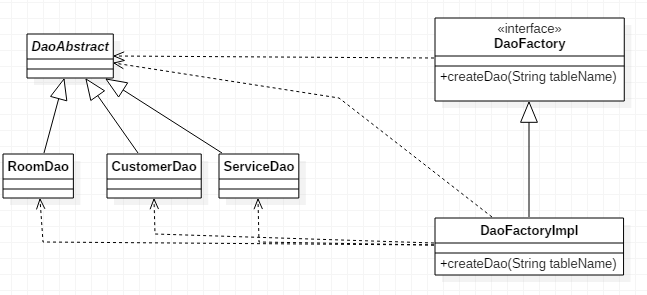


Participants:

* Builder : Specifies an abstract interface for creating parts of an InvoiceRecord object.
* InvoiceBuilder: Concrete Builder class, constructs and assembles parts of the InvoiceRecord by implementing the Builder interface.
* BasicCheckoutController: Director, construct InvoiceRecord object using Builder interface.
* InvoiceRecord, Product class, represents the complex object under construction.

## **4.3 Factory Method**

- Motivation: provide a factory so that the user can get the DAO object basing on the table name only.



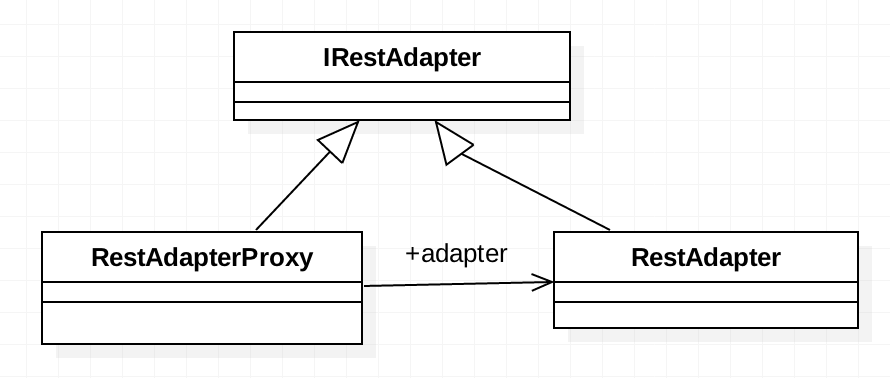
Participants:

* DaoFactory: interface define the create DAO method interface
* DaoFactoryImpl: implement the create method in DaoFactory to return the DAO object basing on the tableName
* DaoAbstract: abstract DAO class.
* RoomDao, CustomerDao, ServiceDao,...: concrete DAO class which extends the DaoAbstract.

## **4.4 Proxy**

Motivation:

We dont want Client to access to RestAdapter directly. Also, there might be additional protection for different client in the future.



Participants:

- IRestAdapter: the interface that client uses to explore the RestAdapter

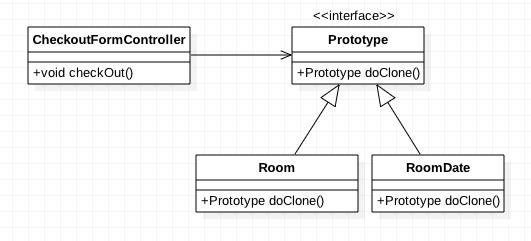
- RestAdapterProxy: the proxy that prevents client talking directly with RestAdapter.

- RestAdapter: rest implementation for the adapter to connect to RestAPI

## 4.5 Prototype

Motivation: Support cloning the Room and RoomDate object to reduce the cost of initializtion and avoid changing the original object an

Implementation:



Participants:

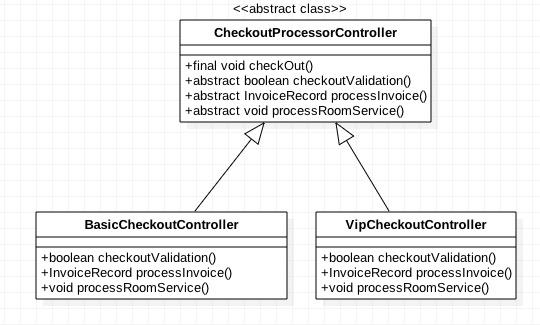
* CheckoutFormController: Controller that does the checkout process. During the process, the Room and RoomDate are cloned
* Prototype: Interface that define the provide the cloning function
* Room and RoomDate: Implement the cloning function

## **4.6 Template method**

Motivation:

Define the skeleton algorithm for checkout process

Implementation:



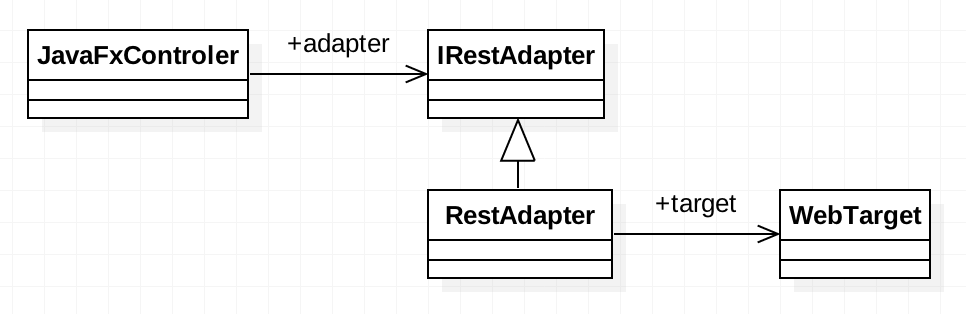
Participants:

* CheckoutProcessorController: The abstract class that define the template method (checkout). The detail processing are defined as abstract methods.
* BasicCheckoutController / VipCheckoutController: Implement primitive operation that specific for each type of checkout.

## **4.7 Adapter**

Motivation:

We need an adapter to convert client request to another request that RestApi can understand.



Participants:

- JavaFxControler: client controlers. It can be JavaFx or any Client application

- IrestAdapter: interface of RestAdapter service

- RestAdapter: the adapter to convert from client API to call WebTarget which is RestAPI

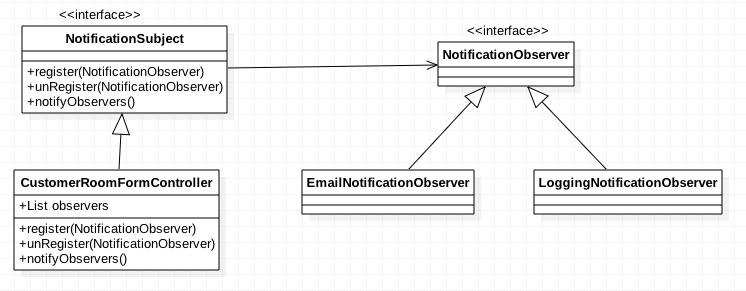
- WebTarget: RestAPI library which connects to Rest Service end point

## **4.8 Observer**

Motivation:

When a booking system recognize a booking that over a specific amount, it will automatically send the email to management as well as logging the information

Implementation:



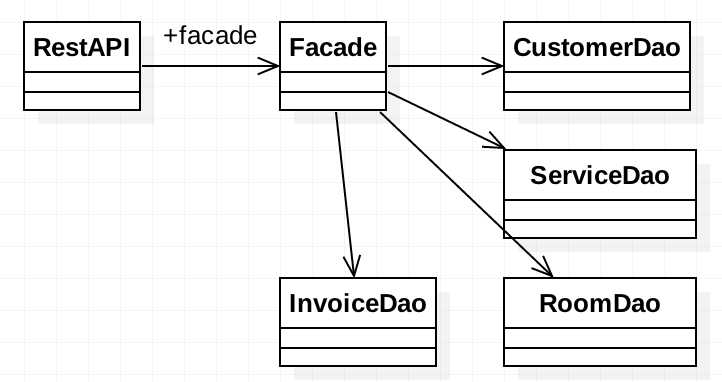
Participants:

* NotificationSubject: Provide interface for registering and unregistering the observer. Holds the list of observers
* CustomerRoomFormController: Implement the interface. Send notification to its observers
* NotificationObserver: Define an updating interface for objects that should be notified of changes in a subject
* EmailNotificationObserver / LoggingNotificationObserver: Implement the observer interface to send email and log the booking information depends on how the CustomerRoomFormController register the observer.

## **4.9 Facade**

Motivation:

We need to centralize Daos into Facade so that Rest only knows Facade operations, not accessing directly to Dao.



Participants:

- RestAPI: the server side interface that published the rest services

- Facade: the facade to centralize the Dao business service. Instead of accessing those Dao directly from RestAPI, it has to go through the Facade

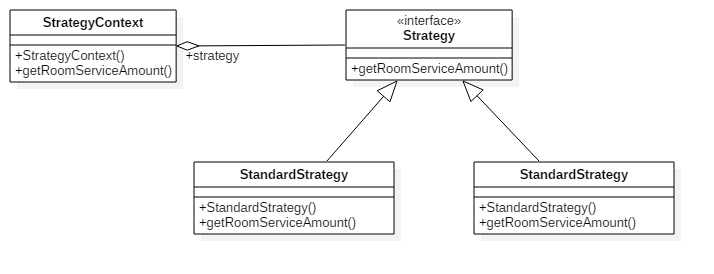
- Daos: the daos which interact with Databse directly via Entity and SQL

## **4.10 Strategy**:

Motivation:

When a customer using different room class, associated services should be different. Using strategy pattern could allow different algorithms for pricing VIP service and Standard service.

Implementation:



Participants:

- Strategy: declare common interface for supported algorithms - getRoomServiceAmount()

- VIPStrategy: Implement VIP Strategy to get Room Service Amount for VIP class.

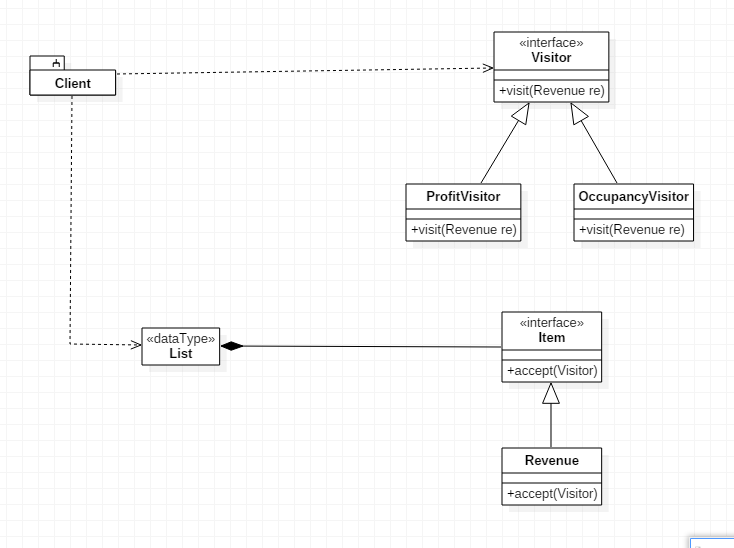
- StandardStrategy: Implement Standard Strategy to get Room Service Amount for Standard class

- StrategyContext: hold concrete strategy to calculate the Room Service Amount when checkout and help to construct Invoice for customer.

## **4.11 Visitor**

Motivation: in a month, we have many invoices with the customer check-in, check-out information. With each invoice, we would like to calculate the total revenue in that month and the occupancy rate within that period of time.

Implementation:



Participants:

- Visitor: the interface defines the visit method to each concrete object.

- ProfitVisitor: implement the Visitor interface to calculate the revenue of the hotel in a month.

- OccupancyVisitor: the visitor to calculate the occupancy rate of the hotel in a month.

- Item: the interface defines the accept method to apply visitor operation.

- Revenue: concrete object to apply operations of the visitor.

# **5. Framework**

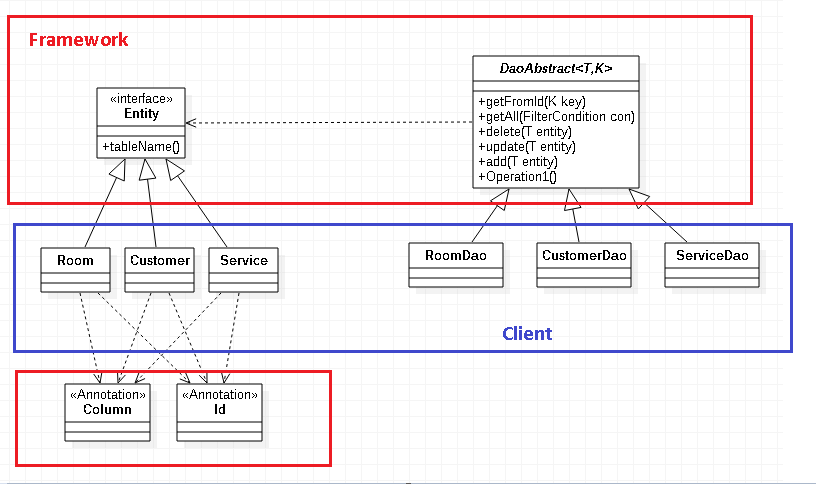
## **5.1 Design Approach**

Framework design approach: From our experience on developing the application with the aim of applying design patterns to solve engineering issue, we came to agree that:

We will try to implement a simple Persistent framework which is similar to Spring/Hibernate framework using the DAO design pattern. In the entities class, we use annotations to define the structure of the table in the database (columns, primary keys, relations,...). The DAO class corresponding to each table will refer to each entity to CRUD data from DB. To make DAO classes to work, they only have to inherit from an abstract DAO class in the framework which already implemented basic operations create, update, select, delete from DB.

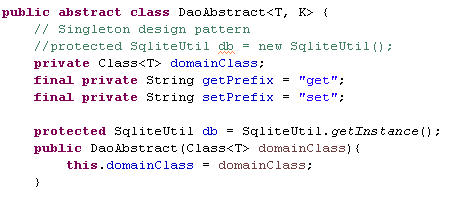
By using this approach, the client code doesn’t have to re-implement every CRUD for each table. By that, we reduce the number of codes the user has to write and make them easier to maintain.

## **5.2 Framework Architecture**



## **5.2.1 DaoAbstract:**

This is the abstract class which implements most of the basic CRUD operation. Because we’re developing a framework which will deal with various Entity classes, so we need to specify the generic data type as a template for this class ( T is entity data type, K is key data type).



Class<T> domainclass: Because of the type erasure in Java when using generic type, we need to keep track the concrete class of the entity. Because this domainClass will be initialized in the constructor so every DAO inherit from this abstract class need to specify the domainClass in its constructor.

SqliteUtil db: a singleton instance to connect to SQLite DB.



Get an entity from DB with the key value.



Get all entities with a defined condition. This condition is optional, if no condition, it returns all rows in table.



Delete entity t from DB, or all rows with the optional condition.



Update values of an entity into DB.



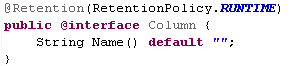
Insert an entity into DB.

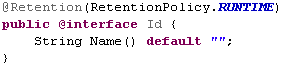
## **5.2.2 Entity:**

All of the entities must inherit this interface. It defines a method to get the table name which link with an entity. ( Because we didn’t develop a tool to generate entities code to DB tables or vice versa, so we don’t have the metadata to let us know which entity belongs to which table).



**5.2.3 Annotations:**





These annotation will be used by the entities class to define the structure of the table. These annotation must be used in runtime so we set the retention policy to runtime.

*Column*: normal columns name

*Id*: primary key columns.

These annotations have a property called Name() to map the member variables name in the entity class to the real columns name in the table. If the Name() is not defined, the variables name are also the columns name.

## **5.3 Usage**

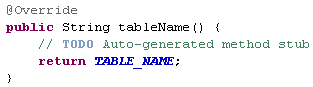
To use this framework, the client code need to follow belows rules:

**5.3.1 Entities class:**

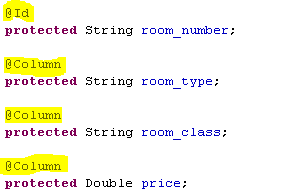
* Entity class must implement from the interface Entity



* So that it must implements a method to return the table name:



* It must map the member variables to column names in DB by using the provided annotation:

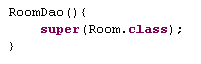


**5.3.2 DAO classes:**

* DAO class must inherit from DaoAbstract class and specify the Entity class it refers to



* In constructor, it must specify the entity class to the superclass:



And that’s it! The client code are ready to CRUD data from DB.

**5.3 Limits:**

* Because the rush of time, we didn’t implement more Annotations to specify relations between tables yet ( one to many, many to many,...)
* It didn’t really support the composite primary keys yet. To work-around, we can use id column or execute methods with defined filter conditions.