MBTCover:
A tool for code-level and requirement-level test coverage measurement of model-based testing

User guide and Design document

Development team:
James Sloan[1], Vahid Garousi[2]
Queen's University Belfast, Northern Ireland, UK
{jsloan19, v.garousi}@qub.ac.uk

First release: May 10, 2020
Revision date of this document: May 10, 2020

www.github.com/vgarousi/MBTCover

Abstract

While model-based testing (MBT) has been around for a few decades and is an effective software testing technique, tool support for MBT in terms of code-level or requirement-level coverage assessment is limited. We have developed and offer an open-source prototype tool, named *MBTCover*, in the form of an easy to understand display system that shows these metrics after running a given MBT test suite. The current implementation of MBTCover is for the GraphWalker[3] open-source model-based testing tool. The document serves as a brief user guide and design document for the tool.

[1] www.linkedin.com/in/james-sloan-3b39a2150/
[2] www.vgarousi.com
[3] graphwalker.github.io

Contents

1.0 User guide

We must now ensure that the JaCoCo Java Agent is attached as an argument to the JVM of our system under test. This is achieve through first running the command:

```
nano ~/.profile
```

And adding the following option before saving the file:

```
export MAVEN_OPTS="-javaagent:/location/jacoco/lib/jacocoagent.jar"
```

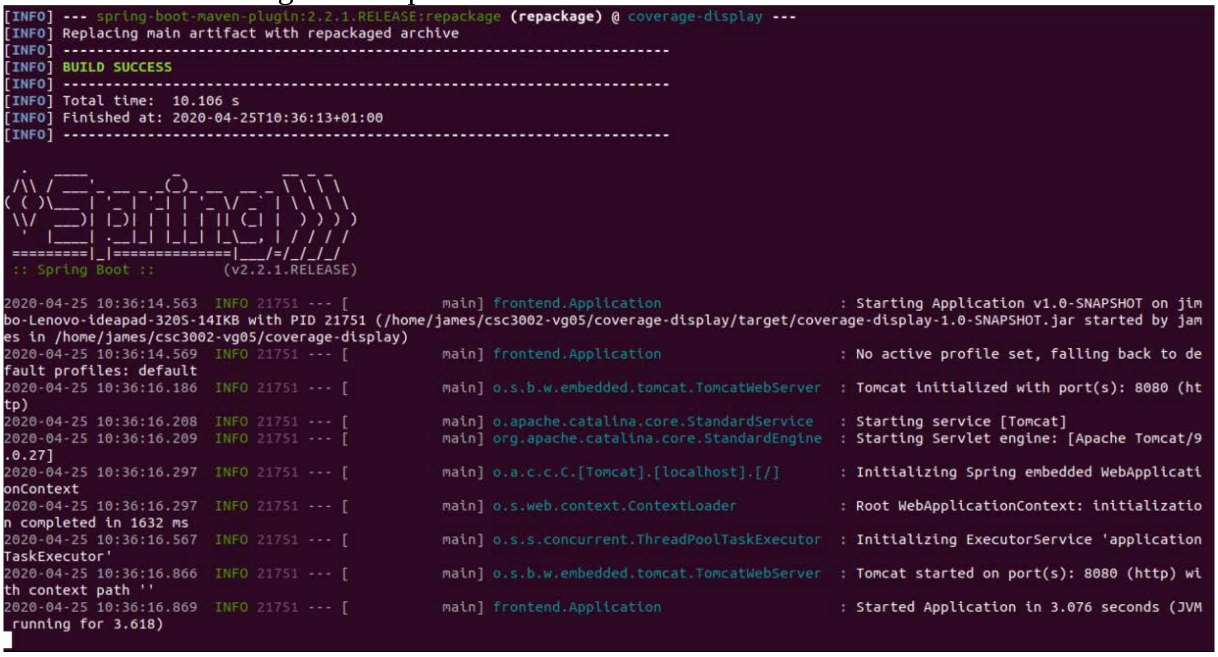
Now that we have done this we can now make use of the system. Open your Linux terminal and navigate to the given project using the following command within the Linux Shell:

```
cd csc3002-vg05
```

Once we are in the required place, we invoke our automation script by entering the following command into our Linux terminal:

```
./run.sh
```

This will kick off the automated process of running our GraphWalker tests against our SUT, and therefore the user will see a browser be opened as the test suite is executed. After the test suite is finished and the browser closes, a number of metrics will be generated. Eventually, the end display system will be launched. The user will know this when they reach this point when the console is flooding with output similar to this:

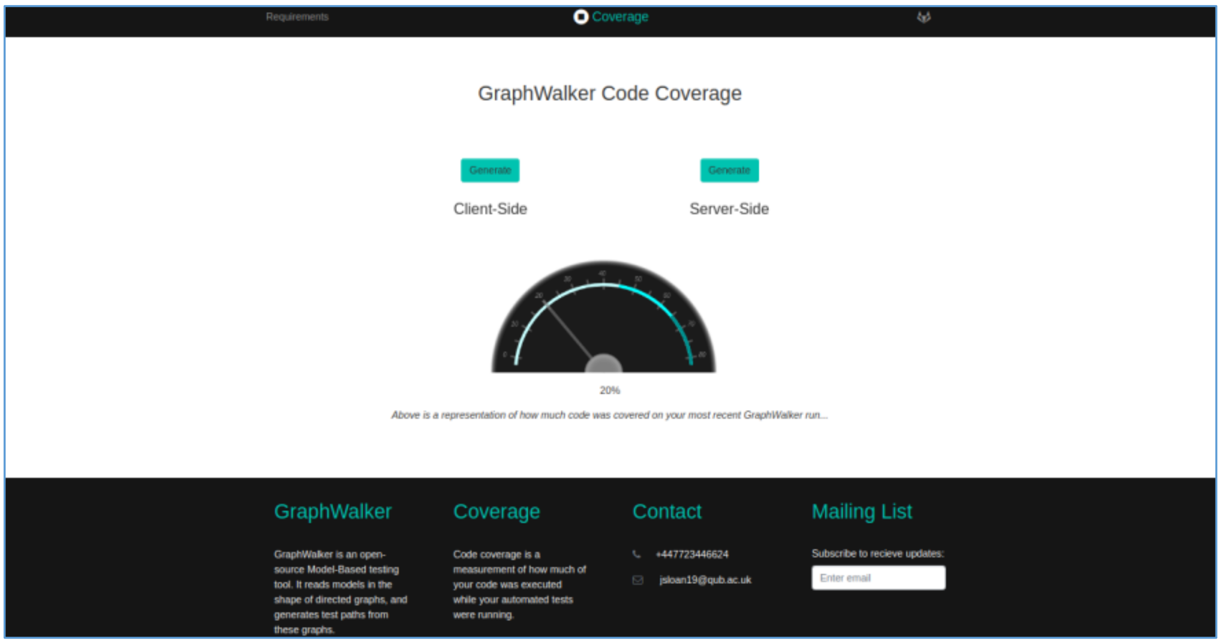


```
[INFO] --- spring-boot-maven-plugin:2.2.1.RELEASE:repackage (repackage) @ coverage-display ---
[INFO] Replacing main artifact with repackaged archive
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  10.106 s
[INFO] Finished at: 2020-04-25T10:36:13+01:00
[INFO] ------------------------------------------------------------------------

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.1.RELEASE)

2020-04-25 10:36:14.563  INFO 21751 --- [           main] frontend.Application                     : Starting Application v1.0-SNAPSHOT on jim
bo-Lenovo-ideapad-320S-14IKB with PID 21751 (/home/james/csc3002-vg05/coverage-display/target/coverage-display-1.0-SNAPSHOT.jar started by jam
es in /home/james/csc3002-vg05/coverage-display)
2020-04-25 10:36:14.569  INFO 21751 --- [           main] frontend.Application                     : No active profile set, falling back to de
fault profiles: default
2020-04-25 10:36:16.186  INFO 21751 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (ht
tp)
2020-04-25 10:36:16.208  INFO 21751 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2020-04-25 10:36:16.209  INFO 21751 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9
.0.27]
2020-04-25 10:36:16.297  INFO 21751 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicati
onContext
2020-04-25 10:36:16.297  INFO 21751 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initializatio
n completed in 1632 ms
2020-04-25 10:36:16.567  INFO 21751 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'application
TaskExecutor'
2020-04-25 10:36:16.866  INFO 21751 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) wi
th context path ''
2020-04-25 10:36:16.869  INFO 21751 --- [           main] frontend.Application                     : Started Application in 3.076 seconds (JVM
 running for 3.618)
```

It is at this point the user can open their browser and navigate to:

```
localhost:8080/coverage
or
localhost:8080/requirements
```

The user will be greeted one of the two webpages, depending on where they route to. Given that the user navigates to the former link, they would be greeted with this:



Above is a representation of how much code was covered on your most recent GraphWalker run...

GraphWalker is an open-source Model-Based testing tool. It reads models in the shape of directed graphs, and generates test paths from these graphs.

Code coverage is a measurement of how much of your code was executed while your automated tests were running.

+447723446624

jsloan19@qub.ac.uk

Subscribe to recieve updates:

The user can navigate between the two pages using the navbar at the top and can generate relevant metrics by making use of the aqua-coloured button(s) that exist above the meter itself.

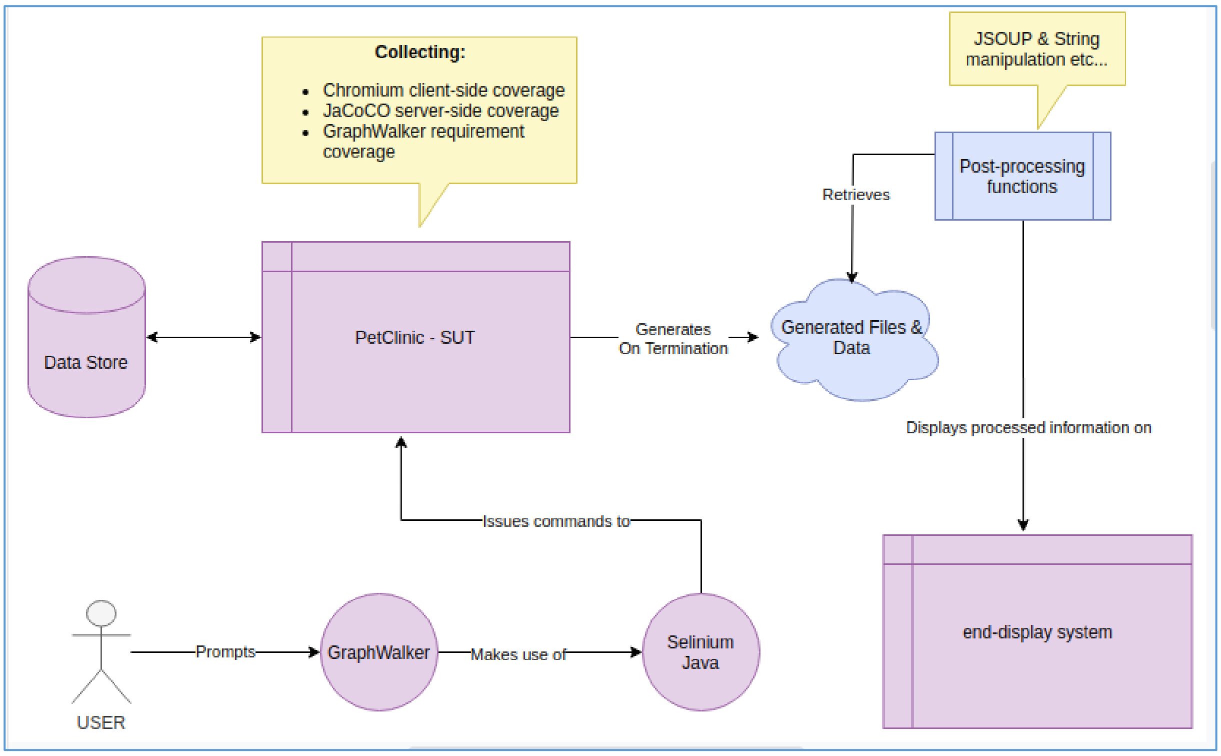
2.0 Design document

2.1 Conceptual Design

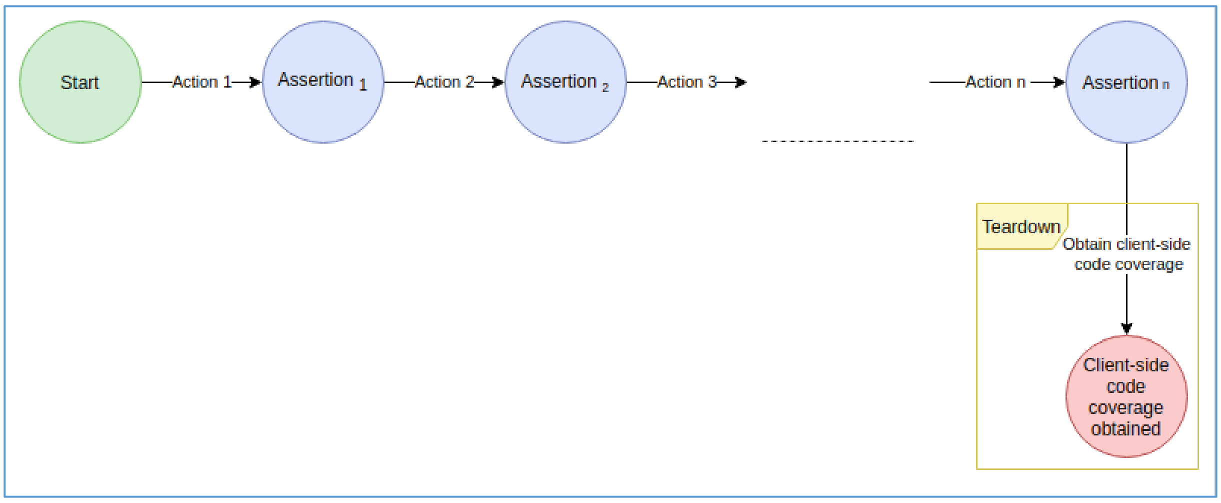
Considering the fact that we now have a finalised concept on how we are going to approach our solution to the proposed problem of this project, we must now focus on the design of the system itself. It is of utmost importance that we look at the design of the system before we go anywhere near any level of implementation as we do not want to spend our time developing a given implementation for a solution that is not fit for a given problem; causing us to fail later into the development stage and therefore have to back-track. Issues caught earlier during the software development process tend to be cheaper in both time and in monetary value.

Software design takes a problem or solution and allows the person or people developing or dealing with it to divide it up to provide a more modular approach to ensure that their bases are covered and to minimize something being overlooked during this phase. The design phase of the development process allows us to take our given requirements and bridge that gap between what we know we need to do and how we are actually going to go about doing it whenever it comes to the production of the code itself.

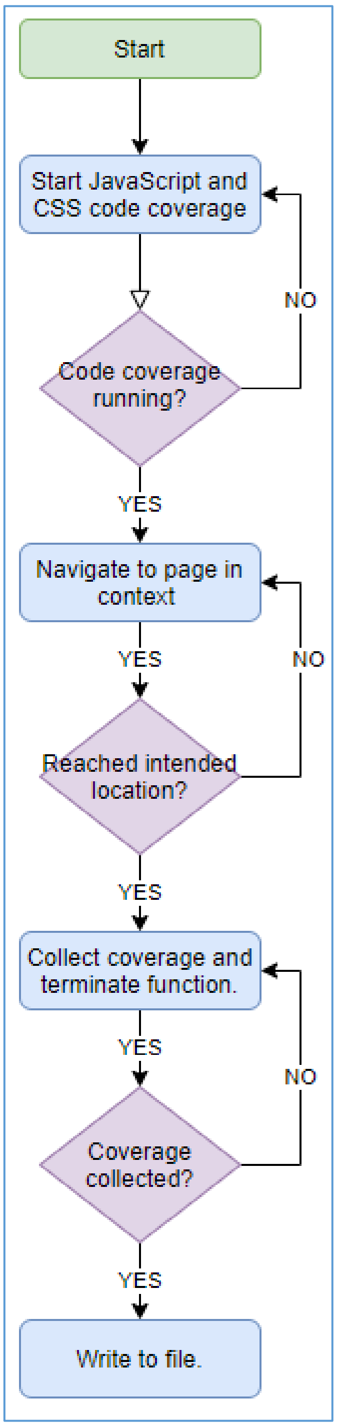
It is with this in mind that we will begin the design process. It is often argued that diagrams speak a thousand words whenever it comes to explaining or laying out your design process. Therefore, we will employ the use of the popular online diagram development tool draw io [9]. In order to ensure the correct modular breakdown of our design it is in our best interests to start with an overall view of how we would expect the flow of our full system implementation. This can be seen below:


Collecting:
Chromium client-side coverage
JaCoCO server-side coverage
GraphWalker requirement coverage
JSOUP & String manipulation etc...
Post-processing functions
Retrieves
Data Store
PetClinic - SUT
Generates On Termination
Generated Files & Data
Displays processed information on
Issues commands to
end-display system
USER
Prompts
GraphWalker
Makes use of
Selinium Java

So as you can see from the above diagram, the user will run the GraphWalker test suite against the SUT and during teardown of the GraphWalker tests and after the termination of the SUT the code-level and requirement-level coverage statistics will be generated. Once this has occurred then the end-display system can be started. This system will find, and process said coverage information so that it can be displayed in a useful and simplified manner. To better understand this process, we will look at the individual components and processes involved in this pipeline. Let us begin with the flow of obtaining the client-side code coverage.

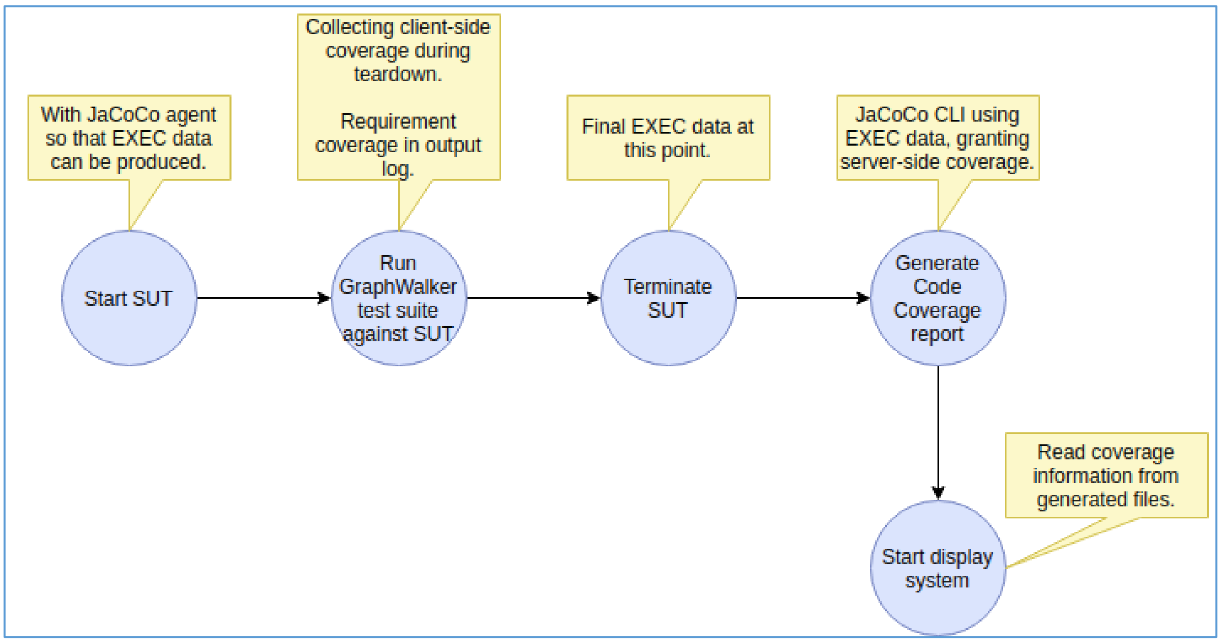


The core of the idea being displayed above is that the client-side coverage will be taken during the teardown phase of the selenium code of the GraphWalker testing suite. This means that any other developer outside of this project can add this to their implementation without having to alter the models that they are running. This stage is done after the actions and assertions have ran their course. However, this poses the question of how exactly would we achieve the client-side coverage collection? Below is a diagram which shows the pseudo flow of the scripting that we will implement to be able to retrieve the client-side code coverage from the Chromium DevTool.

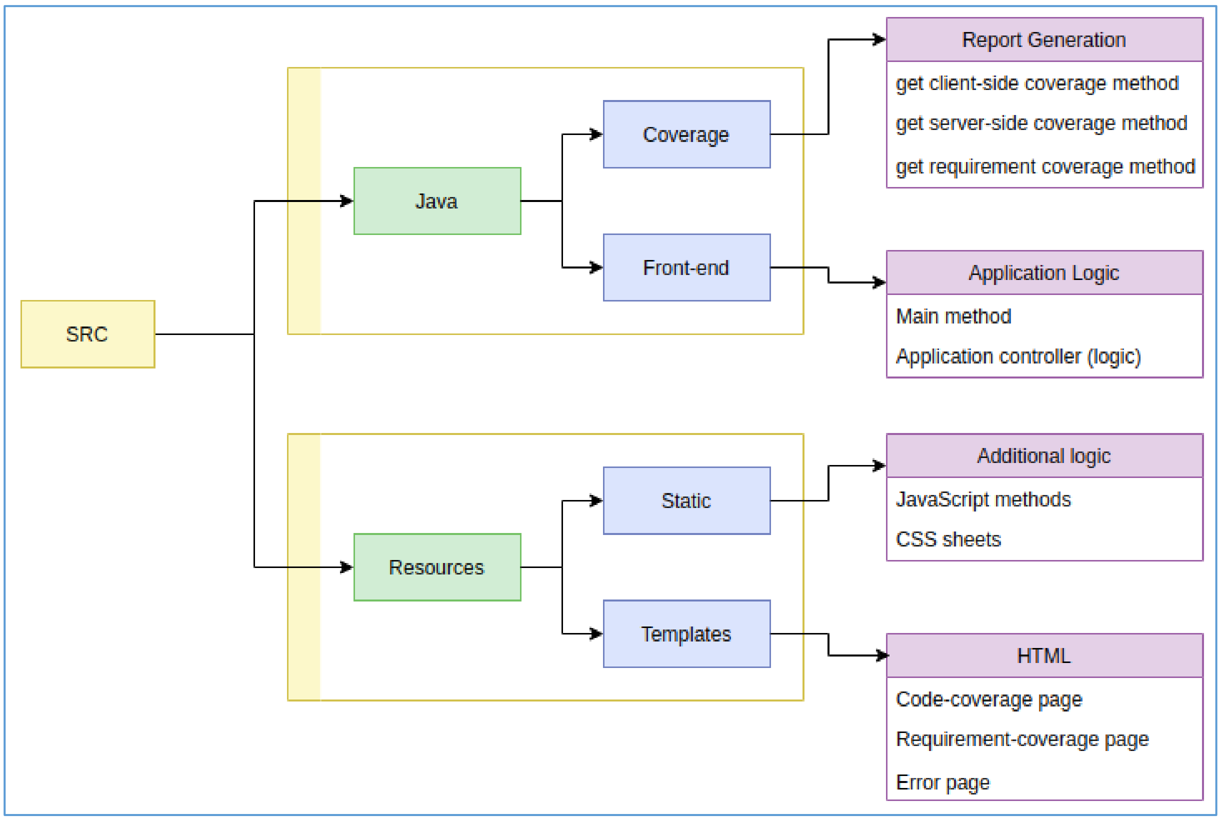

Start
Start JavaScript and CSS code coverage
NO
Code coverage running?
YES
Navigate to page in context
YES
NO
Reached intended location?
YES
Collect coverage and terminate function.
YES
NO
Coverage collected?
YES
Write to file.

This partially fulfils the requirement of retrieving some code-level coverage metric. However, the arguably more important side of the code-level coverage metrics would be the server-side code-coverage statistics. As lightly touched upon in the problem solution section of this report, we will run our SUT with the JaCoCo Agent attached to it so that it will collect execution data as the system is live. Therefore, running the test suite against the live system would therefore inherently alter said execution data. The coverage aspect would come into effect given that the code within the SUT is being tracked, which we will come to within the implementation section of this report.

In order to better clarify the steps that are to be taken in regard to where and when the requirement-level and code-level coverage is going to be generated within the pipeline we will make use of another diagram, as seen below:


With JaCoCo agent so that EXEC data can be produced.
Collecting client-side coverage during teardown.
Requirement coverage in output log.
Final EXEC data at this point.
JaCoCo CLI using EXEC data, granting server-side coverage.
Start SUT
Run GraphWalker test suite against SUT
Terminate SUT
Generate Code Coverage report
Read coverage information from generated files.
Start display system
One notable point raised from the respective figure above is where exactly the end display system will come into scope in regard to the previously mentioned pipeline of events for our solution. Although we have identified that we will perform some level of pre-processing on existing requirement-level and code-level coverage metrics that have been generated in the manner dictated in the diagram above, we must take the time to flesh-out the design for our display system.

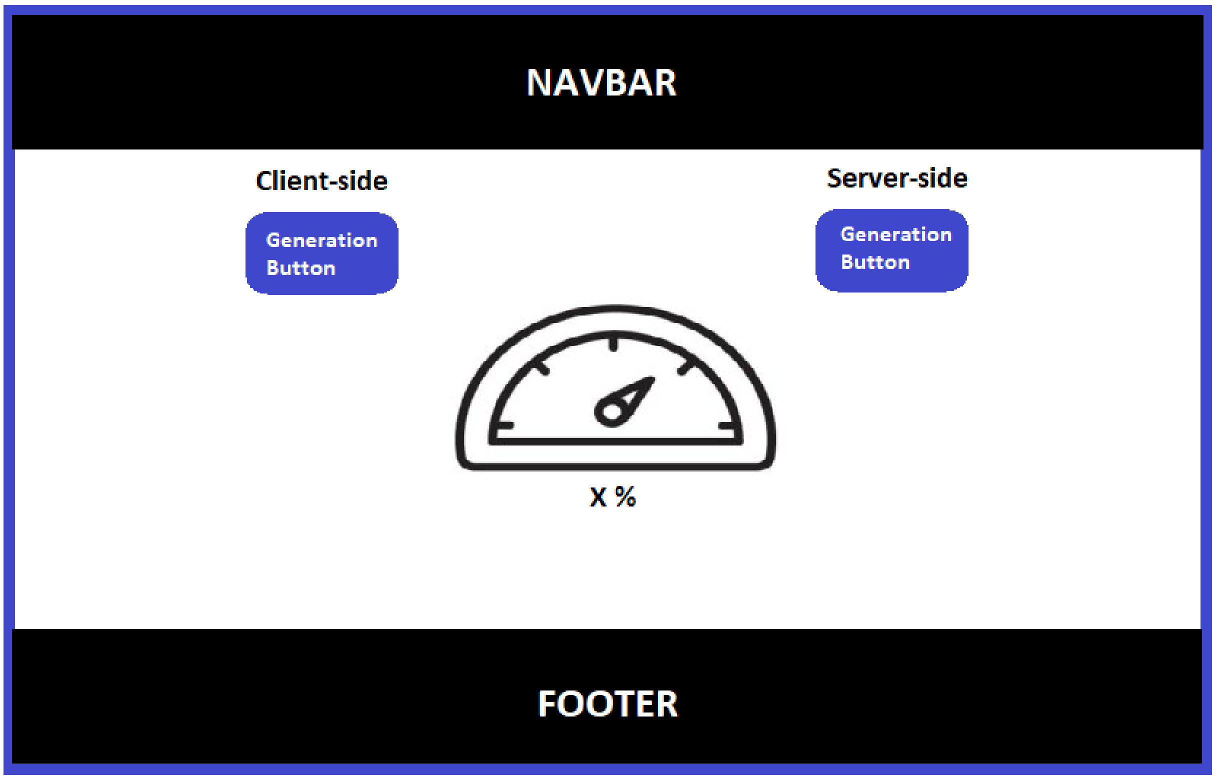
There are two main areas whenever it comes to the design of our said display system; how exactly we are going to achieve it in terms of its technical architecture, and how it is going to look. Firstly, let’s look at the more technical side of the system. This will be shown by a structural diagram below:



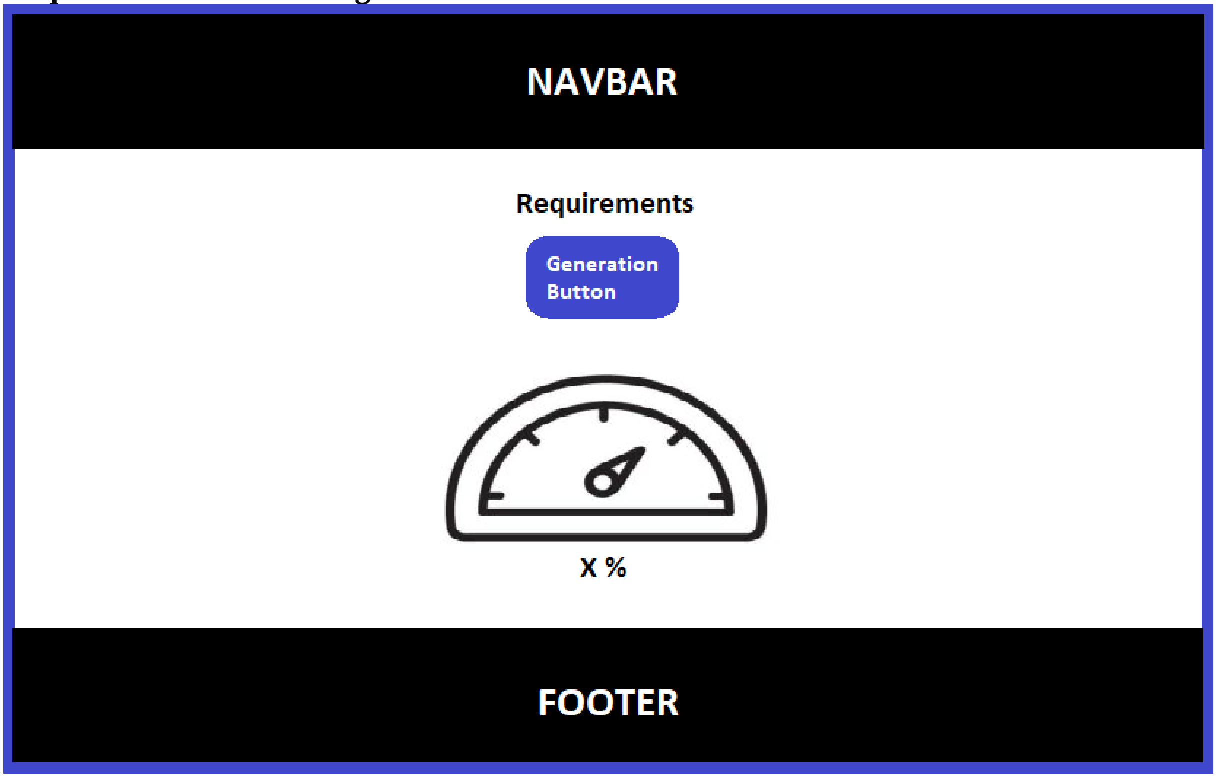
As can be seen from the structural diagram offered above, our source folder will essentially exist of two main modules. These modules are a Java folder, and a Resources folder. The Java folder will contain two main things: a coverage sub-module, and a front-end sub-module. The coverage submodule will contain the prementioned 'pre-processing' methods that will go and access the files that were generated previously by the execution of the GraphWalker test suite against our SUT. These methods would be called from the front-end module of the java folder. Given how a spring-boot application functions, there would be an Application class that holds a main-method entry point to boot the application, and a ApplicationController class that would hold the routing logic, deciding what variables and HTML webpage the user will see. The Resources folder will again contain two main things, a Static folder and a Template folder. The Static folder will contain front-end dynamic scripting logic that will grant functionality to the webpages that will be displayed to the user. These said webpages will be held within the Templates folder.

Now that we have a clear indication of how we are going to structure our display application, it is crucial that we now briefly design how the system is to look, at least to some rudimental level.

Code-level coverage screen



Requirement-level coverage screen



Both screens are similar, the only difference being that each button will call upon different functions that will retrieve the relevant metric. Additionally, the numeric value of this metric will be displayed below the meter.

2.2 Error and Exception Handling

The final consideration for the design process for our proposed solution is to attempt to anticipate any errors that we may encounter within the running of our overall pipeline. In addition to this, it would be wise for us to examine where and how we would implement the according exception handling to deal with the respective errors.

The first notable area in which we can expect an error to occur is the choice of driver within the GraphWalker selenium tests. In general, most selenium implementations make use of the FireFox driver due to its reliability and stability. However, with Chromium DevTools being an area of interest for us we will be using the Chrome driver. The Chrome driver automatically seeks the newest version. However, the newest version of the driver is typically released before the according Google Chrome build is. Although the timeframe is small, it is an existent issue and therefore we will have to account for this as not doing so would result in the execution of our test suit being unstable.

Additionally, with our proposed solution making use of GraphWalker Studio to develop JSON based models rather than the traditional GraphML approach, we must consider that use of guards, shared keywords, and ReqTag functionality will not be the same and therefore we can expect to have to deal with errors whenever it comes to the implementation of this and thus will need some level of verification.

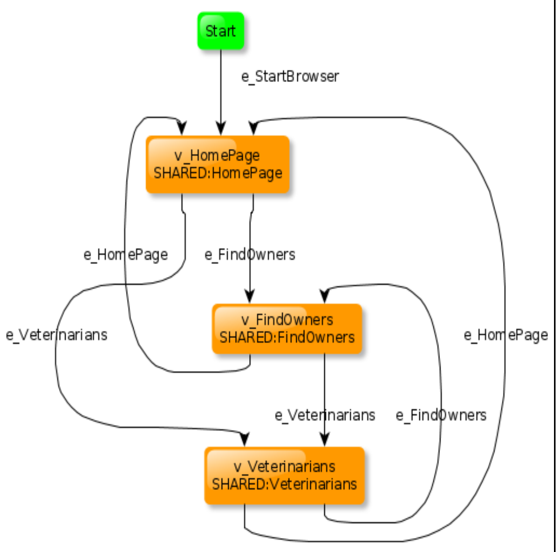
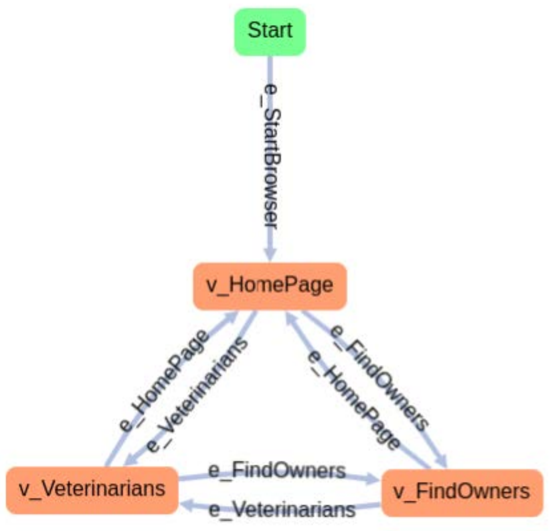
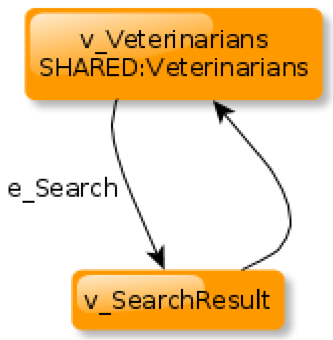
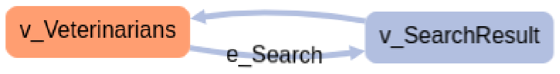
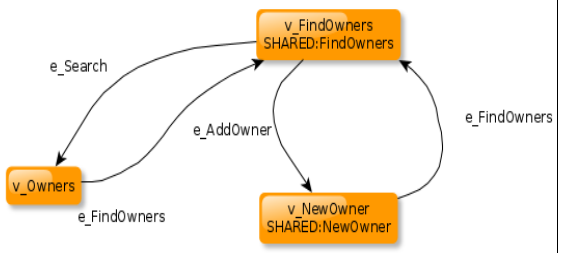
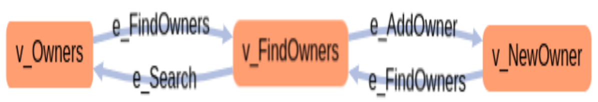
Considering that our code-level and requirement-level coverage display web-application will attempt to search for and process the generated files from the test suite execution against the system under test, if anything were to have went wrong at an earlier stage in our pipeline, the files that we require might not have generated properly or at all. If left alone without being dealt with, this would cause the end-system to terminate unexpectedly. Therefore, we must have some element of error and exception handling. To begin with we must ensure that the files are present; this can be handled by wrapping the post-testing data processing functions in a try-catch statement that will throw a FileNotFound exception. In addition to this, if the file is present, we make sure that the result is not null or NaN (not a number). If this is the case, we will direct the user to an error page whenever they try to route to one of the existing webpages of the project, rather than terminating the program or returning a whitelabel page.

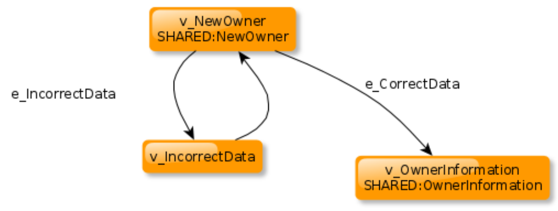
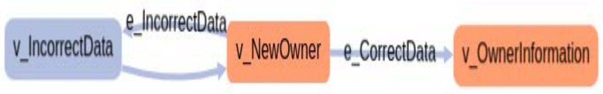
3.0 Implementation details

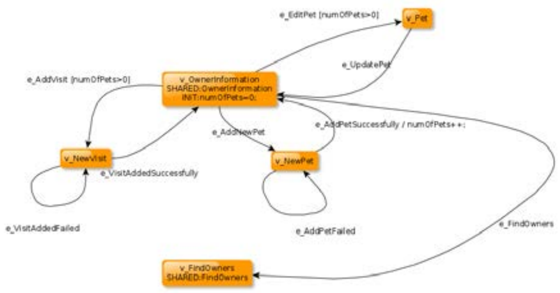
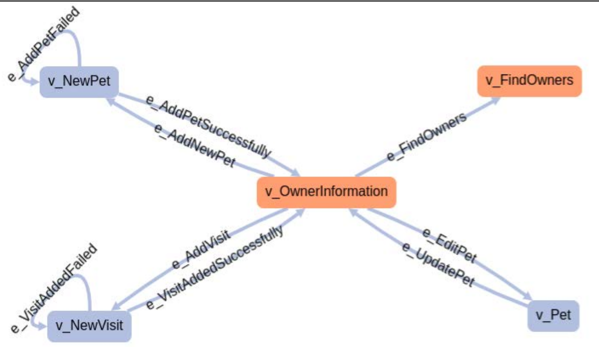
As a result of finishing the design phase of the software development pipeline, we can now look to begin implementation of our proposed solution. In line with our design we will take each individual component at each stage of our pipeline and break down how exactly we achieved the implementation for the functionality we have previously set forth within this document.

3.1 GraphWalker Test Suite

The most logical place to start in regard to the implementation of our GraphWalker test suite are the models themselves. As previously discussed, the existing implementation of GraphWalker for the PetClinic system was developed using arguably outdated tools and technologies. For our project to be the most applicable from an industry standpoint we must make some tweaks to this. In specific, we must re-develop the models in GraphWalker Studio to convert them from the GraphML format to the JSON format whilst also ensuring that our implementation follows suit to retain the integrity of the functionality of the existing project. Below are the older models alongside the ones that were developed during the implementation phase of this project:

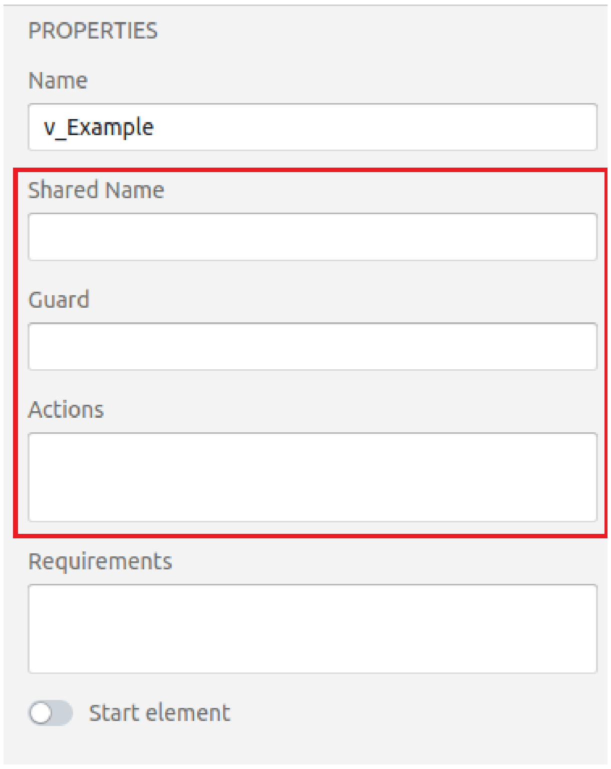
yED GraphML	GraphWalker Studio JSON
Base PetClinic Model	
 Start e_StartBrowser v_HomePage SHARED:HomePage e_HomePage e_FindOwners e_Veterinarians v_FindOwners SHARED:FindOwners e_HomePage e_Veterinarians e_FindOwners v_Veterinarians SHARED:Veterinarians	 Start e_StartBrowser v_HomePage e_HomePage e_Veterinarians e_FindOwners e_HomePage v_Veterinarians e_FindOwners e_Veterinarians v_FindOwners
Veterinarians Model	
 v_Veterinarians SHARED:Veterinarians e_Search v_SearchResult	
FindOwners Model	
 v_FindOwners SHARED:FindOwners e_Search e_AddOwner e_FindOwners v_Owners e_FindOwners v_NewOwner SHARED:NewOwner	 v_Owners e_FindOwners e_Search v_FindOwners e_AddOwner e_FindOwners v_NewOwner
NewOwners Model	

 v_NewOwner SHARED:NewOwner e_IncorrectData e_CorrectData v_IncorrectData v_OwnerInformation SHARED:OwnerInformation	 v_IncorrectData e_IncorrectData v_NewOwner e_CorrectData v_OwnerInformation
OwnerInformation Model	

 e_EditPet [numOfPets>0]
v_Pet
e_UpdatePet
e_AddVisit [numOfPets>0]
v_OwnerInformation SHARED:OwnerInformation INIT:numOfPets=0;
e_AddNewPet
e_AddPetSuccessfully / numOfPets++;
v_NewVisit
e_VisitAddedSuccessfully
v_NewPet
e_VisitAddedFailed
e_AddPetFailed
e_FindOwners
v_FindOwners SHARED:FindOwners
	 e_AddPetFailed
	v_NewPet
	v_FindOwners
	e_AddPetSuccessfully
	e_AddNewPet
	e_FindOwners
	v_OwnerInformation
	e_EditPet
	e_UpdatePet
	e_VisitAddedFailed
	e_AddVisit
	e_VisitAddedSuccessfully
	v_NewVisit
	v_Pet

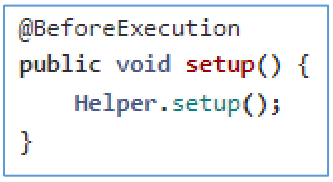
There are two main elements to consider during the implementation of the updated models. Firstly, there are multiple models; and we thus need a way to be able to move between them. Secondly, within the Owner Information model, there is need for a 'guard' to be added in order to prevent moving to a state before a given condition is met.

Whenever it comes to the first issue, you can see by the older models, multiple-model traversal is implemented by adding the SHARED keyword to a given vertex's text; however, this does not offer any functionality in GraphWalker Studio. Similarly, the guard is indicated by a conditional statement on a given edge travelling between two states. E.g. within the OwnerInformation model, numOfPets must be greater than 0 to travel between the NewVisit and OwnerInformation states. Again, simply adding text does not implement the desired functionality in GraphWalker studio. This applies to adding an action on the edge between two states such as incrementing the variable itself. So how do we achieve this? Let us look at this image below:



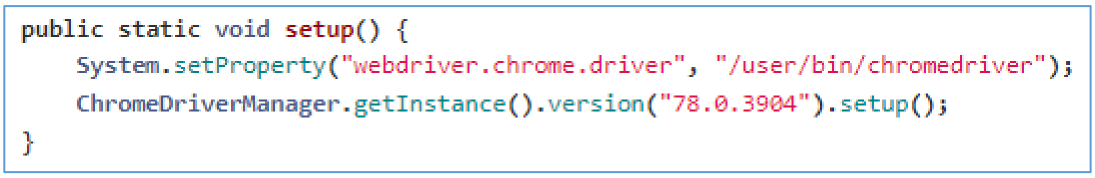
This menu can be accessed by highlighting an individual vertex or edge. As can be seen from the figure above, options exist to add SHARED, guard, and action functionality without having to add it as text within the respective vertex or edge, providing a more readable graph. Additionally, you can also see here that this is where we would dictate if a given vertex was the starting element for the model and if by reaching this state, we would have met any requirements. We will look deeper into the requirement functionality at a later stage though.
Now that we have been able to develop our models it is time to implement the GraphWalker test suite itself. Considering we will be using Kristian Karl's implementation that we have spoken about; this given implementation of GraphWalker will be built in Java, making use of the GraphWalker and Selenium Maven Plugins [10], [11] alongside base-java libraries for file reading and writing (Java.io) to be able to retrieve information and run scripts. The running of the scripts themselves will be discussed within the individual coverage implementation sections later within this implementation phase of this report.

As stated previously, with our intention to make use of Chromium DevTools, we will be converting the GraphWalker implementation to be using Selenium's Chrome Driver. Considering what we have mentioned previously within the Error and Exception Handling section of the design phase of this solution, we must account for the instability of the Chrome Driver release builds. The most logical fix for this is to simply set the version of the driver during the setup phase of the GraphWalker testing suite. This is achieved through using the @BeforeExecution tag that belongs to Selenium to ensure that the correct version of the driver is being used before the browser is ever started, as seen below:



```
@BeforeExecution
public void setup() {
    Helper.setup();
}
```

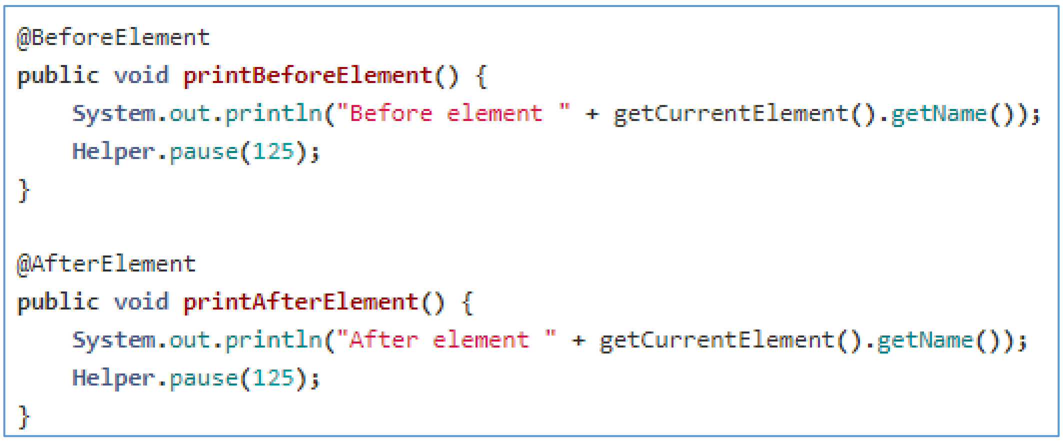
And inside the helper utility class:



```java
public static void setup() {
    System.setProperty("webdriver.chrome.driver", "/user/bin/chromedriver");
    ChromeDriverManager.getInstance().version("78.0.3904").setup();
}
```

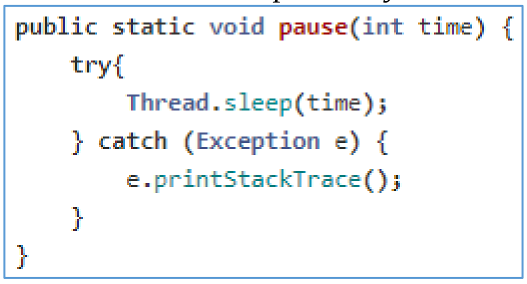
The logic itself can be hard coded into this Setup method in the main model, however we are making use of a modified Helper method in order to allow the same driver to be accessed in multiple methods as seen above.

Following this, it became clear that there was a fundamental issue with the switch due to the fact that the test suite itself was seemingly hanging on certain points and eventually failing; GraphWalker expecting something other than what was in-browser. Without delving heavily into the intermittent results of this, upon closer inspection the expected state was always one state before or after where the browser was. When comparing the FireFox implementation and the Chrome implementation it became clear that the Chrome driver was attempting to do each action at a much faster rate than its FireFox counterpart. This prompted a correction by adding an implicit wait after and before each state to effectively slow the execution of the test steps down slightly. This was done by making use of the @BeforeElement and @AfterElement functionality available by GraphWalker, as seen below:



```java
@BeforeElement
public void printBeforeElement() {
    System.out.println("Before element " + getCurrentElement().getName());
    Helper.pause(125);
}

@AfterElement
public void printAfterElement() {
    System.out.println("After element " + getCurrentElement().getName());
    Helper.pause(125);
}
```

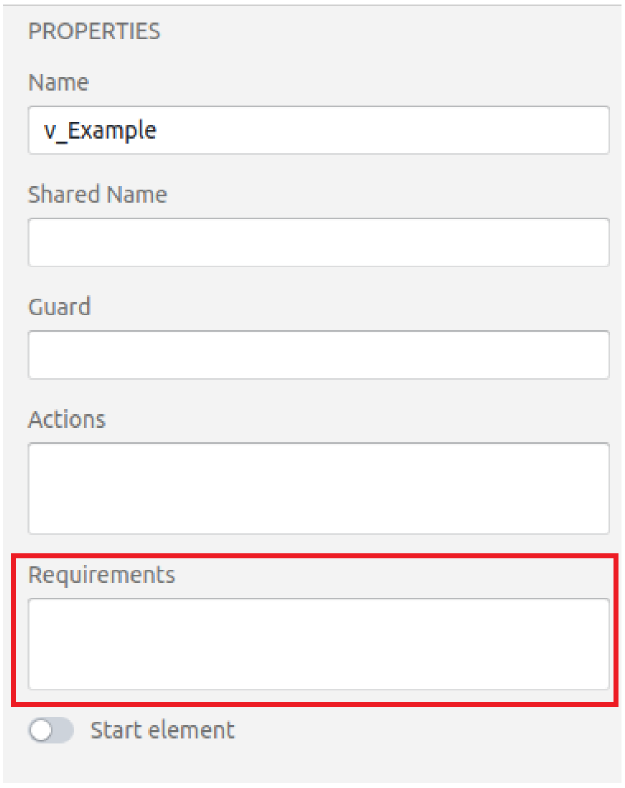
And inside the helper utility class:



```java
public static void pause(int time) {
    try{
        Thread.sleep(time);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

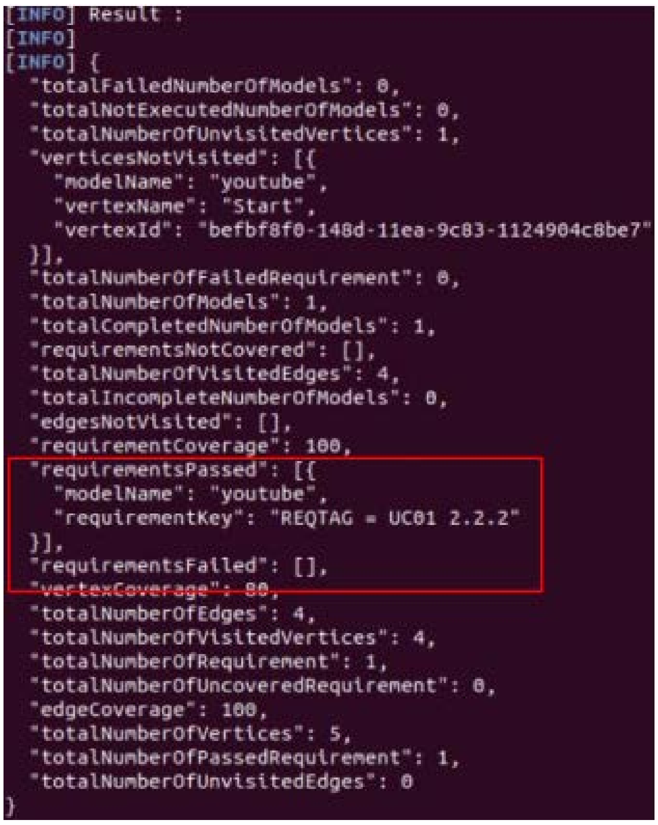
3.2 Requirement-level Coverage

The previous implementation of a GraphWalker suite for the PetClinic system did not include any requirement tracking. In other words, it had no REQTAGS attached to any of the states within the models. As is seen from other GraphWalker implementation examples [12] we would have previously added the tags via text into an individual state of a given model. However, as expressed previously within this report, this is not supported in GraphWalker Studio. Similarly, to how we added functionality for the SHARED keyword, guards, and actions; we will make use of the options available to use in the menu produced whenever a specific state or edge is highlighted, as seen below:



The requirement-tag text would be inserted into the highlighted box shown in the figure above. This would be done in the same format and the previous models (e.g. REQTAG=1.1). For the sake of having the most information available to us in regard to requirement traceability we will add a requirement to each of the states of each of the models.

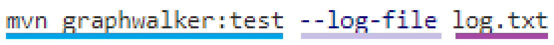
Despite now having active requirements we have not yet provided any implementation on how to track the coverage of them. As stated previously, we run our GraphWalker test suite through the use of GraphWalker CLI. This is achieved through invoking the command 'mvn graphwalker:test' within the Linux shell whenever we are in the directory of our GraphWalker implementation. At the point of test execution, GraphWalker will begin to dump all the output of the tests to the console in JSON format. GraphWalker takes note of how many requirement tags were present in your model, how many were traversed, and how many were not; thus, inevitably calculating how many of your requirements were ultimately covered within a given run of your test suite. This output would look similar to the following:



```
[INFO] Result :
[INFO]
[INFO] {
  "totalFailedNumberOfModels": 0,
  "totalNotExecutedNumberOfModels": 0,
  "totalNumberOfUnvisitedVertices": 1,
  "verticesNotVisited": [{
    "modelName": "youtube",
    "vertexName": "Start",
    "vertexId": "befbf8f0-148d-11ea-9c83-1124904c8be7"
  }],
  "totalNumberOfFailedRequirement": 0,
  "totalNumberOfModels": 1,
  "totalCompletedNumberOfModels": 1,
  "requirementsNotCovered": [],
  "totalNumberOfVisitedEdges": 4,
  "totalIncompleteNumberOfModels": 0,
  "edgesNotVisited": [],
  "requirementCoverage": 100,
  "requirementsPassed": [{
    "modelName": "youtube",
    "requirementKey": "REQTAG = UC01 2.2.2"
  }],
  "requirementsFailed": [],
  "vertexCoverage": 80,
  "totalNumberOfEdges": 4,
  "totalNumberOfVisitedVertices": 4,
  "totalNumberOfRequirement": 1,
  "totalNumberOfUncoveredRequirement": 0,
  "edgeCoverage": 100,
  "totalNumberOfVertices": 5,
  "totalNumberOfPassedRequirement": 1,
  "totalNumberOfUnvisitedEdges": 0
}
```

The relevant information that we are interested in is highlighted in red in the figure that is being shown above. This existing functionality of GraphWalker does indeed lighten our load whenever it comes to necessary implementation. However, this is only useful in a manual setting where a human would be actively looking at the console output. Therefore, we must capture the information so that it can be processed in order to extract the information that we would require.

Fortunately, given that Maven is used to run GraphWalker through cli, we can make use of its respective functionality. Maven offers a log-file hook that we can use when running our Graphwalker test suite through our command-line [13]. The notation for doing this would be implemented as so:



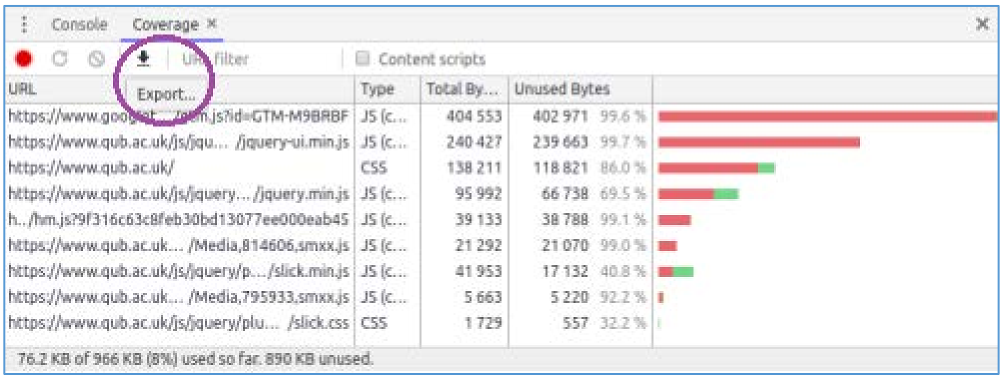
```
mvn graphwalker:test --log-file log.txt
```

The part of the command highlighted in blue is the original command, the part high-lighted in lilac is the hook, and the part high-lighted in purple is the hook in question, and the part highlighted in purple simply being the name and extension of the file where we wish to dump our console output to. We would still need to strip away the information in the log that we are not interested in so that we can retrieve the desired metric. However, given that we are to display this on our end-display-system, we will implement this functionality on its side.

3.3 Code-level Coverage

Client-side Coverage

The first element to implement is the client-side code coverage statistics. Essentially, which functions and whatnot that were hit that the browser itself is aware of without any knowledge or communication with the back-end system and its application code. As we have identified earlier in this report, we will be using the Chromium DevTools code-coverage tool offered in the Google Chrome browser. An example of this is seen below:



URL	Type	Total By...	Unused Bytes	
https://www.goo...t.../...n.js?id=GTM-M9BRBF	JS (c...	404 553	402 971	99.6 %
https://www.qub.ac.uk/js/jqu... /jquery-ui.min.js	JS (c...	240 427	239 663	99.7 %
https://www.qub.ac.uk/	CSS	138 211	118 821	86.0 %
https://www.qub.ac.uk/js/jquery... /jquery.min.js	JS (c...	95 992	66 738	69.5 %
h.../hm.js?9f316c63c8feb30bd13077ee000eab45	JS (c...	39 133	38 788	99.1 %
https://www.qub.ac.uk... /Media,814606,smxx.js	JS (c...	21 292	21 070	99.0 %
https://www.qub.ac.uk/js/jquery/p.../slick.min.js	JS (c...	41 953	17 132	40.8 %
https://www.qub.ac.uk... /Media,795933,smxx.js	JS (c...	5 663	5 220	92.2 %
https://www.qub.ac.uk/js/jquery/plu... /slick.css	CSS	1 729	557	32.2 %

76.2 KB of 966 KB (8%) used so far. 890 KB unused.

The image above is a snippet from the Google Chrome browser of the client-side code coverage from the QUB website [6]. Highlighted in the image is the manual export option that we are able to click. However, we of course need to automate this process. Selenium doesn't offer functionality to be able to open up the Chromium DevTool extension in order to mimic the manual export therefore our implementation must be achieved in a different manner.

In order to do this, we will employ the use of Node JavaScript in order unlock the ability for us to use Google Puppeteer [14] in order to interact with the API that lays behind the interface for the code-coverage tool offered by Chromium DevTools. Google Puppeteer is offered as a package available to Node JS. The first step of this scripting would be to first create a page object that we would be able to manipulate, like so:

```
const page = await browser.newPage();
```

Following this, we now have the core functionality of the script available to us through the use of the member of the page object, 'coverage'. This member allows us to invoke the methods required to begin to track the client-side code coverage from Chromium DevTools. This can be seen below:

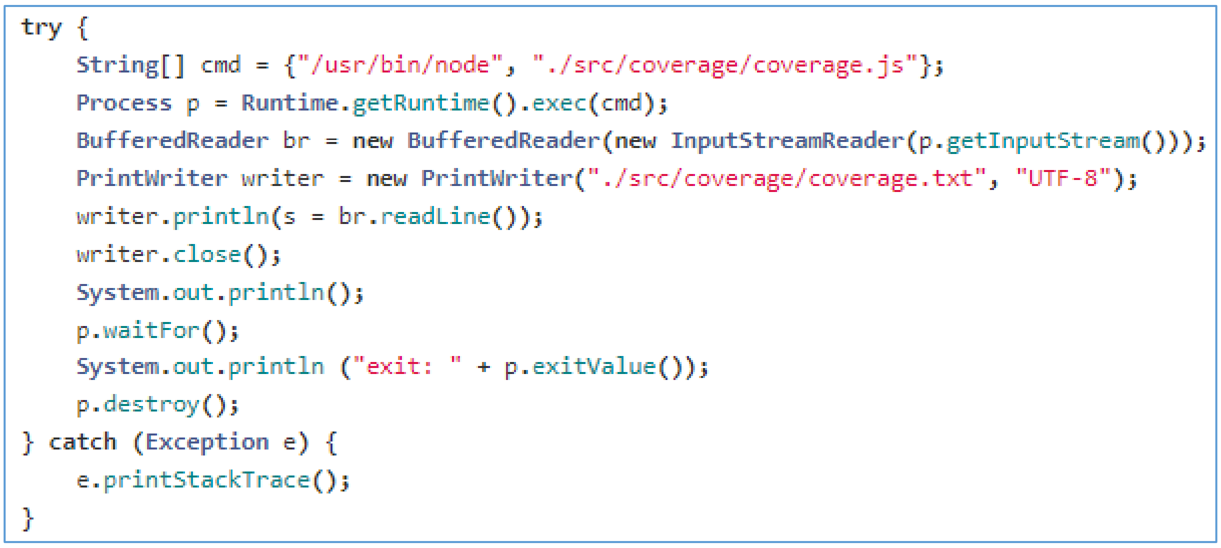
```
  await Promise.all([
  page.coverage.startJSCoverage(),
  page.coverage.startCSSCoverage()
]);
```

And to stop it:

```
//stop coverage trace
const [jsCoverage, cssCoverage] = await Promise.all([
  page.coverage.stopJSCoverage(),
  page.coverage.stopCSSCoverage(),
]);
```

At this point we would only need to perform some calculation to figure out the difference between what it was initially and what it was at the end and write the used figure to a file. The true difficulty comes with how exactly we would be able to run this script in automation during the execution of our GraphWalker test suite in order to be able to satisfy our requirements.

Given that this project is developed in and for a Linux based environment, much of our artifacts and system components are executed through the command-line. In Java, assuming the use of the 'io' base library, we are able to write a function that would be able to achieve this in automation. The implemented function is as seen below:



```
try {
    String[] cmd = {"/usr/bin/node", "./src/coverage/coverage.js"};
    Process p = Runtime.getRuntime().exec(cmd);
    BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));
    PrintWriter writer = new PrintWriter("./src/coverage/coverage.txt", "UTF-8");
    writer.println(s = br.readLine());
    writer.close();
    System.out.println();
    p.waitFor();
    System.out.println ("exit: " + p.exitValue());
    p.destroy();
} catch (Exception e) {
    e.printStackTrace();
}
```

The most notable piece of this function are the first two lines within the try-catch statement. Initially, we create a string array that holds two objects; firstly, the location of where node is installed on the developer's machine (default location is '/usr/bin/node') and secondly where the script is being held exactly. Note here that we make use of relative pathing so that the implemented solution would run on any Linux machine. The second line of code makes use of the in-built Process and Runtime classes that allow us to use our array object members to execute the script in an automated fashion. The code that follows simply captures the output of our floating shell and then writes said output to a file so that we can process it later for display within the Display System component of our solution.

Server-side Coverage

The other, and arguably more important side of code-level coverage analytics is the server-side code coverage. It is likely that if an external developer or team of developers were to make use of the implementation of this project, that they would be interested within the server-side code coverage analysis. The reason for this is that all of the application code of a given system, including the front-end code, inevitably lives on the back-end of the system; thus metrics based on the back-end of the system would give an individual a very clear indication of how well their code is being tested.

At previous points within this report we have set forth what we would like to achieve with server-side code coverage; that it is generated dynamically during the execution of our test suite against our SUT. So, how exactly would be set out to implement our design? The first step is to take our chosen tool (JaCoCo) and attach its agent to our JVM (Java Virtual Machine). This will ensure that the execution data will be generated based on our instrumented classes of our back-end system. This is achieved by adding the location of the JaCoCo Java Agent as a virtual-machine argument for our JVM.

There are a number of ways to do this. The option that may be chosen will depend solely on the individual's personal preference to how they would like to approach their work. The most generic way to achieve this is to add the java agent to you JVM args via your Linux command interpreter. Also known as your ‘profile’ file. This was implemented by opening the Linux console and entered the following command:

```
nano ~/.profile
```

Nano is a simple text editor that is built-into linux. You can optionally find the file and open it with any other preferred text editor. Now that the file is open to be edited, we will insert the following line at the top of the file before saving it:

```
export MAVEN_OPTS="-javaagent:/location/jacoco/lib/jacocoagent.jar"
```

Where “location” represents the pathway for where the JaCoCo jar folder lives. This causes all maven executed java systems to run with this argument connected to its JVM. The other options to achieving this is to add the JVM arguments to a specific SUT through the use of your IDE. This is supported in most popular IDEs such as Eclipse and IntellJ IDEs. However, this is not how we implemented it and therefore we will not go into detail with this.

It is at this point we can actually start up our SUT in order to be tested. Whenever it is running, it will start the process of collecting the execution data (EXEC data). Now, we can actually begin to run our tests against the system. Of course, this will be our standalone GraphWalker testing suite, but it is worthy to note at this point that this methodology would support manual testing also. Once the tests are complete, we will terminate the SUT. This can be done manually, through your IDE, or in an automated fashion with signalling [15] as I will show later within this report.

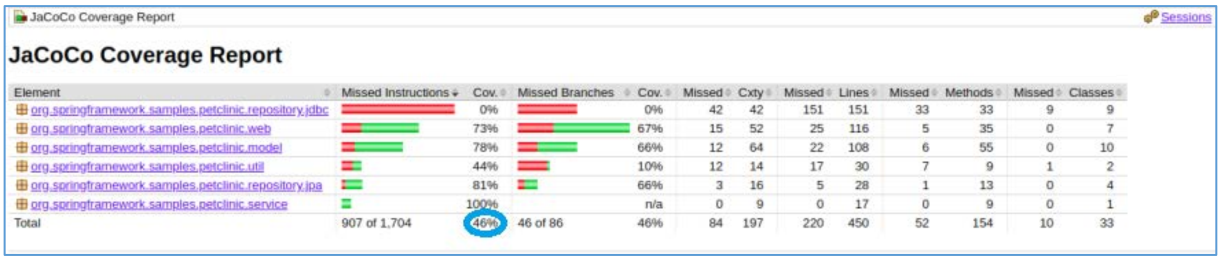
Once the SUT is terminated we will have the final version of the EXEC data file. Now we can generate a report of the server-side code coverage using JaCoCo itself. To remain in line with the rest of the development for this project we will opt to be using JaCoCo CLI for the report generation. As presented in the JaCoCo CLI documentation [16], this is achieved by calling java with the jacococli jar as an argument in our command line whilst passing our classfiles through. An example of this can be seen below:

```
java -jar jacococli.jar report [<execfiles> ...] --classfiles <path>
```

If we have multiple classfiles similar to this project, then we will need to pass the –classfiles option multiple times. In the end, the implemented version of this looks as follows:

```
java -jar ../jars/jacoco/lib/jacococli.jar report jacoco.exec --classfiles
./target/classes/cache --classfiles ./target/classes/dandelion --classfiles
./target/classes/db  --classfiles  ./target/classes/messages  --classfiles
./target/classes/org --classfiles ./target/classes/spring --html coverage
```

There are two things to note here. Again, we can see that we opted to use relative pathing so that it will run on anyone’s machine. In addition to this, we also opted to implement the optional –html option which will output a HTML based report to the ./coverage directory, creating said directory if it did not already exist. Here is the generated report for our SUT after our GraphWalker tests were ran against it:



JaCoCo Coverage Report

Sessions

JaCoCo Coverage Report

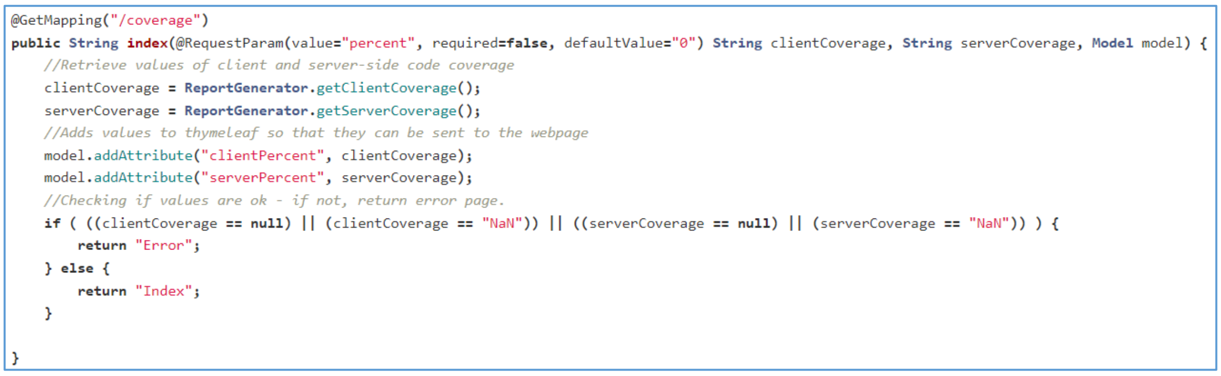
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.springframework.samples.petclinic.repository.jdbc	🟥	0%	🟥	0%	42	42	151	151	33	33	9	9
org.springframework.samples.petclinic.web	🟥🟩	73%	🟥🟩	67%	15	52	25	116	5	35	0	7
org.springframework.samples.petclinic.model	🟥🟩	78%	🟥🟩	66%	12	64	22	108	6	55	0	10
org.springframework.samples.petclinic.util	🟥🟩	44%	🟥	10%	12	14	17	30	7	9	1	2
org.springframework.samples.petclinic.repository.jpa	🟥🟩	81%	🟥🟩	66%	3	16	5	28	1	13	0	4
org.springframework.samples.petclinic.service	🟩	100%		n/a	0	9	0	17	0	9	0	1
Total	907 of 1,704	46%	46 of 86	46%	84	197	220	450	52	154	10	33

Highlights in blue is the metric we are most interested in, the overall server-side code coverage percentage metric. Additionally, an individual can click any of the links on the left-hand side of the report to further investigate what exactly was and was not covered.

3.4 Display System

Now that we have successfully generated each of the individual metrics that we require, it is now time for us to implement the system which will obtain, process, and display these metrics in a simplified and useful manner. As has been discussed, this system will take from in a Java Spring-boot web application. Considering that this system has been built from the ground-up, the available code has been well commended in order to allow for a full understanding. The architectural structure of this web-application will reflect that exactly of the structural diagram we presented for in within the design section of this report.

The main bulk of the functionality of how the webapp operates belongs to the ApplicationController class. Within this class we use the @GetMapping notation to create routes for the user hitting the web application when live. An example of one of these routing methods can be seen below:



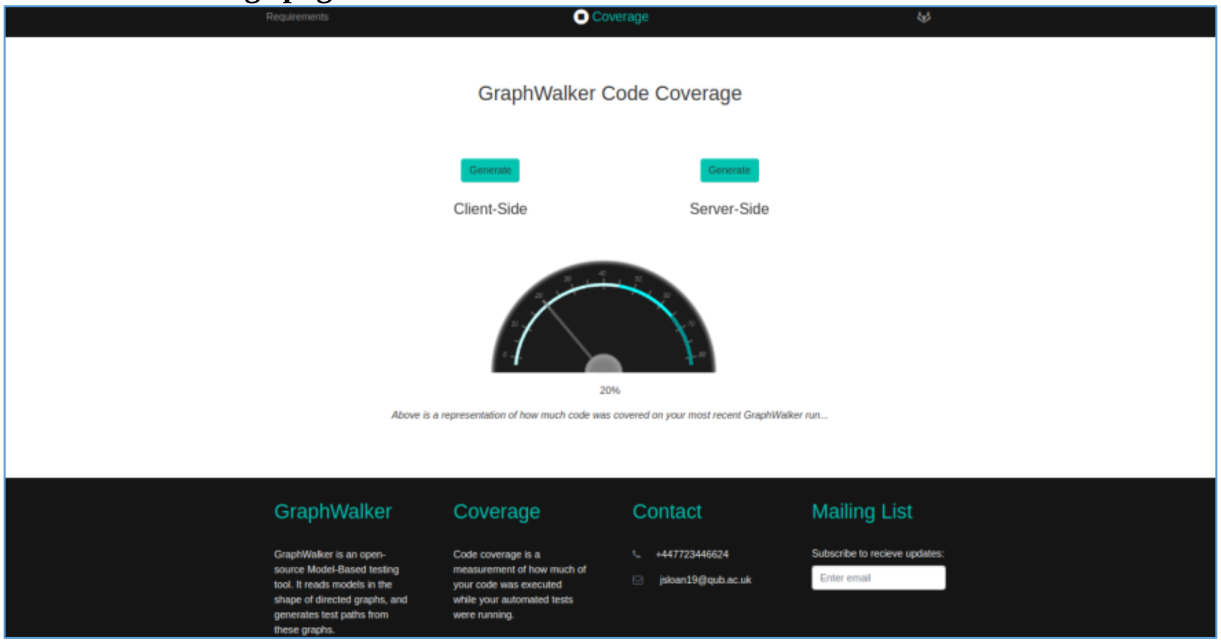
```java
@GetMapping("/coverage")
public String index(@RequestParam(value="percent", required=false, defaultValue="0") String clientCoverage, String serverCoverage, Model model) {
    //Retrieve values of client and server-side code coverage
    clientCoverage = ReportGenerator.getClientCoverage();
    serverCoverage = ReportGenerator.getServerCoverage();
    //Adds values to thymeleaf so that they can be sent to the webpage
    model.addAttribute("clientPercent", clientCoverage);
    model.addAttribute("serverPercent", serverCoverage);
    //Checking if values are ok - if not, return error page.
    if ( ((clientCoverage == null) || (clientCoverage == "NaN")) || ((serverCoverage == null) || (serverCoverage == "NaN")) ) {
        return "Error";
    } else {
        return "Index";
    }

}
```

Depending on what the user routes to (whether it be /requirements or /coverage) will dictate what metric is retrieved from the ReportGenerator class. We will delve deeper into said class at a later stage. Following this, we make use of the Thymeleaf framework that allows us to pass and eventually reference these metrics to and in the front-end HTML pages [17]. Spring-boot allows us to then return a specific string from this method which be resolved to a given HTML page that we have stored within our static resources folder. In order to address our expected error set out in our design phase of the metrics not being there or being the wrong data-type, we have implemented some error handling to check for null or NaN before returning the intended page. Otherwise, we would return an error page as seen above.

In terms of the front-end we made use of Bootstrap in order to produce some professional yet stylish HTML pages. The three main pages are the code-level coverage page, the requirement-level coverage page, and finally an error page to be displayed by default if something unintended has occurred during our pipeline. These pages are to be displayed below before we go into further detail regarding the implementation of this system.

Note, the screenshots are zoomed-out so that we can display all the webpage in one screen.

Code-level coverage page:



GraphWalker Code Coverage

Client-Side

Server-Side

20%

Above is a representation of how much code was covered on your most recent GraphWalker run...

GraphWalker

GraphWalker is an open-source Model-Based testing tool. It reads models in the shape of directed graphs, and generates test paths from these graphs.

Coverage

Code coverage is a measurement of how much of your code was executed while your automated tests were running.

Contact

+447723446624

jsloan19@qub.ac.uk

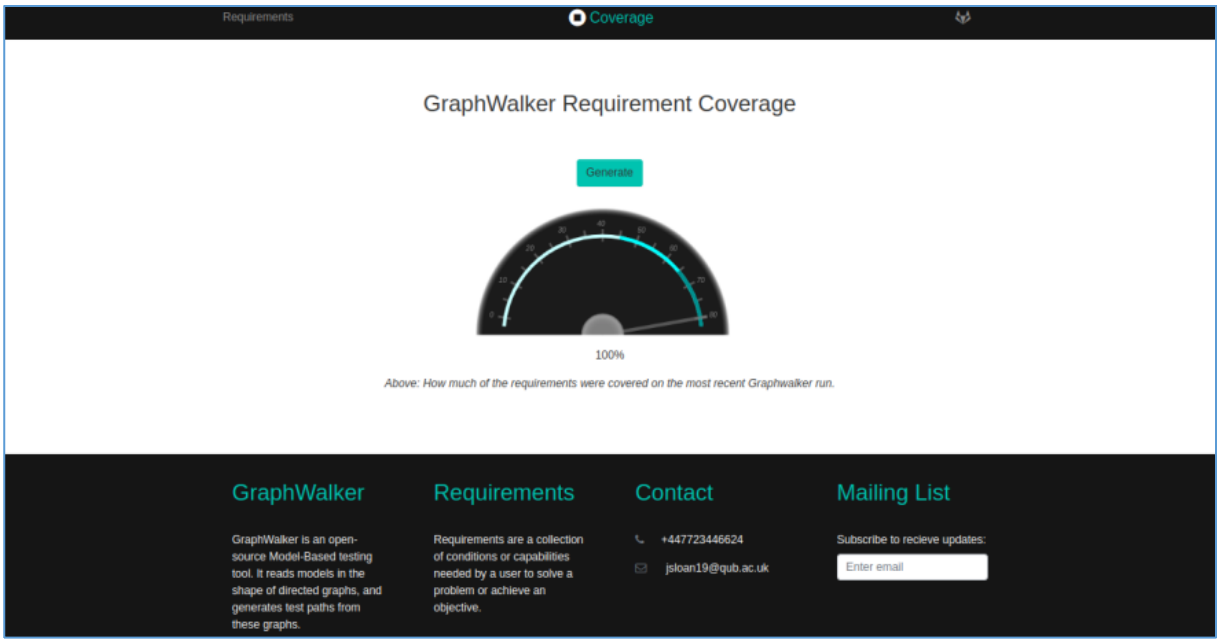
Mailing List

Subscribe to recieve updates:

Enter email

As seen from the image above, the user has the option to check either the client-side and server-side code coverage. The value displayed on and below the speedometer will change accordingly.

Requirement-level coverage page:



GraphWalker Requirement Coverage

100%

Above: How much of the requirements were covered on the most recent Graphwalker run.

GraphWalker

GraphWalker is an open-source Model-Based testing tool. It reads models in the shape of directed graphs, and generates test paths from these graphs.

Requirements

Requirements are a collection of conditions or capabilities needed by a user to solve a problem or achieve an objective.

Contact

+447723446624

jsloan19@qub.ac.uk

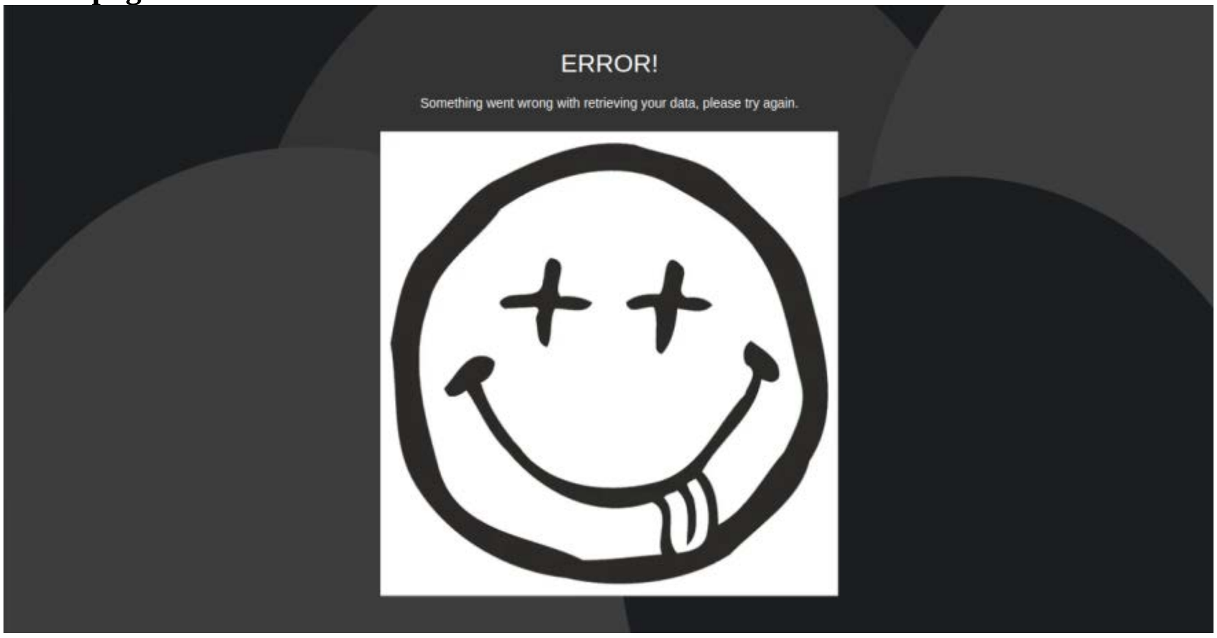
Mailing List

Subscribe to recieve updates:

Enter email

The requirement-level page offers the same design in order to retain a level of consistency. Of course, there is only one button to prompt the generation of the metric since there is only one value required on this page.

Error page:



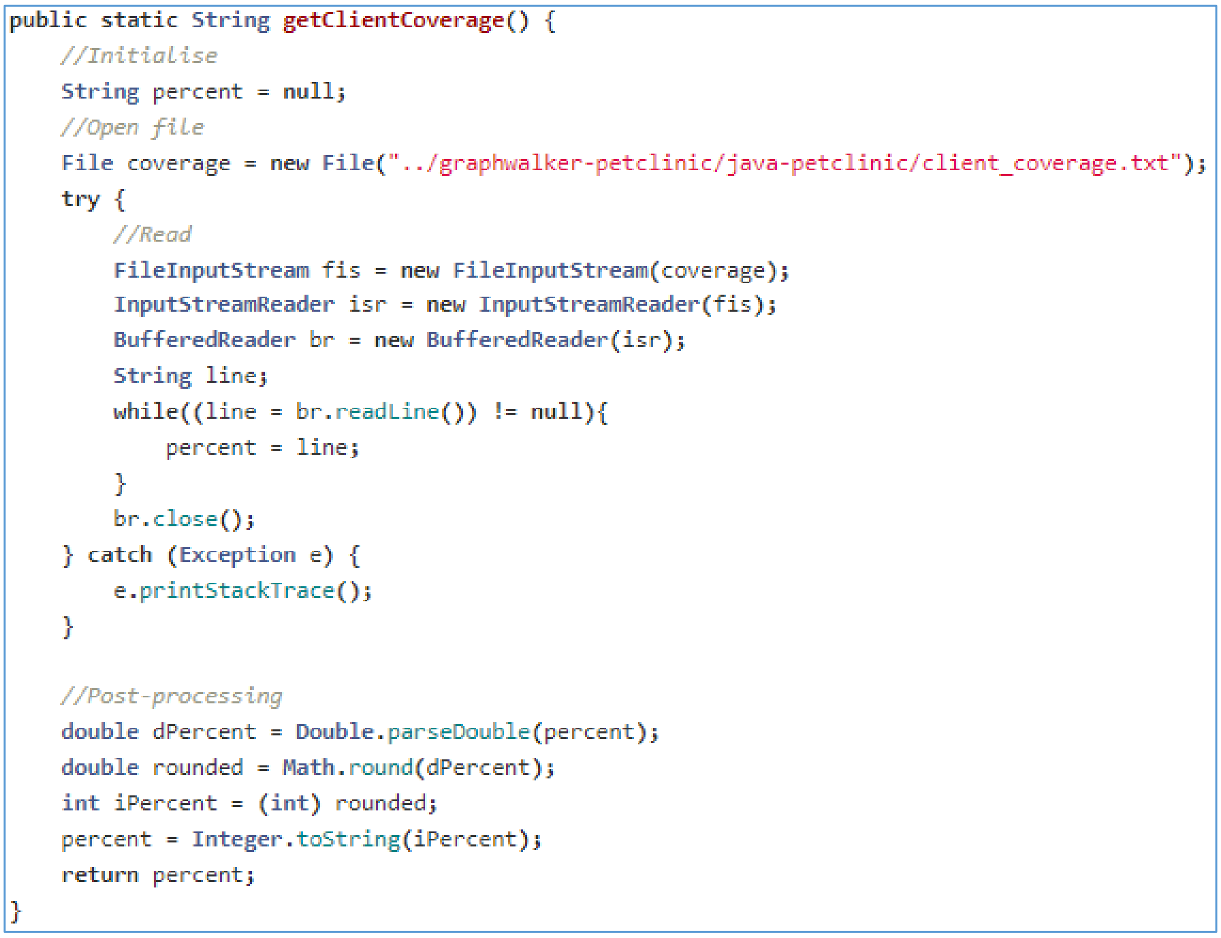
ERROR!

Something went wrong with retrieving your data, please try again.

The error page itself is just a simple rudimentary design as we wouldn't expect the user to end up on this webpage anyhow.

Let us now take the time to break down the functionality behind these webpages that have been displayed. Much of the functionality that exists within the actual HTML files of the front-end webpages are similar and therefore I will be explaining individual pieces of implementation but not showing every variant of it. However, before we look at the implementation of these webpages, let us first look at how we obtain and process the coverage metrics prior to sending them to the HTML level.

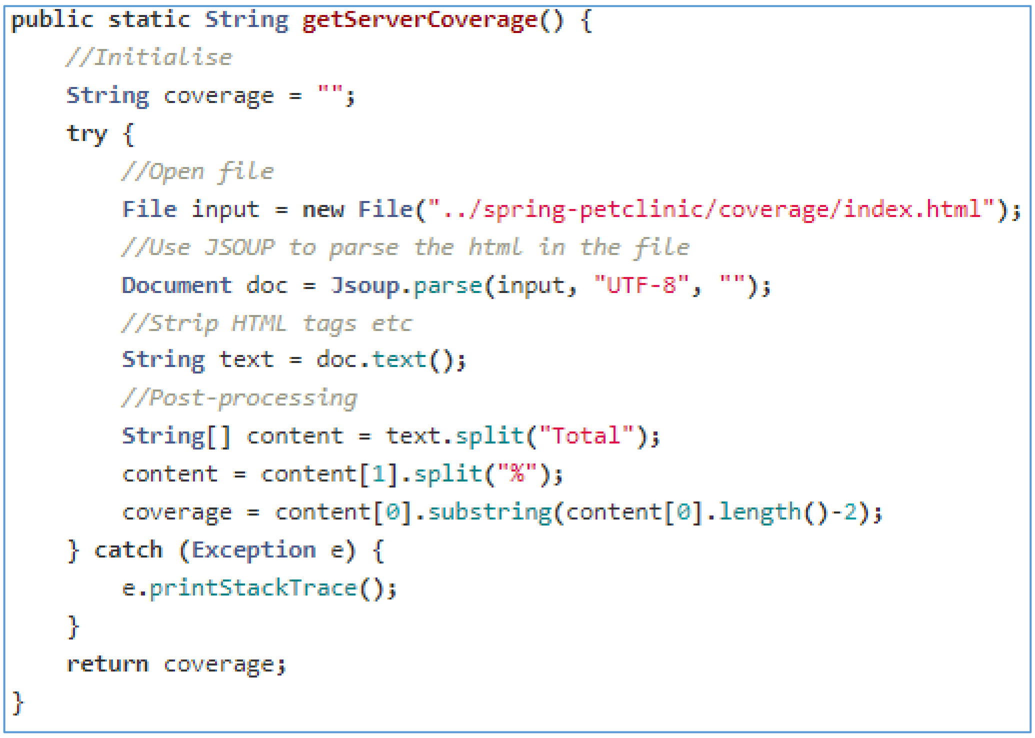
The functionality of retrieving and processing these metrics lives within the ReportGenerator class. This class consists of three methods; getClientCoverage(), getServerCoverage(), and getRequirmentCoverage(). These three different methods look for the files that were generated previously within the pipline by running our GraphWalker test suite against our SUT before performing some level of sting manipulation or processing in order to target the specific integer value for the given metric and then returning it. We will look at these methods individually, starting with the retrieval and processing of the client-side code-level coverage metric:



```java
public static String getClientCoverage() {
    //Initialise
    String percent = null;
    //Open file
    File coverage = new File("../graphwalker-petclinic/java-petclinic/client_coverage.txt");
    try {
        //Read
        FileInputStream fis = new FileInputStream(coverage);
        InputStreamReader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);
        String line;
        while((line = br.readLine()) != null){
            percent = line;
        }
        br.close();
    } catch (Exception e) {
        e.printStackTrace();
    }

    //Post-processing
    double dPercent = Double.parseDouble(percent);
    double rounded = Math.round(dPercent);
    int iPercent = (int) rounded;
    percent = Integer.toString(iPercent);
    return percent;
}
```

The client-side code-level coverage metric is a simple integer value written to a text file during the execution of our GraphWalker tests and therefore we can make use of the Java base-class io. Similarly to what we have seen previously within the report this is achieved through taking a relative file path and using the FileInputStream, InputStreamReader, and BufferedReader class to create an object that we can use to read the text file containing the client-side code-level coverage metric line by line. Once achieved, we simply round the value before casting it to a string so that it can be used on the webpage. One thing to note here is that we have accounted for the errors that we anticipated within our design in that we make sure that the file does exist, and if not, we throw an exception rather than allowing the program to fail.

Following this, we will take a look the implementation of the server-side code-level coverage metric method within this class. The implementation can be seen as follows:



```java
public static String getServerCoverage() {
    //Initialise
    String coverage = "";
    try {
        //Open file
        File input = new File("../spring-petclinic/coverage/index.html");
        //Use JSOUP to parse the html in the file
        Document doc = Jsoup.parse(input, "UTF-8", "");
        //Strip HTML tags etc
        String text = doc.text();
        //Post-processing
        String[] content = text.split("Total");
        content = content[1].split("%");
        coverage = content[0].substring(content[0].length()-2);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return coverage;
}
```

The implementation for the server-side metric is a little different that its counterpart. Considering that the value we are interested in exists within a generated HTML report document, we cannot use the standard java.io libraries and methods available to us. Therefore, we must make use of an external HTML parsing library. As stated previously, our chosen tool is JSOUP. Firstly, we use a relative path to create a file, passing the file and its Unicode datatype into JSOUP's parse() method, assigning it to an object of the Document class. In doing so, we get to make use of the text() method of the Document class, which strips away all of the HTML tags, leaving only the text behind. This value is held within the string variable text and we can now perform some string manipulation on it in order to target the specific metric value we are interested in. Once achieved, we return said metric. Again, we handle our expected errors in a similar manner as the previous method with some simple try-catch error handling.

Lastly, we will look at how we retrieve and process the requirement-level coverage metric:

```java
public static String getRequirementCoverage() {
    //Initialise
    String requirements = null;
    //Open file
    File result = new File("../graphwalker-petclinic/java-petclinic/log.txt");
    try{
        //Read
        FileInputStream fis = new FileInputStream(result);
        InputStreamReader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);
        String line;
        //Look for specified line
        while((line = br.readLine()) != null){
            requirements = line;
            if (requirements.contains("requirementCoverage")) {
                break;
            }
        }
        //Post-processing
        String[] aRequirements = requirements.split(":");
        requirements = aRequirements[1].replace(" ", "").replace(",", "");
    } catch (Exception e) {
        e.printStackTrace();
    }
```

This given metric is retrieved in a similar way to the client-side code-level coverage metric; however, the log-file generated during the test suite phase of this project is considerably larger and simple string manipulation would not cut it. Therefore, we take each individual line within the buffered reader and check if it contains the term “requirementCoverage” as the value we are interested in comes just after this. If found, it will break out of the loop and perform some post-processing string manipulation on it to obtain the return the value. Again, our error handling is completed in a similar manner.

These three different methods of the ReportGenerator class would be called from the individual routing methods from our ApplicationController class that we have spoke about previously. Let us revisit this to show how exactly we pass the metric values to our front-end.

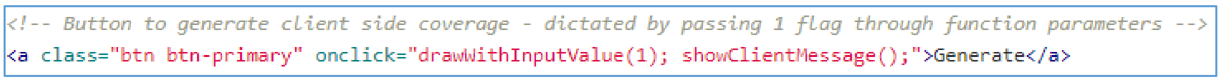
Within the routing methods of our ApplicationController class we make use of the Thymeleaf framework to add the variables containing the metrics to the front-end like so:

```java
model.addAttribute("clientPercent", clientCoverage);
model.addAttribute("serverPercent", serverCoverage);
```

Now that we have done this, we can access these variables dynamically from our HTML by making use of the ${…} notation. An example of this can be seen below:

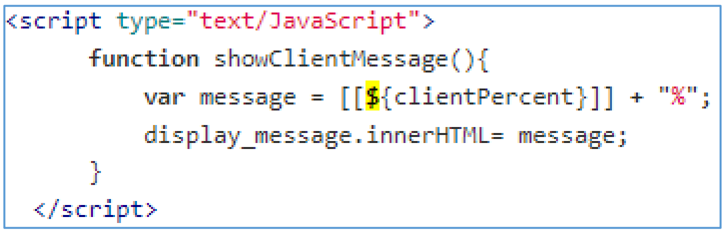
```html
<!-- Getting the values from the backend through thymeleaf and hiding them on the page. -->
<input type="hidden" id="txtSpeed" name="txtSpeed" th:value="${clientPercent}"/>
<input type="hidden" id="txtSpeed2" name="txtSpeed2" th:value="${serverPercent}"/>
```

We employ the use of the ‘hidden’ type to store the value on the webpage without the end-user being able to see it. Whenever the user is to click one of the buttons, two different functions are called as can be seen from this example below:



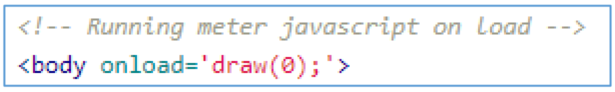
```
<!-- Button to generate client side coverage - dictated by passing 1 flag through function parameters -->
<a class="btn btn-primary" onclick="drawWithInputValue(1); showClientMessage();">Generate</a>
```

The first method drawWithInputValue() is responsible for updating the speedometer-style meter with the correct value, and the second method will display the exact percentage below said meter for clarification. The drawWithInput() method exists within our static JavaScript resources, however the other method simply exists in-line within the HTML webpage. An example of this can be seen below:



```
<script type="text/JavaScript">
      function showClientMessage(){
          var message = [[${clientPercent}]] + "%";
          display_message.innerHTML= message;
      }
  </script>
```

The additional square brackets around the ${…} notation is due to Thymeleaf having different notation for JavaScript files rather than just HTML. The only other notable thing that exists in-line is an onload function that draws the meter initially set to 0 whenever then user navigates to the webpage.

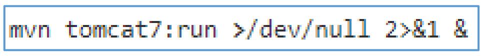


```
<!-- Running meter javascript on load -->
<body onload='draw(0);'>
```

Each of the metrics are treated in the same way regarding their respective buttons and how they are displayed. The only difference is of course their variable names and the flag that is passed through the parameters into the drawWithInputValue() method that updated the meter. This flag dictates which metric we are drawing.

3.5 Automation

The final piece of our implementation is to automate our pipeline so that it doesn’t have to be completed in steps. Considering that our implementation has largely been Linux command-line based, it would be sensible to implement our implementation through the use of a shell script. The main issue regarding this is that up until this point we would have kept the SUT running whilst we executed our test suite against it. However, this takes control of the terminal, meaning we would usually have opened an additional terminal. Given that shell scripts run encapsulated within their own subshell, this is not possible. The fix for this initial issue can be seen below:



```
mvn tomcat7:run >/dev/null 2>&1 &
```

‘/dev/null’ is essentially the trash-can of linux, the ‘>’ pre-fix esserntially redirects the contents of the standard output to this area. ‘2>&1’ ensures that standard error is directed to the same place and standard output, which is ‘/dev/null’. Lastly the final ‘&’ ensures that this is run in detached mode, meaning it does not grab control of the shell that the script is executing on, allowing command inputs to continue as the SUT is running.

Following this, we need a way to terminate the program now that we cannot use a user-interrupt. The initial approach for this was to send the same signal in order to terminate this (SIGINT or its numeric value, 2). However, this prompted issues as it would ask the user for confirmation on whether to terminate the application, making the automation process null and void. In order to combat this, we would opt for the killall java option over the standard kill option, as is seen below:

```
killall -w java
```

One additional thing to mention here is the use of the -w hook. The reason for this is to ensure that the application is definitely closed before the attempts to move onto the next step as this was noticed to cause some intermittent failures.

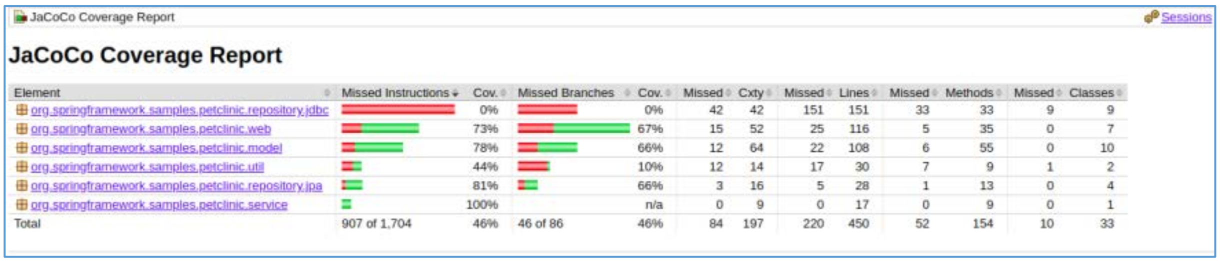
4.0 Example evaluation of the tool

Considering that we have finished the implementation of our solution, at this point it is most logical to have some level of verification. There are two areas in which the term verification can be applied. Firstly, we can have verification in the sense that we can verify the functionality of our project. Additionally, this can mean providing a means of verification for the end-user; for example, providing more in-depth analysis for their testing results to that they can verify the coverage findings.

Given the fact that this project is testing-based, it makes verification a little tricky. The simple solution to this issue is alter the values of our GraphWalker testing suite and see how the code-level and requirement level coverage is affected. Firstly, let's run our suite against our SUT with random edge coverage, aiming to cover 100 percent of all states and edges. This is achieved through this line of code within the main model.

```
@GraphWalker(value = "random(edge_coverage(100))", start = "e_StartBrowser")
```

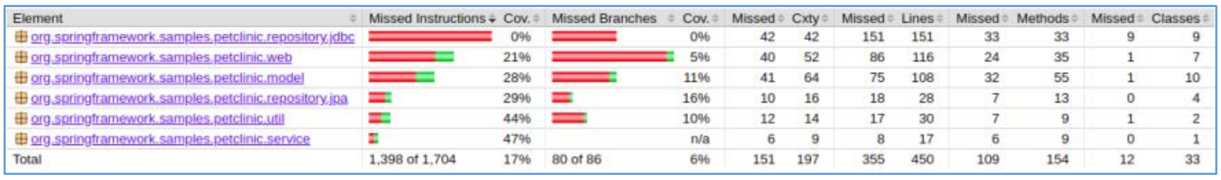
The resulting report was as follows, showing 46% total coverage.


JaCoCo Coverage Report
Sessions

JaCoCo Coverage Report

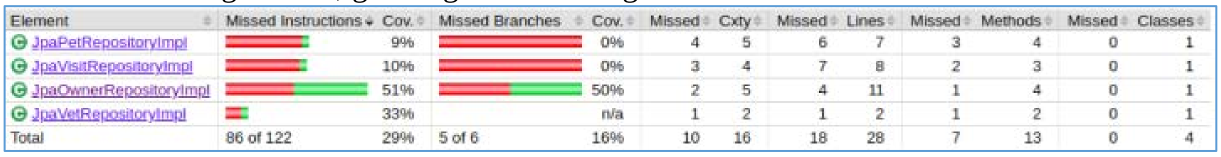
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.springframework.samples.petclinic.repository.jdbc		0%		0%	42	42	151	151	33	33	9	9
org.springframework.samples.petclinic.web		73%		67%	15	52	25	116	5	35	0	7
org.springframework.samples.petclinic.model		78%		66%	12	64	22	108	6	55	0	10
org.springframework.samples.petclinic.util		44%		10%	12	14	17	30	7	9	1	2
org.springframework.samples.petclinic.repository.jpa		81%		66%	3	16	5	28	1	13	0	4
org.springframework.samples.petclinic.service		100%		n/a	0	9	0	17	0	9	0	1
Total	907 of 1,704	46%	46 of 86	46%	84	197	220	450	52	154	10	33

Following this, we ran the same suite but this time manually interrupting it during its execution which should in turn lower the code-coverage percentage. As seen below, this was achieved with the report showing only 17% code coverage.



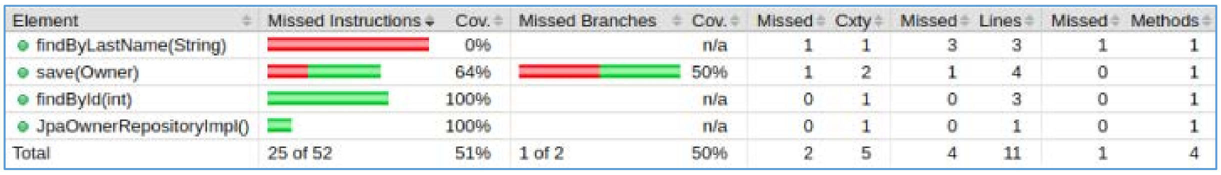
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.springframework.samples.petclinic.repository.jdbc		0%		0%	42	42	151	151	33	33	9	9
org.springframework.samples.petclinic.web		21%		5%	40	52	86	116	24	35	1	7
org.springframework.samples.petclinic.model		28%		11%	41	64	75	108	32	55	1	10
org.springframework.samples.petclinic.repository.jpa		29%		16%	10	16	18	28	7	13	0	4
org.springframework.samples.petclinic.util		44%		10%	12	14	17	30	7	9	1	2
org.springframework.samples.petclinic.service		47%		n/a	6	9	8	17	6	9	0	1
Total	1,398 of 1,704	17%	80 of 86	6%	151	197	355	450	109	154	12	33

If the end-user were to want to verify any of their findings, they are able to click any of the links shown in the image above, granting the following:



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
JpaPetRepositoryImpl	🟥🟩	9%	🟥	0%	4	5	6	7	3	4	0	1
JpaVisitRepositoryImpl	🟥🟩	10%	🟥	0%	3	4	7	8	2	3	0	1
JpaOwnerRepositoryImpl	🟥🟩	51%	🟥🟩	50%	2	5	4	11	1	4	0	1
JpaVetRepositoryImpl	🟥🟩	33%		n/a	1	2	1	2	1	2	0	1
Total	86 of 122	29%	5 of 6	16%	10	16	18	28	7	13	0	4

As you can see from the image above, this offers further in-depth details as it displays the coverage break-down for each induvial class within this section of the code. If any of these class hyperlinks are clicked you will be able to receive a full breakdown of the individual methods, as seen below:



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
findByLastName(String)	🟥	0%		n/a	1	1	3	3	1	1
save(Owner)	🟥🟩	64%	🟥🟩	50%	1	2	1	4	0	1
findById(int)	🟩	100%		n/a	0	1	0	3	0	1
JpaOwnerRepositoryImpl()	🟩	100%		n/a	0	1	0	1	0	1
Total	25 of 52	51%	1 of 2	50%	2	5	4	11	1	4

References

1. James Sloan. (2019). CSC3002 Project. Available: https://gitlab2.eeecs.qub.ac.uk/40131612/csc3002-vg05. Last accessed 24th April 2020.
2. Kristian Karl. (2019). *GraphWalker*. Available: https://graphwalker.github.io. Last accessed 24th April 2020.
3. Spring. (2020). *PetClinic.* Available: https://projects.spring.io/spring-petclinic/. Last accessed 24th April 2020.
4. Kristian Karl. (2020). *GraphWalker PetClinic.* Available: https://github.com/GraphWalker/graphwalker-example/tree/master/java-petclinic. Last accessed 24th April 2020.
5. yworks. (2001). *yED.* Available: https://www.yworks.com/products/yed. Last accessed 24th April 2020.
6. Queen's University Belfast. (2020). *QUB Website.* Available: http://qub.ac.uk. Last accessed 24th April 2020.
7. Node. (2009). *Node JS.* Available: https://nodejs.org/en/. Last accessed 24th April 2020.
8. Kristian Karl. (2020). *Amazon Shopping Cart.* Available: https://github.com/GraphWalker/graphwalker-project/wiki/Amazon-Shopping-Cart. Last accessed 24th April 2020.
9. Draw.io. (2017). *Draw io.* Available: https://app.diagrams.net. Last accessed 24th April 2020.
10. GraphWalker. (2014). *GraphWalker Maven Plugin.* Available: https://mvnrepository.com/artifact/org.graphwalker/graphwalker-maven-plugin/3.0.0. Last accessed 24th April 2020.
11. Selenium. (2011). *Selenium Java Maven Plugin.* Available: https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java. Last accessed 24th April 2020.
12. Kristian Karl. (2016). *GraphWalker Examples.* Available: https://github.com/GraphWalker/graphwalker-example. Last accessed 24th April 2020.
13. Rapt. (2012). *Send maven output to file.* Available: https://stackoverflow.com/questions/9726875/send-maven-output-to-file. Last accessed 24th April 2020.
14. Google.(2019). *Puppeteer.* Available: https://developers.google.com/web/tools/puppeteer. Last accessed 24th April 2020.
15. Zachary Brady. (2016). *List of kill signals.* Available: https://unix.stackexchange.com/questions/317492/list-of-kill-signals. Last accessed 24th April 2020.
16. JaCoCo. (2009). *JaCoCo CLI.* Available: https://www.jacoco.org/jacoco/trunk/doc/cli.html. Last accessed 24th April 2020.
17. Thymeleaf. (2018). *Thymeleaf Documentation.* Available: https://www.thymeleaf.org. Last accessed 24th April 2020.