Team: HvZ App

Members: Kyra Clark, Santiago Rodriguez, Alex Hadley, Matthew Waddell
Project 3C — Test Plan

Github Link:
https://github.com/alexhad6/HvZ-app/blob/master/Administrative/Phase%203/Test_3C.txt
Primary author: Matthew

Component Test Plan: User Interface

I will be testing whether User Interface response to API behavior is as expected and outlined in my specs. In general WhiteBox testing for isolating specific interactions and Block Box for cohesive app behavior.

To be Specific, Whitebox testing I will use Mock APIs to isolate User Interface errors from API functionality. After Whitebox tests of this kind pass If the corresponding integration test dependent on both UI and API fails then assuming my MockAPI was representative of API behavior I will know that the problem is API derived not UI derived.Whitebox tests will also be useful to specific transitions from one display screen to another.

Example of Representative WhiteBox MockAPI:

```
Def mockZombies):
if (s==‘PlayerZ’)
Return True
Elif (s==‘PlayerH’)
Return False
```

Whitebox testing will allow me to check for especially intricate specific cases with more control such as the following:

1.First I will make sure that when the  QR Scanner passes a non-Qr code image or a QR image not saved in the database the API should fail to match this image with any QR code in the database.

2.Second I will make sure that when the QR Scanner passes the API modified the corresponding Player to zombie and kicks the now zombie player from the game I will make sure these functions are called in the correct order with corresponding print statement because if the player is kicked before their attribute is changed then the player may rejoin the game and allow other zombie players to scan them, causing multiple parent nodes in the ancestry tree.

3.Additionally if the human was scanned and correctly and appropriately modified in the database to Zombie without a subsequent kick command being confirmed then I will have a test to decrease the periods of time per which the converted player's phone is incrementally called

and increase the total time. Lastly, I can also test that the tree is appropriately updated when a valid feed code conversion takes place.

Whitebox testing will allow me to check for especially intricate specific cases such as when the mod resurrects a zombie I can print a confirm statement andthe kick function and sets the players attribute to Human. And then check in a different unit test that when the player calls feedcode the API method called is display feed code instead of scan feed code.

Furthermore, the Blackbox testing I will create a mock database instantiated at the start of each test and deleted upon completion on which the API calls its database functions. These tests will immediately inform whether the behavior of the app is cohesively consistent, but will require subsequent whitebox tests when these tests fail in order to pinpoint the bug. In order to test UI functionality in blackbox test after an API method is called I will have written flutter unit tests which search for and then click on (if able)the appropriately subsequently displayed text.

Blackbox testing will allow me to check for cohesive consistency in all cases such as the following:

## First

1.One simpler BlackBox test with a mock database I will include a list of Players strings which will be stored as ids for Player objects in the mock database and a mockMod which only starts game and ends game when after all user paths have been explored individually without interaction between players (ie no feedcode scan or transfer) and only mod paths taken ie announcements kicking and banning.

2.Then to Build the more complicated off of this test I will include player conversion and display of the feedcode and try to run reading tests for changes in the displayed objects such as the ancestry tree.

## Second

3.
Aside from testing individual pieces of functionality, it is helpful to run some integration tests that will explore interactions between mod paths and user paths, for example ensuring that announcement inputs are updated as well as mission files.  BlackBox tests will be especially useful for the Endgame Function since the text saved will be created as the product of various possible orderings of methods which is more representative of an actual game.
To emulate a realistic game for this purpose we would first pre populate the mock database with an amount of player objects equal to the expected amount of users. Then begin conversions

implementing the feedcode functionality, check if the tree is updated properly. Then run alternate between Mod ban kick and ban commands and subsequent conversions. After running the Endgame function we could ensure for instance that the banned player is redacted from the game results.

## Feasibility of Testing:

https://flutter.dev/docs/cookbook/testing/unit/mocking
https://flutter.dev/docs/testing
For my blackbox and whitebox testing I require a flutter framework to prepopulate a mock database, create mockAPI and be able to read display text created after API methods are called and click on the searched for text to autotest requirements for the next screen.
Flutter has the ability to find widgets as well as tap drag and click widgets in tests As well as mock http API protocol.