Final Project Report

Crash Starring Sandra Bullock

Team Eleven

Kelsey Aten

Adam Nastasia

December 13, 2015

Table Of Contents:

Introduction: Project Definition and Overview

This project served as a summary of this class by applying OO design principles to a tangible software development project. Through applying these OO design principles we enhanced our general software development process and familiarized ourselves with the Android platform.

In particular, multiple GoF patterns were implemented in order to produce a working and efficient tank game. The tank game consists of a two-dimensional field containing an assortment of various entities; these include tanks, walls, and bullets. The game architecture is broken into two distinct parts, the client side, and the server side. Our task was to implement the client side. The client side's responsibility is to display the game field based on information received from the server, as well as to provide a control interface.

The server side's responsibility is to manage the battlefield as well as enforce some of the rules. The remaining rules are delegated to the client side. These rules take the form of various constraint checks. While the battlefield does handle overall game rules it falls to the client to monitor and enforce the frequency and correctness of commands sent to the server. An example of this is a tank facing a particular direction. This tank cannot move perpendicular to the direction in which it is facing. However, on the server side it is possible to produce that result. Therefore, the client must check these conditions before sending a move command.

Software Requirements:

Functional Requirements

A functional requirement can be defined by specific system behaviors as well as functions requested by users. In our tank game, the overarching functional requirement is to produce a game that is playable by a user. Within this broader requirement lay multiple sub-functional requirements. These include, a visible updating playing field.

The user should also be able to view the status of the game at any point. The user should also be able to join the game with his or her tank. Another requirement is the user interaction portion. When a user presses the appropriate button, the game should react accordingly. An example of this is the user pressing the fire button. When the fire button is pressed, their tank should fire a bullet.

Non-functional Requirements

A non-functional requirement can be described by the overall qualities or attributes of a system. These criteria can be used to judge the criteria of the system. The non-functional requirements can be broken into different categories. The categories are as follows: accuracy, adaptability, compatibility, documentation, predictability, and response time.

Accuracy

All the information from the server must be parsed correctly. This includes extracting the appropriate entity type. When relevant, the tank id, life from various objects, as well as object location must also be correct.

Adaptability

The finished software must be able to easily support new entity types without a major design overhaul. This is made possible through various GoF pattern, such as the factory pattern.

Compatibility

The client program must interact with the server without any changes to the server code. The system must also be compatible with the most recent version of the android operating system.

Documentation

The source code shall be self-documented by placing the design description in a java doc header.

Predictability

When a user on the client side pressed a button its respective action must be executed and displayed appropriately.

Response Time

As long as an Internet connection is present, the system must query the server once every ten milliseconds. The user interface must be able to update at the same pace to reflect these changes.

Software Qualities

Our client side code must be robust enough to handle receiving packets of data every ten milliseconds. The parsing of this data, and the updating of the user interface should in no way slow the user experience down. The battlefield should match exactly the information received from the server. When the user interacts with the interface, the resulting commands sent to the server should reflect exactly what the user had inputted. The user interface should be intuitive, elegant and attractive.

Design specification:

Class Diagram

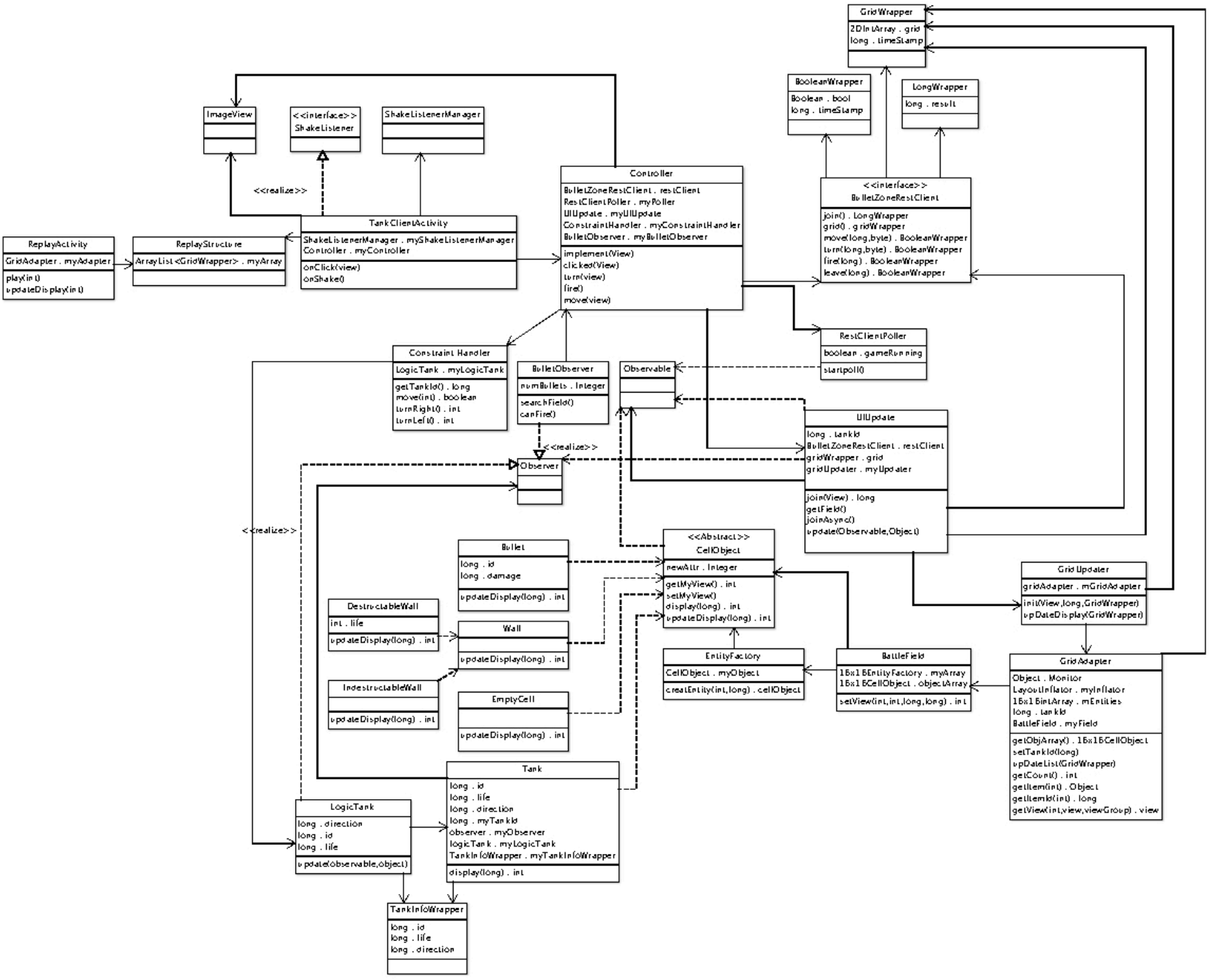

***Photo may be too small to be legible. Please reference the PDF in repository of necessary**

Class Diagram Explanation

At a broad scope, our game "starts" with the tank client activity. This activity contains all of the user interface elements. Such as, buttons and image views. In terms of logic, this class has been decoupled to only handle signaling when and what button has been clicked. What to do on those button clicks is delegated to the controller class.

The controller class contains instances of the bulletZoneRestClient, the restClientPoller, the UIUpdate class, the constraintHandler class, and the bulletObserver class. In terms of logic, the controller class calls turn, fire, or move through the restClient depending on the constraintHandler class.

The constraintHandler class observes the logicTank class. When the controller queries the constraint handler about moving or turning it returns a boolean stating whether or not that operation can be called.

The bulletObserver class observes the battlefield. It returns a boolean stating whether or not the tank can fire depending on how many of its own bullets are on the battlefield.

The restClientPoller class implemented the observable pattern and simply called update every ten milliseconds. The UIUpdate class, which is observing the restClientPoller, then deals with updating the user interface every time update is called.

The UIUpdate class can retrieve the battlefield. It can then delegate the image view updating to the gridUpdater class. The gridUpdater class contains a grid adapter, which can be set to the appropriate grid view.

The gridAdapter contains the battlefield and updates the grid view according to the battlefield it contains when it is applied to the grid view.

The battlefield contains an array of entity factories and an array of cell objects. As the grid adapter updates each image view in the grid view, the appropriate index in the battlefield creates a cell object through the factory pattern based on the information that is passed. The cell object is the parent class of bullets, wall, empty cells, and tanks.

Sequence Diagrams:

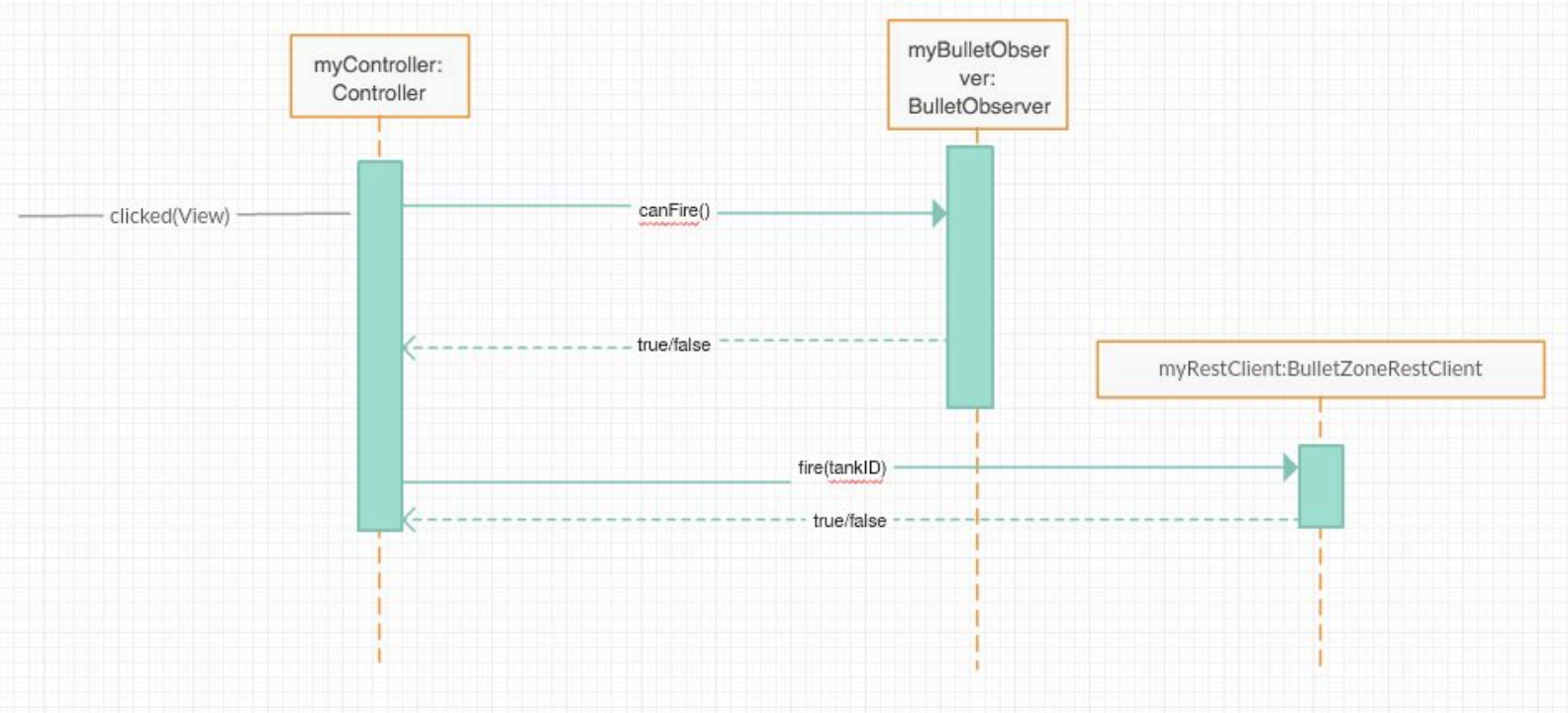

Figure 1: Firing the Bullet

The sequence diagram shows, at a basic level, the sequence of events happening when the tank controller registers that a view has been clicked. When the tank controller determines that the view that was clicked was the fire button, it queries the bulletObserver by calling the method canFire(). The bulletObserver then returns a true or false boolean value. In this case, the value is true and the controller calls the bulletZoneRestClient fire() method. This method also returns a true or false boolean value.

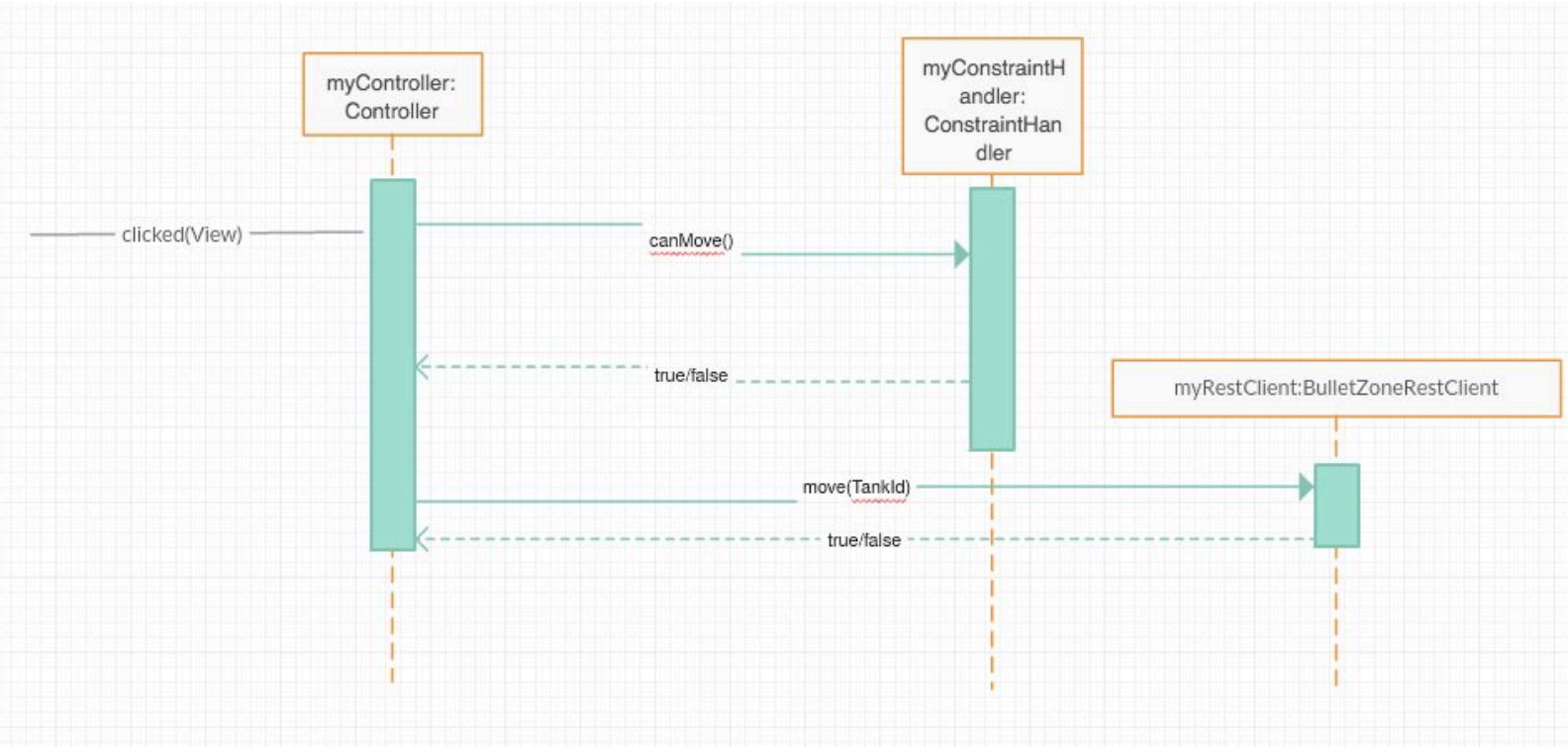

Figure 2: Moving the Tank

The sequence diagram shows, at a basic level, the sequence of events happening when the tank controller registers that a view has been clicked. When the tank controller determines that the view that was clicked was any of the move buttons: up, down, left, or right, it queries the bulletObserver by calling the method canMove(). The class, myConstraintHandler, then returns a true or false boolean value. In this case, the value is true and the controller calls the bulletZoneRestClient move() method. This method also returns a true or false boolean value. If the return value for the constraint checking is false, the controller first turns the tank in the appropriate direction and then calls move.

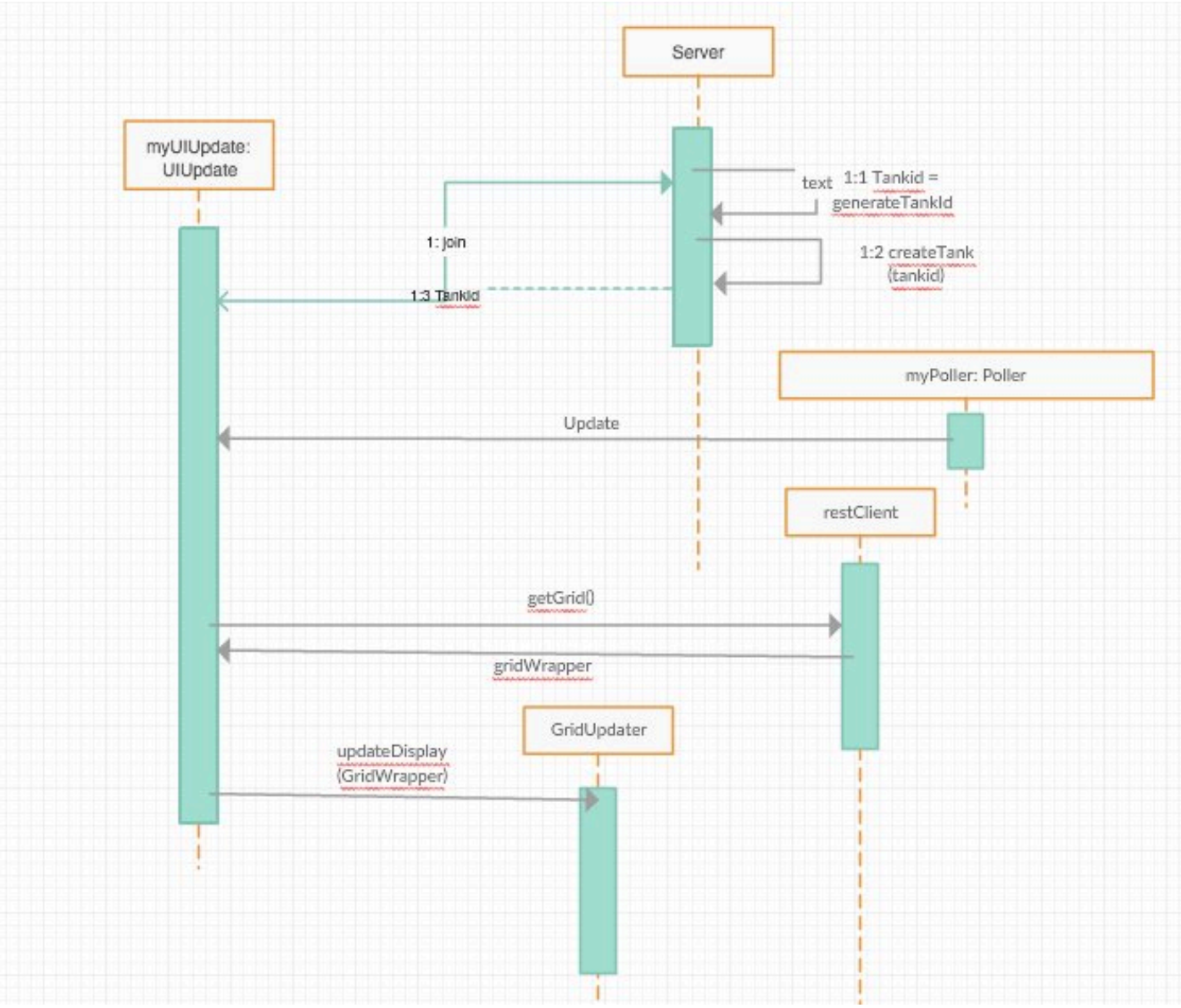

Figure 3: Displaying the Field

The sequence diagram shows, at a basic level, how the grid is retrieved from the server and then delegated to the gridUpdater to then update the user interface. The UIUpdate joins the server, which returns a unique tank identification number. After this happens, the poller is initialized and called update() once every ten milliseconds. When update() is called, the UIUpdate class requests a grid from the restClient, which returns a gridWrapper. Finally, the gridWrapper that was retrieved is then passed to the gridUpdater, which includes more sequences which are not shown in this simple diagram.

Design Patterns

This code snippet shows an implementation of the GoF factory pattern. The factory pattern decouples class instantiation. The factory class contains an instance of the cellObject class. Our cell objects that are used in practice all extend this cellObject class. When the factory class is called to create a cell object, it instantiates a particular cell object based on parameters passed into the create() method.

```
public class EntityFactory {

    CellObject myObject;

    public EntityFactory(){

    }
    public CellObject createEntity( int n ){

        if( n == 0 ){

            myObject = new EmptyCell();

        }else if( n == 1000 ){

            myObject = new IndestructableWall();

        }else if( n > 1000 && n < 2000 ){

            myObject = new DestructableWall(  n - 1000 );

        }else if( n > 2000000 && n < 3000000 ){

            myObject = new Bullet( n );

        }else if( n > 10000000 && n < 20000000){

            myObject = new Tank( n );
        }

        return myObject;
    }

}
```

Figure 4: Implementation of Factory Pattern

Database Design

We were unsuccessful in implementing our database. We sought an alternate solution in order to be able to produce the replay functionality. The following are code snippets from our unsuccessful database.

```
public class DBHelper extends SQLiteOpenHelper {

    public static String DATABASE_NAME = "myDatabase.db";
    public static String DB_TABLE_NAME = "gameStates";
    public static String DB_COLUMN_TIMESTAMP = "timestamp";
    public static String DB_COLUMN_GRID = "grids";

    public DBHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
        DATABASE_NAME = name;

    }


    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("create table gameStates" + "(timestamp INTEGER PRIMARY KEY, grids BLOB)");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        //db.execSQL("DROP TABLE IF EXISTS contacts");
        onCreate(db);
    }


}
```

Figure 5: DatabaseHelper Class

We implemented this databaseHelper class in order to interface with the SQLite database structure.

```
@Background
public void insertGrid( byte[] blob ){

    SQLiteDatabase db = myDb.getWritableDatabase();
    ContentValues cv = new ContentValues();
    cv.put( myDb.DB_COLUMN_TIMESTAMP, (String) null );
    cv.put(myDb.DB_COLUMN_GRID, blob);
    db.insert(myDb.DB_TABLE_NAME, null, cv);

}
```

Figure 6: insertGrid()Method

```
@Background
public void insertIntoDatabase( GridWrapper gs ){


    int[][]grid = gs.getGrid(); // get the grid from event
    try {
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        ObjectOutputStream objectStream = new ObjectOutputStream(stream);
        objectStream.writeObject(grid);
        byte[] blob = stream.toByteArray();

        insertGrid(blob );

    }catch( IOException e ){

    }

    //  getFromDatabase();

}
```

Figure 7: insertIntoDatabase() Method

```
@Background
public void getFromDatabase(){

    try {

        SQLiteDatabase db = myDb.getWritableDatabase();
        String query = "SELECT  * FROM " + myDb.DB_TABLE_NAME;
        Cursor cursor = db.rawQuery(query, null);
        byte[] blob = cursor.getBlob(1);

        ByteArrayInputStream bStream = new ByteArrayInputStream(blob);
        ObjectInputStream oStream = new ObjectInputStream(bStream);
        int[][] dSerialize = (int[][]) oStream.readObject();
        Log.d("HERE", "%d" + dSerialize[0][0]);

    }catch( Exception e ){

        Log.d("humm", e.toString());

    }


}
```

Figure 8: getFromDatabase() Method

Alternate Solution

In order to have a replay functionality, we had to have a way to pass a collection of saved states between activities. This collection had to be easily added to and easily iterated through. While not the best solution in comparison with the database, out solution allowed us to save and pass this vital information between activities and thus have a replay function.

We decided to create a simple class that includes an array list of wrapper objects. We can easily add wrapper to this which mimics saving to a database. We can also extract wrappers at a certain index from an array list.

```
@EBean
public class ReplayStructure {

    private ArrayList<GridWrapper> myWrappers;

    public ReplayStructure() { myWrappers = new ArrayList<GridWrapper>(); }

    public void insert( GridWrapper g ){

        myWrappers.add( g );

    }
    public GridWrapper getIndex( int i ) { return myWrappers.get( i ); }

}
```

Figure 9: Replay Structure

Additional Features

Some additional features we implemented include:

- Our game grid will resize to fit any size screen
- Game includes a spectator mode which allows the user to continue watching the game after they have been eliminated

User Documentation:

Screenshots

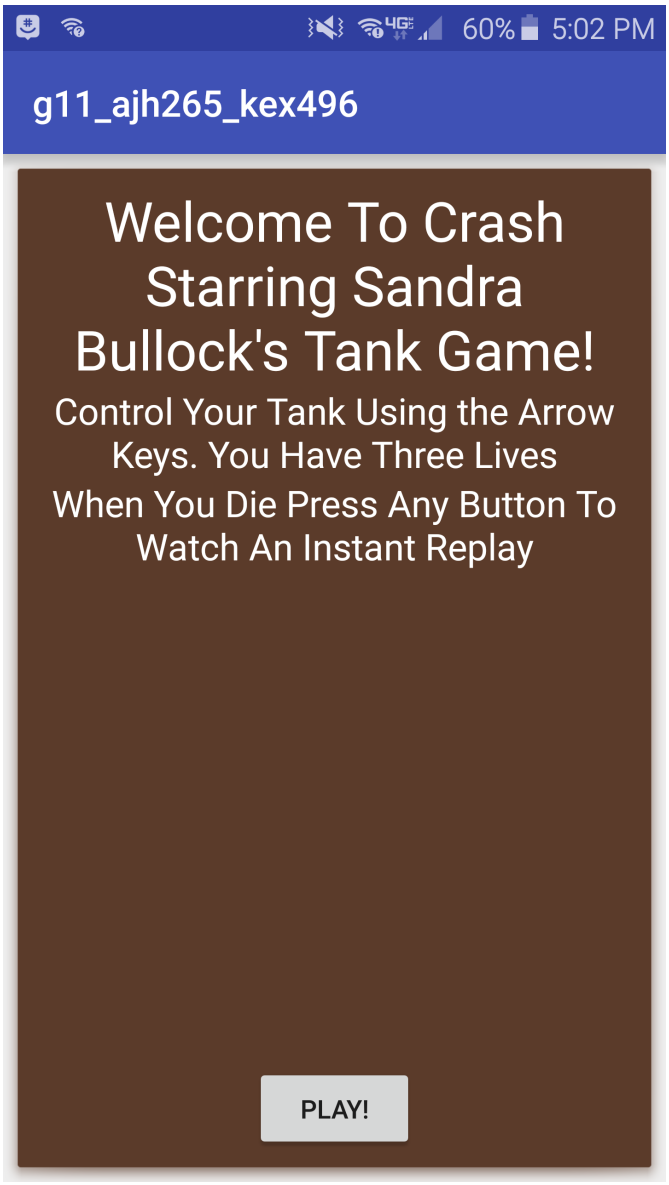

Figure 10: Splash Screen

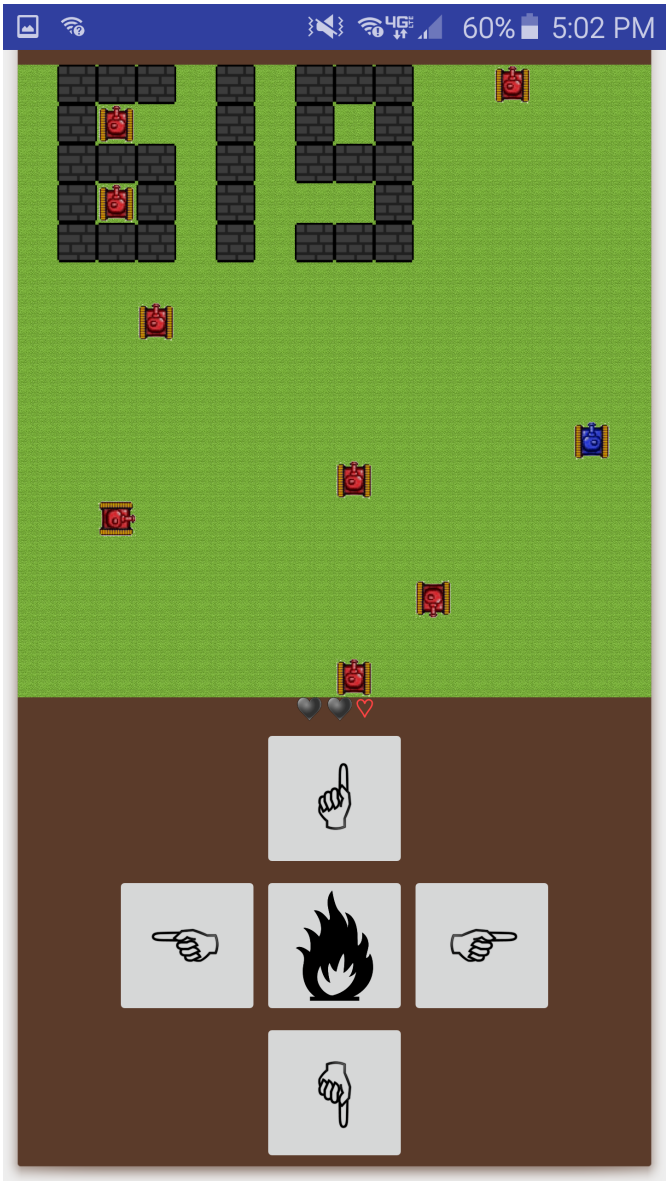

Figure 11: Game Screen

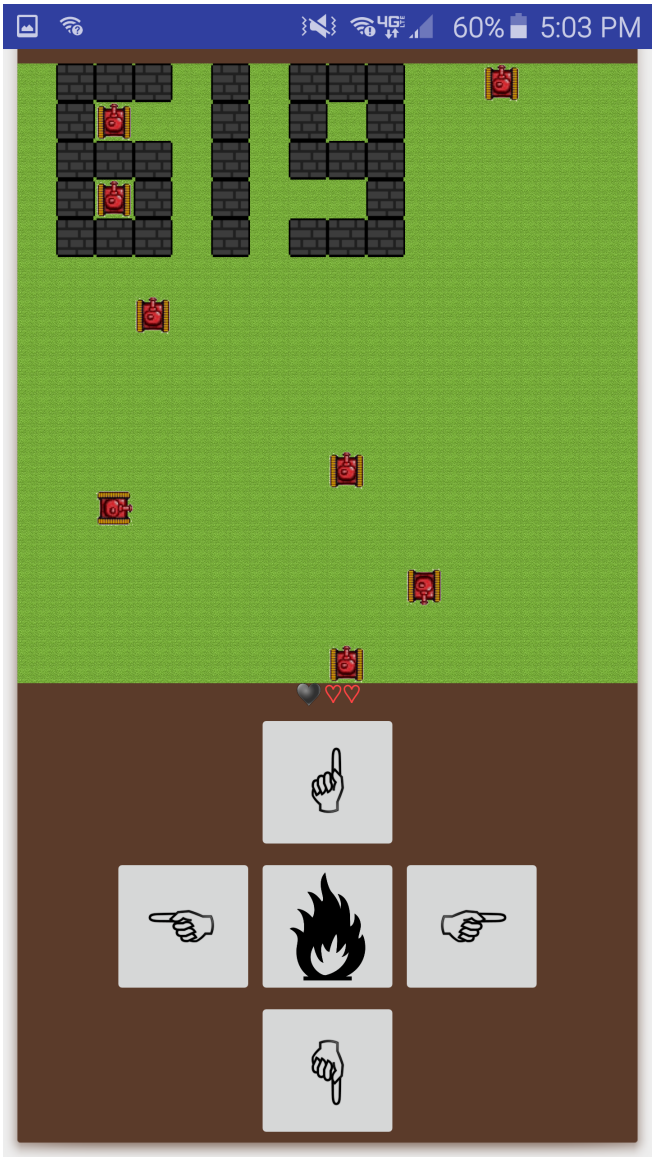

Figure 12: Spectator Mode

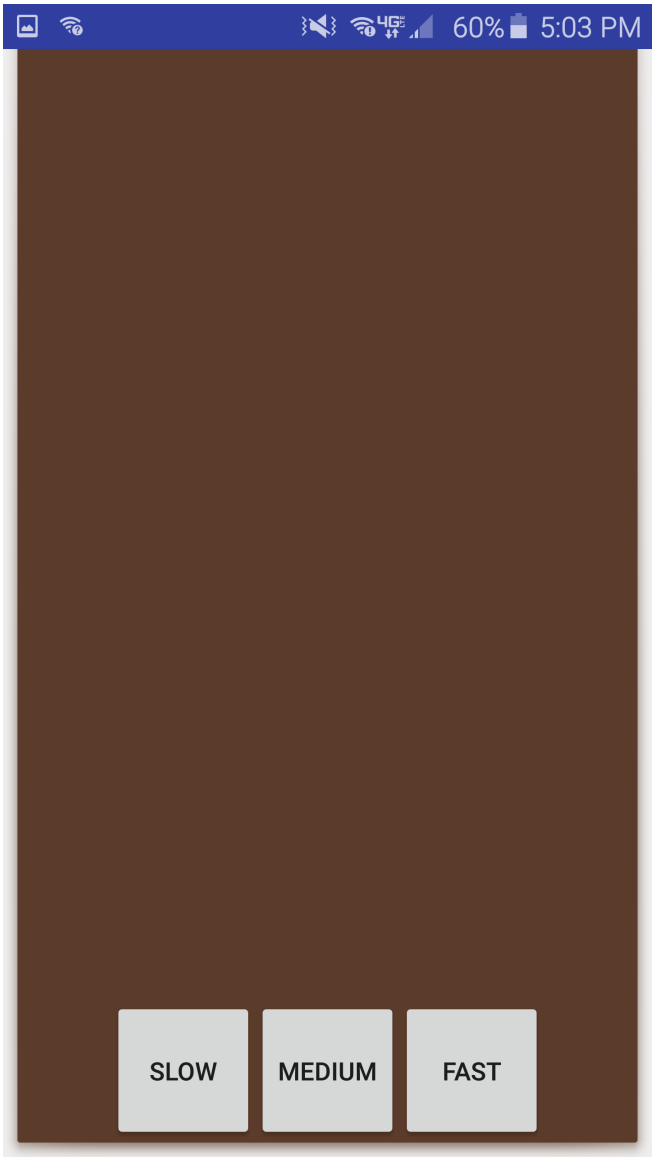

Figure 13: Replay Selection Mode

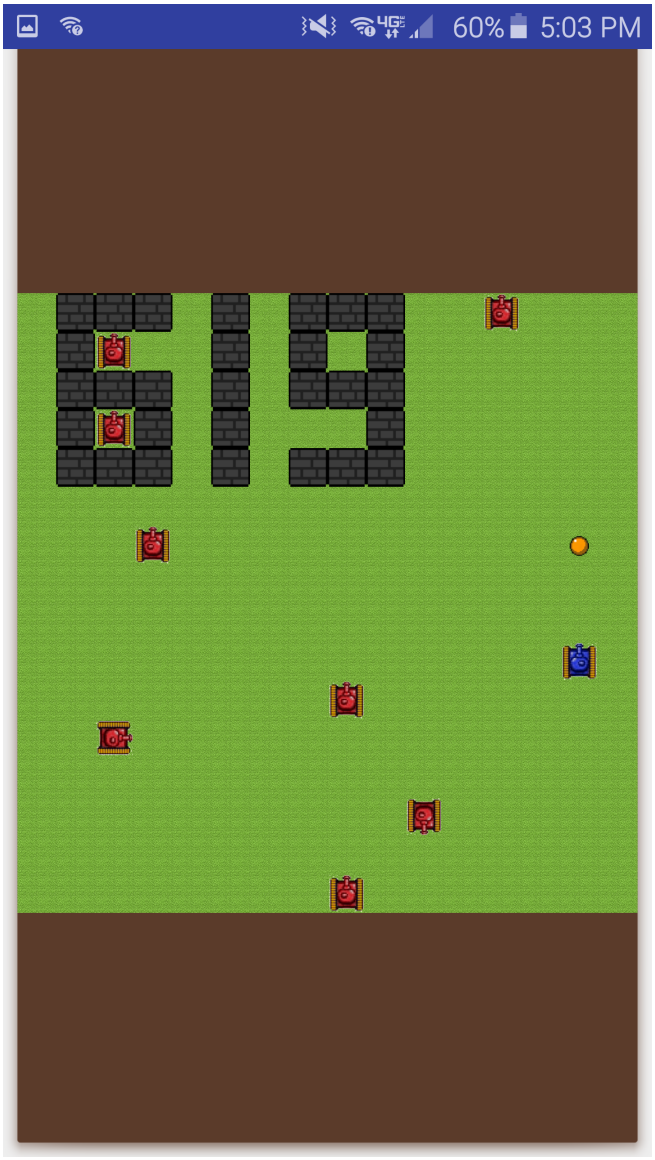

Figure 14: Replay Display Screen

Software Test Plan:

Test Approach and Features Tested

Our testing was done using white box testing as well as black box testing. In terms of white box testing, we were constantly utilizing logcat to print variable states pre and post function call. This allowed us to see exactly where our code was going awry. In particular cases, we did not necessarily have to check edge cases as we assumed that the server output was correct. We just had to check that our parsing of the output was correct.

In terms of black box testing, we tested the move(), join(), and fire() methods. We heavily tested that out buttons functioned as specified and that any combination of user input produces an expected reaction.

Tools

We used logcat, junit tested, and ourselves.

Environment

We tested on the android studio emulator, as well as two different android devices.

Tools: Pros and Cons:

Logcat:

Pros:

- Easy to use
- Can insert printout anywhere necessary
- Dependable

Cons:

- Possibility for user error
- Bug could be hard to find

Junit Testing:

Pros:

- Dependable
- Efficient

Cons:

- Learning curve

Ourselves:

Pros:

- More fun
- Simple

Cons:

- Possibility for user error
- Cannot test within method
- Can only test output

References:

Draw Beautiful Diagrams Flowcharts Wireframes UML Diagrams Mindmaps Org Charts DB Diagrams Infographics Diagrams. (n.d.). Retrieved December 12, 2015, from http://creately.com/

Freeman, E. (2004). *Head First design patterns*. Sebastopol, CA: O'Reilly.

Gamma, E. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley.

Tigris.org: Open Source Software Engineering. (n.d.). Retrieved December 12, 2015, from http://www.argouml.stage.tigris.org/