1. Website Overview

The user is greeted by the homepage where he can choose to either register or login to an account. If the user chooses to register an account, he is expected to input the required fields correctly and the outcome is indicated on the registration page after the information has been submitted.

If the user already possesses an account, he is required to input the correct credentials in the login form which is found on the homepage. After three consecutive invalid logins, the user account is locked for five minutes. After a successful login, the user is redirected to the betting page, where he can than place bets until reaching the account limit or until the user decides to log out.

2. Levels of Testing

Different levels of testing are performed on the system:

- Unit testing – unit tests cover most of the logic of the website
- Integration testing – Integration tests test the database connection and database queries. These tests are performed on a test database (set through a properties file)
- User acceptance testing – User stories are tested using cucumber

3. Testing Methodologies Used

Dependency Injection was employed to inject dependent components into test objects. Both setter injection and constructor injection were used, depending on the case. The use of this methodology required additional setter or constructor methods.

4. Testing strategies

The testing strategies used are the random strategy and the partitioning strategy. Specific examples where these strategies are used are given below.

- Random Strategy – This strategy was used to test credit card numbers, since there are no particular boundaries.
- Partitioning Strategy – This strategy was used to test the function that validates a login request, when an account is locked. This has only two partitions: the difference in time from the moment the account was locked is greater than five minutes or smaller than five minutes.

5. Coverage Report

The following image shows the code coverage statistics as measured by the Emma plugin for Eclipse. The overall code coverage is 82.3%.

Assignment	90.6 %	7,678	799	8,477
src/main/java	82.3 %	2,146	460	2,606
com.assignment.DBObjects	64.7 %	626	341	967
User.java	65.0 %	487	262	749
Bet.java	63.8 %	139	79	218
com.assignment.functionalities	89.8 %	316	36	352
BettingImp.java	80.7 %	151	36	187
LoginImp.java	100.0 %	78	0	78
RegistrationImpl.java	100.0 %	87	0	87
com.assignment.servlets	91.6 %	284	26	310
RegisterServlet.java	90.1 %	109	12	121
BettingServlet.java	94.5 %	121	7	128
LoginServlet.java	88.5 %	54	7	61
com.assignment.util	72.7 %	64	24	88
MenuImpl.java	56.1 %	23	18	41
Props.java	71.4 %	15	6	21
MessagePageImpl.java	100.0 %	26	0	26
com.assignment.validations	96.0 %	529	22	551
RegistrationValidationImp.java	95.0 %	416	22	438
BettingValidationsImp.java	100.0 %	67	0	67
LoginValidationsImpl.java	100.0 %	46	0	46
com.assignment.mongodb	94.6 %	192	11	203
MongoDBConnectionWrapperImpl.ja	75.0 %	21	7	28
MongoDBWrapperImpl.java	96.8 %	122	4	126
MongoDBActionsWrapperImpl.java	100.0 %	49	0	49
com.assignment.requests	100.0 %	135	0	135
BettingRequestsImpl.java	100.0 %	63	0	63
UserRequestImpl.java	100.0 %	72	0	72
src/test/java	94.2 %	5,532	339	5,871

Illustration 1: Code coverage metrics

The following table explains why certain code was not covered.

User	65.0 %
hashCode()	0.0 %
equals(Object)	59.9 %
setCreditcard(String)	0.0 %
setCvv(String)	0.0 %
setDob(String)	0.0 %
setExpdate(String)	0.0 %
setSname(String)	0.0 %

Package Name	Class Name	Method Name	Reason
DBObjects	User Bet	hashCode	These are the default methods generated by the IDE and are rarely used
		equals	
		Various setter methods	These are not covered as they are simple setter methods where no calculations are performed
Functionalities	BettingImp	addBet	This method could not be tested because the return value depends on an instance of type BettingRequestImpl which is created within. Also, this method does not have any decision statements or calculations to perform, it relies on methods of the aforementioned instance.
		getMessage	Simply a getter
Servlets	RegisterServlet BettingServlet LoginServlet	init	This method is used on initialization of the servlet, and its sole purpose is to generate instances of the variables used. To test these classes, mocks are set through setter methods
Util	MenuImpl	getLoggedInMenu GetLoggedOutMenu	The purpose of these methods is to return an HTML string which represent a menu
Validations	RegistrationValidationsImp	Various methods	The only lines of code that are not tested perform null checking
Mongodb	MongoDbConnectionWrapperImpl	connect	Exception handling is not tested as the instantiation of a new object cannot be mocked

1.Web Testing using Selenium Test Suite

a.Approach Adopted

The Page Object approach was adopted to increase the readability of the code. Each webpage has a corresponding class, composed of methods which are responsible for manipulating the page objects of the respective page. When a particular page object needs to be tested for something, an instance of its respective class is

initialised and the appropriate method is called. Each of these classes are initialized using a WebDriver instance to be able to perform operations in that browser.

b.Testing methodology

For the front end, we test to the error messages. We assigned an id for every element being used. Then when we want to assert a particular result we find the respective element by its id and read the contents and verify whether they match the expected contents.

2.Cucumber edits

- Scenario 2: Incorrect registration data
  - Since on our site the error messages have their own div with its own id, the scenario had to be altered a bit, in a way which gave us the possibility to provide the id of the message div where we need to make the assertion on. We also had to provide the error message to which to compare. Scenario generated:
    - Scenario Outline: Change field names

```
Given I am a user trying to register
When I fill in a form with correct data and I change the "<fieldname>"
field to have incorrect input
Then I should be told in "<errorMessage>" that the data in
"<fieldname>" is "<incorrect>"
Examples:
|fieldname     |incorrect           |errorMessage      |
|firstName     |Invalid characters  |name_error        |
|lastName      |Invalid characters  |surname_error           |
|dob           |Please enter date of birth |dob_error|
|creditcard    |Invalid card        |creditcard_error|
|expiry_date   |Invalid Expirary date|expiry_error|
```

1.Scenario 7: Verify that free users can only place low-risk bets

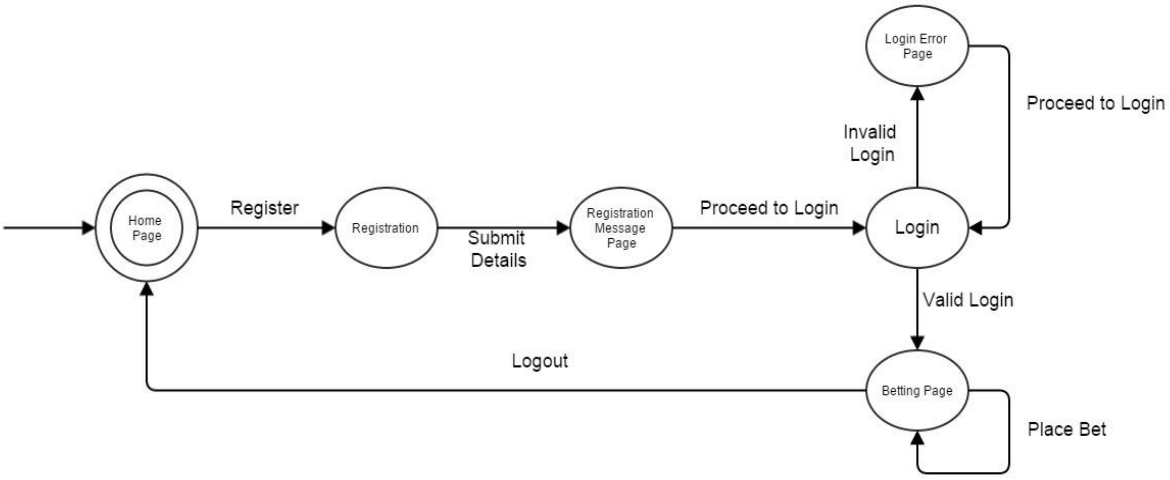
2.when the user tries to make a bet with a risk level that is not available for the his account, he is displayed a message indicating so. the modified scenario:

- Scenario Outline: invalid risks

```
Given I am a user with a free account
When I try to place a "<risk>" bet of 5 euros
Then I should see "<message>"
Examples:
|risk     |message               |
|low      |Bet placed successfully|
|medium   |Invalid risk          |
|high     |Invalid risk          |
```

1.Model Overview

The model simulates the typical behaviour of a user who is using the website for the first time. The model is initiated by loading the homepage. The user must first register to be able to log in. When registration is complete, the details are submitted and the user is redirected to the Registration message page, which notifies the user with the outcome of the registration . The user then proceeds to login. Given the case that the user inputs an incorrect username or password, he is redirected to the Login Error Page, and the user must proceed back to the login. When the user provides correct username and password, he will be redirected to the Betting page, where he will either place a bet or logout.



2.Model Design

The model is composed of states and actions. Every page in the website is represented by a state in the model. Transitions from one page to another and the operations that happen during a transition such as submitting information are represented by the actions. during the execution of the model, the current state is obtained by the current url that the website is using. For example if the website is currently on the betting page, the current state is Betting page.

The model we designed is composed of the following states:

- ●Home page
- ●Registration
- ●Registration Message page
- ●Login
- ●Login Error Page
- ●Betting Page

Action	Description
Register	Navigate to the registration page
Submit Details	Fill the registration form and submit it. The user has a 75% probability of being a free user and 25% of being a premium user.
Proceed to Login	Navigate to the login page
Valid Login	A correct username and password are provided with a probability of 75%, and the user is directed to the Betting Page.
Invalid Login	Wrong username and password are inputted with a probability of 25% and the user is redirected to the Login Error Page.
Place Bet	Place bet with a probability of 50%. If the user is free, a random bet between 0 and 6 is placed, while if the user is a premium one, he will place a random bet between 100 and 2000. While the user does not choose to log out, he will remain on the betting page.
Logout	Navigate to the Home Page with a probability of 50%

3.Test Set Up

The performance test was carried out using multiple threads, each running an instance of the model. Concurrent execution of the model is guaranteed by first opening all the browsers required before executing any model. The related classes used for the model are described below.

Class Name	Description
PerformanceTest	Contains the model of a typical user using the website
TestLauncher	Launches multiple instances of the model over different threads

4.Assumptions

a. The model assumes that the user will never enter a wrong password three times consequently.

b.When registering registration data will always be correct

5.Average Response Time

a.Time Calculation Process

The methodology we employed to calculate the average response time for each page, was by initialising a vector of type long. For each thread created this vector is passed to it, and when a page is to be loaded, the time before the loading is recorded, and when the page is loaded and asserted that it is the correct page, the time is recorded again. The time recorded before the loading, is subtracted from the time after the load and added to this vector. When all the threads finish, the times recorded are added together, and divided by the total times recorded so that to obtain the average response time for each page.

b.Results

Users	Windows avg resp time	Ubuntu
5	176	120
25	1375	719
50	3338	1615
100	14309	710 / 718

c.From the test results provided in table above, one can notice that the response time for a page on our hardware depends on the number of concurrent users using the site. The response time for when having five to 25 users was great. the response time started to degrade when the system touched with 50 users, while with 100 concurrent users is totally not acceptable. It also can be noticed that the response times are different for the two operating systems used for this test. the execution on Linux was faster while on Windows was a little slower.