# Website Overview

The first page the user will be provided is the Home Page, which has one button to be redirected to the Contact Supervisor Page. In our case, the user will be an agent. Whenever the agent wants to use the system, he/she first needs to contact the supervisor to be given a login key. The details that are required to be entered for validation is the agent’s ID and name. If the agent’s ID starts with ‘spy’, he would not be granted a login key. On the other hand, if successful validation occurs, the agent would be redirected to the Login Key Page, were the agent’s details together with the login key are displayed. The agent has 60 seconds to login to the system, otherwise it would become expired and would need to re-contact the supervisor. Once the agent is logged in to the system, the agent can send and receive messages.

The mailbox allows the user to send 25 messages and subsequently receive another 25 messages. If this limit is exceeded, the system will, automatically logout the agent. A message has a timestamp allocated with it so that each message will stay in the mailbox for 30 minutes, after which they will be inaccessible. A message are considered to be invalid under the following validations:

* The Source Agent whom is trying to send the message would not be the one logged in;
* Even though a message containing blocked words such as ‘ginger’ and ‘recipe’ is not considered an invalid message, these blocked words would be removed from the message before sending;
* Messages exceeding 140 characters are considered to be invalid;

After 10 minutes logged in to the system, the user will be automatically logged out and has to proceed in contacting the supervisor again to get a valid login key.

# Testing Techniques

To test the above described system, the following level of testing strategies were implemented:

* Unit Testing – Through Junit;
* User Acceptance Testing – Through Selenium and Cucumber;
* Model based Testing – Through Junit, Selenium and an FSM;
* Performance Testing – Through JMeter;

# Unit Testing and Test Driven Development

While writing both the classes and tests we made sure that each test was testing for one thing only and that they always returned the same result no matter how many times they were ran. Moreover, tests were not to have any conditional logic or depend on some other tests.

## Class Diagram

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **Method** | **Return Type** | **Description** |
| Agent | contactSupervisor | Boolean | Initiate contact with a supervisor to get a login key and subsequently logs into the system. |
| sendMessage | Boolean | Sends a message to the destination agent. |
| getMailbox | Integer | Get target agent's mailbox. |
| Mailbox | consumeNextMessage | String | Returns the next message in the box on a FIFO bases. |
| hasMessages | Boolean | Check if there are any messages in the mailbox. |
| checkTimestamp | Boolean | Check timestamp of all messages in a mailbox. |
| Messaging System | login | String | Logs in a user given an agent ID and key. |
| registerLoginKey | Boolean | Takes a login key and agent ID such that when an agent with that id tries to login she will only be allowed access if the key also matches. |
| getSessionKey | String | Method to generate a session key. |
| sendMessage | String | Sends a message from the source agent to the target agent. Creates a message object and stores it in the target agent's mailbox. |
| checkMessage | String | Check if passed string contains an element from the blocked words. |
| logout | Void | A method to log out the passed user. |

## Test Coverage

As one can see below, through the series of tests that were written, we got 100% class, method and line coverage from all classes. The only exception was the class ‘JohnDoe’ whose unit tests were written in task 2 of this assignment and will therefore be seen later on in the report.

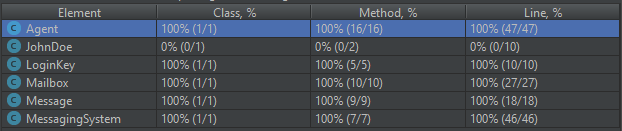


Figure : Coverage Analysis

## Test Patterns

The type of test pattern used in the system is Dependency Injection so that dependent components were injected into test objects. Parameter injection, constructor injection and setter injection were implemented and later on tested through unit testing. An example of each injection can be seen below:

### Constructor Injection

Variables were passed to the test objects via the constructor when the object is instantiated for the first time. The example provided is from the Message class.

1. // Constructor
2. **public** Message(Agent sourceAgent, Agent targetAgent, **long** timestamp, String message) {
3. **this**.sourceAgent = sourceAgent;
4. **this**.targetAgent = targetAgent;
5. **this**.timestamp = timestamp;
6. **this**.message = message;
7. }

### Setter Injection

Private variables are set through a method which can be called at any point during runtime. The example provided is from the Mailbox class.

1. // Getters and setters
2. **public** String getAgentID() {
3. **return** agentID;
4. }
5. **public** **void** setAgentID(String agentID) {
6. **this**.agentID = agentID;
7. }

### Parameter Injection

Variables are passed as a parameter to the method being called. Therefore, in this case, if a test object requires the use of a variable from outside the object itself, it can be passed when the method is called. The example being provided is from the Agent class.

1. // Initiate contact with a supervisor to get a login key and subsequently logs into the system
2. **public** **boolean** contactSupervisor(Supervisor supervisor){
4. // Contact the supervisor
5. **this**.key = supervisor.getLoginKey(**this**);
7. // Return response
8. **if**(**this**.key != **null**){
9. **return** **true**;
10. }
11. **else**{
12. **return** **false**;
13. }
14. }

## Test Doubles

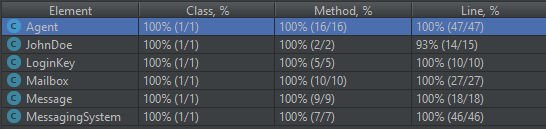
Test doubles are objects which are installed in place of the real component for express purpose of running a test. During almost all tests performed, the use of mock objects was very essential. This is because at this stage, the system still did not have an implementation of a supervisor, only an interface. The mocking framework that was used in this project was Mockito since it is a powerful tool which enables the dynamic creation and management of mock objects. An example of how Mockito was used to mock a supervisor is:

1. // Properties
2. **private** **final** String VALID\_KEY = "0000000000";
4. @Mock
5. **private** Supervisor supervisor;
6. @InjectMocks
7. **private** Agent agent;
9. @Before
10. **public** **void** setup() {
11. StaticVariables.Erase();
12. agent = **new** Agent("001", "Ryan");
14. // Initiate mockito
15. MockitoAnnotations.initMocks(**this**);
16. }
18. @Test
19. // Testing contactSupervisor method in agent with valid ID
20. **public** **void** testContactSupervisorValidID(){
21. // Specify the return of the method without even having an implementation
22. when(supervisor.getLoginKey(agent)).thenReturn(**new** LoginKey(VALID\_KEY, System.currentTimeMillis()));
24. // Exercise
25. **boolean** check = agent.contactSupervisor(supervisor);
26. assertEquals(**true**, check);
27. }

# Cucumber and Automated Web Testing

## Web Application

The web application was built by using Java Server Pages (JSP) for the front end. These JSP files would then communicate to servlets whose aim is to handle HTTP requests from the web application. Finally, these servlets would communicate accordingly with the classes developed in task 1 of this assignment. At this stage, an implementation of the Supervisor interface mentioned in task 2 was developed and unit tests to test the methods in the class were written.



The JohnDoe unit tests passed successfully and yielded a 100% for both class and method coverage, and a 93% for line coverage. The reason that the tests did not result in 100% line coverage is because in the Supervisor implementation, the method ‘getLoginKey’ return a unique 10 character login key. Therefore the possibility of generating a non-unique login key is very low, thus the tests did not manage to enter the else statement of the method. This can be shown in the code below.

1. // Method to generate a session key
2. **public** String key(**int** counter) {
4. Random random = **new** Random();
5. String sessionKey = "";
7. // Generate a session key according to the length passed to the method
8. **for** (**int** i = 0; i < counter; i++) {
9. sessionKey += alphanumerical.charAt(random.nextInt(alphanumerical.length()));
10. }
12. // Check if login key is unique
13. **if**(StaticVariables.keys.contains(sessionKey)){
14. **return** key(counter);
15. }
16. **else**{
17. **return** sessionKey;
18. }
19. }

## Assumptions

Some assumptions that were taken to make the website work are the following:

* Messages did not enter an agent’s mailbox on their own, meaning, that the agent needed to press the ‘Get Next Message’ button which returned the next valid message in the mailbox;
* An agent’s session did not terminate automatically. The system check if the user’s session is valid every time the user interacts with the system;
* A login key can only be used once, even if it still has not yet expired;

## Web Testing

# Model-Based Testing

We designed a Finite State Diagram and then implemented it in classes for the system to work using model-based testing.

# Performance Testing