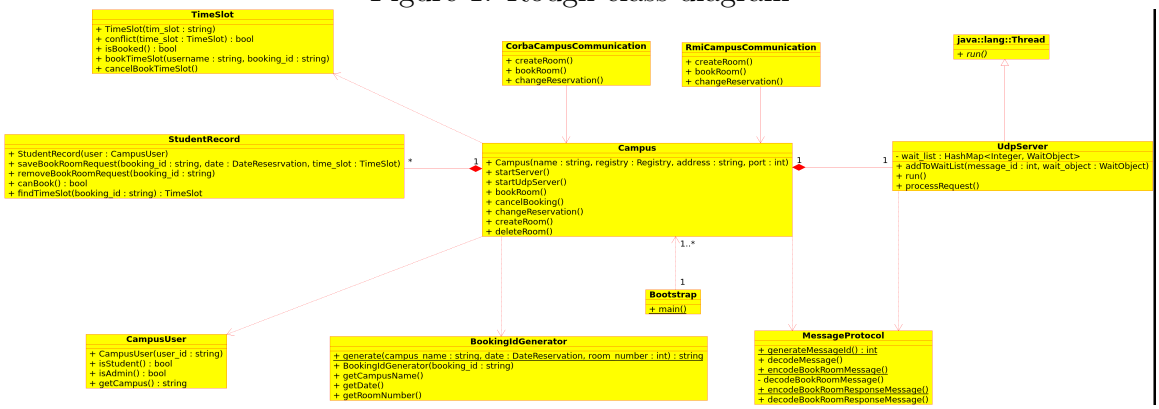
Comp 6231 – Assignment 2
RMI

Saman Saadi Alekasir
Student ID: 40009949

November 5, 2017

Figure 1: Rough class diagram



1 Adopting to New Requirements

I decided to separate business logic from communication technology. In other words, My *Campus* class which is responsible to implement campus servers functionality contain only business logic of the servers. Both RMI and CORBA use this class for handling user requests. Due to checked exceptions in Java, It wasn't feasible to write one interface for both of them. Because they have different classes for exception handling.

Because of this separation I can run both RMI and CORBA with the same functionality. I defined a variable to determine communication technology. At the moment I only support RMI and CORBA and hopefully with a little changes I can support web services also. You can see a rough class diagram in figure 1.

2 Implementing the New Functionality

Since changing room reservation should be a transaction, first I try to book the new time slot and then remove the previous time slot. By doing that I have the concept of a transaction. The only problem is the user now can book more than 3 time slots. So I allow the user temporary book more than 3 time slots and after the transaction completed the amount of booked time slots per user should be at most 3.

3 Synchronization

According to Java SE 8 documentation [1]: "if multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.). This is typically accomplished by synchronizing on some object that naturally encapsulates the map". So it is not safe to use $HashMap.get$ when another thread try to change $HashMap$'s structure [2]. I should also mention $HashMap$ is not thread-safe for different keys [3].

3.1 Main Database

I designed three types of object locks for handling $HashMap$ database for multi-threaded environment. One locks the first key (date of reservation), the other locks the second key (room number) and the third locks a list of time slots for a specific date and room number.

I've noticed *createRoom* is similar to *deleteRoom* but some parts are different. For handling that I design a system which is used in event-handling systems. I define a private method named *traverseDb* which handles all synchronization when we traverse the database and call some abstract methods for handling different implementation between *createRoom* and *deleteRoom* operations. In other words, those methods should implement abstract class methods define in listing 1. I removed the arguments and the return value for simplicity. As you can see all of them are thread-safe (unless we try to access another database or container). In other words, *createRoom* and *deleteRoom* should only implement these thread-safe callbacks. Handling the synchronization is done in *traverseDb* method.

Listing 1: DbOperations abstract class

```
1 private abstract class DbOperations
2 {
3   //All the following methods
4   //are thread-safe for campus time slots database
5   public abstract onNullValue();
6   public abstract onNullSubValue();
```

```
7     public abstract void onSubValue();
8  }
```

You can see a snippet of *traverseDb* method implementation in listing 2. It is important to put lines 2 - 8 in a *synchronization* block. In *createRoom* method it implements *onNullValue* by creating a new *HashMap* for room number and list of time slots. If more than one thread try to execute it, each of them has their own new *HashMap* of room number and list of time slots which is not correct because one of those two threads has stale value.

Listing 2: Snippet from traverseDb method

```
1  synchronized (date_db_lock) {
2     val = db.get(date);
3     if (val == null)
4     {
5        val = db_ops.onNullValue();
6        if (val == null)
7           return;
8        db.put(date, val);
9     }
10 }
```

According to problem statement, we need to update student record when we delete a time slot which that user has booked. I create a thread for every time slot which is booked by a user from another campus which is responsible to get notification about the status of student record's update from another campus. At the end of method I'm going to wait for all created threads to finish their job.

3.2 Student Database

This database store the records of a particular student reservation and make sure it does not exceed 3 times per week. I follow the same pattern as main database. I've noticed *bookRoom* and *cancelBooking* have a lot in common. So I created a private method named *traverseStudentDb* which handles traversing student database and calls some callbacks which are implemented differently in *bookRoom* and *cancelBooking* methods. You can see the abstract class which both methods should implement in listing 3. All these methods are thread-safe, unless we want to access another database or container.

Listing 3: UserDbOperations Abstract Class

```
1  private abstract class UserDbOperations
2  {
3    //All the following methods are tread-safe for user database
4    public abstract boolean onUserBelongHere(record);
5    public abstract StudentRecord onNullUserRecord(user);
6    public abstract boolean onOperationOnThisCampus(time_slots);
7    public abstract void onOperationOnOtherCampus(message_id);
8    public abstract void onPostUserBelongHere(record);
9    public abstract ArrayList<TimeSlot> findTimeSlots();
10 }
```

I also try to lock student's record during the transaction. For example when user wants to book a room on another campus, the student record should be locked until we receive answer through UDP communication. When the campus send booking request to another campus, it create a *WaitObject* object and then calls its *wait* method. It's the responsibility of UDP server to notify it that its response has arrived.

3.3 Container Synchronization

According to *ArrayList* documentation [4]: "If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list.". So I need to synchronize accessing *ArrayList* (or other collections) when I want to access an element or traverse it. According to *Collections* documentation [5]: "It is imperative that the user manually synchronize on the returned list when iterating over it (see listing 4)".

Listing 4: Collections Synchronization

```
1  List list = Collections.synchronizedList(new ArrayList());
2      ...
3  synchronized (list) {
4    Iterator i = list.iterator(); // Must be in synchronized block
5    while (i.hasNext())
```

```
6         foo(i.next());
7     }
```

3.4 Socket Synchronization

It is safe to have two threads which one of them read from socket and the other write into it concurrently [6]. I only have one reader thread so I don't need to do synchronization here. But I can have multiple thread for writing into socket so synchronization is necessary.

4 Testing Scenario

I've changed the client side classes like the server side. So my client classes don't have any information about the underlying communication technology. So it was easy to use previous assignment unit tests. I also write some new unit tests for testing the new functionality in this assignment.

5 Conclusion

Since in three assignments we have the same functionality but different technology, I thought it would be better to separate the core implementation from communication technology in this assignment. So I have a two layer architecture. The main layer is responsible for providing services for communication layer which can be CORBA or RMI.

References

[1] Oracle. HashMap (Java Platform SE 8). `https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`. [Online; accessed 07-October-2017].

[2] StackOverflow. HashMap in multithreaded environment. `https://stackoverflow.com/questions/11050539/using-hashmap-in-multithreaded-environment`. [Online; accessed 07-October-2017].

[3] StackOverflow. Is a HashMap thread-safe for different keys? https://stackoverflow.com/questions/2688629/is-a-hashmap-thread-safe-for-different-keys. [Online; accessed 07-October-2017].

[4] Oracle. ArrayList (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. [Online; accessed 07-October-2017].

[5] Oracle. Collections (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html. [Online; accessed 07-October-2017].

[6] StackOverflow. Do Java sockets support full duplex? https://stackoverflow.com/questions/6265731/do-java-sockets-support-full-duplex. [Online; accessed 08-October-2017].