

# **Project Test Plan**

ECE 493, Group #2

Lawton Spelliscy (1198654)

Kenneth Rodas (1211431)

Dustin Durand (1203271)

Table Of Contents

## I. Introduction

### 1. Bluetooth Poker - Android

#### 1.0 Foreword

#### 1.1 Test Suite

##### 1.1.1 Application

##### 1.2.2 Bluetooth

##### 1.2.3 Client

##### 1.2.4 BluetoothPoker

##### 1.2.5 Database

##### 1.2.6 DataModels

##### 1.2.7 Game

##### 1.2.8 Misc

##### 1.2.9 Networking

##### 1.2.10 Server

#### 1.2 GUI Testing

#### 1.3 System / Function Level Testing

#### 1.4 Failure Testing:

## 2. Web Service

#### 2.1 Test Suite

##### 2.1.1: Data Models

##### 2.1.2: Services

##### 2.1.3: Util

##### 2.1.5: Database Interface

##### 2.6: Automated Test Settings

##### 2.7 System Testing

#### 2.2 Other Formal Testing

# I. Introduction

This document details the formal testing methods, processes, and strategies that were used in testing of the bluetooth poker application. This includes the testing completed on the web service, and the on the android application itself. This document details the testing completed, suites and other formal methods used, manual tests completed, and any other aspects of note in regards of testing the application.

## 1. Bluetooth Poker - Android

### 1.0 Foreword

Our initial plan for the test suite matched the web service test plan of striving for 100% statement coverage in our automated tests for the application except for libraries, the GUI itself, and external interfaces. After about three days of researching and experimenting, the plan was determined to not be feasible due to the current tools available for code coverage in the Android environment.

The only possible solutions for achieving code coverage on Android is using a library called emma, or a library called clover-for-android. Emma is actually built into the android sdk, but is NOT supported for use. It is used by google for testing the SDK / android's internal code, but there is no official support for it on android projects. After looking through many different tutorials and unofficial sources of how to use emma through a command line interface, we attempted to integrate it into our project. After a large amount of experimentation, we were able to get it to run the tests, and even trigger emma - but compiler link errors to the version of emma in the SDK could not be resolved and would cause failure. Some older articles mentioned using build.xml settings to allow its use, but all attempts of these solutions had no effect.

Another options that was attempted was clover-for-android. This project was released from Atlassian in alpha is not considered supported on their site as it is "in alpha stage and therefore it **is not officially supported** by Atlassian". After much pain and tinkering, we were able to get it working for simple data model classes, but the success was short lived. The library was extremely unstable, and would cause eclipse to deadlock when attempting the build your project, or clean your project, or attempting to update the level of coverage, etc. The biggest issue was when more than ~10 tests were successfully run, and it attempted to update the coverage it ALWAYS would enter a state "deadlock" in the sense that it would never complete the update. Since eclipse runs as a series of events, this would means that eclipse would be responsive but any event such as saving would be placed in queue behind this never ending update. The only solution was to manually kill the process and effectively lose all changes. For this reason, along with others, this option was discarded.

Based on our research and personal experiences, we do not believe code coverage can be reliably measured in the android environment. We have mentioned two options were we invested a significant amount of time, but other options, such as **Robolectric** + emma, were also explored in attempting to find a solution. As a group we have made the decision to write our unit tests in the same white box approach we originally planned for, but without the use of a

code coverage tool. We will attempt to achieve as close as possible to 100% coverage based on inspection for our intended objects(which is not a solution we like, but until a stable way to measure coverage is release for the android environment - there is no other option).

## **1.1 Test Suite**

Our general test plan for the android application is to achieve 100% statement coverage on all objects excluding the GUI, and interfaces to external components. We have decided to exclude interfaces to external components since they are dependent on an external system that is outside the control of the automated test suite, or in the case of databases would require a large amount of code functionality rewrite to perform the testing.

The test suite will be achieved through a white box methodology of implementing unit tests in order to achieve the desired level of statement coverage. The test suite is created using JUnit 3 in combination with the Android SDK. The Android SDK provides specific classes that facilitate the use of JUnit 3 on the android platform.

### **1.1.1 Application**

The only object in this package is the PokerApplication object. This object provides a global context for the whole application. An application object is typically used for objects that need to be shared across the whole applications, and are not parcelable or serializable, such as the database interface or data streams. This package has full unit coverage of 100% statement coverage.

### **1.2.2 Bluetooth**

The bluetooth package contains the objects that facilitate the connection between the different android devices over bluetooth. The majority of these objects are heavily *interweaved* with the android bluetooth interface provided by the SDK due to the event styled implementation of the interface. Due to the nature of this interface, and the fact its an interface to an external library & system, we have excluded the majority of these objects from our automated test suite. This functionality will be tested through manual testing via the formal checklists and system level testing as mentioned later in the test plan.

Note that Holder.java, which is a data class for the objects in this package, does have full statement coverage.

### **1.2.3 Client**

The client package contains the infrastructure for the client android devices to communicate information with the web service. This package was important to test since it facilitates all communication from and to the hosting server device. The automated test suite covers all objects in this package, except for a couple interface classes that excluded since their methods have no implementation.

### **1.2.4 BluetoothPoker**

This package contains the GUI logic / controllers for the user interface of the project. We have decided to exclude the GUI and its controllers from the automated test suite, and will instead test these through formal checklists in combination with manual testing as detailed later in the test plan.

### **1.2.5 Database**

The database package provides the interface to the sqlite database that is located within the Android SDK. Since this is an interface to an external component, we will exclude it from our automated test suite. The functionality of this package will be covered through the formal checklists and manual testing of the application.

### **1.2.6 DataModels**

The data models provide data representation for the application. These data objects also contain additional logic for facilitating json serialization and deserialization. These data models have near, if not exactly, 100% statement coverage for the automated test suite.

### **1.2.7 Game**

This package contains the core of the poker application logic, in regards to the game rules and logic. All other packages in the application are built to support the functionality in this package, since the game's ultimate function is to provide the user with a game of texas hold'em poker. For this reason, we strived for near 100% statement coverage for all objects in this package, and will focus a large amount of informal testing (playthroughs) on top of the automated test suite and other formal testing for this functionality.

### **1.2.8 Misc**

The majority of the misc package deals with custom GUI elements, such as custom list adapters. These GUI objects are not included in the automated test suite, and will be tested manually. There are two objects in this package that are used by the list adapters as data models - these two have been tested for full statement coverage.

### **1.2.9 Networking**

This package contains the asynchronous tasks that provide the interface between the web server and the android application. These objects are interfaces to an external system, and will not be explicitly tested through our automated unit tests. These objects will be covered implicitly through the other formal testing completed.

### **1.2.10 Server**

The server package provides all of the architecture for the server to communicate all information from the server to clients, clients to server, server to game engine, and game engine to server through the use of consuming or producing of queued events. The objects in this package provide the foundation to the application and therefore should be in the automation test suite. All objects should have nearly 100% statement coverage, with the exception a few specific exception handling cases that are in place for extraneous situations. The last exception to coverage to this coverage is the server.java object. This object facilitates the creation of all the threads, queues, game engines for the server. This server object can be considered the "top level" where all objects are created, and therefore is equivalent to a system test or high level integration test since the objects present in the server cannot be mocked. For this reason, we have left this object out of the automation suite.

## **1.2 GUI Testing**

Our strategy for testing the behaviour of the user interface was setting up a formal checklist of the most common actions the user(s) would perform, classified by the different screens in the application. It is important to note that we did not focus on testing the GUI components themselves individually, but rather on the action expected when we interacted with them (clicked, swiped, among others).

On the other hand, we can confidently say we achieved an acceptable, but not optimal, degree of GUI testing by the simple fact that we needed to test other components of the application, such as User Creation, Connecting to another table and Playing the game, among others. Every single time we tested other components, we had to interact with the GUI to get to the desired screen; for example, when testing the playing area we would have to go through all the necessary screens to host the table on the server phone in addition to going through all the screens on the client phones as well, for joining the newly created table.

The mentioned checklist for the GUI testing can be found attached under the title “GUI Test Cases”.

### **1.3 System / Function Level Testing**

Due to the nature of our application using bluetooth based connections, much of the more challenging debugging and testing will START at the system and functional level. Once found an issue is found at the system level, they will have to be traced back to their source. Our group found it imperative to mention this in the test plan, despite it only being indirectly related to the plan itself (more of a debugging challenge, but because of it we dedicated more efforts of testing to the informal side to try to discover these more uncommon type of problems).

For our top level system & functional tests, our testing plan is to develop a set of documents that contain formal checklists for the functionality described in the SRS. These tests will be completed manually, and their results recorded prior to the delivery of the project. These documents will be included alongside this test plan document. For the Android portion of the application there will be a GUI checklist, and a functionality / system level formal checklist. These formal checklists will ensure that all functionality is working from a user level, and meets the SRS requirements.

### **1.4 Failure Testing:**

These tests are considered the devious tests that we will attempt to put our application through. These types of tests will be slightly more informal and ad hoc, but some examples include:

- Dropping Internet Connection during Play
- Loss Of Bluetooth
- Disconnect Host
- Disconnect Client
- Phone Call During Game
- Sleep and waking the phone

## 2. Web Service

The following sections detail the test plan & the resulting test suite that was created in order to test the functionality of the web service. This details the automated test suite, including unit tests and system tests, and the other formal methods used in conjunction with the manual testing.

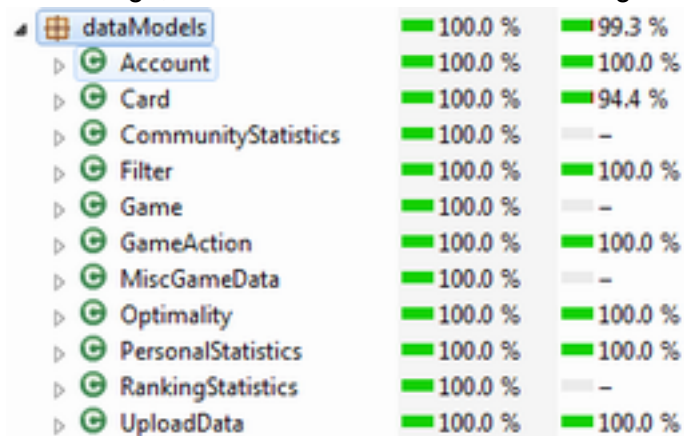
### 2.1 Test Suite

The overall goal of the test suite was to achieve as near to 100% statement coverage as possible. The exceptions to this 100% coverage include exclusions such as libraries or other code that facilitates connections to external elements, or other specific cases that are noted in this section.

To achieve an a near 100% statement coverage of the web service, a variety of tools and libraries were utilized. The primary library used for implementing the test suite was JUnit4. To measure the level of coverage, for the majority of the application, the eclipse plugin of “codecover” was utilized. This tool allows the user to run a test suite against a project and determine the current level of coverage for that test suite(in most cases). The test suites completed in our automated tests are mainly unit tests that are based on a white box approach to testing. A black box methodology was used in the system level testing that is detailed in the later sections.

#### 2.1.1: Data Models

The data models of the application provide the data presentation, and basic manipulation for the web service. These set of objects in the web services are completely covered by the test suite. Based on the measurements from code coverage, the data models have a 100% statement coverage, and also achieve a branch coverage of 99.3%.

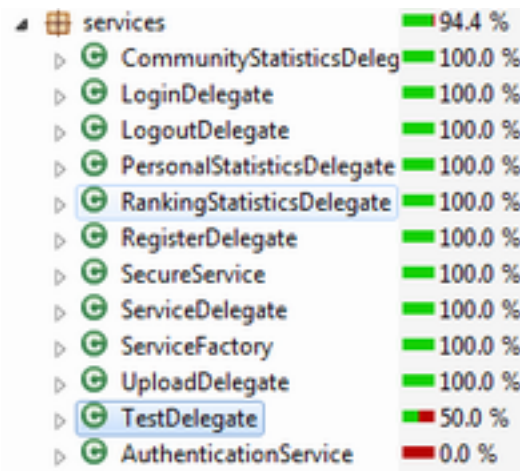


dataModels	100.0 %	99.3 %
Account	100.0 %	100.0 %
Card	100.0 %	94.4 %
CommunityStatistics	100.0 %	-
Filter	100.0 %	100.0 %
Game	100.0 %	-
GameAction	100.0 %	100.0 %
MiscGameData	100.0 %	-
Optimality	100.0 %	100.0 %
PersonalStatistics	100.0 %	100.0 %
RankingStatistics	100.0 %	-
UploadData	100.0 %	100.0 %

There are no notable exclusions of coverage or notes in this specific section.

#### 2.1.2: Services

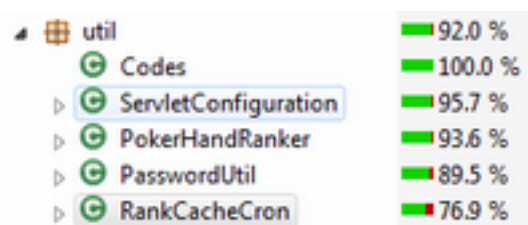
The service and service delegates of the application provide much of the backbone management and logic of the web service. They facilitate the retrieving of information from JSON or external data sources, processing and in some cases storing of this information, and generation of the final response from the web service. The test suite also provides full statement coverage for the services section of the web application.



The two notable exceptions or exclusions for our test suite are the testDelegate, and the AuthenticationService. These two objects are not for production and are only used for development purposes. Test Delegate is used in conjunction with test servlet for development purposes, and the AuthenticationService is used for the login test page to check the status of the databases.

### 2.1.3: Util

The util package provides a set of miscellaneous functionality, which is essential for the web service. The functionality provided by the objects in this package varies greatly, but is still critical to the web service. The report by codecover shows that the automated test suite provides statement coverage to the vast majority of the package, but there are some notable exclusions.



- In the ServletConfiguration there is an additional check in a synchronized method to ensure that only the singleton instantiation can only occur from the first call. This method enforces that only the first call creates the singleton, and if multiple methods are put into queue for the method - due to multiple calls at the same time - only the first is run. This check cannot be explicitly tested in any reasonable manner.



- In PokerHandRanker, there are several exceptions that relate to if one of the lookup tables not being present. These lookup tables are exported with the rest of the files in the project and should always be present. If they are not present, the project will not export or start correctly. We have excluded these tests since they would require compiling the project separately for this test, and thus these exception catches have been excluded.
- In PasswordUtil we are explicitly stating that we want to use utf-8 and the SHA-512 algorithm for the encryption of the passwords. If UTF-8 or SHA-512 is not supported on the current server, then an exception is thrown, before recovering to use the system's defaults. Since this would require additional hardware to test explicitly, we have excluded these tests.
- In RankCacheCron, the functionality of the object is tested, but there are exception handlers that are tested by the test suite. If any exception occurs an error is written to the error log, so this functionality is not explicitly tested in the test suite.
- NOTE: The CRON test is not explicitly included in the AllTest.java suite file since it can effect the database.

#### 2.1.4: Servlets

The servlets acts as the main interface between the internet and our web service's various service objects and delegates. These elements are critical to the application, and facilitate the communication between the http client and server for receiving the initial json data, and outputting the json result. These servlet objects can only run with a context that is created when the server is running, which means JUnit4 is unable to properly test the servlets. For this reason, a different library - httputil is used for testing these elements. The one caveat to using this library is that it's contacting the web service after it has exported and running, and therefore we cannot get a statement coverage report. Based on our looking at our servlet tests, we can make the following statements:

- All the servlets are tested, except the test servlet.
- All statements are currently executed in those servlets, with the exception of a database exception, which is caught and an error reported. This is not tested since its a last line of defence and shouldn't occur. The only possible way this exception should occur is if a major database failure occurs and an unknown error occurs. Since this is the case, this case is not explicitly tested.

#### 2.1.5: Database Interface

The database interface acts as the interface between the external database and the web service. This interface is, from the point of view of the server, the external storage of the application. Due to the nature of the database requiring a set of code to access, retrieve data and check any test code written, would provide a majority of overlapping functionality of the original code. Several libraries were explored to determine if they could be a viable to test the database interfaces, but most wouldn't work with the servlet setup. For this reason the database interface is not tested as a separate entity. It is tested implicitly through the system/integration tests, which tests the system as a whole. These system tests follow a black box approach to testing the various functionality of the system, but the database will be implicitly tested through them.

## 2.6: Automated Test Settings

The following section is critical for the deployment and use of the test suite. These settings are required for the test suite running correctly.

The system tests - `system.SystemTest.java` - and `cron - CronTest.java` - are not included in the `allTests.java` suite file since they can change the database. They should be run separately when a test database is connected to the web service. A note for these files is that there is also a `DatabaseOverride.java` object that contains an override for the database. This is required since the `datasource` pool is only generated when the service is running - therefore we have to manually create a single connection in this file. When testing `cron`, this object must be edited to point to the correct database.

The most critical setting in regards to running the automated test suite are the ones provided to the JVM when starting the application. In order for the tests to complete correctly when running any of the tests, including `AllTests.java`, make sure to pass “-Xms1024m -Xmx1024m” to the JVM. This informs the JVM to increase the max heap size. Due to the lookup tables needed by the application, the test suite will fail to complete a variety of tests due to this issue.

## 2.7 System Testing

In our automated test suite we have also included a set of system level tests for the web service. These tests cover the main functionality of the web service, and are assumed to run on a live implementation with a test database. These tests are based on a black box methodology through the use of a partial oracle with a set input and expected output from the web service. These tests are run as a series, with the effect of one possibly effecting the other. For this reason one should always look at the first test to fail for the source of an error in these system level tests. These tests can be found the “system” package of the test source folder of the web service project.

## 2.2 Other Formal Testing

For the web service, we have also included a formal checklist based on the functionality detailed in the SRS. This checklist can be seen as the final check before delivery; these tests are done at the highest level and are ensuring the functionalities of the system are completed. Many of the checklist items for the web service will be indirectly overlapping with the formal checklists for the android application, due to applications dependency on the web service. The web service’s completed checklist is included with the other test documents.

