CPS3230 Fundamentals of Software Testing

Assignment

Miguel Dingli (49997M)

Mark Said Camilleri (306697M)

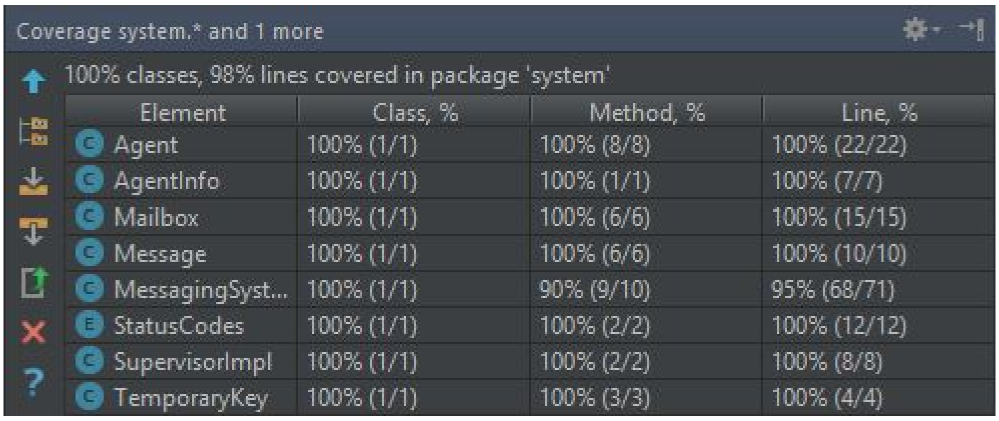
Table of Contents

Task 1 – Unit Testing and Test Driven Development

1.1 Description and Evaluation of the Testing Strategies Used

For this task, the system design was used in creating unit tests for a skeletal system. After making sure that the tests were a good reflection of the system design, the system itself was implemented.

The test suite implemented has an average statement score of above 99%. A breakdown of this is provided in Figures 1.1 and 1.2 (the third column represents the statement score). Analysis of the untested lines showed that this was a default constructor that is not used for testing, however it makes use of the same constructor used for testing, and a return of an error that is theoretically unable to happen. Note that `SupervisorImpl` was a pat of the additions for Task 2.

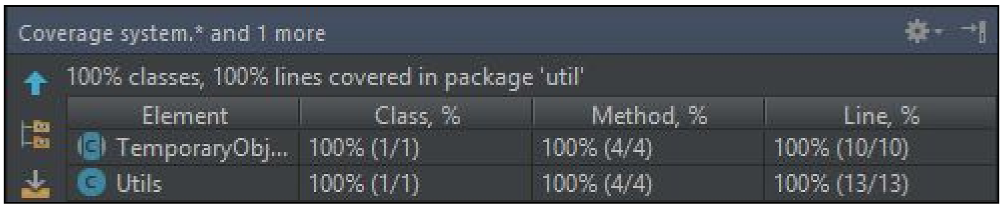


Coverage system.* and 1 more

100% classes, 98% lines covered in package 'system'

Element	Class, %	Method, %	Line, %
Agent	100% (1/1)	100% (8/8)	100% (22/22)
AgentInfo	100% (1/1)	100% (1/1)	100% (7/7)
Mailbox	100% (1/1)	100% (6/6)	100% (15/15)
Message	100% (1/1)	100% (6/6)	100% (10/10)
MessagingSyst...	100% (1/1)	90% (9/10)	95% (68/71)
StatusCodes	100% (1/1)	100% (2/2)	100% (12/12)
SupervisorImpl	100% (1/1)	100% (2/2)	100% (8/8)
TemporaryKey	100% (1/1)	100% (3/3)	100% (4/4)

Figure 1.1: Test coverage analysis for the system classes' unit tests.



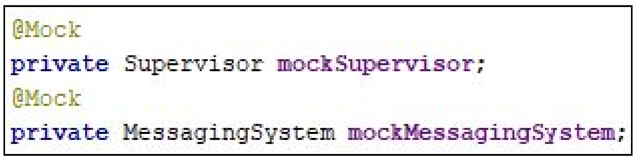
Coverage system.* and 1 more

100% classes, 100% lines covered in package 'util'

Element	Class, %	Method, %	Line, %
TemporaryObj...	100% (1/1)	100% (4/4)	100% (10/10)
Utils	100% (1/1)	100% (4/4)	100% (13/13)

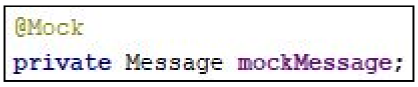
Figure 1.2: Test coverage analysis for utility classes' unit tests.

The implemented unit tests make use of mocking with Mockito. We created mock objects of classes that are not under test so as to be able to control their output from within the test suite. This ensures that our tests are not subject to any possible misbehaviours of other classes and allows for the isolation of problems in the system. If a test fails, then it should only be due to the specific method being tested and with the conditions tested for. Some examples of mock objects are shown in Listings 1.3, 1.4 and 1.5.



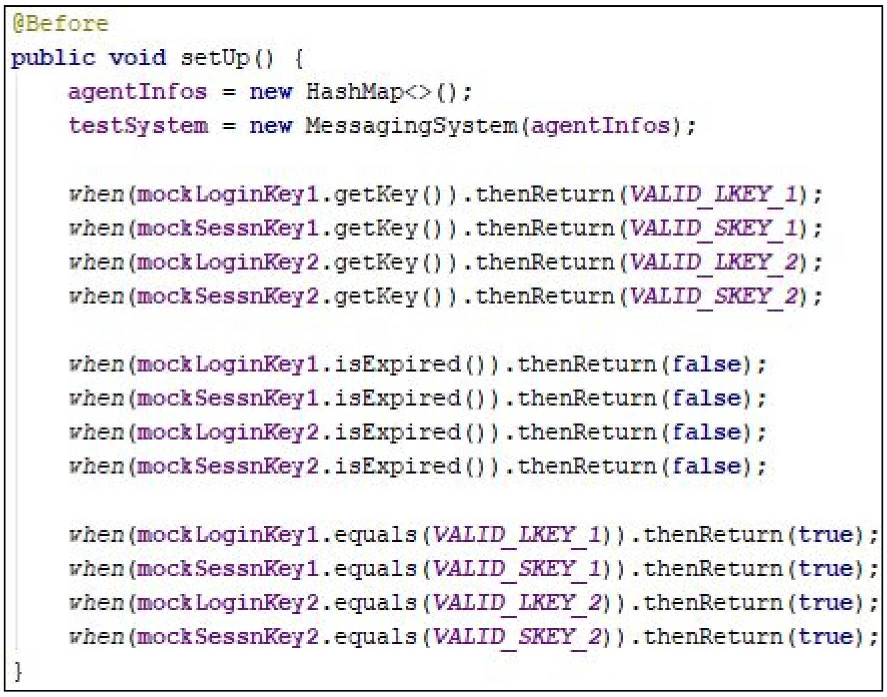
```
@Mock
private Supervisor mockSupervisor;
@Mock
private MessagingSystem mockMessagingSystem;
```

Listing 1.3: Mock objects in `AgentTest`.



```
@Mock
private Message mockMessage;
```

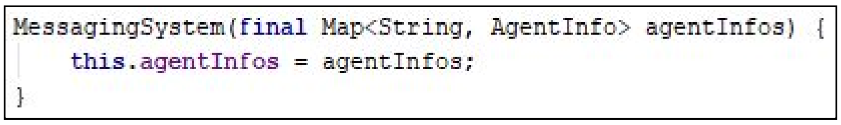
Listing 1.4: Mock object in `MessagingSystemTest`.



```
@Before
public void setUp() {
    agentInfos = new HashMap<>();
    testSystem = new MessagingSystem(agentInfos);

    when(mockLoginKey1.getKey()).thenReturn(VALID_LKEY_1);
    when(mockSessnKey1.getKey()).thenReturn(VALID_SKEY_1);
    when(mockLoginKey2.getKey()).thenReturn(VALID_LKEY_2);
    when(mockSessnKey2.getKey()).thenReturn(VALID_SKEY_2);

    when(mockLoginKey1.isExpired()).thenReturn(false);
    when(mockSessnKey1.isExpired()).thenReturn(false);
    when(mockLoginKey2.isExpired()).thenReturn(false);
    when(mockSessnKey2.isExpired()).thenReturn(false);

    when(mockLoginKey1.equals(VALID_LKEY_1)).thenReturn(true);
    when(mockSessnKey1.equals(VALID_SKEY_1)).thenReturn(true);
    when(mockLoginKey2.equals(VALID_LKEY_2)).thenReturn(true);
    when(mockSessnKey2.equals(VALID_SKEY_2)).thenReturn(true);
}
```

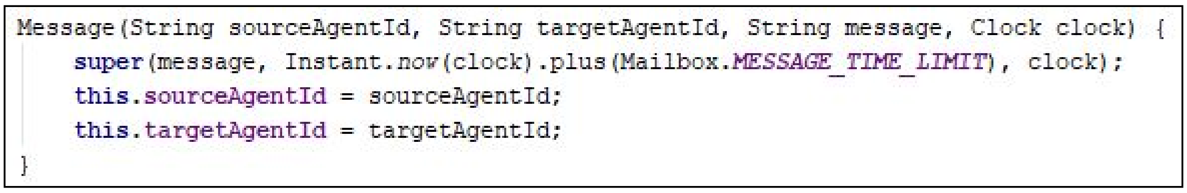
Listing 1.5: Setting up the mock object's outputs in `MessagingSystemTest`.

In the implemented test suites, we also used dependency injection by designing constructors that accept dependencies as a parameter, as shown in Listings 1.6 and 1.7. This allowed the unit tests to input custom dependencies into the class being tested to further ensure that we are only testing the class in consideration and to have more control over its inner workings. We did not see the need to use dependency lookup in the testing.



```
MessagingSystem(final Map<String, AgentInfo> agentInfos) {
    this.agentInfos = agentInfos;
}
```

Listing 1.6: Dependency Injection of `AgentInfo` map in `MessagingSystem`



```
Message(String sourceAgentId, String targetAgentId, String message, Clock clock) {
    super(message, Instant.now(clock).plus(Mailbox.MESSAGE_TIME_LIMIT), clock);
    this.sourceAgentId = sourceAgentId;
    this.targetAgentId = targetAgentId;
}
```

Listing 1.7: Dependency Injection of `Clock` in `Message`

In the `TemporaryObject` test suite, a test stub was used. The `TemporaryObject` class is a container for an object that should be deleted after a set amount of time. To test the various timing criteria of the system, there was either the option of waiting for the specific amounts of time (and delaying our tests which could take long and does not scale that well), or extending the `Clock` class and injecting this into the `TemporaryObject`. By choosing the latter option, the result was a `StepClock` class. At each call of `instant()`, `StepClock` outputs sequential ticks that are "*X*" time units apart, with *X* being specified on creation of the `StepClock` object. This could be seen as akin to bending time, since it allows us to simulate a long duration. The code for `StepClock`, defined in `TemporaryObjectTest`, is provided in Appendix A.

1.2 Assumptions and Important Changes to the Design

In the `Agent` class, the agent's name was removed as it was assumed to be suitable to refer to an agent just by his ID. The `Agent` also keeps track of his own login and session keys, which he uses when any of the his methods (register, login, and sendMessage) are invoked.

The `AgentInfo` class is a new class which is used by the messaging system to keep track of the login and session keys generated for an agent, which it checks against when the agent attempts to register, login, send a message, etc using the keys stored in Agent. This class also stores a reference to the agent's mailbox and the number of messages sent and received by the agent, so that the agent does not send or receive above the quota.

The `MessagingSystem` keeps a map from agent IDs to `AgentInfo`s to keep track of registered agents. A `logout(agentId)` method was also added to give the messaging system the ability to logout a user, such as when more than 25 messages have been sent, or also for when the user decides to logout manually.

Due to the requirements that some objects are only valid for a specific time period, such as the login key and messages, the `TemporaryObject` class mentioned in Section 1.1 was created. The two classes `TemporaryKey` and `Message`, both of which are meant to store temporary strings, extend this class. The `TemporaryObject` class essentially checks whether the generic temporary object that it holds has expired, in which case it sets it to `null` as soon as an attempt is made to access the expired object. The new `TemporaryKey` class is used in `AgentInfo` as the login and session keys, so that when these expire, the agent is not allowed to login, or use the messaging system, respectively.

Task 2 – Cucumber and Automated Web Testing

2.1 Supervisor Test Double

For this to task to be achieved, an implementation was needed for the Supervisor. This lightweight implementation acts as a test double and it was built according to the provided specifications such that all agents are accepted as long as their id does not start with “spy-”.

2.2 The Web Interface

The web interface was intended to provide a simple front end to the messaging system. This was done with the use of Jetty [1], a lightweight java web server and a servlet container.

The interface first shows a registration screen, where the agent enters their ID for the supervisor to issue them a login key, if it is deemed safe to do so. Figures 2.1 and 2.2 below show the registration page as it is shown to the user. Figure 2.2 also showcases the message that appears when the supervisor decides that it is unsafe to login. In this case, this is when the ID starts with “spy-”.

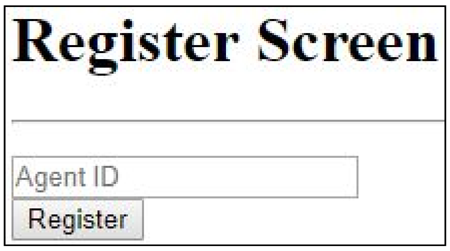
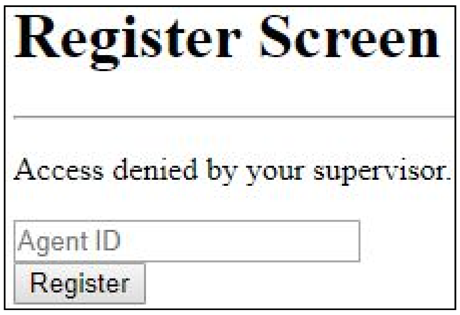


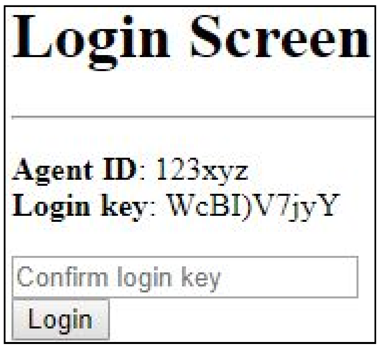
Figure 2.1: Registering


Register Screen

Access denied by your supervisor.

Figure 2.2: Registering as a spy.

Figure 2.3 below is a screenshot of the next step the agent has to take once successfully registered. The agent is presented with a login key given by the supervisor, which they then use to login. Figure 2.4 showcases the message displayed when either the login key was entered incorrectly or it has expired.



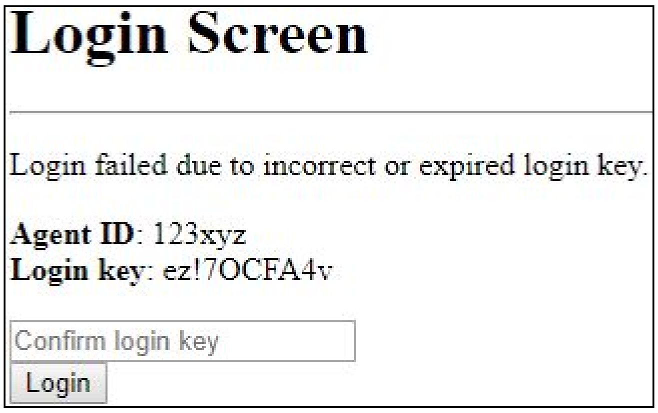
Login Screen

Agent ID: 123xyz
Login key: WcBI)V7jyY

Confirm login key

Login

Figure 2.3: Login screen.



Login Screen

Login failed due to incorrect or expired login key.

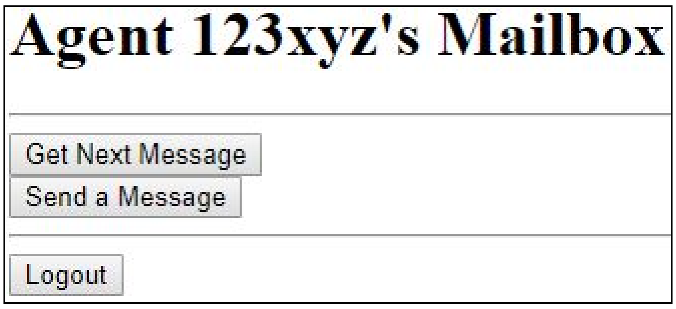
Agent ID: 123xyz
Login key: ez!7OCFA4v

Confirm login key

Login

Figure 2.4: Login screen when an incorrect login key was input by the agent, or the login key was expired.

Once the agent has logged in, they are presented with the option to send a new message to another agent, or to read and consume any messages in their mailbox. The user can also log out with a convenient button in this page as shown in Figure 2.5.



Agent 123xyz's Mailbox

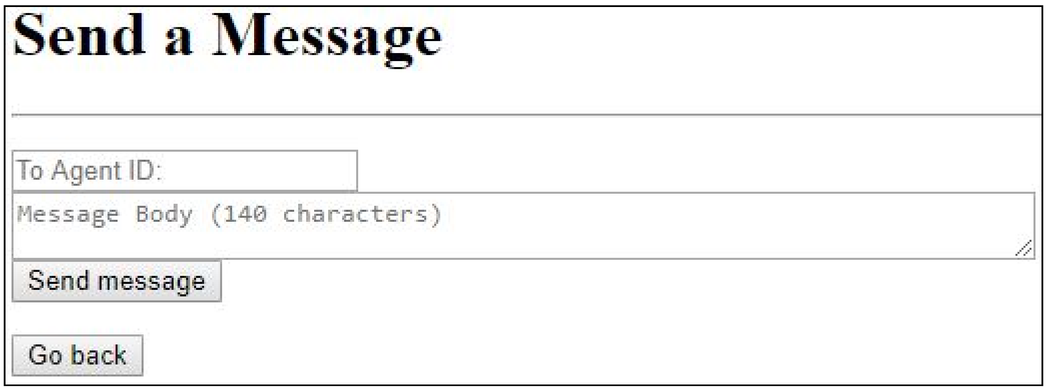
Get Next Message

Send a Message

Logout

Figure 2.5: Mailbox screen after successful login.

To send a message, the agent is limited to 140 characters which is indicated in the message form. Figure 2.6 shows the form as the agent would initially see it. Figures 2.7, 2.8 and 2.9 depict what the same page shows with a status message added, which is displayed after the user has attempted to send a message. Figure 2.7 depicts a successful send. Figures 2.8 and 2.9 show what happens if the user sends a message to a non-existent agent or when the message is too long, respectively.



Send a Message

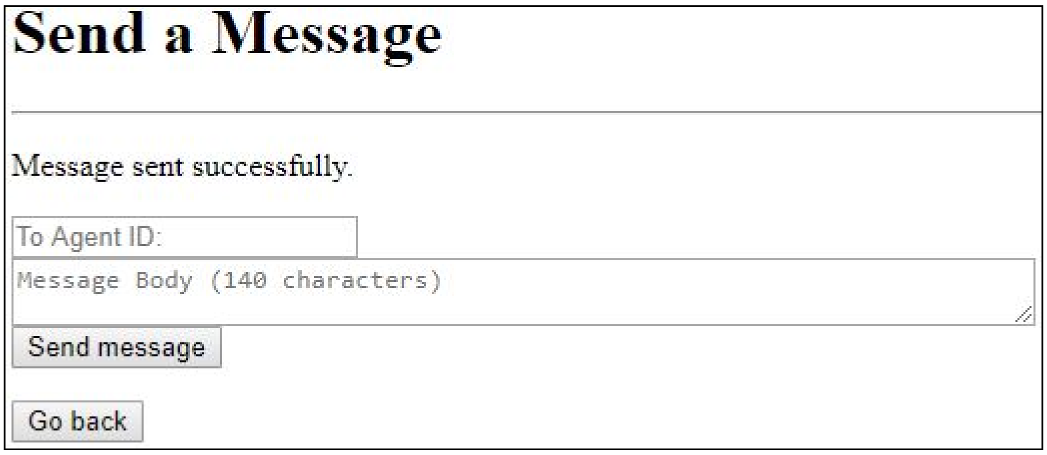
To Agent ID:

Message Body (140 characters)

Send message

Go back

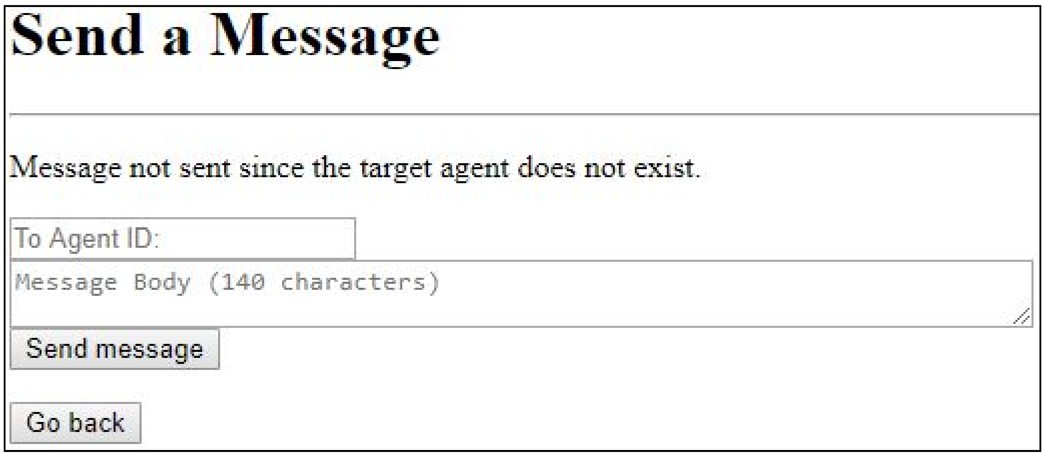
Figure 2.6: Sending a message.



Send a Message

Message sent successfully.

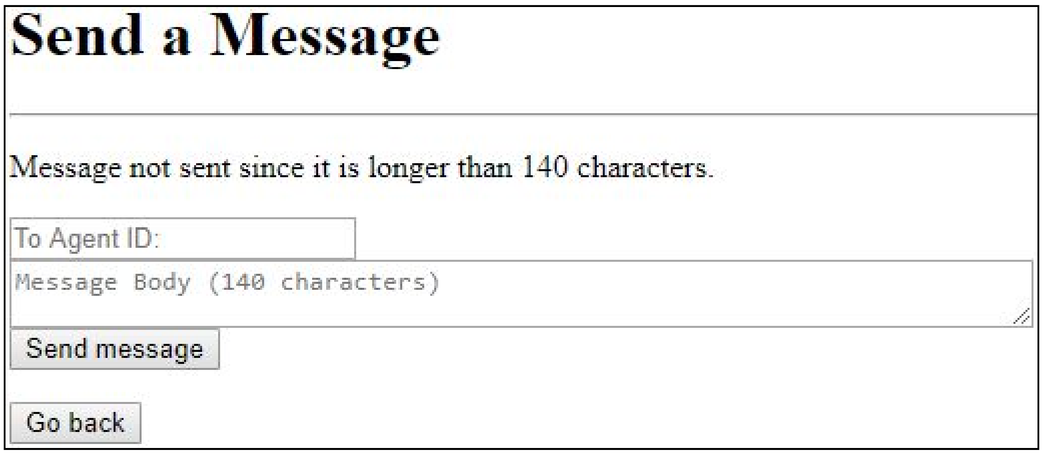
Figure 2.7: Message sent successfully.



Send a Message

Message not sent since the target agent does not exist.

Figure 2.8: Message not sent when the target does not exist.

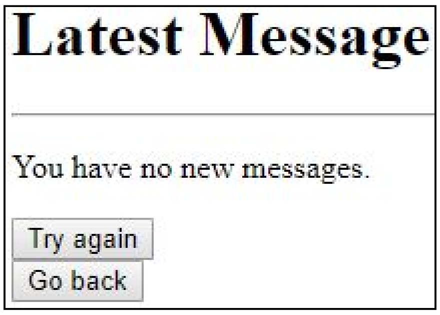


Send a Message

Message not sent since it is longer than 140 characters.

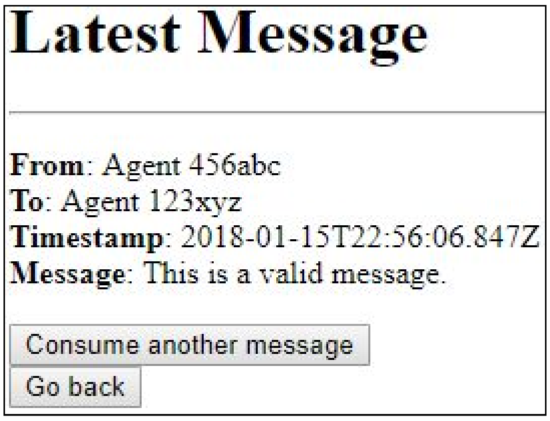
Figure 2.9: Message not sent when the message is over the limit.

The message consumption screen displays the earliest message that was sent to the agent. This message is immediately deleted from the mailbox queue. Should the agent not have any messages, then a screen like in Figure 2.10 will be shown to the user. Otherwise, the received message would be displayed as shown in Figure 2.11.


Latest Message

You have no new messages.

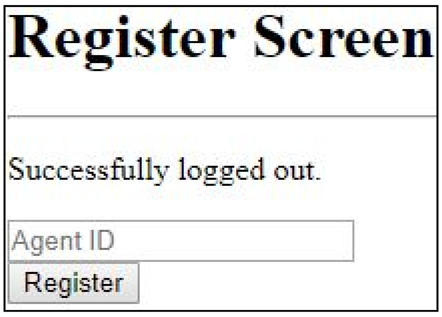
Figure 2.10: Reading a message when the mailbox is empty.


Latest Message

From: Agent 456abc
To: Agent 123xyz
Timestamp: 2018-01-15T22:56:06.847Z
Message: This is a valid message.

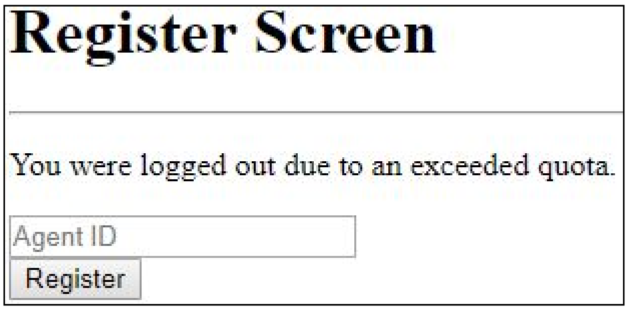
Figure 2.11: Reading a message when the mailbox is not empty.

There are three ways a user can be logged out of the system. The first is when the user voluntarily logs out, upon which they are greeted at the register screen as shown in Figure 2.12. If they were logged out automatically by the system due to sending too much messages, they are shown the register screen as depicted in Figure 2.13. Finally, if they were logged out automatically due to receiving too much messages or due to a session key expiry, then they are shown the register screen like in Figure 2.14.


Register Screen

Successfully logged out.

Figure 2.12: Logging out manually.


Register Screen

You were logged out due to an exceeded quota.

Figure 2.13: Source agent logged out due to an exceeded sending quota.

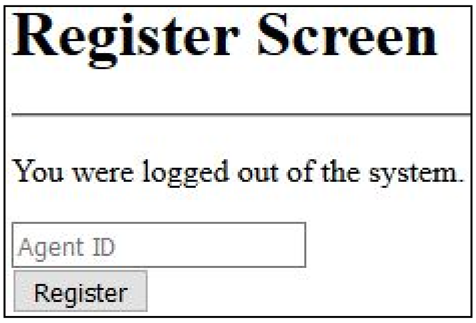


Figure 2.14: Agent logged out due to receiving too many messages or an expired session key.

2.3 Automated Web Testing

To test this system, Selenium was used to interact with a web browser (we chose Google Chrome), and Cucumber to split our tests into clauses that allowed us to reuse test code. This also helps make test creation more efficient.

To be able to test the web server, chromedriver [2] must be present in the project root folder. It has not been provided with the artefact since it is platform dependant. The web server must first be run manually by calling `webapp.StartJettyHandler.main()`, after which the tests can be run.

The cucumber test scenarios used were those given in the Assignment specifications, however a change was made to Scenario 3. Before changes, Scenario 3 was as follows:

```
Given I am a logged in agent
When I attempt to send 25 messages
Then the messages should be successfully sent
When I try to send another message
Then the system will inform me that I have exceeded my quota
And I will be logged out
```

The last clause, `I will be logged out`, was changed to `I should be logged out` which matches the clause in Scenario 5, shown below:

```
Given I am a logged in agent
When I click on “Log out”
Then I should be logged out
```

As a result of using the same clause, the same code is used to check if the agent is logged out. A benefit of this would be that should there be a need to change how this is done, one must only change it once and all tests that use this clause will be affected. Also, should a new test need to be created using clauses already implemented, creating tests takes less time and tests are easier to read.

Task 3 – Model-Based Testing

3.1 Description of the Model

The model used for these tests serve as a mathematical likeness to the specifications of the web app as the user should experience it. The transitions of this model represent the actions the user may take when at a particular state in the web app, whereas the states represent where in the web app the user is. ModelJUnit [3] was used to mechanize this model, which is shown in Figure 3.1. Like in Task 2, the server needs to be running before running this test.

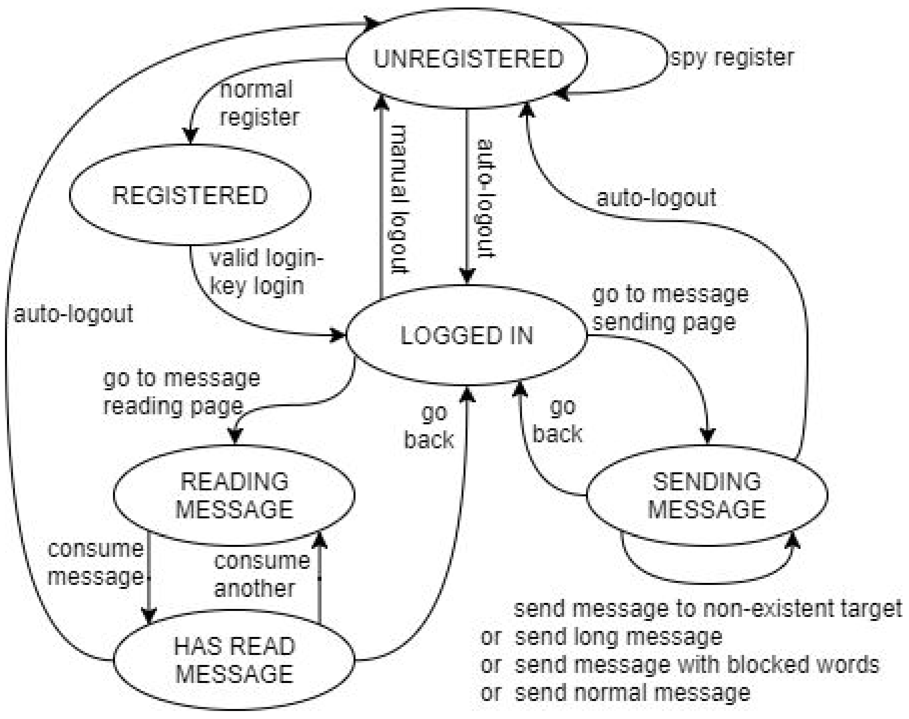


Figure 3.1: Model of the Web Application

3.2 Issues Discovered

During testing, an issue with sending a message was discovered. If the user was automatically logged out in the backend while on the message composition screen, after typing and sending the message, the method tied to POST request of the form was not checking that the user was still logged in. As a result, the message would appear to have been sent either way. This behaviour was corrected and the tests were all successful, such that the model tests ran for 15 minutes without crashing.

3.3 Results

The model tests were set to run for 15 minutes in succession. Coverage metrics for the last run of the model test were recorded as follows:

- action coverage: 15/15
- state coverage: 6/6
- transition coverage: 21/21
- transition-pair coverage: 80/80

References

[1] jmcconnell, 'Jetty - Servlet Engine and Http Server'. [Online]. Available: https://www.eclipse.org/jetty/. [Accessed: 16-Jan-2018]

[2] 'ChromeDriver - WebDriver for Chrome'. [Online]. Available: https://sites.google.com/a/chromium.org/chromedriver/. [Accessed: 16-Jan-2018]

[3] ducnh, 'Model-Based Testing with JUnit', *SourceForge*. [Online]. Available: https://sourceforge.net/projects/modeljunit/. [Accessed: 17-Jan-2018]

Appendix A - StepClock in TemporaryObjectTest

```java
private class StepClock extends Clock {

   private final Instant baseTime;
   private final Duration step;
   private int stepMultiplier = 0;

   StepClock(Instant baseTime, Duration step) {
       this.baseTime = baseTime;
       this.step = step;
   }

   @Override
   public ZoneId getZone() {
       return ZoneId.of("UTC");
   }

   @Override
   //Ignored
   public Clock withZone(ZoneId zoneId) {
       return this;
   }

   @Override
   public Instant instant() {
       return baseTime.plus(step.multipliedBy(stepMultiplier++));
   }
}
```