## 3.2. Test Strategy Plan

A test strategy is an outline that describes the testing approach of the software solution. It is a crucial piece of document which helps the entire development team better understand key issues of the testing process. On top of that, it helps the managers organize and handle risky areas of the project, as well as keep track of possible issues and development progress.

### 3.2.1. Project Scope

The purpose of this project is helping the Georgia tech library in their most usual tasks, of lending books to several thousands of users, by creating a software solution that can achieve just that. Thus, the business goals of this project are:

* Improved response times, when it comes to members loaning and returning books
* Better material tracking
* Easy, fast and reliable inter-institutional communication

Due to lack of time, we have decided to test our UI using only the leader of browser market share, Google Chrome.

### 3.2.1. Test Approach

On top of the aforementioned testing methodology, we have decided to conduct our tests using the following code standards:

* Tests will be written using the AAA (Arrange, Act, Assert) pattern, as it helps with readability by clearly defining sections within a testing method.
* Test method names are created as follows: UserStory\_TestCase\_ExpectedBehaviour
* Test classes are created and organized within dedicated folders, for each type of tests (Unit, Integration, Acceptance)
* Tests are created and are able to run independently
* Tests are running on fresh environments, unless otherwise specified

Alongside those code standards, testing documents, diagrams and other non-code related material, will be tested using informal reviews and/or walkthroughs, with each member taking one of the roles typically encountered in such meetings (moderator, author, reviewers), intentionally omitting the scribe, as the value such a role brings is minimal, given the team size.

### 3.2.2. Risk Analysis

In order to ensure that Ares runs as smoothly as possible and to reduce the risk that, possibly encountered problems, play a minor role in both user experience and will not hurt the Georgia Tech Library, we have decided to group our user stories under different risk levels and deal with them accordingly.

The following table depicts our organization of possible risks that each of our program’s features has. The numbers in the column “user story” reference the numbers present in the column “ID” in the “user stories” table, that can be seen in the appendix of this paper.

|  |  |  |  |
| --- | --- | --- | --- |
| Likelihood -> | Impact -> | | |
| Low | Medium | High |
| Low | Medium | Medium |
| Low | Low | Low |

To better understand the way we determined risks, we used a risk matrix, which perfectly visualizes how a risk level is chosen based on likelihood of it occurring the impact it has on the software

|  |  |
| --- | --- |
| User story | Risk level |
| 1. Catalogue of books | High |
| 16. Checkout books |
| 17. Return books |
| 20. View books |
| Book count checkout limit |
| Book checkout time limit |
| Prevent checkout of unauthorized material |
| 2. 3. Exchange books with other libraries | Medium |
| 4. 5. Exchange user information with other libraries |
| 12. List of books that need to be acquired |
| 18. 19. Add new members to library |
| Add new books |
| 15. Renew member card | Low |
| 8. Lending statistics |

The risk organization was made based on perceived risks. As the main function of any library is lending materials, tasks like: “Checkout books”, “return books” as well as some business decisions like: “5 book limit per person”, “lend time restrains” and “lending of rare material” have been placed under a High risk level; where tasks like “create new person”, “add new book”, “get book from remote library”, etc. although still important, have been placed under lower risk levels.

### 3.2.3. Test Environment

As there are several tests that we have decided to work with, the test environment varies widely from the safe development environment of the IDE, to brand new, fresh and isolated nodes in the continuous development solution, to actual running Azure servers on which the application can be Alpha tested.

The following table describes how each of the tests are spread across which types of environment:

|  |  |  |
| --- | --- | --- |
| Test type | Environment | Environment description |
| Unit | Local / Appveyor | Personal laptops and virtual nodes using Visual Studio 2017 |
| Integration | Local / Appveyor |
| Acceptance | Local / Azure Cloud | Microsoft Azure free servers, on which the application is deployed after each release. |
| Alpha | Azure Cloud |

### 3.2.4. Testing tools

In order to make testing easier and faster, we have decided to automate it using the following tools:

* N-unit tests
* Selenium with chrome webdriver
* Appveyor

These tools have been chosen for their unique abilities and ease of use, in order to amplify our project’s testability and in order to ensure it works with as little impediments as possible.

In terms of test automation, we have compared several tools that might help us, tools like: selenium, katalon, Squish, Tricentis, or TestCraft. And we decided to go with a Page Object Model-based framework, rather than a “record-n-play" framework, as although the record-and-play function of katalon, might be beginner friendly and not require any programming skills, we consider the flexibility offered by a Page Object Model-based framework, to be vastly superior and freeing at the same time as code snippets might be reused in future tests or easily changed and refactored as the product evolves, where recorded tests must be constructed from scratch, with every single change. In the end, we have chosen selenium as our test automation tool due to the following reasons:

* Compatibility across Operating Systems;
* Support for multiple scripting languages (including C#, which we are most familiar with and are using for this project);
* Support for running tests on multiple browsers;
* Large and open community (in case we needed assistance);
* Extended documentation (in case we needed assistance);
* Open Source (free to use and a large community behind it);

Often, checking that all tests are still successfully executed and the made changes did not affect any of the previously implemented functionality, is forgotten. Thus, we decided to Appveyor as our continuous integration solution for Ares. With Appveyor, the “burden” of rechecking tests and functions, is taken from the testers shoulders and put on remote, fresh and isolated nodes. We have integrated Appveyor with our GitHub repository and configured it such that it will automatically run with every new push to an open pull request, allowing for the pull request to be merged with the main branch only, if the build was successful and if the tests run successfully. This integration has helped us ensuring that new code changes don’t affect previous functions, as well as making sure that the code will run on other machines.

### 3.2.5. Release Control

Following the decision of working under an agile framework, we have decided to release new updates at the end of every sprint. Updates that would bring both product’s stability and value to the customer.

We have also decided to use a cloud hosting service (Microsoft Azure) as grounds for our release environment, where the customer can get a hands-on experience of the new changes.

During a release, the following procedure would typically be followed:

* Merge Development branch in Master branch
* Continuous integration checks
* Deploy Master to Azure
* Product owner reviews changes
* Feedback session

### 3.2.6. Review and Approval

In order to ensure that new features are added correctly, we have decided to instate several guidelines upon how things should be added to the project:

* To keep a level of traceability, always work in new branches, based on the Development branch, with the same name as the task you're currently working on (including the Trello card number) ex: T-cardNumber cardTitle
* When you believe a task is finished, create a pull request (always write a short description of what you did when creating the pull request)
* Once the branch is approved (passes Appveyor checks and gets at least one positive review), the owner of said branch, will merge with development branch.
* Merging with master will be done once every sprint.

As for approval, we have decided to employ the use of Acceptance tests, where, at the end of each sprint, the customer would get to see and experience the new changes, followed by their feedback and approval/dismissal.