4. Verification Test Plan

4.1 Introduction

4.1.1 Purpose

The purpose of this test plan is to verify the functionality of all aspects of the software so that the requirements can be fulfilled. Tests will be conducted to target at the unit, functional, and system levels.

4.1.2 Scope

The test plan covers the entire architecture of the system, including the website main page, login page, account manipulation, theater data manipulation, movie data manipulation, review page, movie browser, search bar, seat map, and payment menu.

4.2 Test Objectives:

The goal of the testing process is to catch any immediate problems as well as identify areas that may pose future problems. At the most fundamental level, the unit tests seek to ensure each section of the code performs its intended task which is separate from the overall functionality of the program that the user experiences. For example, one of the unit tests, tests whether or not the theater admin class properly updates the theater objects name. This type of testing can be done in the software and does not need to be done via the actual software. The next stage of testing, functional testing, will seek to ensure that the program functions well from the level of using the software as a user would without looking at the code. This is a type of black box testing. Testing should be performed by looking at all the functions a user can perform and attempting to use them as intended and even as not intended to see how the software performs. Examples include things like creating a user account and then attempting to make updates to user info such as the username and password. Success would be seen in-website. The final stage of testing would be system testing which would seek to test whether all functions of the system work in unison. This includes things such as testing whether the bank api for payments works and whether the system can handle a large amount of traffic. This stage of testing might need to be performed at both the user and coding level. For example high amounts of traffic could be simulated using test objects whereas testing whether the software works on different platforms would be performed from the UI level. At each level of testing, attempts should be made to break the software or system to ensure things will run smoothly. For example testing weird inputs to see whether the software allows it should be done. If done improperly, this could slip by and not cause any compiling errors but it could break the functionality of the system. This is why both unit and functional testing need to be done.

4.3 Test Environment

4.3.1 Hardware:

These tests will take place on mobile devices, laptops, and desktops. However among these devices of various different types will be tested that operate off different operating systems. This will be addressed in more detail in section 4.3.2. Laptops and desktops will both produce relatively the same results so main testing will be performed on a desktop computer. However extra cautionary testing will be performed on the laptop but it will be minimized to reduce time and costs.

4.3.2 Software:

Among mobile devices the main operating systems will be tested. So the website will be tested using the native browser of IOS devices, Android devices, and HarmonyOS that runs on Huawei devices. Among computers the website will be accessed on Microsoft Windows, Linux, and MacOS. Since browsers among these systems will produce relatively the same results we just need to make sure that the pages display properly among the three operating systems but main functionality testing will occur in Windows. Among browsers the main ones to be tested will be Google Chrome, Microsoft Edge, Microsoft Explorer, Firefox, Safari and Android browser. Tests will be performed within each of these browsers. One important note is that while tests will be performed on the devices, operating systems, and browsers listed above; the most in-depth and comprehensive testing will be performed on a desktop computer running a windows operating system and utilizing the google chrome browser.

4.4 Test Scenarios:

Our testing will need to be very extensive, however some of the specific testing scenarios will be described here.

Our first test case example is testing the payment menu. This is a unit test that tests whether payment info can only enter correctly such as the CCV only being an int. This can be done with a test object.

The next thing we will test is whether the createTheater function can update the database properly. This can also be done as a unit test with an object.

The edit movie test will see if the editMovieInfo function updates the movie object's field variables. This is a unit test and can be done with a theater admin object. A simple print output test with expected versus actual results could work for all the unit tests.

The final unit test example is to test whether the buyTicket Function works. This can be done within the guest and customer class and checking to see if a ticket object is created.

The changePasswordTest will be a functional test to see if from a user's perspective their password changes in the login info section. This can be done by acting as a user and seeing if the password shows up correctly.

The removePurchase_dataBase_update will be another functional test to see if an IT support user can delete a payment and the corresponding payment object gets deleted.

The movieSearchTest will be another functional test to see whether the search function works properly. This can be done by attempting every scenario a user might run into. This includes the tester trying stuff like entering a word, entering a phrase, and entering a genre. Any errors may require unit testing to resolve.

The next test is the seatAvailabilityUpdate test. This is a unit test to test the whole system functionality. When a user buys a ticket, the system needs to almost instantly update the database to show that seat as taken so that if another user is trying to buy that ticket it will show up as taken. Huge issues would arise if a ticket that no longer is available is able to be purchased.

The accountInfoUpdate test will be a system test to ensure that when information is updated in one section of the system, it is concurrently updated in other areas. For example if a user leaves a review, and then updates their username, the old review should show as left under the updated username and not the old one.

Another test example is the ChangeofOS test. This is a third system test that will ensure that the software can be used equally well on any operating system. Further testing beyond these specifics will be necessary.

We will need to test that redundancies in the database do not occur so the system will not permit two of the exact same theater to be created. Testing will involve ensuring the system compares theater addresses to make sure duplicates do not exist
We will need to test that a valid email address is required for the user to make an account. An invalid email address should give the user some sort of message to inform them their email is invalid.
Guest class users should not be able to leave a review on any movie. The system must ensure this by checking and making sure that the ID type is not of the guest type before allowing them to leave a review.

A test scenario for account creation is to verify that a user should only be able to create an admin account as long as they have a valid admin ID. This test would be a unit test to make sure that only a valid ID already in the database can be used to create an admin account.

Testing should be done to ensure that the deleteAccount method wipes the corresponding account data from the database. Holding on to unnecessary information creates security risks for the user and company. This can be done by deleting a test account object and then attempting to access that information and hopefully not finding it.

When a user selects to leave a review the system will check if the user has a purchase for a ticket of that movie in their ticket history. If they do not they will be notified that they have not watched this movie so they cannot leave a review

Removing a review from the review page also needs to be tested. This would be a unit test to verify that the data from the review was deleted from the database. This would protect the user from having a permanent review on the website.

4.5 Testing Approach:
Much of the content held and displayed on this website is user generated. The accounts, the movies, and reviews are some of the core data on this system which is all heavily reliant on user input. With such being the case much of the testing will have to be done manually.
Some pre-programmed inputs can be developed and then executed simultaneously to check for edge cases where there are two conflicting inputs such as for buying the same seat. This can also be utilized to load test the website and ensure it operates smoothly with moderate levels of usage. Testers will have to manually check much of the functionality such as systems for creating movie listings, theaters, leaving reviews, and editing accounts and purchase information.

4.6 Test Schedule:
Unit Tests will take approximately 1 week to complete. Bug fixes will be applied in the midst of testing.
Functional Tests will be performed afterwards and will take 4 weeks to complete. This ensures cohesion between the software modules. The first week will be used to identify errors within the program. The second week will be used to resolve issues found within the code. Then the last 2 weeks will be spent confirming the proper functionality of the code and fixing any smaller issues found during the testing. System Tests will be performed afterwards for the next 5 weeks. The first 2 weeks will be spent on administrative side testing. The next 2 weeks will be spent on client side testing. The last week will be spent ensuring systemwide communication operates properly.

4.7 Risks and Mitigation:
One risk is receiving invalid payment information.
To solve this problem simple misinputs that do not meet the required amount of characters can be caught by the system. However invalid cards that are of proper length will need to be checked through a bank before a bank API before the transaction is finalized.
Another risk is with identical account creation. If hypothetically two users were to create an account with the same name and password how would the system differentiate the two at login time. The solution to this issue is having the userID that is associated with every user account to be required to be entered at login. This way everyone has a unique identifier and no information could completely overlap.
Another risk is when people purchase the same seat. If two users bought the same seat at about the exact same time there would be issues. This issue can be solved by having a condition that if two users bought a seat at the same time and neither of them was warned that the other already bought the seat then while the payment for the seat is pending the system will look at the userID for both customers and whoever's userID has the lower value will get the seat and the other will have their transaction canceled and be notified that the seat was taken.
One more risk is with payment security. There is no simple solution to this problem. The only proper way to manage the risks is to keep payment information in a smaller database that is much more secure and has much higher restricted access. Along with that it will have to be maintained with up to date security patches.

4.8 Conclusion:

After the steps depicted in the verification test plan are completed the website will be in a state where there is a high confidence level for the system to execute as intended. Any major issues should not arise. However with the increase in usage, less severe issues may be found that can be resolved with the help of the IT class and later fully patched by the development team.