Dots and Boxes Project Design Document

Abstract:

Within this design document one will find the design details of a web based game where registered and logged in users can offer games, accept offered games, play games, and see completed games. In addition, the web application will allow concurrency through it being a multi-threaded environment handling synchronization when necessary. The approach I will take to explain my design includes diagrams depicting architecture overview, internal design of storage collection classes, control flow of web pages for basic user actions, synchronization of the storage collection classes, as well as authentication and authorization. Furthermore, this document will cover design decisions (good and bad), control flow, testing, and lessons learned. In the end, this document will serve as yet another technical brochure to the overall system design, as well as, the decision making to get to the current system as it stands today.

Overall System Architecture:

The overall architecture is very similar from homework three assignment, however, there are a few more things to consider. First, we now have to think about authentication and authorization. Second, storage of game and user state. Lastly, how data is transferred from one servlet/JSP to another while keeping in apparently consistent. Illustration 1 below demonstrates my architecture at a very high level view, and addresses all of the requirements I just mentioned. As one can see the first four operations in the simple process is exactly like the previous

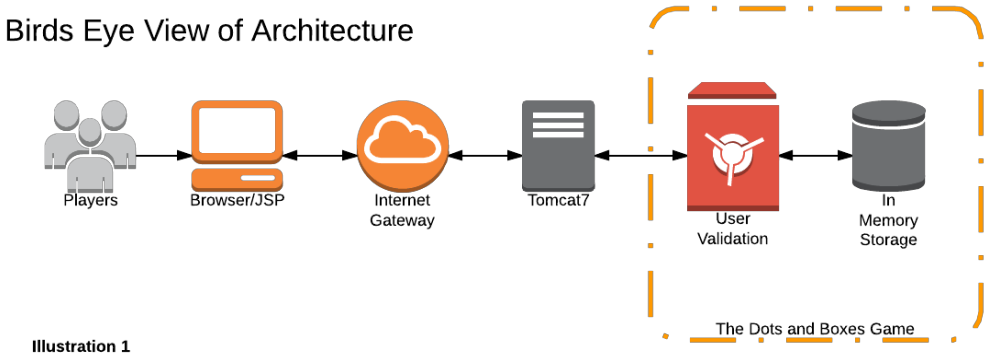

Illustration 1

assignment, however, our servlets are doing much more processing than before. As seen from the diagram the “User Validation” vault is the internal validation done for users who are 1. logged in, or 2. who need to register for an account. This simple validation/registration is done on both the servlet level and the JSP level. Also, included in the diagram is our storage facility (labeled

as In Memory Storage), which handles storing both Users and Games, since we need to obtain this information a lot. Since the diagram is a birds eye view of the overall system I will now go into the details of such a system.

Web Page Control Flow

Control flow was something that I mapped out from the start. There are a lot of combinations on where the user can go once logged in, so I mapped the flow of some of the routes which helped immensely. Just to make note, I only created flow maps for what are called “happy paths,” which are paths that don't result in error. For the first route, I knew I needed to offer a game, so I could test accepting it later. When making these maps I combined the project assignment documentation, and started visualizing the route. Using these two resources I came up following illustration shown below.

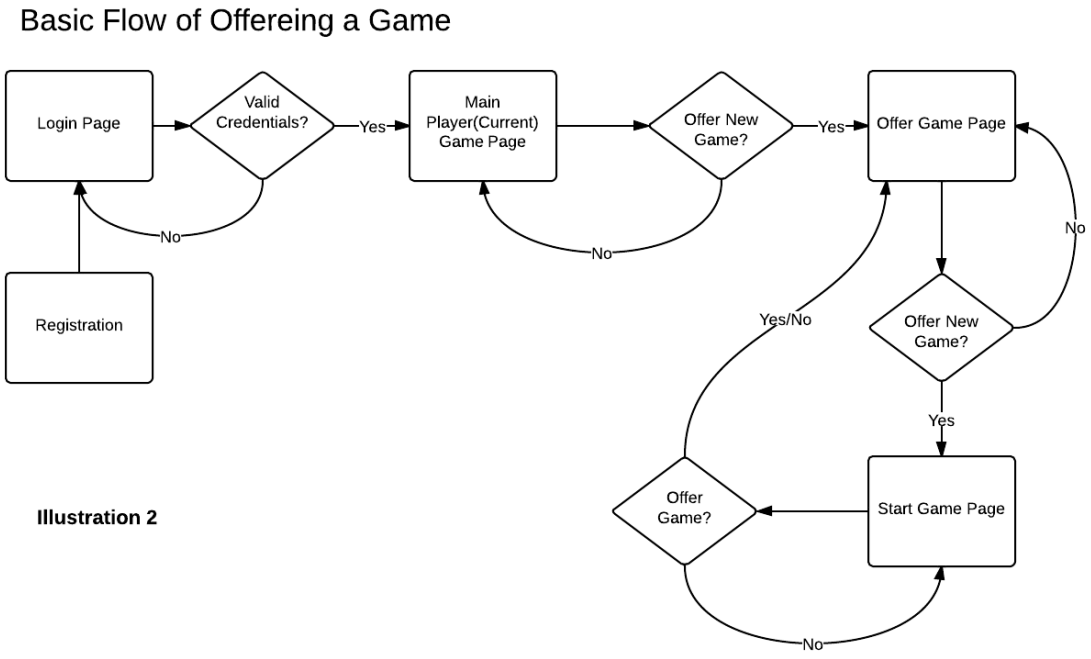

Illustration 2

As one can see, the flow is pretty easy to follow.

1. If a user has not registered before then the application will direct him to the login page with an option to register.
2. Once validated the user is displayed the MainPlayer page, or what is just current games page. Here a player can offer a new game which takes him/her to the OfferGame page.
3. The OfferGame page actually doesn't contain the functionality for offering a game. It just is used to display games that are already offered, and to let a player offer a new game. The functionality for starting a game is in the StartGame page.

4. Once in the StartGame page a player can either start a new game to offer or go back to the OfferGame page. If a new game is created then offered then this returns us back to the OfferGame page where our new game is now displayed as an offered game.

That concludes the happy path for offering a new game, however, another path that is important is the acceptance of an offered game. This is important to note because it is the next large part of functionality. The flow of control for a player logging in and accepting a game is illustrated below.

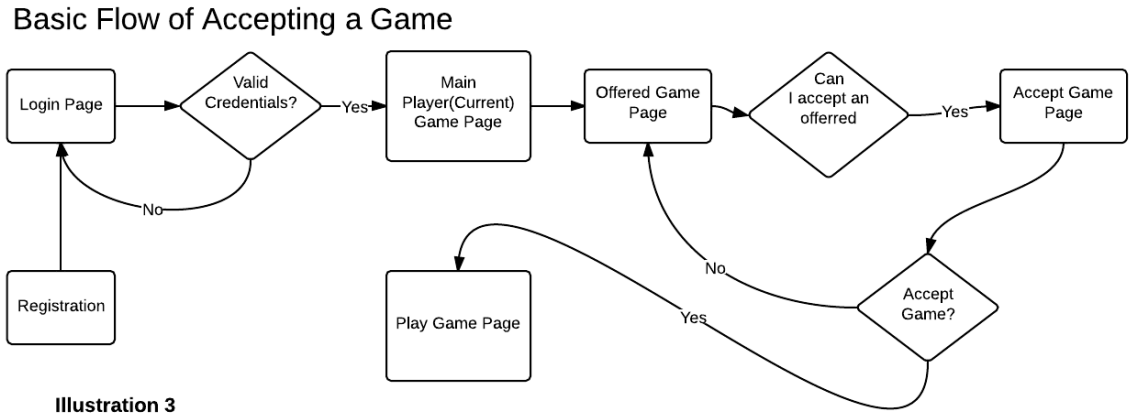

Illustration 3

Since I described the login and registration process above I will skip it here and start right from the MainPlayer page.

1. From the MainPlayer page we can enter directly to the OfferedGame page.
2. Once on the OfferedGame page where all the offered games are listed a player can choose which offered game he/she wants to accept/play which takes the player to the AcceptGame page
3. On the AcceptGame page the player will have the ability to accept the offered game in either a Normal or Reversed fashion.
4. If the game is accepted the player is then transferred to the OfferedGame page, and if he doesn't accept the player is navigated back to the OfferedGame page.

From the beginning of designing this application I knew that these flows of pages were going to be the bulk of the work, and if I got this done I would be in good hands for finishing the latter part of this project which happens to be error handling, user message delivery, and user authentication/authorization. Going through the overview will help describe the details of each page and the internal system structure which will be discussed in the following section.

Application Internals

When I think of application internals I think of the guts or core of the system. The internals are the ‘what’ and ‘how’. They are components that when put together make the system as a whole work seamlessly. For this application I would divide some of these parts into categories such as, authentication/authorization, synchronization of Game and User objects, application storage, session storage, and user interface.

Authentication/Authorization

The path to become a valid user is that the player must register. After filling out the required fields and submitting a registration form the player can now login and access the system. Upon login the user is stored in the session object. This object is persisted across the pages or until the a session timeout which is 30 minute for my application. With this session object set like so:

```
session.setAttribute("loggedInUser", username);
```

We can now just reference it within each jsp by checking if the logged in user is the same as the one stored in the session object. Illustration 4 below demonstrates just this:

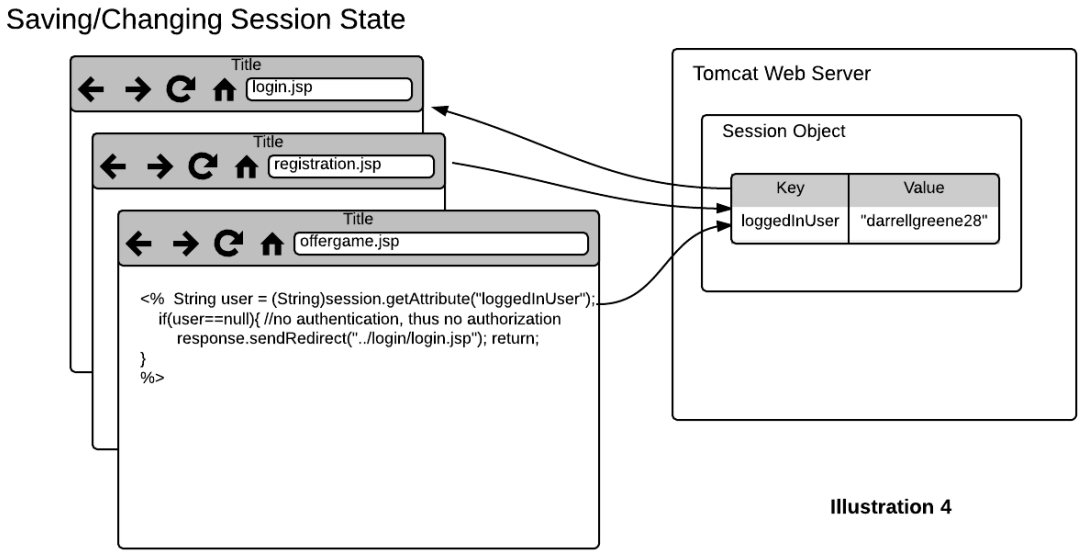

Illustration 4

Every page of this application has a check to make sure the logged in user is allowed or ‘authorized’ to view/access the page. As seen from the listing above the code snippet inside the browser window is exactly what's on top of every JSP page except for the login and registration because they do not need to know/care who’s logged in. The arrows in the diagram show that

both the server and the JSPs use the session object, since the JSPs are just compiled down into servlets that run on the server anyways, however, this is all behind the scenes of the user.

Session State Storage

Luckily this is a simple task that the web container/server handles for us. Sessions are created at first request to the server for a resource. The server populates this object with initial values like a session ID which is unique to each session. The session object is then looked up by this session ID and this will be unique for each session. One issue here is when testing to make sure the session state is what you expect because the session is cached in a cookie within the browser, so making sure two session act 'friendly' together can prove to be a difficult task. Within this project we were only really concerned with one value within the session object, which happens to be the logged in user described in the Authentication/Authorization section above. Since we handle messaging the the user interface we dont need to worry about setting a property for that.

Application State Storage

Much like the data structure for the session object the application object is also a key value pair. In this project the application object is the most used object, and contains/stores a lot of the information the game needs. Without the application object, there would be no way to persist game data amongst all the sessions. Actually, one delineation of the application and session object is that session objects are only tied to the browser session whereas the application object is more like a global object that is shared amongst all sessions. This is both a good thing and a bad thing. One reason its a good thing is because you have a reference to an object that is shared amongst all other users, which means you have a shared storage to pass data through. One reason why this is a bad thing is because this object can be mishandled and now become a storage for confusion. This means that since all session are accessing this one object we must be careful on when and how we use it and what we store in it. The two main objects we will be accessing within the application object are actually wrapped java Collections also known as the Game and User Collection objects.

Game Collection

This collection has a HashMap data structure, and its key is a Game ID and its value is a Game object. This game object what makes the application work, since it contains/stores all the information on the server with the ability to retrieve it easily. The Illustration below is a demonstration of what is described above.

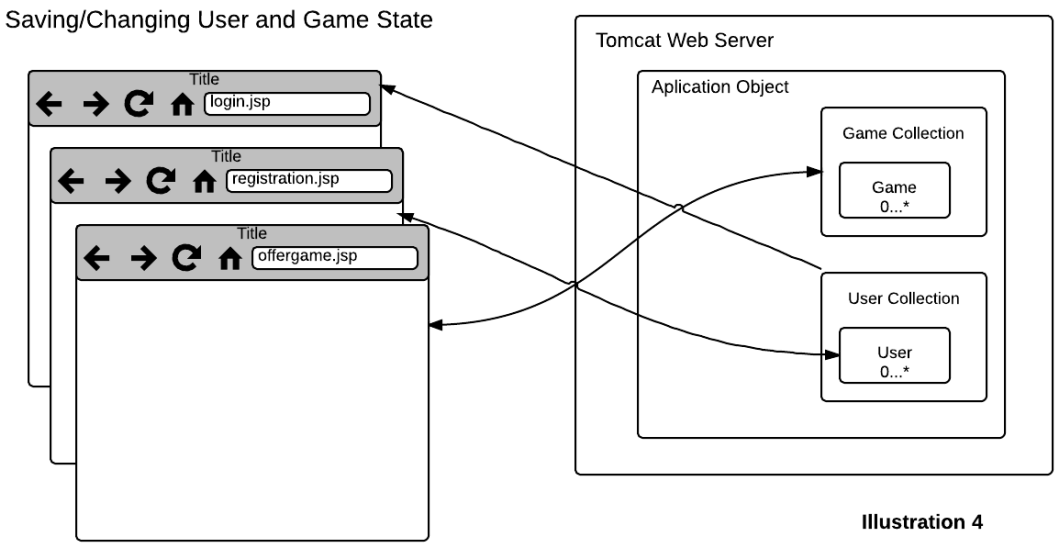

Illustration 4

While this Game Collection object is used throughout the almost whole application, with exception of login and registration, I only drew the three above JSPs. I just wanted to note that the GameCollection isn't just limited to these three JSPs. Now, there are a few more things to note here for the Game Collection. As one can see, and as described above, the GameCollection holds all the games that are both in offered and accepted state. This collection has a HashMap data structure, so it acts as a key/value store. For my design, I was hovering around the decision to use a TreeMap for the mere reason that it is already sorted by design, however, I decided with a HashMap just because I know this data structure well and knew it would work. Another thing to note about the GameCollection is that it is just a wrapper class where there are a few helper methods. The internals are not too complex and this next section I will briefly go over them.

GameCollection Internals

As stated above, this class is just a wrapper class for the actual HashMap that contains the games. This is done by it having a private member variable that is accessed through an synchronized accessor method that helps it from being accessed by multiple threads at one time. This is demonstrated by the code snippet below:

```
private final Map<Integer, Game> gameMap;

    public GameCollection(Game game){
        this.gameMap = new HashMap<>();
        this.gameMap.put(game.getGameId(), game);
}
```

```
public synchronized Map<Integer, Game> getGameMap(){
    return this.gameMap;
}
```

The synchronized keyword associated with the method signature provides the chance of thread leak and solidifies concurrency for the application. Another important piece of information to note here is that the constructor, highlighted in yellow, takes a Game object and initializes the map with that Game object. As the developer/designer I needed to make sure I only instantiated this GameCollection object once and only once, for I would be dealing with all sorts of missing/new games if I had GameCollection object overwriting the other in the application object. When referring to GameCollection Internals one cannot forget about the Game Object, however, I believe this needs its own section to explain its internals.

Game Object

Game objects are really the most used objects within my dots and boxes game. They contain all the logic to manipulate the game board as well as keep information about the game. For example, one game object has the ability to start a game, keep who offered the game, keep track of whose turn it is, know who accepted the game, and much more. Now, the Game is accessed all over the application, and because of this access this can be a difficult object to keep thread safe because of how immensely it's created, read, updated, and deleted. To keep this object concurrent we need to make sure its synchronized well. First, I will show a diagram that paints a picture of the synchronization that was done to accomplish thread safety. Then I will explain what the picture depicts, as there is a lot of information within it.

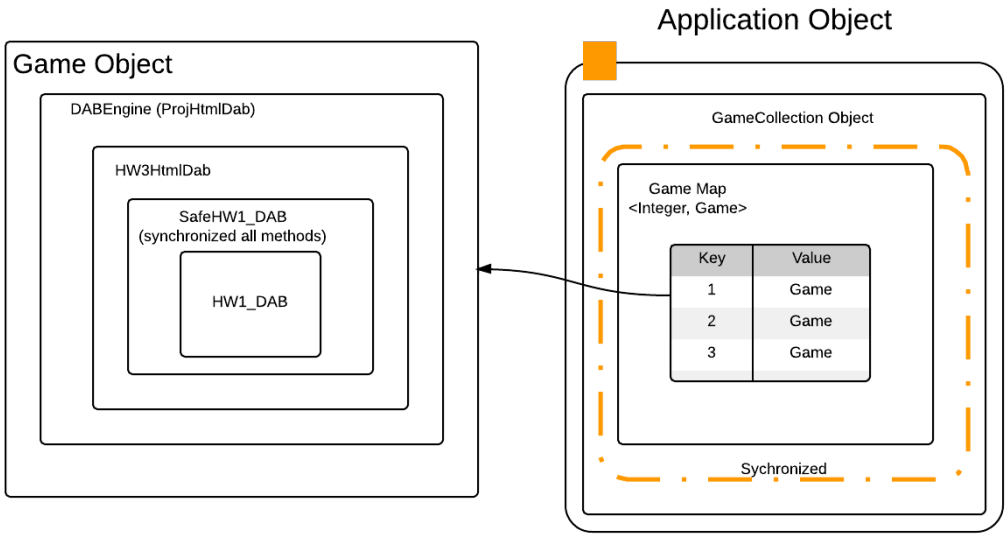

Illustration 3

As one can see, the GameCollection object that I described above stores all the Game objects within its member variable's Map data structure named 'gameMap.' This gameMap variable itself is accessed only through its synchronized accessor method. The reason for this is because we only want one thread to access this gameMap at a time because we don't want any concurrent modification on this map. One will also notice that the Game object is actually the DABEngine (the interface to the HW1_DAB object where all the logic to control the behaviour of the game itself as well as other game relevant information). To make sure the only one state changing method is fired one at a time we must find someway to synchronize this DABEngine instance or its underlying implementation. I accomplished such a task by wrapping the implementation in a SafeHW1_DAB instance that implements the DABEngine. Within the SafeHW1_DAB I synchronized every method that the DABEngine exposes. The code snippet below demonstrates just this.

```
public class SafeHW1_DAB implements DABEngine{

    private DABEngine wrappedDAB;

    public SafeHW1_DAB(DABEngine unsafeDAB){
        wrappedDAB = unsafeDAB;
    }

    @Override
    public synchronized void init(int i) {
        wrappedDAB.init(i);
    }

    @Override
    public synchronized int getSize() {
        return wrappedDAB.getSize();
    }

    @Override
    public  synchronized Set<Edge> getEdgesAt(int i, int i1) {
        return wrappedDAB.getEdgesAt(i, i1);
    }
  ...
}
```

Using this method of synchronizing every method ensures we have a thread safe object which we can access freely without the need to worry about thread safety. Another structure that also shares similarities to the GameCollection/Game objects are the UserCollection/User objects.

UserCollection

The user collection is really a subset of the GameCollection object except it only has one accessor method. Since I didnt want one user object to be modified by two threads at one time the accessor method to the userMap member variable (another Map data structure to hold User objects) I made this method synchronized. To sum up, the UserCollection object is really just around 30 lines of code, which makes it very simple to explain.

User Object

The User object contains all relevant information about the user. For example, the user's password, username, and phone number. I would say the major difference between the Game object and the User object is that the User object does not need to be synchronized. This is mainly because we really only care about the UserCollection allowing one thread at a time to access/change one user at a time, and since no object within the user is changing state of the application I didn't need to worry about making it synchronized.

Testing

For testing this application I did follow the normal change then retest procedure, however, I included no test suite/JUnit tests. Basically, I implemented a functionality then ran through a tedious step-by-step process to see if my change/fix solved the need of the functionality being tested. This was very time consuming, and wish I had time to set up end-to-end tests using some kind of framework like Selenium or Mockito to assist in testing front-end as well as backend functionality.

Lessons Learned

I believe the following long list counts for lessons learned, because I learned a lot from things I had to fix after knowing what I did wrong. I will walk through the list as it it very long by page/servlet that I learned my lessons from.

Login/Registration

1. forgot to wrap usermap in its own collection and add synchronized methods for private member vars.
2. coded partial authentication but commented it out because of professor's continuous reminder to wait till project core functionality is done as to make sure we didn't need to register users everytime we modified the servlet.
3. started with doing regular strings from session object but watched class on week eight and saw forum on displaying messages to JSP using a status code instead.

Start Game and OfferGame Pages/Servlets

1. Synchronization is a tedious don't really understand when and where to include it, but with help from forum and class I learned where it needed to go and implemented it.
2. Messaging on all JSPs should be just like the Registration page
    a. I realized this after the fact that I logged into a second account and found that the old session message attribute was old and for the other user.
    b. switched to using a status code being sent from the servlet on the redirect url
3. On StartGame page I decided to assume that if the size was being sent on query string that I would initialize the board and game with that size rather than check for the init command, and used this as a default command.
4. Created a getNext private method that runs through the gameMap and returns the max key that is in the collection then I add one to get the next available game id
5. synchronized every method within the GameCollection as my design choice instead of performing
    a. using the `getNextGameId` synchronized method which is used to obtain the next id in the map, so I could grab a unique number to store game.
6. To get offered games, I first I grabbed all games within collection then soon realized that i just needed the offered ones, basically where the offerer is equal to null in each game.

Main Player Page

1. Needed another subset of the collection so added another synchronized method in the GameCollection class to return a hashmap where all the game's acceptor is the logged in user. This should give me back all games accepted by the currently logged in user.

PlayGame Page/Servlets

1. My whosturn method is coming in as a good decision as i can display the clickable/non clickable grid with a single method call to whosTurn.
2. Adding messages to deal with exceptions was more difficult that I imagined because there are a lot of moving parts within the application let alone the playing game page.

Extra Credit References

1. Updated the UI to include a more visual user experience.