Final Individual Report

G53SQM coursework by Harry Mumford-Turner

Contents

1.0 Introduction & Related work

Testing software quality relies on testing against an agreed set of standards that are globally used across organisations and the software engineering life cycle.

IEEE is an organisation that creates leading industry standards, produced a set of standards to test software quality. They state a test plan is needed with the follow levels to cover a wide range of test areas. Unit Test Plan, Integration Test Plan, System Test Plan, and Acceptance Test Plan. Using this standard ensures a high-quality approach to testing that can be communicated to other developers.

Each part of the software needs to be thoroughly tested. Strategic testing plans will be constructed to outline every aspect of testing each part of the software. They will be based on the 'IEEE 829 Standard for Software and System Test Documentation' [1], to cover the most test coverage and measure the quality of the software.

2.0 Server Analysis - Part 1

2.1 Approach to testing

Testing plan

The first level of testing will be Unit testing that will concentrate on each method of current server software. JUnit [2] will be used with Eclipse [3] to achieve this because it is lightweight and fast to write tests.

This will be followed by Integration testing that will focus on inputs and outputs to try to see how well the architecture design of the Server copes by performing lots of possible actions.

Setup

The code was downloaded from the GitHub [4] repo, the contents extracted and the Readme file analysed. Following the instructions in the Readme file, the chatServer.jar file was tested to see if the instructions were correct, using the command `java -jar chatServer.jar '.

Using first thoughts and intuition the scope of testing the software was realised. The available commands that were meant to be used from a client were extracted from README file. Telnet (one of the recommended clients) was used to connect to Server and run the commands `telnet localhost 9000'.

Exploration and prior knowledge of chat servers was needed to define each command as no descriptions were given in the Readme file, for each command. (see Appendix 6.1)

Software States

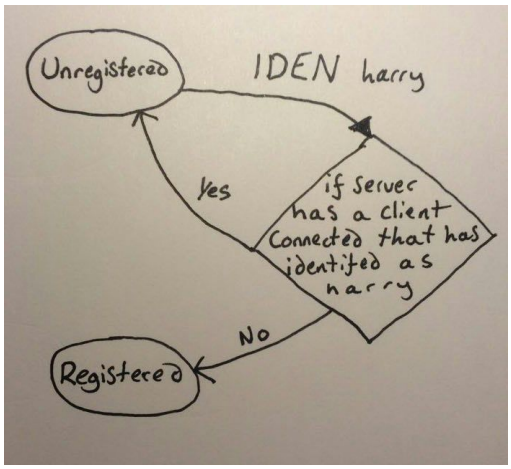
During this exploration stage 2 client states were discovered, where each state provided different server responses.

State Unregistered / Not logged in
The default state of the client when they first connect to the server. All the commands have restricted actions/responses all of which hints towards the user needing to identify themselves and logging in.
Running command `IDEN harry`, moves to the client to the Logged in state.

State Registered / Logged in
The state in which all commands perform their action and show the output detailed above, although the IDEN command can only be used once for each connection.

The expected response of the server for both states is detailed (see Appendix 6.2).

State transition diagram
The path to connect the states is visualized below in a state transition diagram.
The diagram shows the only way to move to the Registered state is by running the command `IDEN` with a username that doesn't belong to another connected identified client.



ChatServer state transition diagram

Chat Server methods
A bottom up approach to testing was taken so bugs are more easily found in the software. Unit testing is used to test the individual methods of the classes. The documentation of the individual method calls are scare, so descriptions are realised and added. (see Appendix 6.3)

Proposed Unit tests
Equivalence partitions are established to reduce the time cost of testing all possible combinations of return and parameter values for the method calls. (see Appendix 6.4)

Unit testing
Unit tests were difficult to add to this software due to the constraints in the architecture. When the Server class is initiated an infinite loop is started that handles incoming client connections. This means unit testing the Server.java class methods and the Connection.java class methods is difficult because we cannot exit this loop without stopping the chat server process, so the Server and Connection methods cannot be directly called.

In order to test the chat server methods integration testing on the chat server commands is used to indirectly test the method calls. The table below pairs up the chat server commands with the Server and Connection method calls. To aid in identifying the method calls each command ran, print statements were added to the start of every method call in Server.java and Connection.java. (see Appendix 6.5)

Integration testing
Integration tests are used with the help of JUnit and eclipse.
To remove human timing errors a client test harness was created so clients can be programmatically created with multiple commands programmatically issued.

Client test harness
The client test harness can run a series of commands and compare the output. This guarantees the integration tests are run the same every time. (see Appendix 6.6)

During the end of the integration testing modifications to the ChatServer were needed to gracefully stop the Server. (see Appendix 6.7)
The modifications allow the Server's while loop to run in it's own thread so it is possible to exit from it when a volatile variable is changed.
This makes it easy to stop and start the Chat server programmatically and results in the integration testing completely automated. Although this isn't the best way to stop the server it was the fastest method to make the server testable.

2.2 Testing Data and Results
The test data was divided up into 3 separate parts based on the different server responses and commands.
The registered client state, unregistered client states and each possible server command, created a logical divide for the Integration tests.

JUnit was used to write integration testing with each JUnit class independent of one another. Each server command could be separately tested in it's own class in it's registered state and unregistered state using this approach. (see Appendix 6.8)

Each server command was worked through while comparing what Server methods would be called when using this command and using the suggested valid/invalid unit test equivalence partitions for those methods, if they were exposed enough.

2.3 Errors and Improvements

Suggested improvements made to the server have been detailed with the steps to reproduce detailed. (see Appendix 6.9.1)

Command feedback

Each command didn't provide much user feedback if entered incorrectly, with commands such as 'HAIL hello' in the UnRegistered state telling the user they haven't logged in, but not telling them *how* they can log in.

Stress testing

A stress test was conducted to test the number of clients the server could handle at once. An OutOfMemoryException was thrown when just over 1000 clients were connected. This was not an error, but a machine testing limitation due to the number of threads the operating system has allowed the Java Virtual Machine to use. A machine with large amounts of memory could allow more clients to be connected. So the hardware of the machine the server is run on, is directly linked with the specification for the number of clients that can connect at once.

Server assumptions

Using intuition and investigation, several core methods of the server were found to rely on lots of assumptions about the list of Connection objects. The IDEN command uses Server.doesUserExist() that only works if the user they are matching against is last in the ArrayList of Connections in the Registered state.
This is dangerous when changes are added to the server, because other software engineers might not be aware of these assumptions as they are not implied by the code. Unit tests could highlight this assumption and make it easier for new developers to understand the codebase.

Lowercase Commands

The parsing of the server is very strict not even allowing lowercase commands (e.g. stat) to be entered. This would be a small addition would make the user experience better because they wouldn't have to keep switching to and from capital letters.

Command Parameters

When running commands that require parameters, e.g. HAIL, MESG or IDEN, writing a wrong or invalid parameter throws an error, rather than gracefully giving user feedback on how to run the command.

IDEN usernames

The tests to the server using invalid inputs did not throw errors, although the IDEN <username> command can run with a space as a username. This becomes problematic when trying to private message this user using MESG <username> <message>, because a blank space separates the parameters, an error is thrown when running 'MESG hi'.

Usernames are also split by a space that means users can run the command like ‘IDEN harry mumford-turner’ and get different output than they expect ‘OK Welcome to the chat server harry’.

HAIL Consistency
A new line is sent after a broadcast command. This is unusual as it breaks the consistency with every other command pattern and requires additional tests to check this.

Command Consistency
The commands of the server did not match a certain standard, meaning exploratory experimentation was needed to understand each command. Perhaps changing the commands to a set of standard chat server commands would mean developers could understand the software more easily.

Layout Consistency
Typos and inconsistent layout from brackets to operands was discovered (e.g. function{, function {, a+b, a + b). This should be changed to match a code style guide so developers reading the code can do so more easily.

Errors
Several concurrency errors were found with the most noticeable when QUIT was called in quick succession with multiple clients. A server ConcurrentModificationException error was thrown because the server quit command was not thread safe, a Connection item was trying removed from the Connections ArrayList while another client is iterating around it.
Possible solutions to this are synchronized locks around iterators and remove commands. So it is only possible to remove an item from the ArrayList when a client isn’t iterating around it. More errors are detailed with possible solutions in results. (see Appendix 6.9.2)

Limitations to my testing
Due to the lack of Unit Tests, all areas of the server methods cannot be tested, but the integration tests provide enough indirect testing to cover most areas that can be reached.

2.3 Server Metrics Analysis

Metrics were testing on the common software measures using the Eclipse plugin ‘Metrics Feature’ [6] that generates a large range of metrics about the server. (see Appendix 6.10).

Comparing some of these metrics generated with metrics of the tests, showed the Total lines of code for the Server (318), was just under three times the size of the integration tests Total lines of code (925). This shows that writing tests impacts the time cost to writing software.

The Weighted methods per Class combined with McCabe's Cyclomatic Complexity [7] metrics show that Connection.java is the most complex class in the Server. Looking further into the Complexity of the methods of Connection.java, the metrics show that validateMessage() method has the highest complexity. This is probably due to the switch statement that select which method to run, based on the command entered.
The integration tests for the server showed several errors that occurred within validateMessage() which proves that metrics like this are useful to identify where to spend the most time on testing and improving code.

As detailed above the Server has been split up into two parts, with the while loop extracted into it's own class. This means the Cyclomatic Complexity for the previous Server class has been split up into two parts. The metrics show the while loop class has a higher complexity than the rest of the Server class, thus reducing the individual complexity of each proving that the changes made reduced the complexity of the server.

3.0 Client Development - Part 2

3.1 Approach to development

When testing the Server, it was clear that it was not developed with thought to testing. Changes needed to be made in order for it to be programmatically tested. This changes the approach to developing the client and Test Driven Development [8] will be used during the development along with Travis CI [9], Continuous Integration testing. This will mean writing tests for the client will not happen after the development of the software, but during. This also means any changes to the client can be made fearlessly because everything is tested so we know it works.
Github will be used with Maven [10] a package manager for dependencies, in order to work with Travis and provide a cloud version control.

Unit Testing and Integration testing will be written alongside the client development as detailed below.

3.2 The Implementation

The client code uses Java swing to create the GUI elements. When the client is launched these elements are instantiated, a Start button and a Send button have click handlers attached that listen to presses on the buttons.

When the start button is pressed, a click handler is called that runs a method 'connectToServer()', this creates a new ClientStub that interacts with the server and saves the responses so the GUI can retrieve them.

When the send button is pressed, text is retrieved from a message box that gets sent to the server. The server last response is extracted and appended at the end of the text area.

Before the command is sent to the server it's first validated by the validCommand(cmd) method that checks to make sure the command entered follows some of the basic server rules. This check occurs so less server processing is required.

3.3 Approach to testing

Similar to the server testing, client Unit testing is the first level of testing. Because the development of the client use Test Driven Development, Unit tests have already been written and no additional changes required. Again Eclipse and JUnit were used with the addition of Travis CI added for Continuous Integration to verify each commit to our version control passes all the written tests.

Client GUI testing will be written with the assumption that the integration tests for the Server are all valid and succeed. Once the command has reached the server the test has passed, so the cost of testing is separated to the GUI only. FEST [11], a Java Library that abstracts higher than Robot [12] will be used to test the actions of the GUI.

Further testing will be conducted afterwards, to provide assurance that the software meets all functional and behavioral requirements. This will be tested through user testing to evaluate the final GUI and give direct input on how users use the system.

3.4 Testing Data and Results

Unit Tests

The client implementation had very little methods, resulted in a small amount of test data. Because the client had been written with testing in mind, the test results were uninteresting and when comparing these results to the Server unit tests, if the developer had created that with testing in mind, perhaps not all test edge cases would've been found. A solution to test more edge cases, would be to get another developer to write tests for the client.
(see Appendix 6.11)

GUI Tests

Unit tests were combined with the Java Library FEST to conduct automated GUI testing. These tests assume the Server integration tests all pass, so this is testing the GUI elements rather than the server responses, which is why the test data is reduced. (see Appendix 6.12)

Travis

Both GUI tests and Unit tests were pushed to Travis to verify tests work for each commit. This has been used in conjunction with pull requests. For experimental features to the client, I branched off and created a pull request on Github, Travis ran tests on the pull request to verify it passes them all. This means I can safely merge the pull request into the main branch with knowledge that it will work. (see Appendix 6.13)

User tests
Sets of conditions were created to determine if the client GUI features were working as intended. These conditions were given to another developer to get their take on the conditions and to see if their test results matched.

3.5 Errors and Improvements

The majority of errors are minimal because Test Driven Development was used, so the software tester is writing their own tests and all realised edge cases were programmed against.

The limitations to the ClientStub code was the mechanism it used to retrieve the server response. The client listens for a server message and overwrite a variable as the last response. This meant some loss of data with the HAIL command, as it returned the expected response, followed by a new line. A solution was used to store an array of responses, but this wasn't mapped to the GUI because the edge case was only highlighted in user testing.

Another limitation to the client was the use of Thread.wait, before retrieving a server response. This wait time was adjusted so it could guarantee the server has processed the message and returned a response before that response was tested against. A more event based model would be more appropriate and would be guaranteed to catch every response.

User Testing Results
The user testing highlighted several issues with the order in which a user interacts with the server. An error was found whereby if the user doesn't press the Start button first, instead typing a valid server command then presses the send button, a client error occurs.
These results show the limitations of software engineers who write the code, also writing the tests. Although the software was easier to test because it was developed with testing in mind, some issues were still left uncovered.

3.6 Client Metrics Analysis

Cyclomatic Complexity
Metrics were ran using the same tool as metric testing the chat server for ease and so the metrics of the two parts (server and client) were comparable.
Again using McCabe's complexity values, the results show that the ClientGUI.java method to validate a user entered command (validCommand()) had the highest complexity of any method in the server/client code (16). This is well above the limit McCabe stated 'was a good starting point' and should be refactored to reduce complexity.
This was because the client-side error checking of the user input was high and the server-side checking was minimal. In order to make software that performs complex actions, some parts of the system will be more complex than others. But reducing that complexity by dividing the problem into smaller pieces each with reduced complexity makes it easier to understand.

Nested Block Depth
ClientGUI.validCommand() has a nested block depth of 6 meaning the method has over the standard amount of nested loops (5) [13]. This metric tells us that we should break the method up because it's current state lead to worse readability and more complex solutions.

LOC vs Test LOC
Under the assumption that the server unit tests pass the client unit tests and client GUI tests can be reduced. This is shown by comparing the ClientGUI.java Lines of code (126) with the tests Lines of code, ClientGUITests.java (58) plus ClientUnitTests (70). For the client GUI there is approximately an equal amount of LOC compared to its tests, but when comparing the whole software tests server and client side, the LOC statistic is tripled for all tests. So low LOC for test cases does not result in incomplete testing. [14]

4.0 Discussion - Part 3

4.1 Critical reflection

I had to write Unit Tests for the existing chat server given by this coursework. I spent quite a bit of time trying to blackbox test it in the way that I have seen in the lectures.
After realising I couldn't Unit Test the server because I didn't think I could modify the Server class, I decided to write integration tests to indirectly test each method. I really wanted to start the coursework early and do it really well, so I wrote lots of tests well in advance of the deadline.

Whenever I tried to revisit the coursework, I got frustrated that I was unable to test the server like I had tested other software in the past, my housemates saw that I was worried. I need to think about how this coursework problem could have such an effect on me.
A few weeks later it was highlighted in lectures that modifications of the chat servers was allowed. I felt annoyed at myself that I had missed that from the specification and had spent lots of time writing integration tests that were potentially of no use.
I asked in lectures about this method of testing and if it was an ok method to use. I felt relieved when the coursework was explained to be more about using 'a' method of testing, not a specific one.

I am feeling more positive about courseworks, I need to think about different solutions to problems, analyse the specification to see what I could do better in the coursework.
The change in my attitude towards the coursework was interesting to view after this highlight. I need to think from the beginning of the coursework about the process of testing software.

For the first part of the coursework writing integration tests made me frustrated by the way it was implemented. I wanted to change the Server class in almost every test I wrote. Writing tests for software after it has been developed was cumbersome.
But when I adopted a Test Driven Approach, I could change the implementation when I wrote the test, because the two were conducted in parallel.

This made writing tests interesting because I was searching for every edge case whilst writing the method and when I needed to change the method, I could make fearless improvements safe in the knowledge my tests pass.
I definitely will take the Test Driven Approach with software development in the future.

I have never used any GUI testing tools before and needed to conduct a lot of exploratory research with the different GUI tools outlined in the lectures and others I had found before I started.
This cost a lot of time and I struggled without any prior knowledge. I got discouraged when I had spent lots of time and made little progress.
I asked my friend who was experienced in this area and used his advice on what technology to use. He told me of a more modern library that builds on top of Java Robot, FEST. Next time I will ask advice before trying to experiment myself.

5.0 References

[1] “IEEE Standard for Software and System Test Documentation: 829-2008”, IEEE Foundation.

[2] “Junit,” Junit, [Online]. Available: http://junit.org/.

[3] “Eclipse”, Eclipse Foundation [Online]. Available: http://www.eclipse.org/.

[4] “GitHub,” GitHub, Inc., [Online]. Available: https://github.com/.

[5] Kalt, Christophe (April 2000). Internet Relay Chat: Client Protocol. IETF. RFC 2812. Retrieved 2009-10-30.

[6] “Metrics Feature,” State of Flow, [Online]. Available: http://eclipse-metrics.sourceforge.net/.

[7] “IEEE Transactions on Software Engineering: 308–320”, IEEE Foundation. McCabe (December 1976). "A Complexity Measure"

[8] "Extreme Programming". Computerworld. Lee Copeland (December 2001).

[9] “Travis CI,” GmbH (Travis CI), [Online]. Available: https://travis-ci.

[10] “Maven,” Apache, [Online]. Available: https://maven.apache.org/.

[11] “FEST,” [Online]. Available: https://code.google.com/p/fest/.

[12] “Java.Robot”, Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html

[13] “Nested Block Depth”, Marcio F. S. Oliveira, Ricardo Miotto Redin, Luigi Carro, Luís da Cunha Lamb, Flávio Rech Wagner. [Online]. Available: ftp://ftp.inf.ufrgs.br/pub/simoo/papers/mompes08.pdf

[14] Lines of Code Metric. [Online]. Available: https://en.wikipedia.org/wiki/Source_lines_of_code

6.0 Appendix

6.1 Description of all possible server commands

Command	Explanation
STAT	Displays statistics about the server including the number of connected clients and the number of messages this client has sent.
LIST	Lists all the connected clients that have **identified** themselves using the IDEN command.
IDEN <username>	Identifies your connection on the server as the username.
MESG <username> <message>	Sends a private message to another user on the server.
HAIL <broadcast message>	Broadcasts a message to all users on the server.
QUIT	Removes the client identification and disconnects the client from the server.

6.2 All possible output from both client states

Command	Unregistered State	Registered State
telnet localhost 9000	OK Welcome to the chat server, there are currently 1 user(s) online	n/a
STAT	OK There are currently 1 user(s) on the server You have not logged in yet	OK There are currently 1 user(s) on the server You are logged in and have sent 0 message(s)
LIST	BAD You have not logged in yet	OK harry,
HAIL hello all	BAD You have not logged in yet	Broadcast from harry: hello all
MESG harry hi	BAD You have not logged in yet	PM from harry:hi OK your message has been sent
MESG bob hi *bob is a username that a client hasn't*	BAD You have not logged in yet	BAD the user does not exist

identified with		
IDEN harry	*If a client has already identified with username harry* BAD username is already taken *else* OK Welcome to the chat server harry *Move to Registered state*	BAD you are already registered with username harry
QUIT	OK goodbye	OK thank you for sending 0 message(s) with the chat service, goodbye.

6.3 Each chat server method description

Connection.java

Method	**Description**
Connection()	Setup the connection object values.
void run()	Handles the input and output of the client connected.
void validateMessage(String message)	Validates the message a client sends to the server.
void stat()	Displays statistics about the server including the number of users and number of messages the connected client has sent.
void list()	Lists all the identified users on the server.
void iden(String message)	Identifies the connected client to the server with the given message username.
void hail(String message)	Sends a broadcast message to the server and all connected clients.
boolean isRunning()	True if the connection is still connected to a client and running.
void mesg(String message)	Sends a private message to a user.
void quit()	Disconnects and removes the connected client.

void sendOverConnection (String message)	Sends the message to the client.
void messageForConnection (String message)	Used for private messages to send to a connected client.
int getState()	Gets the current state of the connection, Registered, or UnRegistered.
String getUserName()	Gets the current client's username.

Server.java

Method	**Description**
Server (int port)	Starts a server listening on the given port, creating a new Connection object in a new thread for each new client connected.
ArrayList<String> getUserList()	Creates and returns the list of all registered clients.
boolean doesUserExist(String newUser)	Checks to see if a username has been registered as a client.
void broadcastMessage(String theMessage)	Sends a message to each connected client.
boolean sendPrivateMessage(String message, String user)	Sends a private message to a connected client with the username. Returns true if client exists, false if not.
void removeDeadUsers()	Removes clients that are not running (disconnected).
int getNumberOfUsers()	Gets the Connection ArrayList size.
void finalize()	Close the server socket.

6.4 Proposed Unit Tests

Connection.java methods with either parameter or return value(s)

Method	**Equivalence partitions test cases**
Connection(Socket client, Server serverReference)	**'Socket client' partitions**: **Valid**

	● The output of a ServerSocket accept() method call ● A Socket object **Invalid** ● null **'Server serverReference' partitions:** **Valid** ● A Server object **Invalid** ● null
void validateMessage(String message)	**'String message' partitions:** **Valid** ● 4 uppercase characters of either {STAT,LIST,QUIT} ● 4 uppercase characters of either {MESG,HAIL,IDEN} followed by 1 character or space **Invalid** ● < 4 lowercase characters ● < 4 uppercase characters ● 4 lowercase characters of either {stat,list,mesg,hail,iden,quit} ● 4 characters, all numerical ● empty string ● null
void iden(String message)	**'String message' partitions** **Valid** ● >= 1 alphanumeric characters or spaces ● empty string **Invalid** ● 1 or more ' ' ● null
void hail(String message)	**'String message' partitions** **Valid** ● ' ' ● empty string ● >= 1 alphanumeric characters or spaces **Invalid** ● null
void mesg(String message)	**'String message' partitions** **Valid** ● ' ' ● empty string ● >= 1 alphanumeric characters or spaces **Invalid** ● null
void sendOverConnection (String message)	**'String message' partitions** **Valid** ● ' ' ● empty string ● >= 1 alphanumeric characters or spaces **Invalid** ● null

void messageForConnection (String message)	**'String message' partitions** **Valid** ● ' ' ● empty string ● >= 1 alphanumeric characters or spaces **Invalid** ● null
boolean isRunning()	**Output** Will either be True or False, will be False if client has disconnected.
int getState()	**Output** Will either be 0 or 1, matching to the constants defined. STATE_UNREGISTERED, or STATE_REGISTERED
String getUserName()	**Output** Not null String with: ● No spaces

Server.java methods with either parameter or return value(s)

Method	**Equivalence test partition**
Server (int port)	**'int port' partitions**: **Valid** ● 0 and 65535, inclusive ● Odd/even numbers up to 65535 **Invalid** ● Less than 0 ● More than 65535
ArrayList<String> getUserList()	**Output** A not null ArrayList of Strings with each element: ● Being a not null string with no spaces
boolean doesUserExist(String newUser)	**Output** True if a client has registered with same username as String newUser, false if not. **'String newUser' partitions:** **Valid** ● >= 0 alphanumeric characters or spaces ● empty string **Invalid** ● null
void broadcastMessage(String theMessage)	**'String theMessage partitions:** **Valid** ● >= 0 alphanumeric characters or spaces ● empty string **Invalid** ● null

int getNumberOfUsers()	**Output** Number of clients connected to the server, don't have to be in the registered state.
boolean sendPrivateMessage(String message, String user)	**Output** True if 'String user' matches a registered client's username, False if not. **'String message' partitions:** **Valid** ● >= 0 alphanumeric characters or spaces ● empty string **Invalid** ● null **Invalid** **'String user' partitions:** **Valid** ● >= 0 alphanumeric characters or spaces ● empty string **Invalid** ● null

6.5 Server commands mapped to Server java methods

Unregistered State

Command	**Method call(s)**
java -jar chatServer.jar	Server.Server()
telnet localhost 9000	Connection.Connection() Connection.run() Server.getNumberOfUsers() Connection.sendOverConnection()
STAT	Connection.validateMessage() Connection.stat() Server.getNumberOfUsers() Connection.sendOverConnection()
LIST	Connection.validateMessage() Connection.list() Connection.sendOverConnection()
HAIL <broadcast message>	Connection.validateMessage() Connection.hail() Connection.sendOverConnection()
MESG <username> <message>	Connection.validateMessage() Connection.mesg() Connection.sendOverConnection()
IDEN <username>	Connection.validateMessage() Connection.iden() Server.doesUserExist()

	Connection.getState() Connection.sendOverConnection()
QUIT	Connection.validateMessage() Connection.quit() Connection.sendOverConnection() Server.removeDeadUsers() Connection.isRunning()

Registered State

Command	Method call(s)
STAT	Connection.validateMessage() Connection.stat() Server.getNumberOfUsers() Connection.sendOverConnection()
LIST	Connection.validateMessage() Connection.list() Server.getUserList() Connection.getState() Connection.getUserName() Connection.sendOverConnection()
HAIL <broadcast message>	Connection.validateMessage() Connection.hail() Server.broadcastMessage() Connection.messageForConnection() Connection.sendOverConnection()
MESG <username> <message>	Connection.validateMessage() Connection.mesg() Server.sendPrivateMessage() Connection.getState() Connection.getUserName() Connection.messageForConnection() Connection.sendOverConnection() Connection.sendOverConnection()
IDEN <username>	Connection.validateMessage() Connection.iden() Connection.sendOverConnection()
QUIT	Connection.validateMessage() Connection.quit() Connection.sendOverConnection() Server.removeDeadUsers() Connection.isRunning()

6.6 Client Test Harness

See attached ClientStub.java in attached ZIP file.

6.7 Modifications to Server.java

```
// Added to the top of the Server class
public volatile boolean shouldServerListen = true;

// Added the new thread start to the bottom of the Server
constructor,
// instead of the while loop
public Server (int port) {
    ...
    list = new ArrayList<Connection>();

    new Thread(new ServerListener(this)).start(); // Added, starts
listener
}
```

```
// Located inside of the Server class
class ServerListener implements Runnable {
    Server serverRef;

    public ServerListener(Server ref) {
        this.serverRef = ref;
    }

    public void run() {
        …
        while(shouldServerListen) { // The while loop
           ...
           c = new Connection(server.accept(), serverRef);
           ...
         }
         server.close(); // Added, Force the server to close
    }
}
```

6.8 Integration test results

6.8.1 General

Test description	Test dataset	Results
Test empty command	‘’	BAD invalid command to server
Test unicode characters	❤☀☆	BAD invalid command to server
Test unicode characters greater than a command length	❤☀☆☂☻🙍	BAD command not recognised

Test another command, in similar structure	SHUT message	BAD command not recognised
Test the number of clients connected	**while(true) { new client.connect() }**	An OutOfMemoryException is thrown around 1000 clients connected, was was due to the number of threads the operating system has allow the JVM to use.

6.8.2 UnRegistered

Cmd	Test description	Test dataset	Results
STAT	Test STAT with 1 user on the server	STAT	OK There are currently 1 user(s) on the server You have not logged in yet
STAT	Test STAT with other users logged in	Connect 2 clients STAT	OK There are currently 2 user(s) on the server You have not logged in yet
STAT	Test STAT with parameters	STAT abc	OK There are currently 1 user(s) on the server You have not logged in yet
STAT	Test STAT with unicode parameters	STAT ❤☀☆☂☻🔬	OK There are currently 1 user(s) on the server You have not logged in yet
STAT	Test STAT in lowercase	stat	BAD command not recognised
STAT	Test STAT with space before command	‘ STAT’	BAD command not recognised
HAIL	Test HAIL with parameters in unregistered state	HAIL hello	BAD You have not logged in yet
HAIL	Test hail parameter as empty string	‘HAIL ‘	BAD You have not logged in yet
HAIL	Test just HAIL with no space or parameters	‘HAIL’	Server throws StringIndexOutOfBoundsException
HAIL	Test HAIL with unicode parameters	HAIL ❤☀☆☂☻🔬	BAD You have not logged in yet
HAIL	Test HAIL with lowercase	hail	BAD command not recognised
HAIL	Test HAIL with space before	‘ HAIL’	BAD command not recognised
LIST	Test LIST with ideal input.	LIST	BAD You have not logged in yet
LIST	Test LIST with parameters.	‘LIST abc’	BAD You have not logged in yet
LIST	Test LIST in lowercase	list	BAD command not recognised
LIST	Test LIST with space before	‘ LIST’	BAD command not recognised

LIST	Test LIST with unicode parameters	LIST ❤☀☆☂☻♘	BAD You have not logged in yet
MESG	Test MESG with ideal parameters	‘MESG aUser hi’	BAD You have not logged in yet
MESG	Test MESG to a user that does exist with ideal parameters	‘MESG clientB hi’	BAD You have not logged in yet
MESG	Test MESG with parameter empty string	‘MESG ‘	BAD You have not logged in yet
MESG	Test MESG with no parameter or space	‘MESG’	Server throws StringIndexOutOfBoundsException
MESG	Test MESG with unicode parameters	LIST ❤☀☆☂☻♘	BAD You have not logged in yet
MESG	Test MESG with lowercase	mesg	BAD command not recognised
MESG	Test MESG with space before	‘ MESG’	BAD command not recognised
MESG	Test MESG with multiple spaces	‘MESG ‘	BAD You have not logged in yet
IDEN	Test IDEN with ideal parameters	‘IDEN harry’	OK Welcome to the chat server harry
IDEN	Test IDEN with numeric parameters	‘IDEN 123’	OK Welcome to the chat server 123
IDEN	Test IDEN with spaces in username	‘IDEN harry mumford’	OK Welcome to the chat server harry
IDEN	Test IDEN with new lines in username	‘IDEN harry\nmumford’	OK Welcome to the chat server harry BAD command not recognised
IDEN	Test IDEN with invalid characters in username	‘IDEN !@£$%^&*()_+-={}[]\"|'\\:;?/><.,'~`’’	OK Welcome to the chat server !@£$%^&*()_+-={}[]\"|'\\:;?/><.,'~`
IDEN	Test IDEN with unicode characters	IDEN ❤☀☆☂☻♘	OK Welcome to the chat server ❤☀☆☂☻♘
QUIT	Test QUIT with ideal input	QUIT	OK goodbye
QUIT	Test QUIT with params after	QUIT abc123	OK goodbye
QUIT	Test QUIT with params and no space	QUITabc	OK goodbye
QUIT	Test QUIT has removed client from number of	**ClientA**: Login	OK There are currently 2 user(s) on the server You have not logged in yet

	users on server, by running STAT.	**ClientB**: Login **ClientA**: STAT QUIT **ClientB**: STAT QUIT	OK goodbye OK There are currently 1 user(s) on the server You have not logged in yet OK goodbye

6.8.3 Registered

Cmd	Test description	Test dataset	Results
STAT	Test STAT with 1 user on the server	STAT	OK There are currently 1 user(s) on the server You are logged in and have sent 0 message(s)
STAT	Test STAT with 2 users on server, from first user	STAT	OK There are currently 2 user(s) on the server You are logged in and have sent 0 message(s)
STAT	Test STAT with 2 users on server, from second user	STAT	OK There are currently 2 user(s) on the server You are logged in and have sent 0 message(s)
STAT	Test STAT message count with 2 users on server, from user who sent message.	'HAIL test', STAT.	Broadcast from harry: hi OK There are currently 2 user(s) on the server You are logged in and have sent 1 message(s)
STAT	Test STAT message with odd number of users, from odd client (client C)	STAT	OK There are currently 3 user(s) on the server You are logged in and have sent 0 message(s)
HAIL	Test HAIL with numerical parameter	'HAIL 123'	Broadcast from harry: 123 \n
HAIL	Test HAIL with ideal input	'HAIL hello'	Broadcast from harry: hello \n
HAIL	Test just HAIL with no space or parameters	'HAIL'	Server throws StringIndexOutOfBoundsException
HAIL	Test HAIL with multiple clients	'HAIL hello'	ClientA: Broadcast from harry: hello \n ClientB: Broadcast from harry: hello \n
HAIL	Test HAIL with an odd number of clients	'HAIL hello'	ClientA: Broadcast from harry: hello \n ClientB: Broadcast from harry: hello \n ClientC: Broadcast from harry: hello \n

HAIL	Test HAIL with unicode characters as the parameter	‘HAIL ❤☀☆☂☻♘’	Broadcast from harry: ❤☀☆☂☻♘ \n
HAIL	Test HAIL with new line in parameter.	‘HAIL hi**n**everyone’	Broadcast from harry: hi \n BAD command not recognised - The server separates this into 2 commands, the second command being ‘everyone’.
HAIL	Test HAIL with invalid characters	‘HAIL !@£$%^&*()_+-={}[]\"|'\\:;?/><.,'~` ‘	Broadcast from harry: !@£$%^&*()_+-={}[]\"|'\\:;?/><.,'~` \n
LIST	Test LIST with ideal parameters	LIST	OK harry,
LIST	Test LIST with random parameters	LIST abc	OK harry,
LIST	Test LIST with unicode parameters	LIST ❤☀☆☂☻♘	OK harry,
LIST	Test LIST with 2 clients connected	LIST	OK harry, clientB,
LIST	Test LIST with odd clients	LIST	OK harry, clientB, clientC,
MESG	Test MESG by sending a message to yourself	‘MESG harry hi’	PM from harry:hi OK your message has been sent
MESG	Test MESG by sending to a client that hasn’t registered	‘MESG john hi’	BAD the user does not exist
MESG	Test MESG by sending a message to another client	MESG clientB hello	Main Client (harry): OK your message has been sent ClientB: PM from harry: hello \n
MESG	Test to see if MESG does send a private message.	MESG clientB hello	Main Client (harry): OK your message has been sent ClientB: PM from harry: hello \n ClientC: OK Welcome to the chat server clientC
MESG	Test MESG with empty string as a parameter	‘MESG ‘	BAD Your message is badly formatted
MESG	Test MESG with both unicode parameters	‘MESG ❤☀☆☂☻♘ ’	BAD the user does not exist

MESG	Test MESG with message as unicode characters	‘MESG harry ☂☻☭’	OK your message has been sent
MESG	Test MESG with no parameters or space	‘MESG’	Server throws StringIndexOutOfBoundsException
MESG	Test MESG with only 1 parameter with an empty string as the second parameter	‘MESG harry ‘	Main Client (harry): OK your message has been sent ClientB: PM from harry: \n
MESG	Test MESG with only 1 parameter of a valid user and no space after it	‘MESG harry’	BAD Your message is badly formatted
MESG	Test MESG with ‘ ‘ as username.	‘MESG hi’	Server throws ArrayIndexOutOfBoundsException
MESG	Test MESG sending new lines in message	‘MESG bob hello \n bobby’	Main Client (harry): OK your message has been sent Client bob: PM from harry: \n Main Client(harry): BAD invalid command to server - The server separates this into 2 commands, the second command being ‘bobby’.
MESG	Test MESG invalid parameter characters	‘MESG harry !@£$%^&*()_+-={}[]\"|'\\:;?/><.,'~`	PM from harry:!@£$%^&*()_+-={}[]\"|'\\:;?/><.,'~` OK your message has been sent
IDEN	Test IDEN registered after registering with different username	IDEN bob	BAD you are already registered with username harry
IDEN	Test IDEN after registering with same username	IDEN harry	BAD you are already registered with username harry
IDEN	Test IDEN after registering with same username, in lowercase	IDEN HARRY	BAD you are already registered with username harry
QUIT	Test QUIT with ideal input	QUIT	OK thank you for sending 0 message(s) with the chat service, goodbye.
QUIT	Test QUIT’s message count.	HAIL hello QUIT	OK thank you for sending 1 message(s) with the chat service, goodbye.
QUIT	Test that the number of users decrease from STAT command, after we quit.	**ClientA, ClientB:** connect **ClientA:** STAT	**ClientA:** OK There are currently 2 user(s) on the server You are logged in and have sent 0 message(s) **ClientB:** OK goodbye

		QUIT **ClientB:** STAT QUIT	**ClientA:** OK There are currently 1 user(s) on the server You are logged in and have sent 0 message(s) OK goodbye
QUIT	Test that we get removed from the list of users (LIST) after we quit.	**ClientA,** **ClientB:** connect **ClientA:** LIST QUIT **ClientB:** LIST QUIT	**ClientA:** OK harry, bob, **ClientB:** QUIT **ClientA:** OK harry,

6.9 Errors and Improvements of Server

6.9.1 Improvements

Cmd	Improvement	Steps to realise	Suggested fix
IDEN	Consider disallowing ' ' as username with the command IDEN	1. telnet localhost 9000 2. Type in 'IDEN ' and press enter	In Connection.iden(String message) consider checking if username != ' '
HAIL/MESG	Consider allowing a new line to be sent in a message.	1. telnet localhost 9000 2. Type in 'IDEN harry' and press enter 3. Type in 'HAIL hello \n everyone' 4. Where \n is a new line	Send command through by converting new line to a token, to display the message, convert all the tokens into new lines.

6.9.2 Errors

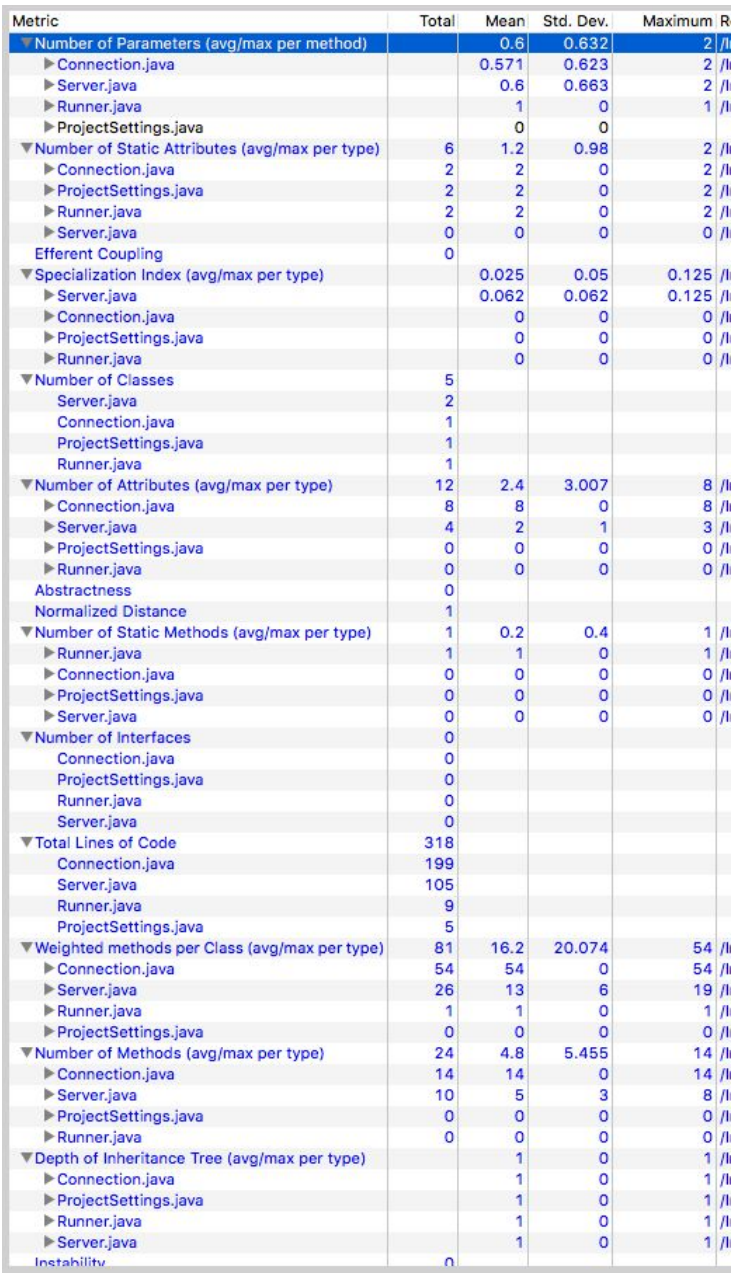
Cmd	Error description (with Stack Trace)	Steps to reproduce	Suggested fix
STAT	Sometimes when running the QUIT command from multiple clients over a short period of time. the server throws a ConcurrentModificationException, because we are removing an item from the Connections ArrayList, while we are iterating around it. It is not thread safe. java.util.ConcurrentModificationException at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901) at java.util.ArrayList$Itr.remove(ArrayList.java:865) at	1. Connect with 3 clients, A, B, C 2. Run QUIT from each client in quick succession 3. Except a Server error thrown	Set a synchronized lock around the iterator and the remove command, this means you can only remove an item from the ArrayList when a client isn't iterating around it.

	chatserver.Server.removeDeadUsers(Server.java:89) at chatserver.Connection.quit(Connection.java:211) at chatserver.Connection.validateMessage(Connection.java:83) at chatserver.Connection.run(Connection.java:47) at java.lang.Thread.run(Thread.java:745)		
HAIL	Running HAIL with no parameters or space, throws an exception in Connection.validate message, when substring(5) is performed on a string of length 4. Exception in thread "Thread-1" java.lang.StringIndexOutOfBoundsException: String index out of range: -1 at java.lang.String.substring(String.java:1931) at chatserver.Connection.validateMessage(Connection.java:75) at chatserver.Connection.run(Connection.java:47) at java.lang.Thread.run(Thread.java:745)	1. Connect 2. Run 'HAIL' 3. Except a error thrown in Connection	In Connection.validateMessage(String message). Check to see if the parameter *message > 5* before running *message.substring(5)*
IDEN	Calling IDEN with no parameters. -- Exception in thread "Thread-1" java.lang.StringIndexOutOfBoundsException: String index out of range: -1 at java.lang.String.substring(String.java:1931) at g54ubi.chat.server.Connection.validateMessage(Unknown Source) at g54ubi.chat.server.Connection.run(Unknown Source) at java.lang.Thread.run(Thread.java:745)	1. telnet localhost 9000 2. Type in 'IDEN' and press enter	In Connection.validateMessage(String message). Check to see if the parameter *message > 5* before running *message.substring(5)*
IDEN	IDEN uses Server.doesUserExist, this function only works if the user you are matching against is last in the ArrayList of Connections.	1. Connect with 3 clients, A,B,C 2. Client A `IDEN harry` 3. Client B `IDEN bob` 4. Client C	In Server.doesUserExist, consider returning true from the function if the user matches, and false if we can't find it.

		`IDEN harry` 5. Except a BAD response, but server accepts same username	
IDEN	Consider disallowing newlines or spaces in username, and telling the user that is the case.	1. Connect to server 2. IDEN harry mumford 3. See that username is only harry not harry mumford	In Connection.iden() check to see if the user has entered more parameters and tell them this.
MESG	that'll also mean that if you sent a private message it would just send it to the first client with a matching name		
MESG	Running MESG with no parameters or space, throws an exception in Connection.validate message, when substring(5) is performed on a string of length 4. Exception in thread "Thread-1" java.lang.StringIndexOutOfBoundsException: String index out of range: -1 at java.lang.String.substring(String.java:1931) at chatserver.Connection.validateMessage(Connection.java:79) at chatserver.Connection.run(Connection.java:47) at java.lang.Thread.run(Thread.java:745)	1. Connect 2. Run 'MESG' 3. Except a error thrown in Connection	In Connection.validateMessage(String message). Check to see if the parameter *message > 5* before running *message.substring(5)*
MESG	Sending a MESG to the user ' ' or running the MESG command like 'MESG hello', throws an ArrayIndexOutOfBoundsException server error. Exception in thread "Thread-3" java.lang.ArrayIndexOutOfBoundsException: 0 at chatserver.Connection.iden(Connection.java:136) at chatserver.Connection.validateMessage(Conn	1. Connect to Server 2. IDEN harry 3. 'MESG ' 4. Expect error thrown in Connection	Add extra checks to the message before it is split, in Connection.iden(). Or enforce that a username cannot be a space.

	ection.java:71) at chatserver.Connection.run(Connection.java:47) at java.lang.Thread.run(Thread.java:745)		
Stress testing	When stress testing the server with just over 1010 clients connected, an OutOfMemoryError was thrown server side. This was an issue with the machine running the server because it was out of native threads / number of threads the OS will allow the JVM to use. Exception in thread "Thread-1" java.lang.OutOfMemoryError: unable to create new native thread at java.lang.Thread.start0(Native Method) at java.lang.Thread.start(Thread.java:714) at chatserver.Server$ServerListener.run(Server.java:50) at java.lang.Thread.run(Thread.java:745)	1. Connect just over 1000 clients to the server 2. Expect a OutOfMemoryError to be thrown server side.	Use a machine with large amounts of memory, so it can allow more clients to be connected.

6.10 Metric Analysis Server results



Metric	Total	Mean	Std. Dev.	Maximum	R
Number of Parameters (avg/max per method)		0.6	0.632	2	/I
Connection.java		0.571	0.623	2	/I
Server.java		0.6	0.663	2	/I
Runner.java		1	0	1	/I
ProjectSettings.java		0	0		
Number of Static Attributes (avg/max per type)	6	1.2	0.98	2	/I
Connection.java	2	2	0	2	/I
ProjectSettings.java	2	2	0	2	/I
Runner.java	2	2	0	2	/I
Server.java	0	0	0	0	/I
Efferent Coupling	0				
Specialization Index (avg/max per type)		0.025	0.05	0.125	/I
Server.java		0.062	0.062	0.125	/I
Connection.java		0	0	0	/I
ProjectSettings.java		0	0	0	/I
Runner.java		0	0	0	/I
Number of Classes	5				
Server.java	2				
Connection.java	1				
ProjectSettings.java	1				
Runner.java	1				
Number of Attributes (avg/max per type)	12	2.4	3.007	8	/I
Connection.java	8	8	0	8	/I
Server.java	4	2	1	3	/I
ProjectSettings.java	0	0	0	0	/I
Runner.java	0	0	0	0	/I
Abstractness	0				
Normalized Distance	1				
Number of Static Methods (avg/max per type)	1	0.2	0.4	1	/I
Runner.java	1	1	0	1	/I
Connection.java	0	0	0	0	/I
ProjectSettings.java	0	0	0	0	/I
Server.java	0	0	0	0	/I
Number of Interfaces	0				
Connection.java	0				
ProjectSettings.java	0				
Runner.java	0				
Server.java	0				
Total Lines of Code	318				
Connection.java	199				
Server.java	105				
Runner.java	9				
ProjectSettings.java	5				
Weighted methods per Class (avg/max per type)	81	16.2	20.074	54	/I
Connection.java	54	54	0	54	/I
Server.java	26	13	6	19	/I
Runner.java	1	1	0	1	/I
ProjectSettings.java	0	0	0	0	/I
Number of Methods (avg/max per type)	24	4.8	5.455	14	/I
Connection.java	14	14	0	14	/I
Server.java	10	5	3	8	/I
ProjectSettings.java	0	0	0	0	/I
Runner.java	0	0	0	0	/I
Depth of Inheritance Tree (avg/max per type)		1	0	1	/I
Connection.java		1	0	1	/I
ProjectSettings.java		1	0	1	/I
Runner.java		1	0	1	/I
Server.java		1	0	1	/I
Instability	0				

6.11 Client Unit Tests

Test description	**Test dataset**
Test all valid server commands	STAT LIST IDEN harry MESG harry hi HAIL hello QUIT
Test IDEN invalid parameters	‘IDEN’

	‘IDEN ‘ ‘IDEN ‘
Test HAIL invalid parameters	‘HAIL’ ‘HAIL ‘
Test MESG invalid parameters	‘MESG’ ‘MESG ‘ ‘MESG ‘
Test invalid text	❤☀☆☂☻♘

6.12 Client GUI Tests

Test description	**Test dataset**
Show an invalid command from client	Type into text box ‘MESG ‘ Click send button See output textbox update with ‘>Invalid Command’
Show a correct command from client expecting a BAD response	Type into text box ‘LIST’ Click send Button Check that output starts with BAD
Show a correct command from client expecting a OK response	Type into text box ‘STAT’ Click send Button Check that output starts with OK

6.13 Travis Output

✓ master Start the Server for GUI tests

Rename file

Commit d1322a8

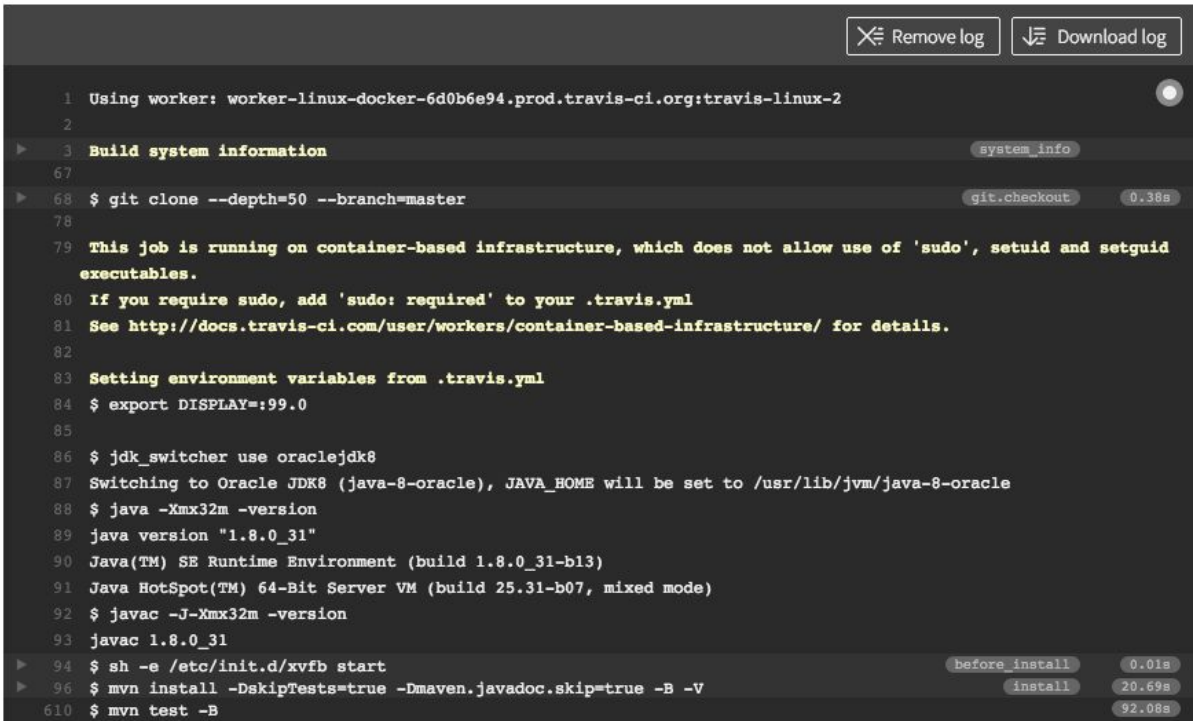
Compare 460cea0..d1322a8

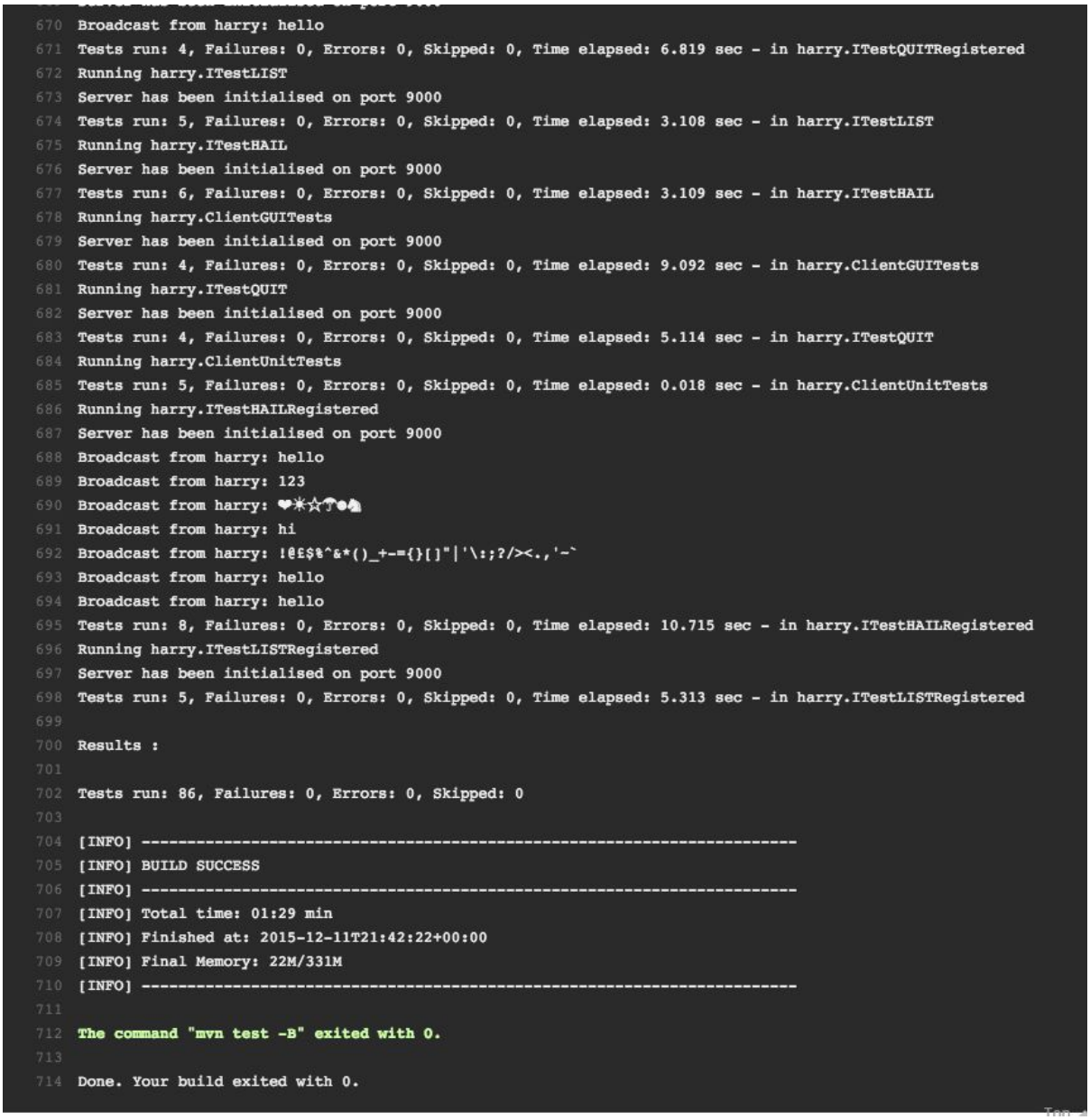
Harry Mumford-Turner authored and committed

#11 passed

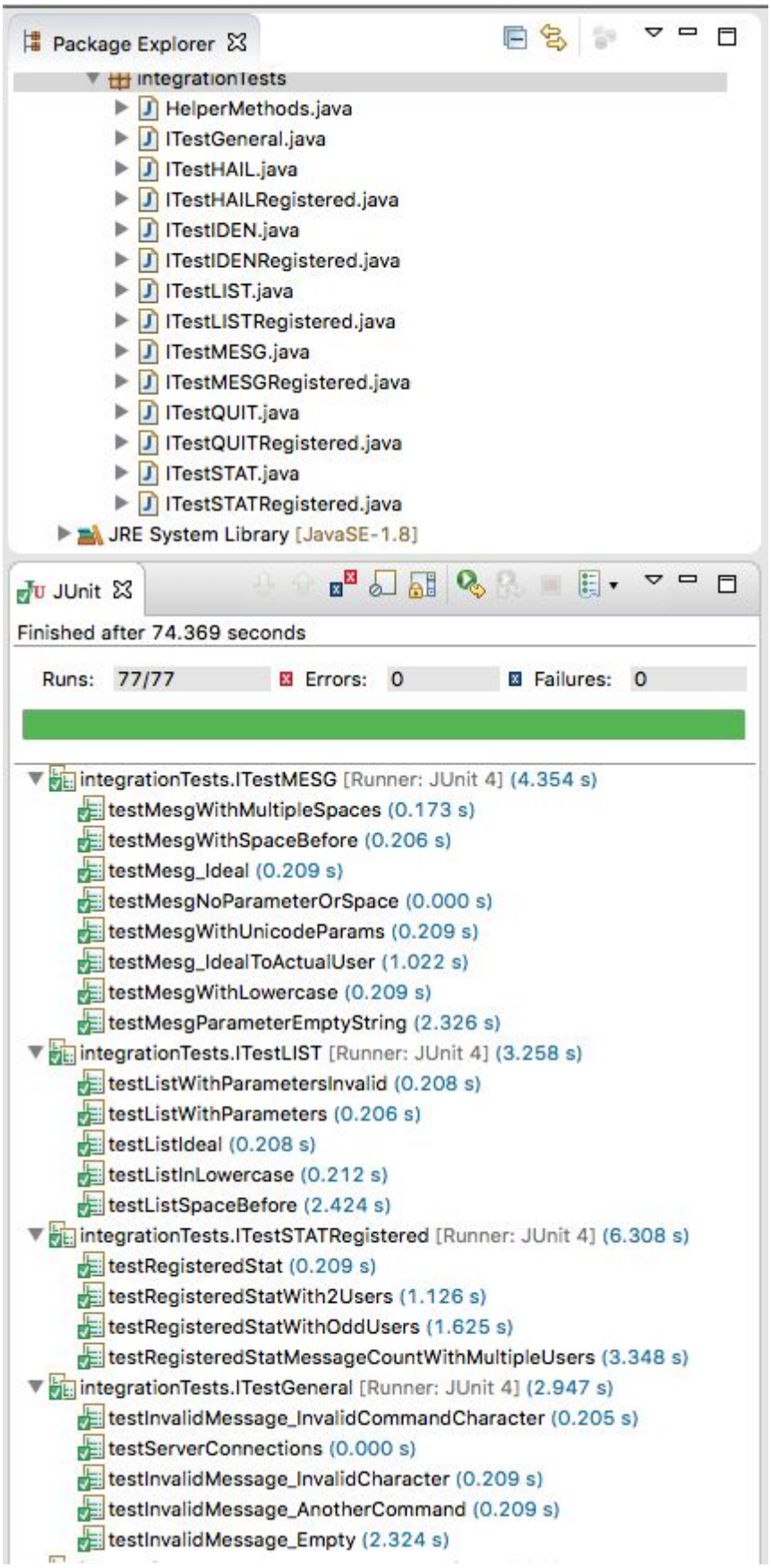
Elapsed time 1 min 58 sec

less than a minute ago

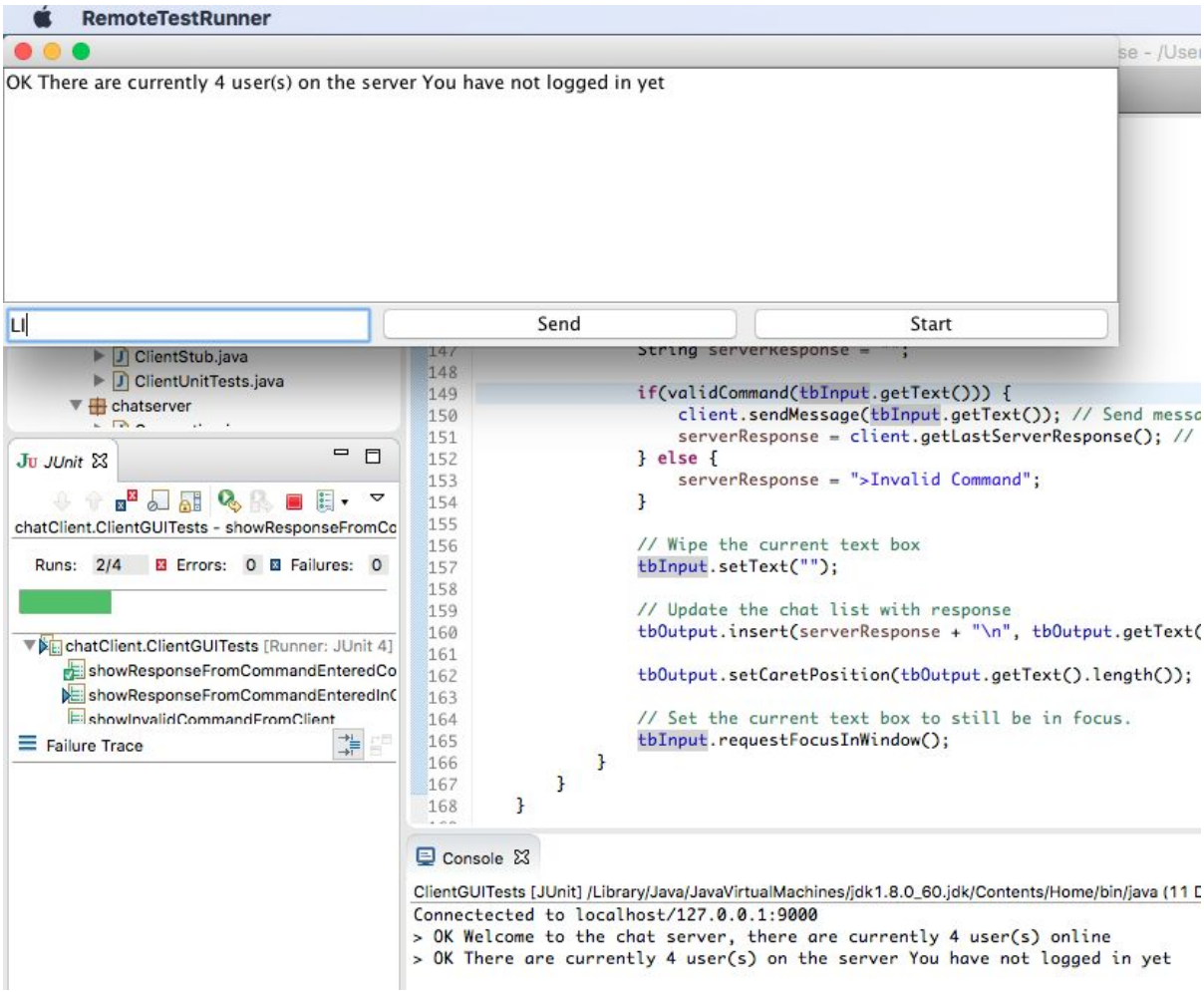




6.14 JUnit example test


Package Explorer
integrationTests
HelperMethods.java
ITestGeneral.java
ITestHAIL.java
ITestHAILRegistered.java
ITestIDEN.java
ITestIDENRegistered.java
ITestLIST.java
ITestLISTRegistered.java
ITestMESG.java
ITestMESGRegistered.java
ITestQUIT.java
ITestQUITRegistered.java
ITestSTAT.java
ITestSTATRegistered.java
JRE System Library [JavaSE-1.8]
JUnit
Finished after 74.369 seconds
Runs: 77/77
Errors: 0
Failures: 0
integrationTests.ITestMESG [Runner: JUnit 4] (4.354 s)
testMesgWithMultipleSpaces (0.173 s)
testMesgWithSpaceBefore (0.206 s)
testMesg_Ideal (0.209 s)
testMesgNoParameterOrSpace (0.000 s)
testMesgWithUnicodeParams (0.209 s)
testMesg_IdealToActualUser (1.022 s)
testMesgWithLowercase (0.209 s)
testMesgParameterEmptyString (2.326 s)
integrationTests.ITestLIST [Runner: JUnit 4] (3.258 s)
testListWithParametersInvalid (0.208 s)
testListWithParameters (0.206 s)
testListIdeal (0.208 s)
testListInLowercase (0.212 s)
testListSpaceBefore (2.424 s)
integrationTests.ITestSTATRegistered [Runner: JUnit 4] (6.308 s)
testRegisteredStat (0.209 s)
testRegisteredStatWith2Users (1.126 s)
testRegisteredStatWithOddUsers (1.625 s)
testRegisteredStatMessageCountWithMultipleUsers (3.348 s)
integrationTests.ITestGeneral [Runner: JUnit 4] (2.947 s)
testInvalidMessage_InvalidCommandCharacter (0.205 s)
testServerConnections (0.000 s)
testInvalidMessage_InvalidCharacter (0.209 s)
testInvalidMessage_AnotherCommand (0.209 s)
testInvalidMessage_Empty (2.324 s)

6.15 FEST example output



6.16 Metrics Results

Metrics - ClientStub.java

Metric	Total	Mean	Std. Dev.	Maximum
Number of Parameters (avg/max per method)		0.429	0.728	2
Number of Static Attributes (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0	0	0
Number of Classes	2			
Number of Attributes (avg/max per type)	7	3.5	3.5	7
Number of Static Methods (avg/max per type)	0	0	0	0
Number of Interfaces	0			
Total Lines of Code	106			
Weighted methods per Class (avg/max per type)	27	13.5	7.5	21
Number of Methods (avg/max per type)	7	3.5	2.5	6
Depth of Inheritance Tree (avg/max per type)		1	0	1
McCabe Cyclomatic Complexity (avg/max per method)		3.857	1.884	7
Nested Block Depth (avg/max per method)		2.286	0.881	4
Lack of Cohesion of Methods (avg/max per type)		0.414	0.414	0.829
Method Lines of Code (avg/max per method)	76	10.857	5.514	18
Number of Overridden Methods (avg/max per type)	0	0	0	0
Number of Children (avg/max per type)	0	0	0	0

Metrics - ClientUnitTests.java

Metric	Total	Mean	Std. Dev.	Maximum
Number of Parameters (avg/max per method)		0	0	0
Number of Static Attributes (avg/max per type)	1	1	0	1
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Attributes (avg/max per type)	0	0	0	0
Number of Static Methods (avg/max per type)	2	2	0	2
Number of Interfaces	0			
Total Lines of Code	70			
Weighted methods per Class (avg/max per type)	9	9	0	9
Number of Methods (avg/max per type)	7	7	0	7
Depth of Inheritance Tree (avg/max per type)		1	0	1
McCabe Cyclomatic Complexity (avg/max per method)		1	0	1
Nested Block Depth (avg/max per method)		1	0	1
Lack of Cohesion of Methods (avg/max per type)		0	0	0
Method Lines of Code (avg/max per method)	33	3.667	3.771	12
Number of Overridden Methods (avg/max per type)	0	0	0	0
Number of Children (avg/max per type)	0	0	0	0

Metrics - ClientGUI.java

Metric	Total	Mean	Std. Dev.	Maximum
Number of Parameters (avg/max per method)		0.571	0.495	1
Number of Static Attributes (avg/max per type)	0	0	0	0
Specialization Index (avg/max per type)		0	0	0
Number of Classes	4			
Number of Attributes (avg/max per type)	5	1.25	2.165	5
Number of Static Methods (avg/max per type)	1	0.25	0.433	1
Number of Interfaces	0			
Total Lines of Code	126			
Weighted methods per Class (avg/max per type)	28	7	6.964	19
Number of Methods (avg/max per type)	6	1.5	0.866	3
Depth of Inheritance Tree (avg/max per type)		2.25	2.165	6
McCabe Cyclomatic Complexity (avg/max per method)		4	5.014	16
Nested Block Depth (avg/max per method)		2.429	1.678	6
Lack of Cohesion of Methods (avg/max per type)		0.25	0.433	1
Method Lines of Code (avg/max per method)	87	12.429	11.312	35
Number of Overridden Methods (avg/max per type)	0	0	0	0
Number of Children (avg/max per type)	0	0	0	0

Metrics - ClientGUITests.java

Metric	Total	Mean	Std. Dev.	Maximum
Number of Parameters (avg/max per method)		0.143	0.35	1
Number of Static Attributes (avg/max per type)	2	2	0	2
Specialization Index (avg/max per type)		0	0	0
Number of Classes	1			
Number of Attributes (avg/max per type)	3	3	0	3
Number of Static Methods (avg/max per type)	2	2	0	2
Number of Interfaces	0			
Total Lines of Code	58			
Weighted methods per Class (avg/max per type)	8	8	0	8
Number of Methods (avg/max per type)	5	5	0	5
Depth of Inheritance Tree (avg/max per type)		1	0	1
McCabe Cyclomatic Complexity (avg/max per method)		1.143	0.35	2
Nested Block Depth (avg/max per method)		1.143	0.35	2
Lack of Cohesion of Methods (avg/max per type)		0	0	0
Method Lines of Code (avg/max per method)	25	3.571	1.678	5
Number of Overridden Methods (avg/max per type)	0	0	0	0
Number of Children (avg/max per type)	0	0	0	0