## 

## 

## 

## 

## 

## 

## 

## Large scale seeking game using android phones

Group # 4

Colby Ansel Horn - cahorn - cahorn@uwaterloo.ca

Prerna Angrish - pangrish - pangrish@uwaterloo.ca

Yaron Yaacov Milwid - yy2milwi - yaron.milwid@uwaterloo.ca

Haoyuan Zhang - h524zhan - h524zhang@uwaterloo.ca

Chen Xu - c222xu - c222xu@uwaterloo.ca

# Architectural description

## **Overview**

The system has a heterogenous architecture which is comprised of (1) a client-server architecture, (2) a three-tiered architecture, and (3) a blackboard architecture. The client-server and three-tiered view of the architecture can be seen color coded in figure 4.

In the client-server model, the client is the device on which the application is run, and the server is the Firebase Database, which provides data synchronization over all the clients and data storage in the cloud. The server acts as the medium of interaction between all the clients and only stores data. For example, the location of the target client is uploaded to the server, and is requested by the seeker client, which in turn is displayed as “hints” to help find the target. Firebase utilizes an event-based mechanism to notify connected devices whenever the data is changed. The client stores all the code that is needed for data synchronization and utilizes a blackboard locally within the client. The blackboard interface acts as a sole communication mechanism between the various components of the client. The client components, consist of: a game logic component, a firebase communication component and a graphical user interface component, each of which are implemented with multiple classes.

The three-tiered model contains a data tier, a logic tier, and a presentation tier. The data tier consists of the Firebase database and the logic for communicating with the database. The logic tier is the main game logic for our application and the graphical user interface forms the presentation tier.

## **The Blackboard**

The blackboard architectural style is a shared state architectural style. It is essentially a global data repository, which is accessed by all the individual classes, via a direct memory reference, a procedure call, or a database query. In our system, the blackboard is local to every client. All the components attached to it listen for changes to the blackboard data. Each of these components accesses or changes only some of the blackboard data, which makes it easier to determine various states of the system.

Having the blackboard as our main architectural artifact provides many benefits to the system. It formalizes the format of any data passed between components; and hence, all the components can be developed in parallel. From a structural and planning perspective, due to this formalization, the result of an operation can be isolated to one field of the blackboard, thus improving cohesion. This facilitates reasoning about program behavior. Each of the components of our system interact only via the blackboard, which reduces coupling, and as a result, also adds testability to the components. Due to this decoupling of communication, adjusting the application is much easier. For example, if we decide a new component needs access to certain data, or that a different component needs to be able to write certain data, that component accesses the blackboard, and implementing this change does not affect any other components.

## **Non-Functional Properties**

The system supports various non-functional properties. The primary ones being modularity, robustness, decoupling, cohesion, data consistency, efficiency and testability.

Modularity refers to the degree to which a system’s components may be separated and recombined. The blackboard architecture promotes modularity because each component can be independently developed and tested without breaking down the application, and as a result the system is also testable. Modularity is improved through decoupling components from each other and improving the cohesiveness within each component. Higher modularity often reduces the complexity of a system. Modules also add the ability to adapt to new changes, thus making the system evolvable. Our application is also able to handle any type of seeking game the user wishes to play.

Since, the blackboard is the sole means of interaction between the various modules, it makes it easier to respond to faulty run-time conditions, resulting in a more robust system.

Our application is scalable to many users since the component shared between users is the Firebase system. This system is designed to scale automatically to meet user demands. Additionally, our application is scalable to adding functionality as a result of the high modularity of our design.

The Firebase system guarantees eventual consistency. Despite the fact that consistency is not guaranteed at any given moment, the use of Firebase transactions ensures that race conditions do not cause inconsistencies between clients. This means that if a transaction is completed, it would show up on the database and if it is aborted at any time, the changes will be reattempted based on the new state of the database. This requires our system to have fewer transaction to ensure that we do not have too many collisions. We do this by having the clients interact with the server every 5 seconds.

Limiting the frequency of transactions to once every 5 seconds also improves the network usage efficiency of the system. Our system is also efficient in terms of power consumption and speed efficiency. Our Game Logic components ensure that all the sensors are turned off when the game is either paused or has ended. Additionally, since no updates are made to these values on the blackboard, no location updates are made to the Firebase server, thus improving network efficiency. The speed efficiency of our system is a result of having the code entirely on the clients and only storing data on the server. This means that all our function calls are local to the device, and hence, our application is much faster.

# 

# Architectural diagrams

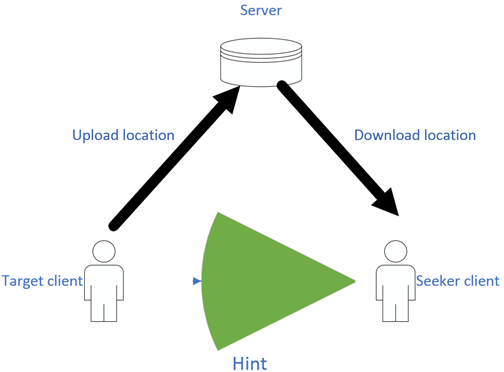


Figure 1: Infographic describing data transfer between clients

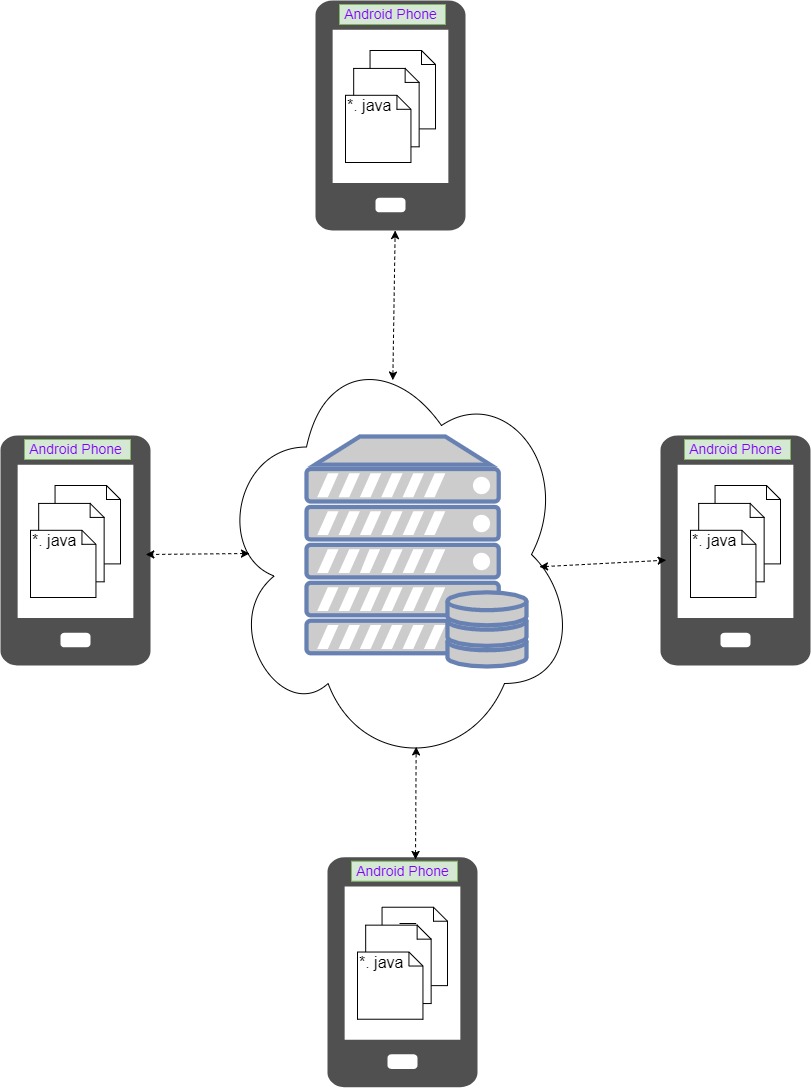


Figure2: Physical representation of the system

Android phones communicate with each other through the server.

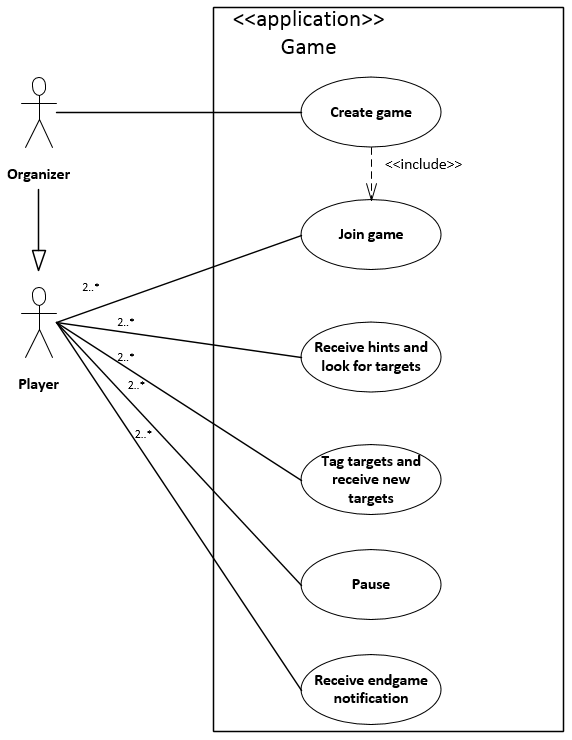


Figure 3: Use case diagram of the game

After the organizer creates the game, other players can choose to join the game. Their location and orientation information will then be sent to each other. They can also choose to pause at any time. If the time runs out, all users will receive a notification.

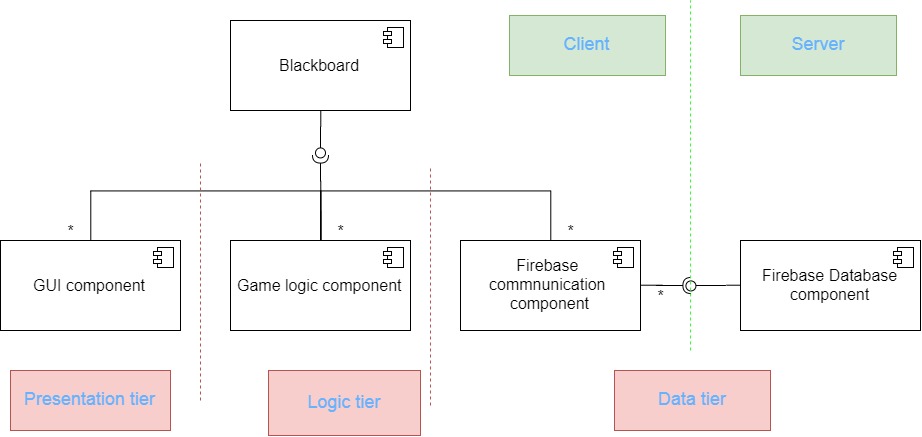


Figure 4: High level component diagram

Our design uses the observer pattern to share the data between components. When a component sets a value to a specific field, all other component listening on that field will receive a notification and run the appropriate method.

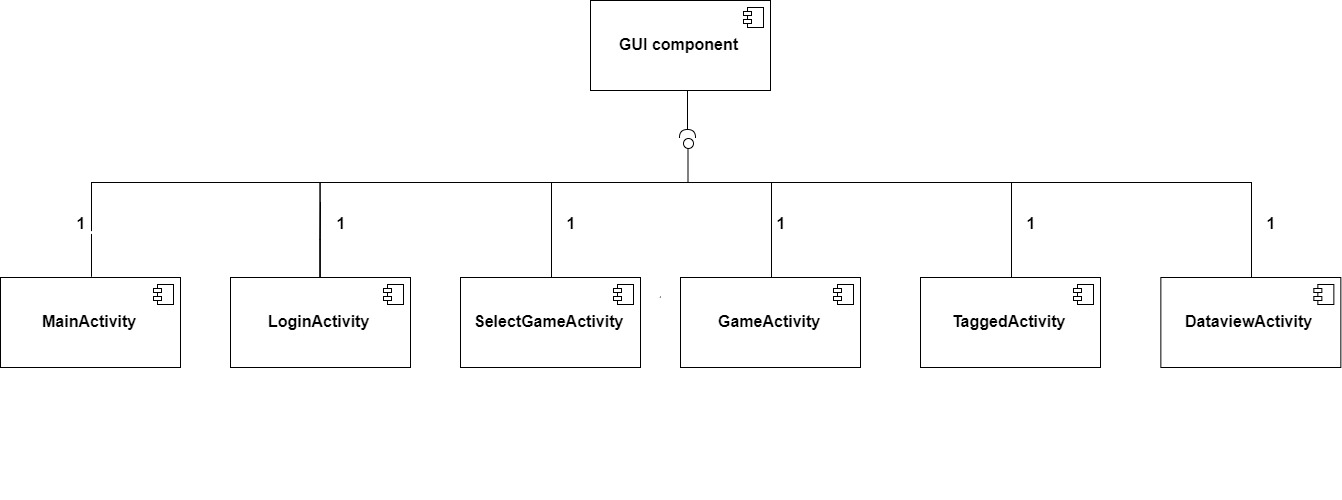


Figure 5: GUI component diagram

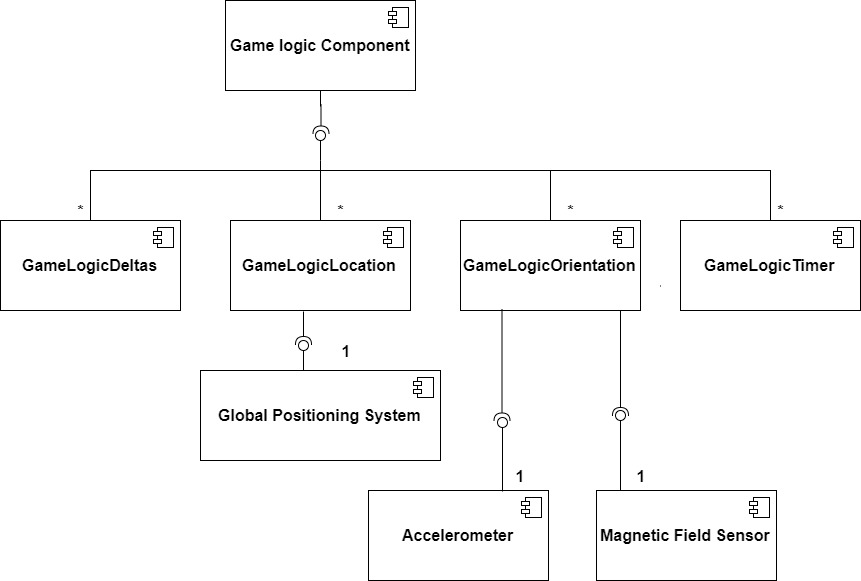


Figure 6: Game logic component diagram

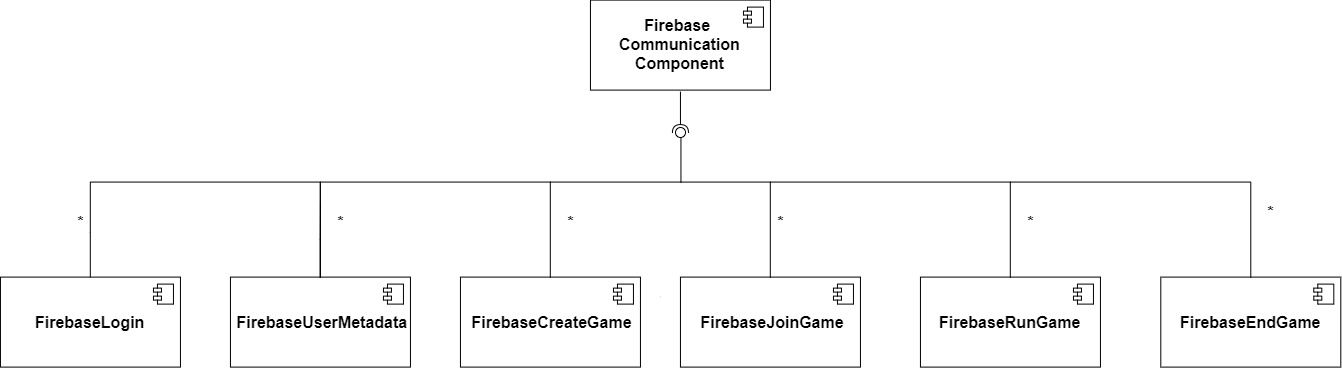


Figure 7: Firebase communication component diagram

# System design description

## **Public interface and class design**

The design utilizes two interfaces, one that is specific to the design and one that is a result of a prebuilt component used. The design uses the Google Firebase server as the database backend. This component has a very large interface, and is not germane to this document. The interface that is specific to this design is the Blackboard interface, found in Blackboard.java. This interface is made up of getters for various BlackboardData objects. Each Blackboard data object has a getter and a setter method, as well as a method to add an observer to the object. Upon calling the setter method, all observers are notified there has been an update to the BlackboardData object and must react accordingly.

The components in this application are separated into three tiers, as shown in figure 4 in red. Each component consists of multiple classes, as shown in figures 10-13. For all classes, the class-name is made up of the responsibility (either state, or action), and the tier. Presentation tier classes have the suffix ‘Activity’, while other tiers are prepended by the tier. In cases where the responsibility spans multiple tiers, there are multiple classes, for example there are both a ‘FirebaseLogin’ class and a ‘LoginActivity’ class.

## **Reduction of coupling**

The primary non-functional properties considered when designing this product were increasing cohesion, and reducing coupling. Additionally, because of this decoupling, it was possible to minimize the number of classes active at any given point. Because of this decoupling, it is also relatively easy to change, or extend the functionality of the design. Finally, wherever possible, the dependency inversion principle was respected, with all variable declarations depending upon abstract classes.

Decreasing coupling was primarily achieved in three ways. The first method of decoupling classes was by using the blackboard architectural style. The centralized blackboard was used for all communication between objects running on the client. The second method of decreasing coupling was through the single responsibility principle [1]. The third and final method of decoupling classes was through using a state chart to describe the control flow of the application. The state chart provides guarantees as to the minimal condition of the system at the beginning of each state. During, each state, anything can change in the system, provided the conditions for that particular state are maintained. Classes can check this state to determine what actions they should perform.

As can be seen from figure 9, the blackboard was implemented using the Java observable class. Each blackboard data object can have multiple classes updating it, and multiple classes listening upon it. This means that (a) different classes may be active at any given point, and (b) when classes are changed in the system, they just need to ensure they adhere to the contract of the state diagram, and access the blackboard fields correctly, but do not need to change any other classes.

The single responsibility principle can be seen when comparing the different versions of the code-base. In the initial monolithic implementation, each non-GUI tier consisted of one class. This implementation was very difficult to debug and test. As a result, it was replaced by the current implementation, where each tier is composed of multiple classes. Many of the classes are controlled by the game state. The classes all listen to the game state, and upon changes, the classes either toggle on, or toggle off.

## **Comparison to alternative designs**

The application was designed in three stages. The first stage determined that since data needed to be exchanged between devices, and the physical locations of devices was unknown, a client-server model was necessary.

The second stage was to determine the client-side architecture. The chosen three-tiered architecture was compared to a model-view-controller architecture. It was determined that the strength of the model-view-controller was in handling multiple views. This application, has relatively few views; and therefore, the lower coupling of the three-tiered architecture was more important.

Once this was determined, the method of communication between tiers was discussed. A pipe-and-filter architecture was rejected since the data in each tier was fundamentally different. Publish-subscribe was rejected since it required too much coupling, additionally, it restricted which components could create which data. This reduces the ability to easily extend the design and change it in future. While the event-based architecture increased decoupling, it involved too much noise and meant classes would be notified of events that did not affect them. As such, it was determined that the blackboard was an acceptable compromise of decoupling and lack of noise. Classes could subscribe to only certain fields, while all classes could update any field, and classes could be removed without notifying or changing other classes.

## **Support for future changes**

The architecture of the design simplifies the process of changing the system. There are three types of changes possible, (1) removing functionality, (2) adding functionality and (3) changing existing functionality. Changes (1) and (2) are performed primarily through the blackboard, while change (3) requires changing the code itself. It is possible to have combinations of multiple types of changes, but for such combinations, the actions are just a combination of the actions described for each base-case.

To remove functionality, the first step is to determine which class contains that functionality. Due to the low coupling of the system, this determination can often be performed based on the name of the class. The assumption of a type (1) modification is that removing the functionality does not disrupt the state diagram. Based on this assumption, removal is trivial. The instantiation of this class is removed from the ‘BlackboardApplication’ class.

To add functionality, there are multiple decisions that must be made. The first is whether to add a new state to the state diagram, the second is whether more data needs to be passed via the blackboard, or whether the current blackboard structure exposes enough data, and the third is which tiers need new classes. Any new functionality should be added via one or more classes, and the classes should be separated based on tier. Once the classes have been implemented, these classes just need to be instantiated in the ‘BlackboardApplication’ class, and the classes listen on the blackboard for important events. The process is demonstrated in the below example.

A probable type (2) change is adding the ability to view the remaining time in the game. For this change, no new state is required, and a new blackboard field is required to store the current time. One new game logic class is required to retrieve the current time every five seconds and update this new field. Once this class has been created and instantiated in the ‘BlackboardApplication’ class, the ‘GameActivity’ class needs to be changed to listen on this value and display the time remaining.

To change functionality, it is important to first determine which classes contain the functionality to be changed. It is then important to determine whether this change will violate the state diagram, and if so, update the state diagram and any classes dependant on states whose contracts have been changed, to avoid errors. If the change does not violate the state diagram, the functionality can be changed. This process is demonstrated in the following example.

One type (3) change would be changing open-game creation in such a way that instead of the game being started upon a certain number of players joining, the game is begun after a certain amount of time. This change would require a new blackboard field to contain game start time. However, it also violates the contract which requires that to enter the ‘CREATING’ state, either the number of players, or the other players in the game must be set. This means that in addition to changing the ’SelectGameActivity’ class, and adding the functionality to both the ‘FirebaseCreateGame’ and the ‘FirebaseJoinGame’ classes, these classes must be changed to not raise errors if no data about other users is provided.

These three examples demonstrate the types of changes possible; however, in general, changes will be heterogeneous. To make any changes, the state diagram and blackboard should generally be consulted to determine the effects of the change, and how to implement the change.

# 

# System design diagrams

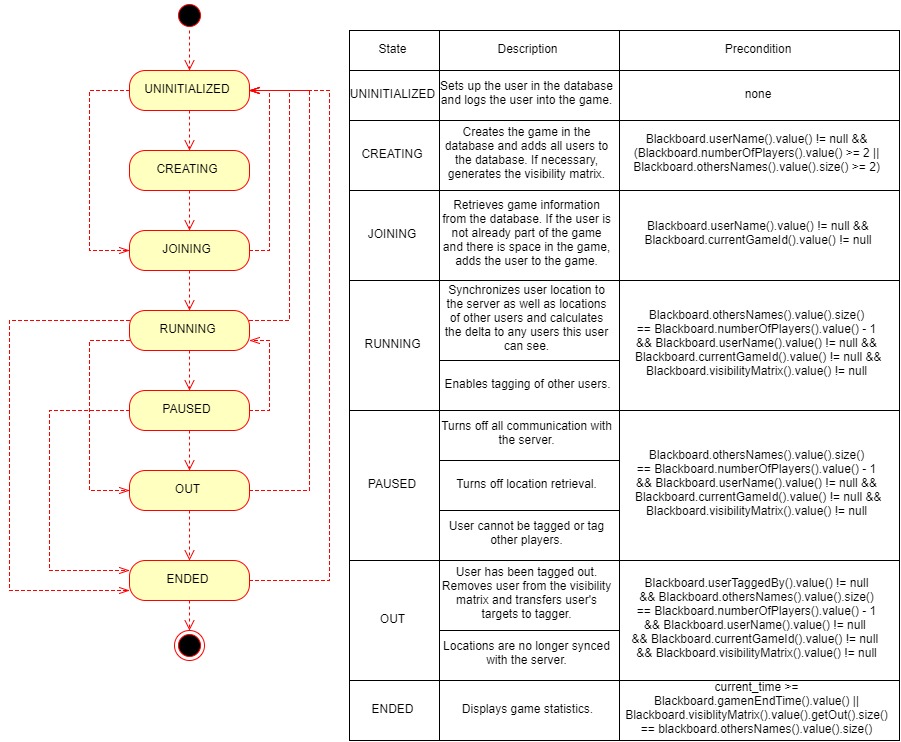


Figure 8 State diagram demonstrating system control flow

# 

Figure 9 Class diagram of the blackboard component

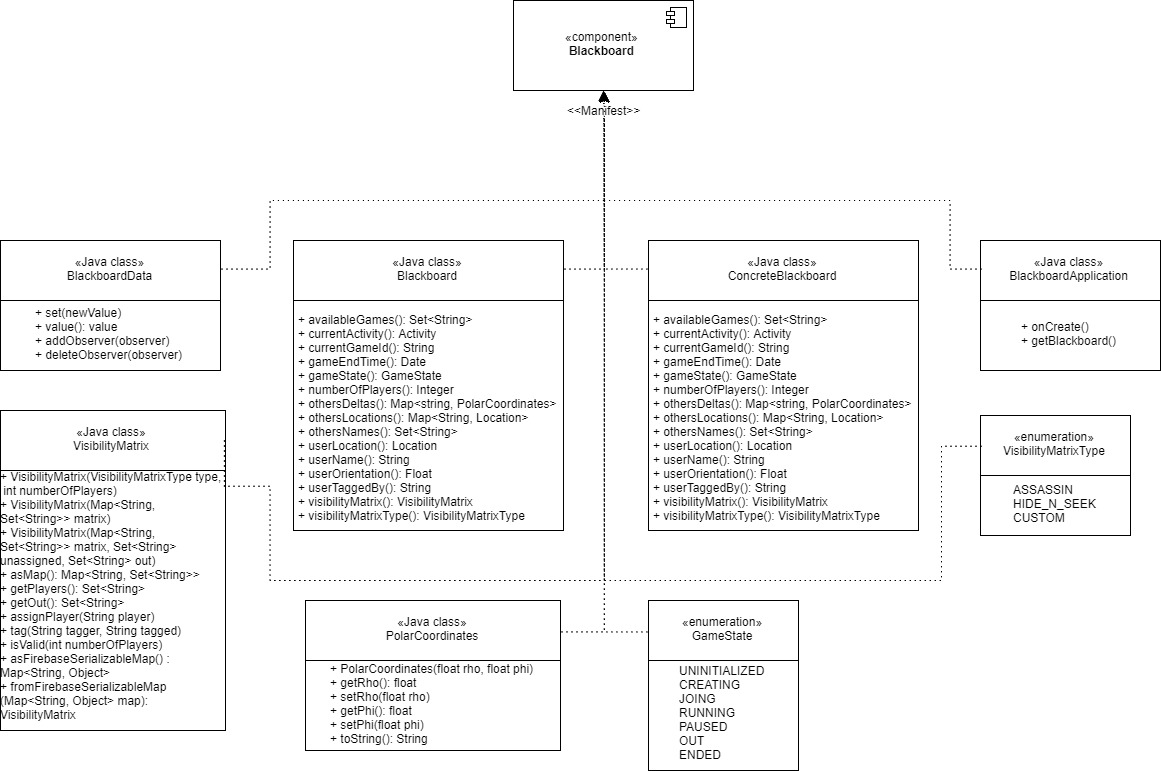


Figure 10 Manifest diagram of the blackboard component

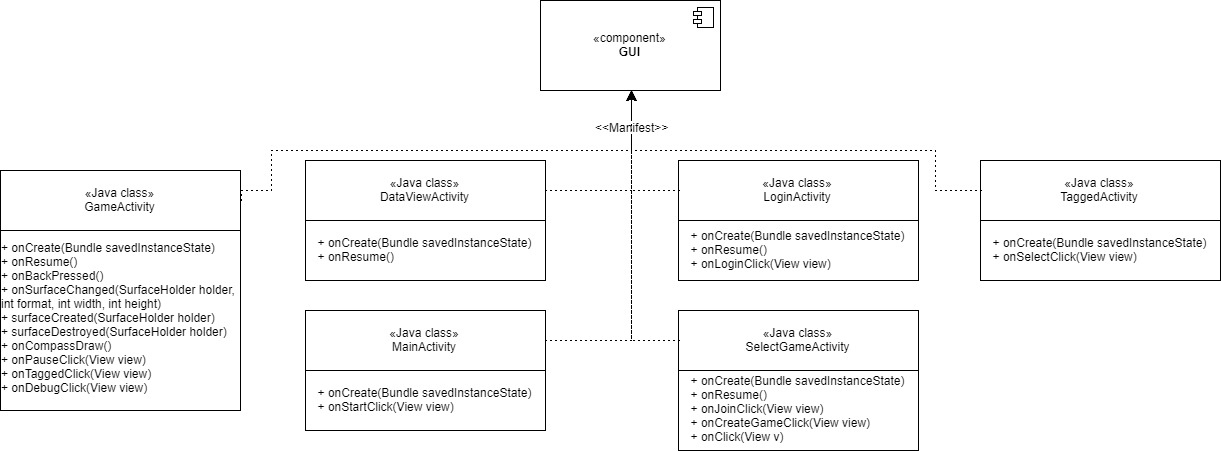


Figure 11 Manifest diagram of the GUI components

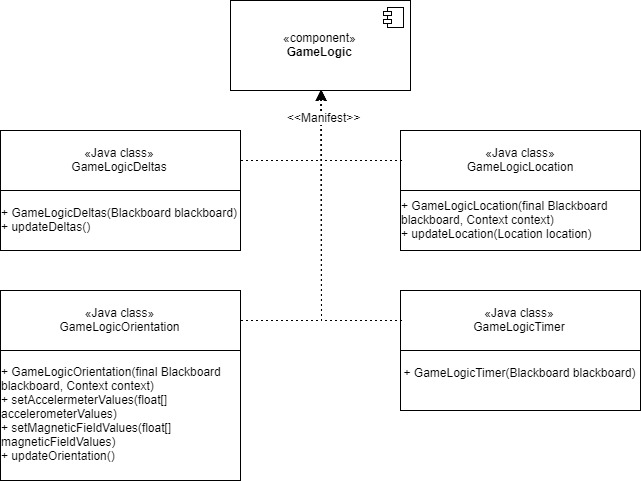


Figure 12 Manifest diagram of the game logic components

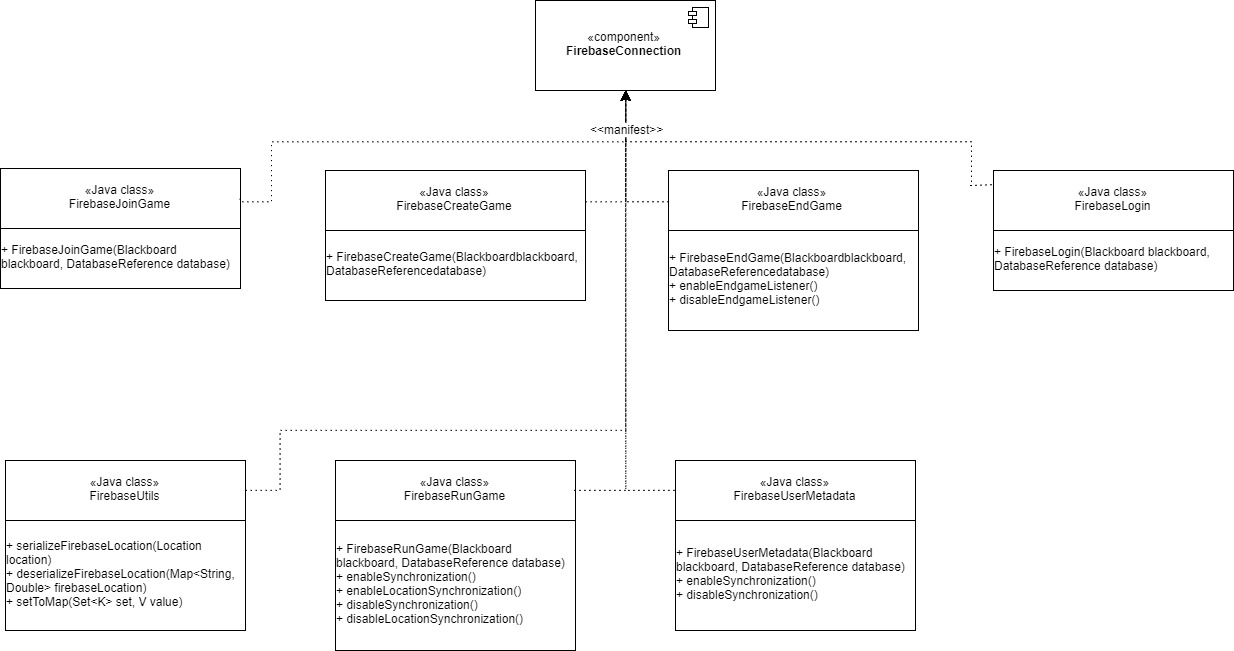


Figure 13 Manifest diagram of the Firebase connection components

# Participation Journal

Table 1 displays the responsibilities of of each group member. In addition to the specific responsibilities in the table, there were group meetings on a weekly basis, with multiple meetings some weeks. All group members attended these meetings, and they were used for planning the architectural design, as well as short term goals. Towards the end of the semester, some group members continued developing the application, while other group members worked on individual deliverables.

|  |  |
| --- | --- |
| Group Member | Responsibilities |
| Chen (Terry) Xu | * Game logic components * Architectural diagrams |
| Colby (Ansel) Horn | * Blackboard * Second version of Firebase communication components * Helped with GUI * Helped with Game Logic * Testing/integration * Proposal:   + Functional properties   + Use case 1   + Non-functional properties   + Possible extensions |
| Haoyuan Zhang | * Game logic components * Deliverable 3 presentation * System design diagrams |
| Prerna Angrish | * GUI components * Proposal:   + Graphical depictions of application * Architectural description |
| Yaron Milwid | * Initial version of Firebase communication components * Status reports * Deliverable 3 presentation * Proposal:   + Introduction   + Use case 2 * System design description |

Table 1: Participation Journal

# **References**

|  |  |
| --- | --- |
| [1] | R. C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2003, pp. 95-98. |
| [2] | W. Durand, "From STUPID TO SOLID Code," 30 July 2013. [Online]. Available: http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/#dependency-inversion-principle. [Accessed 27 November 2017]. |