Design Document-Auctioneers

**Technologies Used:**

1- Native Android SDK

2- ORMLite (Lightweight Object Relational Mapping (ORM) Java Package)

3- JUnit4 ( Simple framework to write repeatable tests.)

4- Mockito (Mocking framework for unit tests written in Java)

**Why ORM?**

* When applying ORM model entities become based on real business concepts, and this facilitates implementing the Domain Model pattern.
* Saves time by avoiding boilerplate code for CRUD operations.
* Rich query capability. ORM tools provide an object oriented query language.
* And for the sake of extensibility of our app, as it's planned to add Category entity in which each item will belong to one of these categories (1:m relationship) One to many relationships are easier to be handled by ORM.

**Why Junit4?**

* As recommended by the official documents of android testing.
* Supported by android studio.
* Works in harmony with Mockito for mocking.

**Why Mockito?**

* Mocking gives you the ability to test your components in isolation of the rest of the system.

**Design Patterns Used:**

**MVP for the presentation layer:**

* In which the activities act as passive views.
* Then the presenters contains the whole presentation logic + executing the suitable use cases for each view.
* I skipped the model as there's no need for any kind of mapping between the data in the business layer and the data in the presentation layer. (Date is presented the same way it's stored)

**Presenters**

* All presenters extend PresenterImp.java in which it implements PresenterInterface.
* In order to respect the Dependency rule of Uncle Bob's clean architecture Only The   
  Presenter-Interface is passed to the Use-Cases (business layer)

**The Domain (Business layer)**

**UseCases:**

* Each use case class can be mapped into a real life use case (feature or sub-feature)
* All UseCases extends UseCaseImp in which it implements UseCaseInterface.

The only available constructor for UseCaseImp is:

**public** UseCaseImplementation(Context applicationContext,PresenterInterface presenter)

The UseCase is responsible to notify the presenter with the   
 requested data by invoking the   
 presentationInterface methods

**presenter**.onSuccess(data);

Or

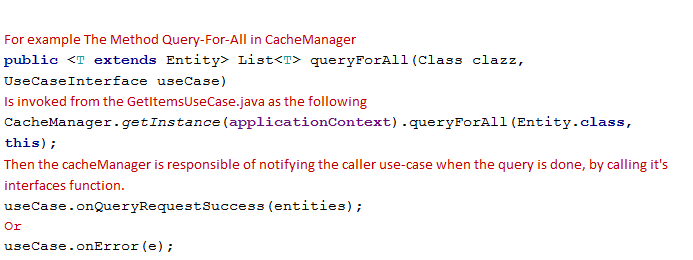
**presenter**.onError(e);

**Singleton CacheManager:**

* This component encapsulates ORMlite, it's the only component that is aware of ormlite existence, and by sequence the only component that is aware of SQLite existence.
* Then if we woke up and decided to use another ORM, or even Decided to drop SQLite for the favor of Flat file database (or any other persisting mechanism). All the changes will be in and only in that class.

Why Singelton?  
1- Because it controls concurrent access to a shared resource (Sqlite Database instance in our case).

2- Access to that resource (sqlite Database instance) will be requested frequently while the app Iis running.

\*The CacheManager is only accessible from the Business layer (Use-cases), in which the CacheManager itself is unaware of anything in the business layer other than the UseCaseInterface.

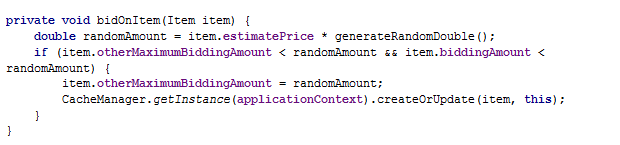
# 

# Bidding Bot: \*Each Item in the system has an estimate price.

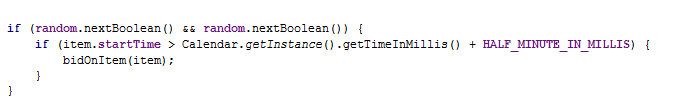
The bidding bot is merely another use case that works in the background.  
Every time the user request the items, the bot will use a simple algorithm to place random bids on items.  
This algorithm is as follows:

Generate Random Bidding amount at least half the estimate price of the item, and at most double  
the estimate price. *(In order to make the Bot sounds sane*)

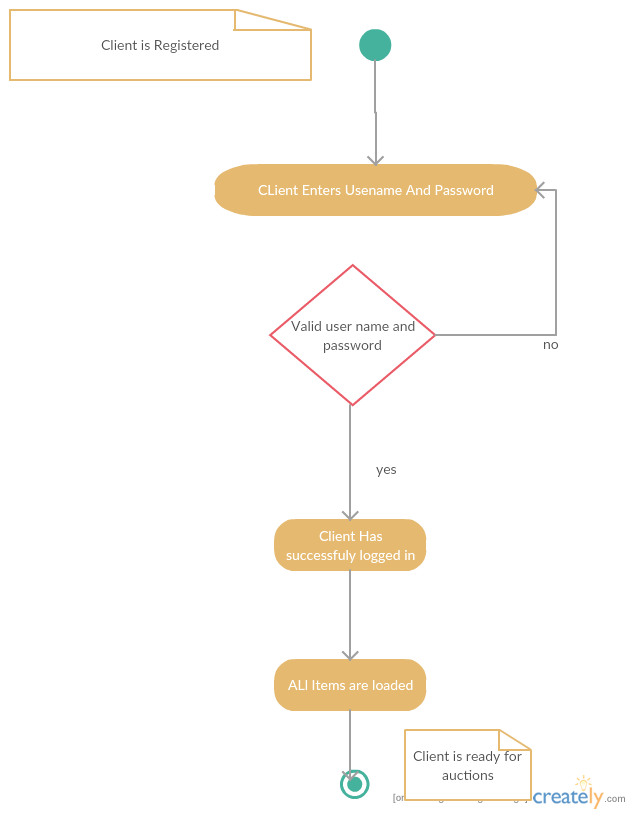
If the Bot tries to bit on the same item twice, then the latest price must be larger than



To rationalize the number of items the Bot will place bidding on, the following simple algorithm is used  
also the item must not be expired. (At least has 30 seconds left)



# Login (Activity Diagram):



# List Won Items (Sequence Diagram)

