GaugeDependExtended: A tool for automated visualisation of BDD test suites and their web service dependencies

Cameron Brush, Vahid Garousi
Queen's University Belfast, UK
{cbrush01, v.garousi}@qub.ac.uk

Abstract:

Software testing is an industry standard for web developers and software engineers. When developing a web application, engineers have found that the test code often has errors and inefficiencies. To help prevent this, continual quality assessment and maintenance is necessary to ensure that the tests are repeatable, predictable, and efficiently executed. Building on the application, GaugeDepend [11], the primary aim of this project is to extend the tool in order to create a new tool, "GaugeDependExtended", that will visualise dependencies between test suites and web services. The visualisation of client-side and server-side test dependencies will allow software test engineers to conduct dependency analysis and assist with quality assessment and maintenance of the test suites. We have applied the tool, "GaugeDependExtended", to test suites of 3 Systems Under Test (SUT) and have observed the unique benefits of the tool.

Contents

1 Introduction and Problem Area

1.1 Test Quality

Every programmer or software analyser must test code at some point in their career. It is the best point of action to assure the quality of software. So, the importance of clear and concise tests with easily understood results cannot be overstated. As tests are written, especially for web applications, each test can become dependent on other methods and test suites. For web developers these particular problems arise with client-server architectures, that makes testing more challenging [1]. So, as well as writing the tests software engineers must also keep track of the dependencies between tests and (for web applications) what objects and calls are being tested.

In today's landscape with the ever-growing popularity of web applications [3], it is becoming more difficult to analyse the calls between each service. Furthermore, with these programs becoming larger, most of, if not all, the program's test suites are automated. In these large systems the scale of the automatic test suits can be, and usually are, massive. When adding the short time-to-market of applications, often resulting in a lot of crunch time, the testing of web applications is usually neglected [1]. This makes analysing the code quality much more difficult, especially with the added complexity of calling web services.

1.2 Dependency analysis

As the test suits gather more dependencies between classes, web services and each other, the clarity of these dependencies becomes complicated. With the added time restraints [2] of human analysers investigating the test results, the reasoning behind each dependency becomes lost.

End-to-end test automation techniques are widely used by web developers e.g., Gauge. These are used to assert a correct outcome from given method using automated test scripts. This scripts automate the manual process that the user will conduct on a web application's interface, usually pressing buttons or filling out forms [13] – this is where dependencies will quickly become overwhelming.

To make these dependencies more digestible and to use them more effectively, to pinpoint faults, a common feature of dependency analysis tools is the ability to generate a dependency graph. A dependency graph is a visual representation of the relationships between different parts of a system. Such attempts have been made before to create applications that will automatically generate these dependency graphs e.g. Tarantula [4] or TeCReVis [9], an eclipse plugin. TeCReVis is a good example of a dependency analysis tool, but I would like to develop a tool that does not necessarily need to be used with eclipse. Furthermore, there has also been attempts to visualise the architecture of service calls being used within programs e.g. Kizceral: A Dynamic Real time Architecture Diagram [6].

However, there is a need to combine these programs to make an open-source application that visualises dependencies between test suits and the web services called by them. More and more test researchers are starting to believe that the visualisation of test suits and the overall coverage of tests will be extremely beneficial for testers when it comes to understanding them effectively [5].

1.3 Web development testing and Gauge Test code analysis

Model based testing is used a lot in web development. MBT testing allows the user to follow the flow of the application and test accordingly. The test engineer will use a state-diagram to represent the flow of the application. It is stated that when considering MBT in particular "Model Based Testing is the automated dynamic verification of software behavior under programmed model control" [14]. As described in the paper "Test automation with the Gauge framework: Experience and best practices" [8], MBT is the most suitable approach for testing web applications. We try to keep it as simple as possible when writing tests, so the best case scenario is to use a streamlined MBT approach to build test cases and then have human testers write Gauge test scripts focusing on them.

Thus, test engineers can work without having to use a lot of complicated equipment. There are tools that have been developed to help make testing more understandable from the root, the writing of tests. Gauge, as mentioned above, is a free and open-source framework for writing and running automated acceptance tests. It has a simple syntax that allows tests to be written in plain text and is broken down into "Specification" files and "Concept" files [7][8]. Each Gauge step must by implemented by code in order to execute. Gauge step implementations can be written in Java, Python, Ruby or Java Script. How Gauge is used is shown in the *Figure 1* below.

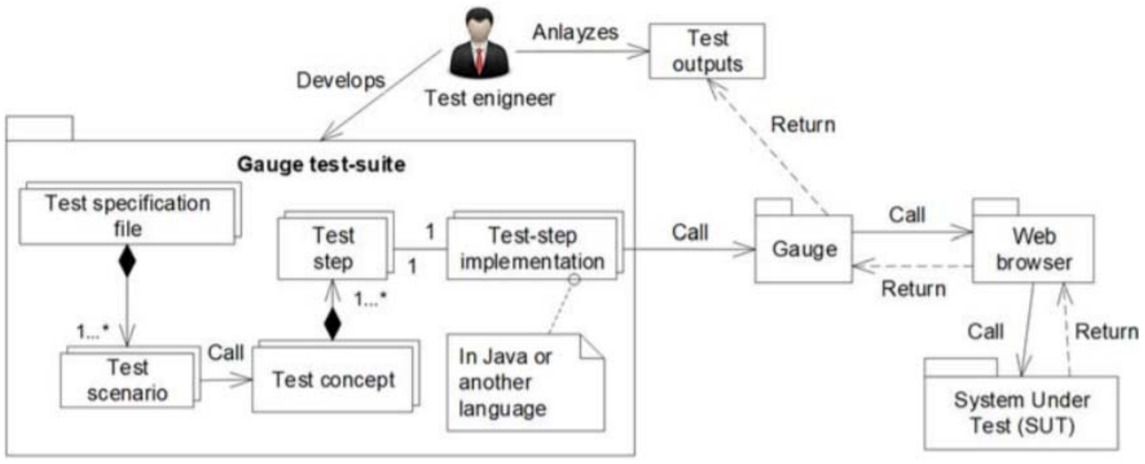


Figure 1. Architecture of test automation and the building blocks of Gauge test scripts [7]

Gauge has many benefits compared to other automated testing frameworks:

- It can help you write the specs in the language you prefer in your business which does not need to be English
- The reusability of Gauge is one of the main features which makes it stand apart. It allows the reuse of test steps from one scenario to another scenario. It can also define steps to run before each scenario while writing the specifications.
- Gauge has huge support of plugins. Plugins in gauge can be written in any language of your choice and can be used to extend its functionality.

Although, having tests written in plain text still is not enough to visualise the complexity of code dependencies. Poor quality Gauge tests can still be written, which allows for all the associated issues of poor-quality test code.

1.4 GaugeDepend

GaugeDepend [11] attempts to combine the usefulness of dependency analysis graphs and the simplicity of Gauge testing to generate dependency graphs from Gauge specification files. The app allows test engineers to conduct dependency analysis and quality assessment of a given Gauge test suite using graphs. It contains a GUI and command line interface that allows users to customise and render the graphs, as well as explore and manipulate these graphs. The graphs show the relationships between Gauge Scenarios, Steps and Concepts. If a Scenario calls a step, then it is said that that scenario has a dependency on that step. If a Concept includes a step, then it is said that that concept has a dependency on that step. Then the dependency between scenarios and concepts are modelled the same way as the dependency between scenarios and steps.

GaugeDepend is a useful app but it still does not tackle the problem of analysing dependencies with the web applications under tests. When analysing test dependencies for web application it is much more useful to have more detailed breakdowns of each dependencies, it is not enough to just show the dependencies between step, concepts, and scenarios. A better analysis tool should also show what objects are being tested on each step i.e., Client-side tests or server calls i.e., server-side tests, that are being made during the step. These functions are not a part of GaugeDepend in its current version.

Below, in *figure 2*, is a screen shot take from the current demo of GaugeDepend showing a snipped of one of the dependency graphs mentioned above

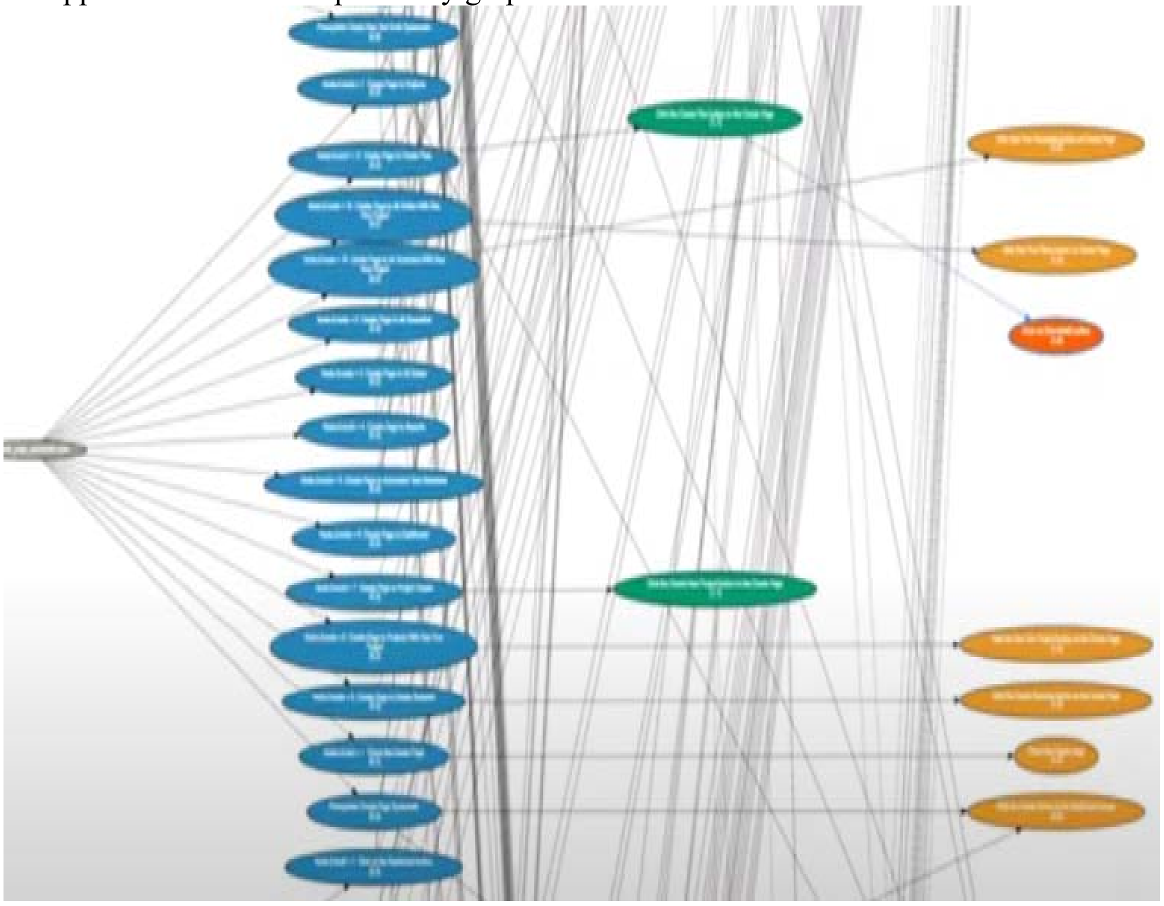


Figure 2- Mock Gauge Depend dependency analysis graph

2 Solution Description and Requirements

2.1 Solution

The aim of this project is to improve and build upon GaugeDepend producing a new application called GaugeDependExtended, that will be a more useful application for web developers and test analysists who are working with web applications.
The app GaugeDepend, as mentioned above in section 1.4, helps with the problem of analysing test dependencies and this project extends this functionality by adding new

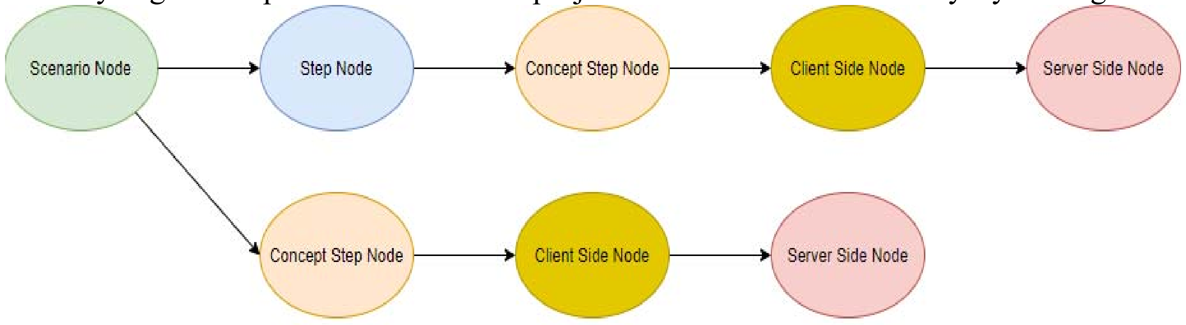


Figure 3- Mock dependency analysis graph with web service nodes

nodes to show the dependency between client-side tests and server-side test against the already implemented test suite dependencies. A mock graph showing how these nodes will be added to the previous nodes is shown above in *figure 3*. As well as showing the location of client-side objects using paths or element names by reading premade resource files. A GUI has been made that allows users to select specific test suite and resource files locations to then generate dependencies graphs from specifically selected specification and concept files. Users can also customise these graphs to personal preference. The graph is rendered and displayed in the GUI.

This solution will help solve the problems mentioned in section 1 by helping software engineers understand the dependencies between client-side and server-side tests and their test suits, so that dependency analysis can be done more efficiently and effectively.

2.2 Advantages of this Solution

Like the previous project due to the language agnostic nature of Gauge, there is still the possibility that the code in the underlying step implementation is of subpar quality. In most cases when a node has unusual text that is not related to a step name, client-side object, or a server call, this is usually related to syntax errors in the specification. Other than that, it cannot help with implementation errors. Although it can help reduce redundant testing of the same web page or server calls, as objects that depend on multiple tests are easily identifiable. The project can automatically analyse dependency graphs for a test suite and produce a graph identifying the client-side and server-side tests.

It can also help with code comprehension. The application combats the problem mentioned above in section 1.1 where, after creating a large, automated test script, the clarity of the tests becomes lost. For example, if a new person were to join a team, having a clear way to examine the relationships between all the existing scenarios and web services using the generated graphs, would assist in getting them up to speed with the test suite. Also helping them understand the consequences of modifying or updating existing tests, as removing, or adding tests can completely change the dependencies among the entire test suites.

Being able to easily see the location of objects with test dependencies is also beneficial. Large web applications often have hundreds of objects such as buttons and textboxes, that can easily become difficult to keep track of thus having the location of these objects that are being tested will reduce time spent searching the SUT for specific objects. Requiring users to create a json file listing objects and their locations is also beneficial for future development as it will make code handovers run a lot smoother.

Clearly being able to see the test steps that have both client-side and server-side relationships is also very beneficial when following MBT. For example, if a test step has an edge connected to a client-side node that represents a button object which, when clicked, is supposed to link the user to a different page, we should be able to see that that same step is linked to a server-side node that calls that page. This lets the user verify that a connection in their state-diagram has been met.

Evidence of these advantages are shown below in section 6

2.3 Requirements

I have broken the requirements into sections, as many of the requirements are in similar categories to each other.

Graph aesthetics: This is simple requirements for the look of the graphs produced by the application.

- The application should be able to generate and display graphs for single spec files, single concept files and combinations of any number of spec and concept files.
- Scenarios, concepts, steps, client-side steps, server-side steps will be unequally identifiable from visuals e.g. colour coded, different shapes.
- Must be able to easily see each dependency
- Aesthetically pleasing graphs without loss of information

Graphical representation: These requirements are for what information the graph represents.

- Vertices should represent scenarios, concepts, steps, client-side steps, server-side steps.
- Client-side nodes must show object location e.g. xpath, tag name
- Edges between vertices will represent dependencies.
- Should be able to distinguish between different forms. For large graphs made up of lots of source files the user should be able to easily identify which form a scenario is from

Usability and Customisation: These are requirements and features that allow the user to change and customise how their graphs look and what they are viewing.

- Must be able to easily import resource folders for client-side paths
- Must be able to filter graph to view specific dependencies e.g. One specific form.
- Must be able to change distance between nodes
- Must be able to change the size of the font
- Must allow for a reasonable amount of further customisation

Scalability and Extendibility: These are needed to allow the application to be used and modified by the wider community.

- Must be scalable. Handle large inputs and display them proportionally. If a large spec file is input, it should not be displayed in a way that is difficult to navigate (a problem with the previous project).
- Must be open source. Contribute to the worldwide software engineering community
- Must be extendable. Meaning, it should be easy to add additional nodes with minimal cod changes.

3 Design

3.1 Program workflow

The core workflow of the previous project has not changed drastically. There have been various steps added to each stage of the workflow, that change or add functionality to each step. Shown below in *figure 4* is the previous workflow with changes that have been made. The process starts by turning the selected spec or concept file into a markdown string. Then the markdown string is combed using regex expressions to search for unwelcome strings, things like comments and or quotations. The markdown also splits some desirable strings. This sanitisation needs to be done to remove any unneeded information from the specification files and to insure there any text that could interfere with graph generation is removed.

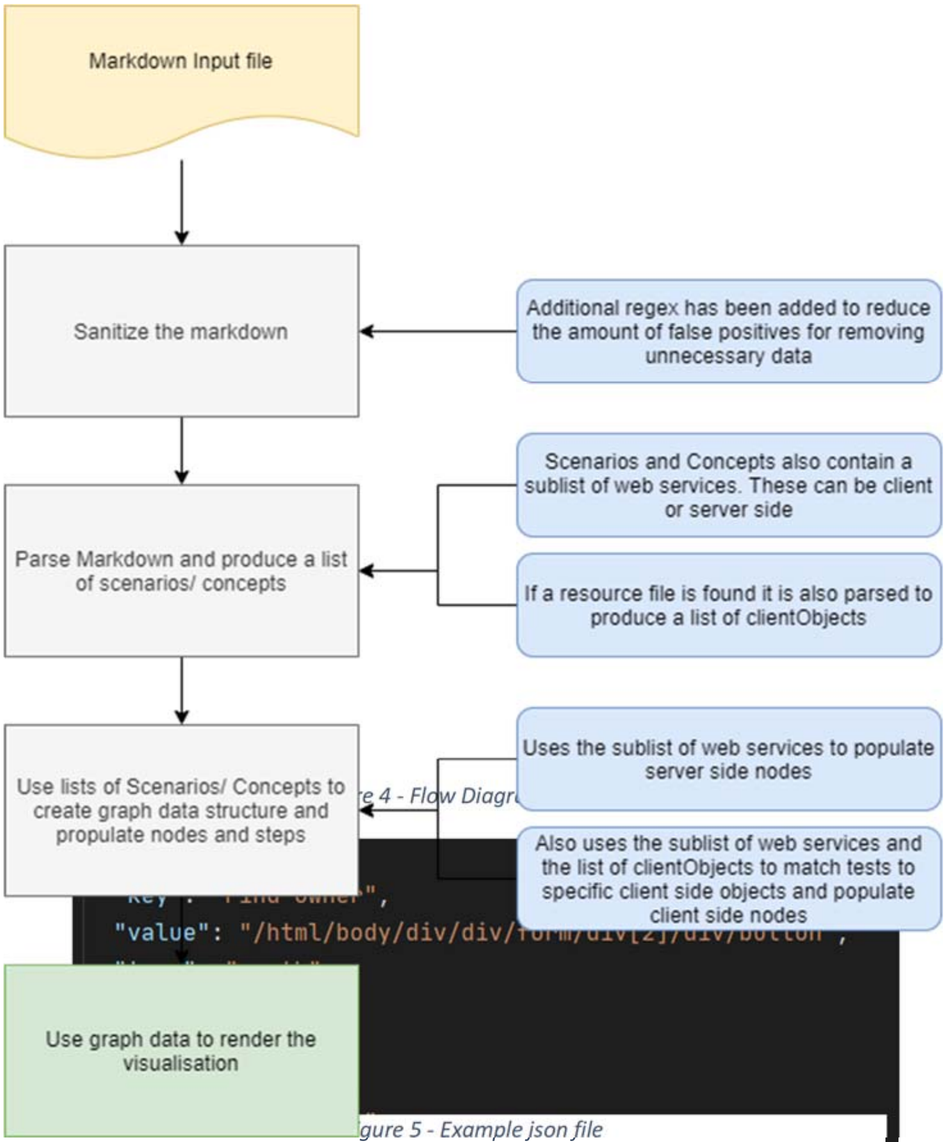
Decisions had to be made on how client-side and server-side nodes would be identified as there are no key features specified in Gauge Specification for these web services. The identification starts during the markdown processing where steps that contain "http" or "https" are separated. These are separated to then be processed during parsing and graph generating.

Once the markdown is sanitised to a format that is desirable it is then converted to html to be parsed. The parsing process uses three models that are defined in a data class: Scenarios, Concepts and Client Objects. There are populated in three steps:

- Spec parsing. Gauge specification files after being put through the markdown process are split into Scenarios containing two lists of steps and web services.
- Cpt parsing. Gauge concept files after being put through the markdown process are split into Concepts containing lists of steps and web services
- Resource parsing. This is an optional step, but it is required for client-side nodes. It takes a provides json file with a specific format (shown in *figure 4*) and appends it to a client Object model. The model contains three lists of "keys", "values", and "types". The use of a resource file for the client-side nodes is useful as it allows the testers to provide exact locations for objects being tested. Requiring the user to present a list of objects with their paths can also help increase test code quality.

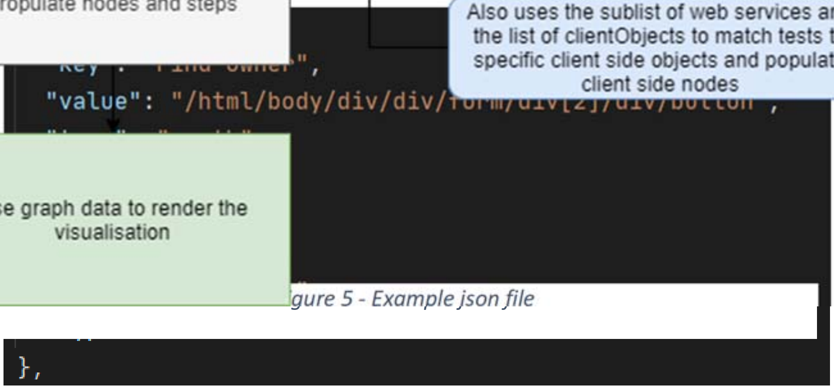
The next step is to take this data and use it to generate the graphs. During this step, the graphs are populated with edges connected nodes representing the relationship described in the above sections. The web services are now split into server-side and client-side nodes. If the web service includes "http" or "https" it is put into the server-side category. Then it checks if there is a resource file found for this Gauge Spec or Concept, if there is one, it then checks if each step references an object in the client Object list. Previously all variables were removed from step strings during markdown processing but since variables are often related to object they are now kept. Edges and nodes are then made for the client-side object. The Graph objects are saves for each specification and concept and then can later be combined using a method in the Generator Class.

Finally, the graphs are rendered in a hierarchical manner for visual representation.



Markdown Input file
Sanitize the markdown
Additional regex has been added to reduce the amount of false positives for removing unnecessary data
Parse Markdown and produce a list of scenarios/ concepts
Scenarios and Concepts also contain a sublist of web services. These can be client or server side
If a resource file is found it is also parsed to produce a list of clientObjects
Use lists of Scenarios/ Concepts to create graph data structure and propulate nodes and steps
Uses the sublist of web services to populate server side nodes
Also uses the sublist of web services and the list of clientObjects to match tests to specific client side objects and populate client side nodes
Use graph data to render the visualisation

e 4 - Flow Diagr



```
"value": "/html/body/div/div/
},
```

gure 5 - Example json file

3.2 Graph aesthetics

One requirement for this application was to be able to easily identify each node. The easiest way to do this was through colours. Various colours were tried and tests but in the end a set of bright unique colours were chosen so that each node would be easily identifiable.

- Source Nodes: Gray

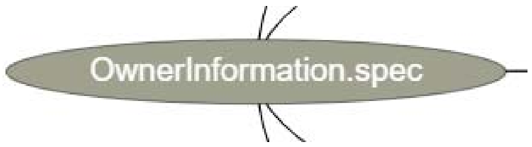


Figure 6-Specification node

- Scenario Nodes: Blue

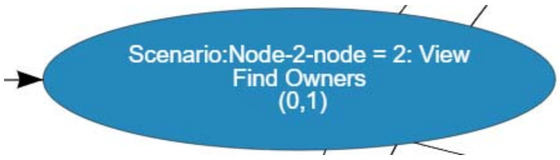


Figure 7- Scenario Node

- Step Nodes: Dark Green

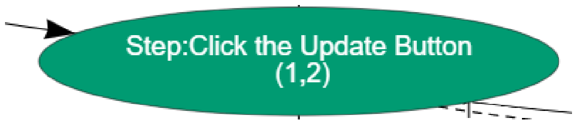


Figure 8- Step Node

- Sub Steps/ Concept file steps: Orange

Step:"Update Pet" button click
"30"
(1,0)

Figure 9- Sub Step node

- Client-Side Nodes: Light Green/khaki

Client:Add new pet
xpath: /html/body/div/div/a[2]
(1,0)

Figure 10- Client-side node

- Server-side Nodes: Purple

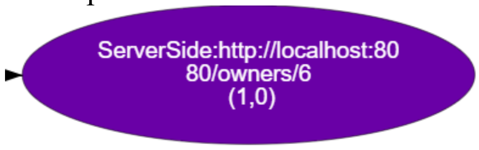


Figure 11- Server-side node

A node is then highlighted in red and edges in blue when it is clicked on, shown in *figure 12*.

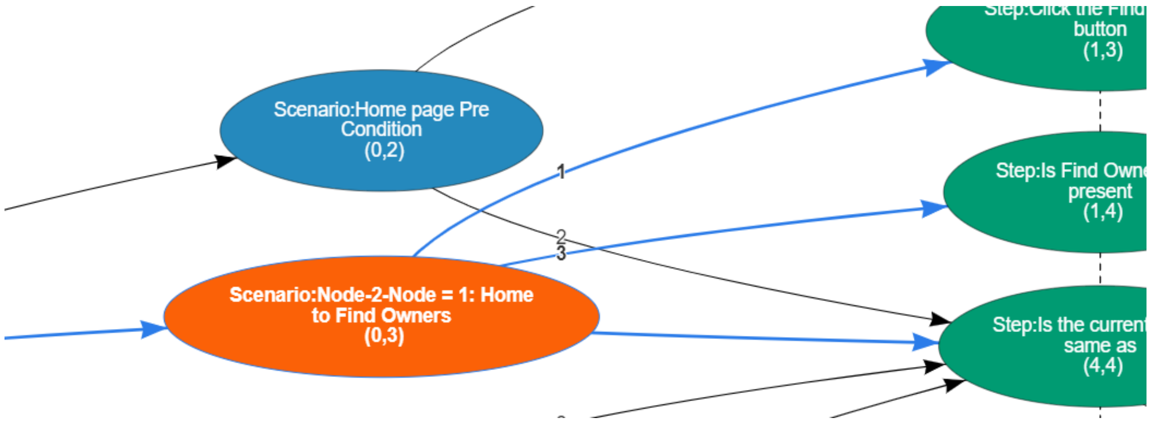

Scenario:Home page Pre Condition (0,2)
Scenario:Node-2-Node = 1: Home to Find Owners (0,3)
Step:Click the Find button (1,3)
Step:Is Find Own present (1,4)
Step:Is the current same as (4,4)
1
2
3

Figure 12- A selected node example

A problem GaugeDepend suffered from was spacing issues when Generating large graphs. In the new application this problem was combatted by changing the physics model used by the graphing library. Nodes now load closer together compared to the older project as showcased below. This physics model unfortunately stops nodes from moving freely from each other, however, it creates a much cleaner and easier to use look from the first generation of the model. The physics model had several iterations, but through manual testing the best model was selected and is currently being used. More details about the physics model can be seen in section 4.5.

Below in *figure 13* and *14* there are examples of the old and the new generation. On some extremely large graphs with a large number of test steps this can cause overlapping of the step nodes, but this can be easily fixed by changing node spacing in the customisable options settings.

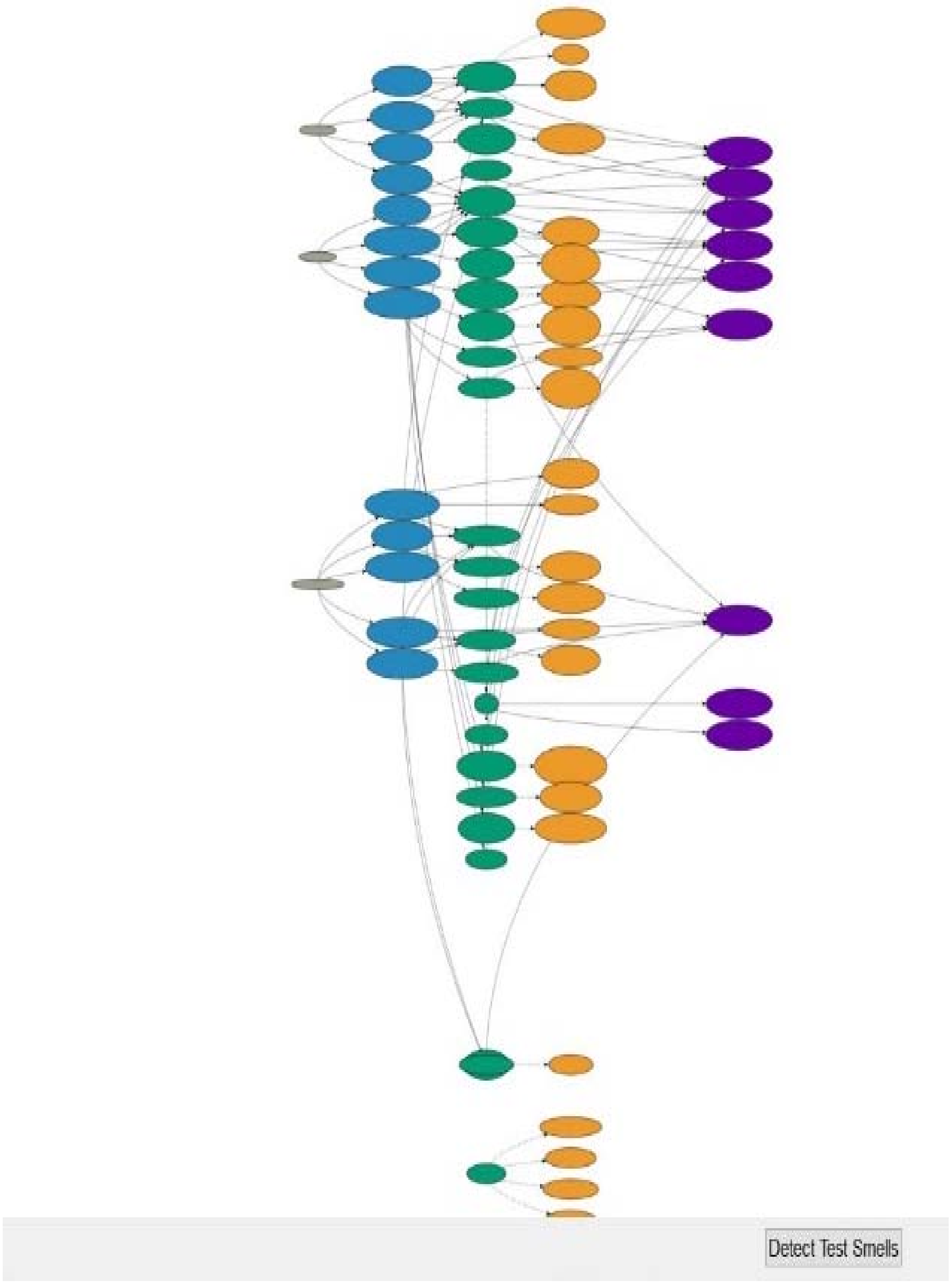


Figure 13- Old Graph spacing

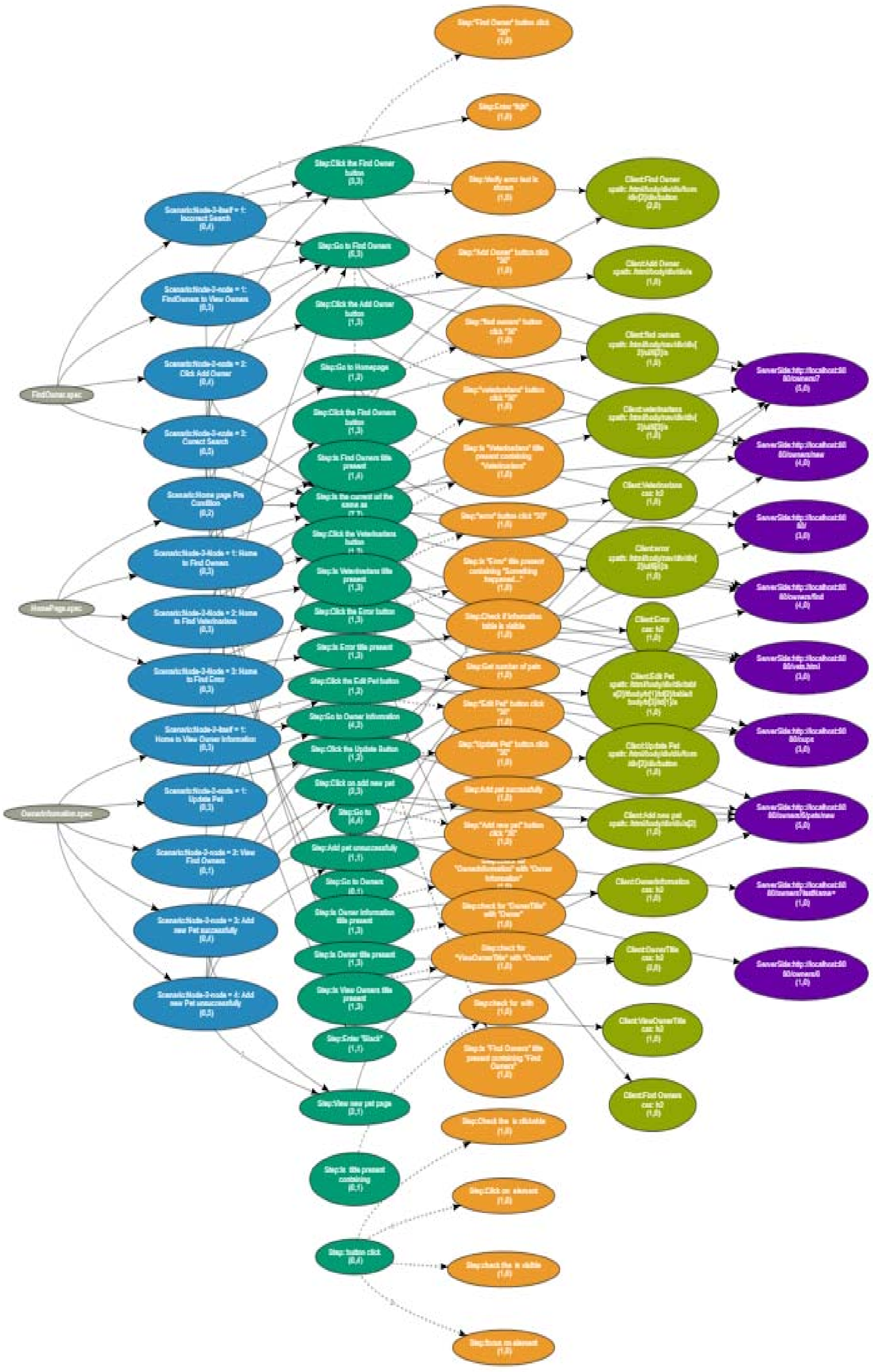


Figure 14- New Graph Spacing

3.3 User Interface

As specified in the user requirements this application must be able to show client-side tests and the location of the client-side object that is being tested. Unfortunately, Gauge specs do not have any features that would give details of html object. To work around this, the user must present a resource file with the object paths defined. It was not ideal to require the user to create these json files manually, but it is good practice to create a list of object names and positions when testing web application.

The UI shown below in *figure 15* is the previous projects UI design. It is a very clean design and did not require many changes. Although, to allow the user to add a resource file as an addition file path, an extra drop down was required. This is of course still optional, and the UI's functionality works without a specific resource file selected.

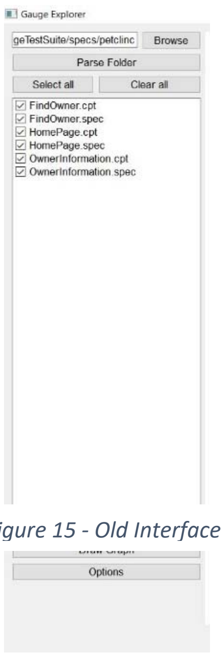


Figure 15 - Old Interface

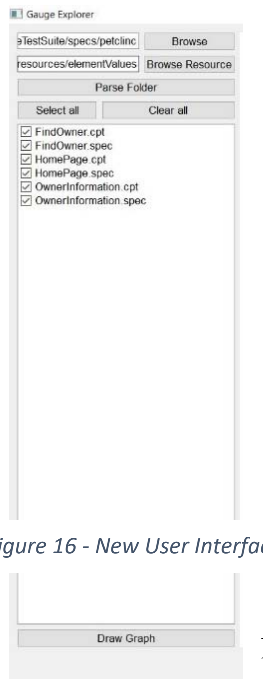


Figure 16 - New User Interface

The user can manually select t older where their Gauge specification and Concept files are by typing the file path into the text box, or by using the browse button to open the file navigator. Then, optionally, they can select the folder where their resource files are stored (this option is required for client-side nodes), this can be done manually or by pressing the browse resource file as well. The user can then press "Parse Folder", this displays a list of specification and concept files from the directory they have specified. Here they can select any combination of files, from one to all of them, and press draw to display the graphs. More details on the UI are touched upon in the Implementation section 4.5.

4 Implementation

4.1 Previous Implementation and Setup

As this project is building on a previous application, all the packages and libraries where already in use, all written in python. Mainly markdown, python GUI, Pyvis and Networkx libraries are used in this project. To understand and build on GaugeDepend I used IntelliJ as my IDE of choice, as I have the most experience with it and it is quite an advanced IDE. The Gauge plugin for IntelliJ also provided a lot of useful functionality for writing Gauge tests.. Other than the Gauge sample test suite, which was written in Java, and the sample Gauge specifications and concept files which were written in plain text, all additions were also written in python.

4.2 Sample Gauge Test Suite Implementation

To develop the application, a Gauge Test suite was needed to work on. As the project is for web development tests, a simple developed web application was required. A Gauge Test suit was created for the spring application Pet Clinic [10]. Pet Clinic is a simple application that has good examples of client-side and server-side web services, while also being small enough that a Gauge Test suite could easily be developed for it.

Below in *figure 17* I have designed a state diagram that shows the flow of the user interface for Pet Clinic. This diagram was used to structure my Model Based Testing approach. A model-based test design methodology offers a straightforward blueprint for test engineers and makes it easier to track progress on the test suite's production. It also allows the user to see what additional test cases need to be derived and helps them remove the randomness that can occur with none model based testing. It is expected that users will be following a MBT approach, so it was appropriate to also follow this approach when created example tests.

Using this model-based testing approach four different types of test can be created:

1. Transition from current node (page) to its next neighbouring nodes
2. Node to itself (usually error checking for example invalid username or password)
3. Input UI tests (in single node/path/unit level). Client-side tests
4. End to end testing: testing a path of nodes.

During testing there has only been Transition and Node-to-itself tests created. Four specification and concept file pairs were made for the application. This did not cover Pet Clinic end-to-end, but it was decided that this would be enough to test the application during development.

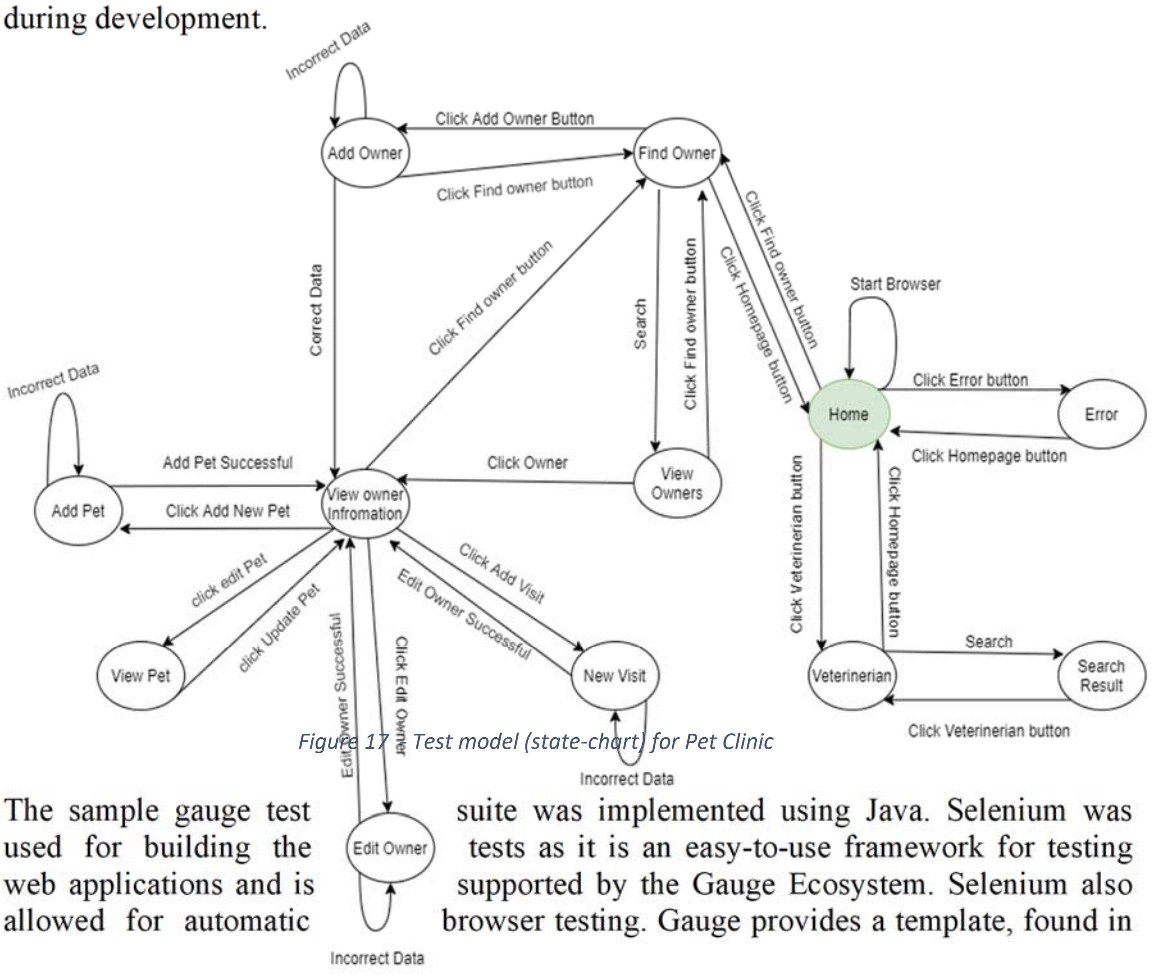


Figure 17 : Test model (state-chart) for Pet Clinic

The sample gauge test suite was implemented using Java. Selenium was used for building the tests as it is an easy-to-use framework for testing web applications and is supported by the Gauge Ecosystem. Selenium also allowed for automatic browser testing. Gauge provides a template, found in the Gauge Documentation, which helped structure the Selenium and Gauge Dependencies. Maven was also used as the build tool.

The Test suite contains several Java classes; A Base Step class that is used for any tests that spread across multiple pages being tested, each individual page test classes, helper classes and a model class. All the step classes (base and webpages) follow a standard manner. Each Gauge step is implemented in a method which is annotated with each step name. The helper and model classes have been created to facilitate the use of elements stored in a json file. The model class is used to convert each item in the resource json files into java variables. Next the "storeHelper" class is used to create a list of the obtained element values from the json files, that have now been converted into java variables. Finally, the "elementHelper" class contains method that allows the program to get an individual element from the list. This method is called from whatever Gauge step that requires it.

4.3 Gauge Parsing Test Suite Implementation

Gauge Depend was already working with Gauge parsing classes, although many changes were needed. The original application was processing the Gauge markdown files using simple regular expression to remove or replace any unwanted patterns. Some of the regex was complicated so to help understanding of the regex and to facilitate the create of new patterns a web tool called "Regex101" [15] was used. Some of the patterns being removed or replaced were:

- Tags
- Comments, not needed for dependency graph
- Tables, these often result in step duplications
- Dividers such as '#####' or '------' or '……'
- Any text not related to a step or scenario
- '*' , stars are used to denote Gauge steps. So any text after the star is moved to a new line.

One of the main problems of the original class was that it replaced any variables used in steps with a place holder value e.g. "$VARIABLE$". This was a feature of the previous application as the variables were not needed when looking at dependencies. This became an issue as many of the variables in Gauge test steps were variables related to the element values of web services (client-side and server-side) objects. The solution to this was to create a method that manages the variables on each markdown string. It checks to see if it starts with http or https, if true the variable will be put on a new line, this will be used later as a server-side node. If it does not start with http or https, it is then returned with its original text.

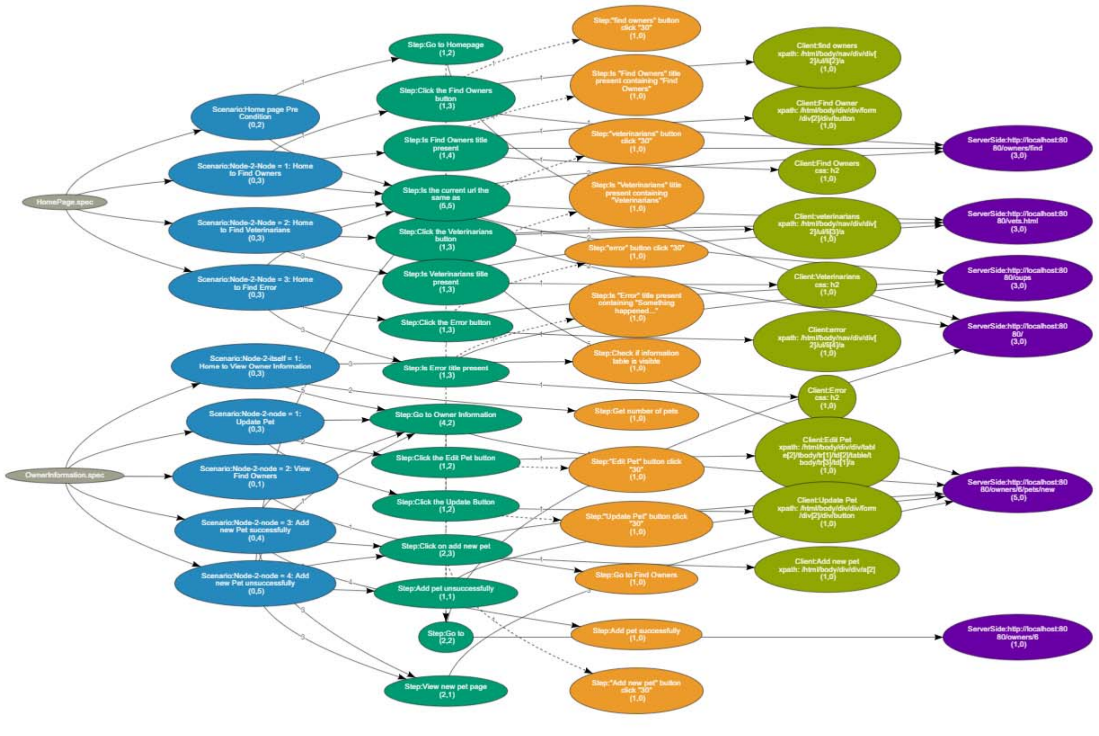
Once the markdown has been processed it is then converted into HTML to then be converted into the desired models. The application already had models for scenarios and concepts, both models just required an additional list for web services. An extra model for Client Objects was added. The decision to add Client Objects as their own model was because each client Object requires three lists, with values unique to each object (Keys, Values, and Types).

4.4 Graph Generator Implementation

Gauge Depend uses a mix of Networkx and Pyvis. Pyvis allowed the generation of interactive graphs in the form of HTML pages and Networkx provides graph objects, that can be saved in yaml format with the ability to directly combine graphs. YAML is a human-readable data serialisation standard that works with any programming language and is frequently used to create configuration files.

Most of the changes were needed in the methods for creating the Networkx yaml files. When creating the Graph for Scenario files extra nodes were added to account for server-side nodes. When creating the Graphs for concept files we needed to incorporate the resource files for client-side objects as previously mentioned. The method takes a path to a file, given by the user, and checks if the files exists, it then starts creating the nodes for concept file steps. After the steps it then compares these steps to the client-side objects and connects the step to the client-side object if that step contains a variable with the object name. This, of course, is optional and the concept file graph can be created if no resource file path is given.

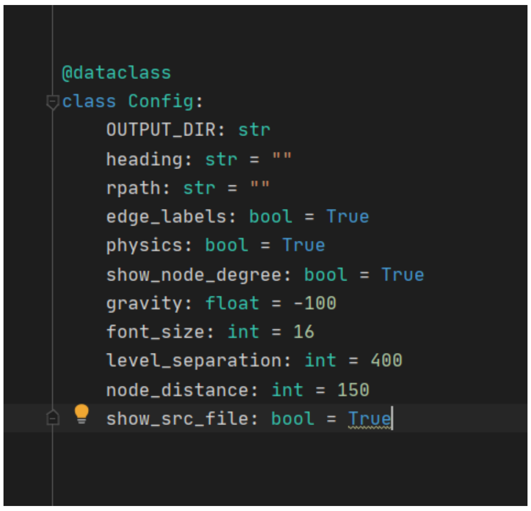
Pyvis then generates visuals directly from the Networkx yaml file. As shown below in *figure 18*.



HomePage.spec
OwnerInformation.spec
Scenario:Home page Pre Condition (0,2)
Scenario:Node-2-Node = 1: Home to Find Owners (0,3)
Scenario:Node-2-Node = 2: Home to Find Veterinarians (0,3)
Scenario:Node-2-Node = 3: Home to Find Error (0,3)
Scenario:Node-2-itself = 1: Home to View Owner Information (0,3)
Scenario:Node-2-node = 1: Update Pet (0,3)
Scenario:Node-2-node = 2: View Find Owners (0,1)
Scenario:Node-2-node = 3: Add new Pet successfully (0,4)
Scenario:Node-2-node = 4: Add new Pet unsuccessfully (0,5)
Step:Go to Homepage (1,2)
Step:Click the Find Owners button (1,3)
Step:Is Find Owners title present (1,4)
Step:Is the current url the same as (5,5)
Step:Click the Veterinarians button (1,3)
Step:Is Veterinarians title present (1,3)
Step:Click the Error button (1,3)
Step:Is Error title present (1,3)
Step:Go to Owner Information (4,2)
Step:Click the Edit Pet button (1,2)
Step:Click the Update Button (1,2)
Step:Click on add new pet (2,3)
Step:Add pet unsuccessfully (1,1)
Step:Go to (2,2)
Step:View new pet page (2,1)
Step:"find owners" button click "30" (1,0)
Step:Is "Find Owners" title present containing "Find Owners" (1,0)
Step:"veterinarians" button click "30" (1,0)
Step:Is "Veterinarians" title present containing "Veterinarians" (1,0)
Step:"error" button click "30" (1,0)
Step:Is "Error" title present containing "Something happened..." (1,0)
Step:Check if information table is visible (1,0)
Step:Get number of pets (1,0)
Step:"Edit Pet" button click "30" (1,0)
Step:"Update Pet" button click "30" (1,0)
Step:Go to Find Owners (1,0)
Step:Add pet successfully (1,0)
Step:"Add new pet" button click "30" (1,0)
Client:find owners xpath: /html/body/nav/div/div[2]/ul/li[2]/a (1,0)
Client:Find Owner xpath: /html/body/div/div/form/div[2]/div/button (1,0)
Client:Find Owners css: h2 (1,0)
Client:veterinarians xpath: /html/body/nav/div/div[2]/ul/li[3]/a (1,0)
Client:Veterinarians css: h2 (1,0)
Client:error xpath: /html/body/nav/div/div[2]/ul/li[4]/a (1,0)
Client:Error css: h2 (1,0)
Client:Edit Pet xpath: /html/body/div/div/table[2]/tbody/tr[1]/td[2]/table/tbody/tr[3]/td[1]/a (1,0)
Client:Update Pet xpath: /html/body/div/div/form/div[2]/div/button (1,0)
Client:Add new pet xpath: /html/body/div/div/a[2] (1,0)
ServerSide:http://localhost:8080/owners/find (3,0)
ServerSide:http://localhost:8080/vets.html (3,0)
ServerSide:http://localhost:8080/oups (3,0)
ServerSide:http://localhost:8080/ (3,0)
ServerSide:http://localhost:8080/owners/6/pets/new (5,0)
ServerSide:http://localhost:8080/owners/6 (1,0)

Figure 18- Pyvis HTML Gauge Dependency Graph

To achieve the customisation specified in the requirements section there is a configuration class. This is simply a python data class that holds the necessary configuration values, and this configuration class is passed in as a parameter to the graph generator class. The configuration class has a set of default values that were decided based on the average size of individual specification graphs shown in *figure 19*.



```python
@dataclass
class Config:
    OUTPUT_DIR: str
    heading: str = ""
    rpath: str = ""
    edge_labels: bool = True
    physics: bool = True
    show_node_degree: bool = True
    gravity: float = -100
    font_size: int = 16
    level_separation: int = 400
    node_distance: int = 150
    show_src_file: bool = True
```

Figure 19- configuration file

This was found through manual testing of several specification files. When using larger graphs made up of combinations of multiple specification and concept files it is very easy to change these default values, to suite your needs.

4.5 Graph Physics Implementation

As previously mentioned, an error in GaugeDepend was how nodes were displayed in relation to each other on first generation. The empty space between some nodes seemed to generate randomly. This was caused by the physics model being used by Pyvis, which helps generate the structure of the graphs. Pyvis allows for the architecture of the network graph to be managed by a front-end physics engine that can be configured using a Python interface, allowing for precise positioning of graph components. It is currently being managed from a html string.

Fixing this took changing the physics model and trying several iterations of a new models. One of the main changes was including nodes in the physics simulation. Below in *figure 20*, it shows how the graphs spreads out more when nodes are not taken into consideration

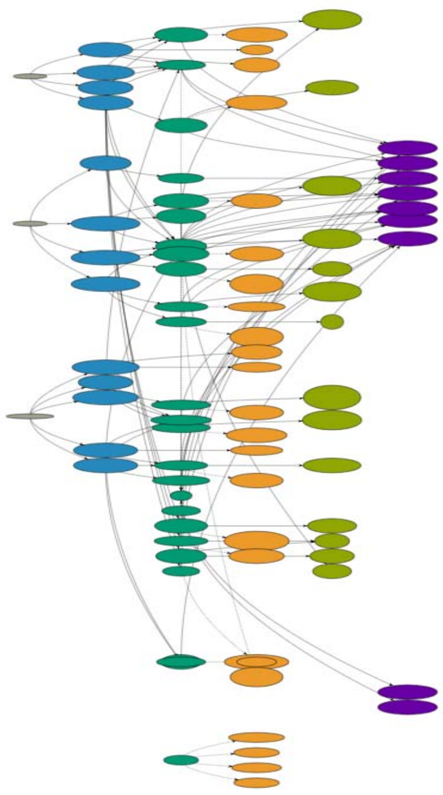
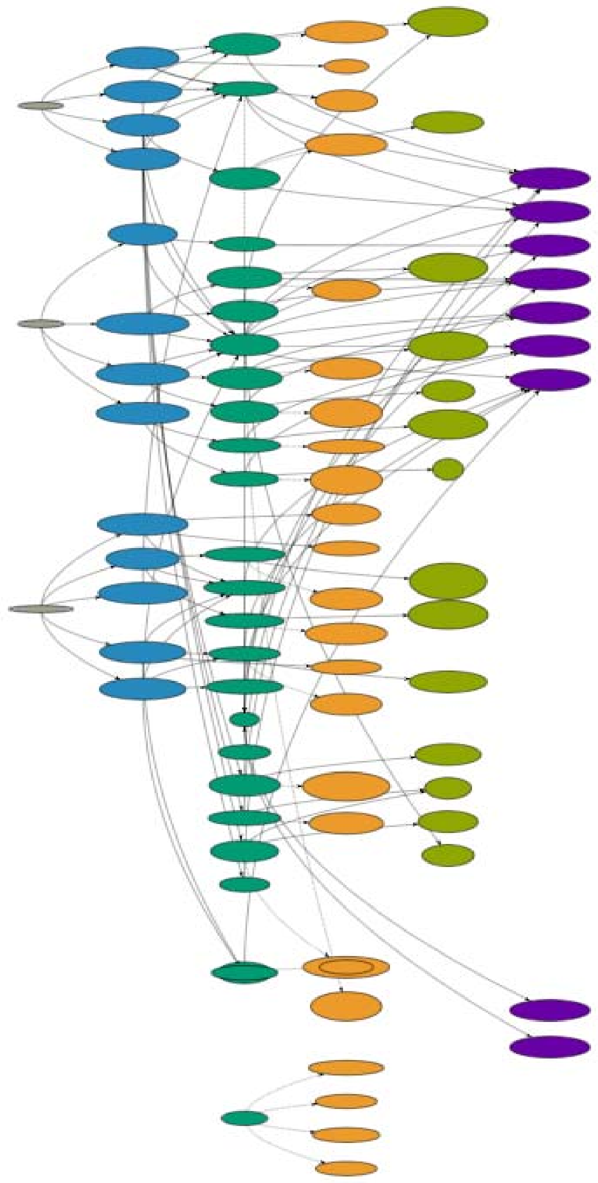


Figure 20- Example of Pet Clinic Graph without Node physics

A small change that had a major impact was adding central gravity to the physics model. This keeps all the nodes at a central point, allowing for the nodes to be easily read together. Unfortunately, this causes the nodes to move after being rendered, but using Pyvis's stabilisation options, the graph will now wait until stabilised before rendering. This does mean some graphs have a slightly longer load time, but this is reasonable for a more aesthetically pleasing graph. Below shown in *figure 21*, shows a graph without central gravity. It is similar to *figure 20* allow with the nodes being apart of the simulation, they are not overlapping as much. *Figure 22* shows the graphs with Central Gravity set to 10.



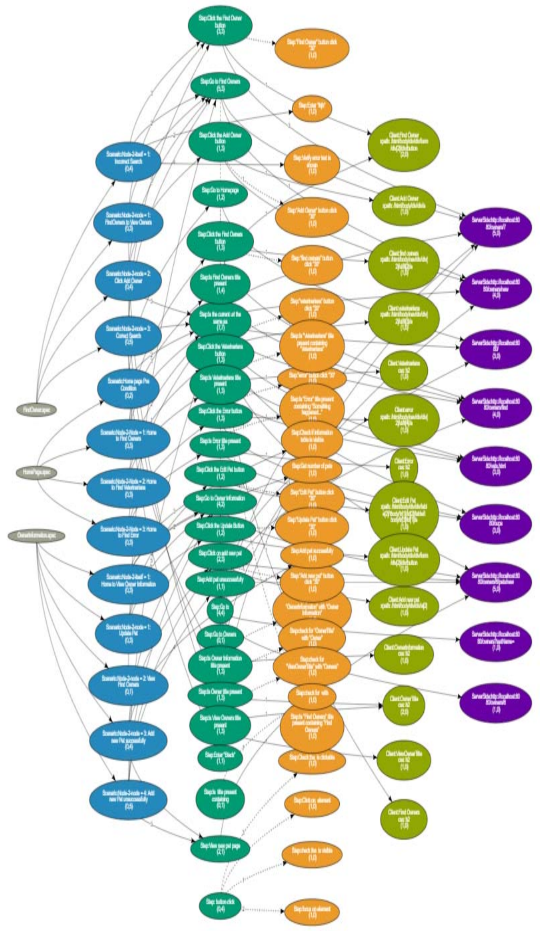


Figure 22- Example of Pet Clinic Graph with Central Gravity

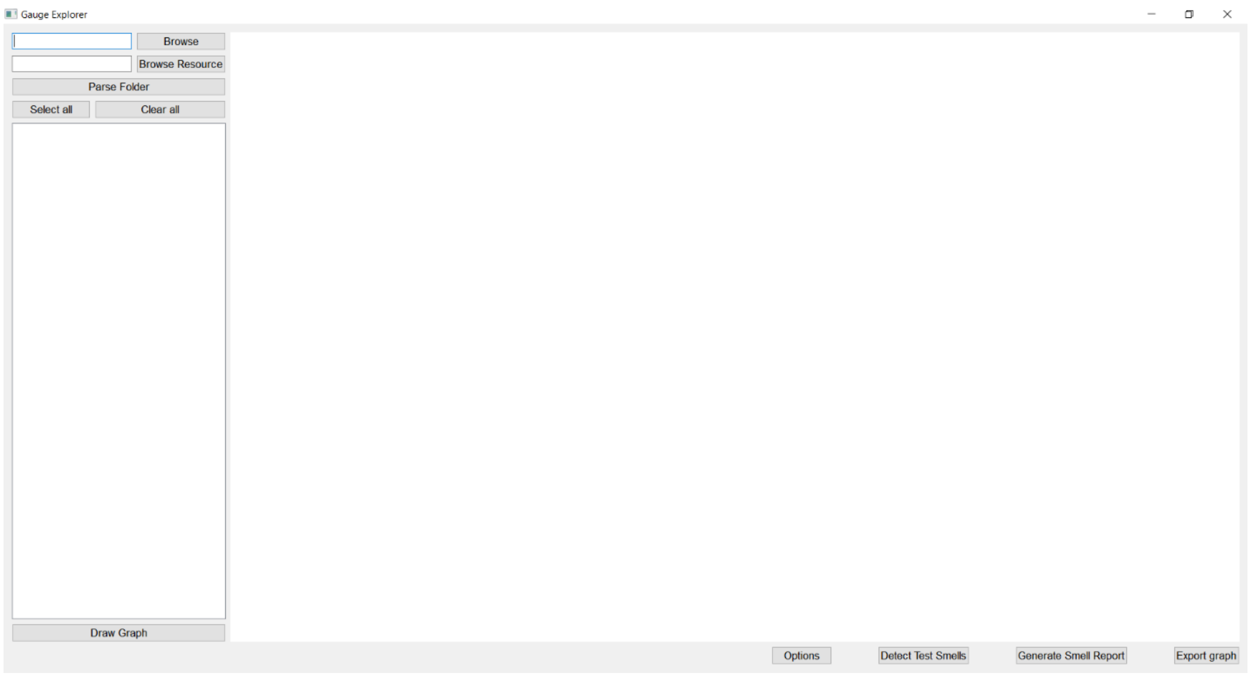
Figure 21- Example of Pet Clinic graph with no central gravity

4.6 GUI Implementation

As this project was building on GaugeDepend most of the GUI was already functioning to a reliable standard. To begin with GaugeDepend had a Main Window class that creates the required layout and buttons and has methods to handle each function. There is a textbox for the user to input the path to the directory containing their Gauge files. There is a parse button, a Draw graph button, options button, two test smell related buttons and an export graph button. The main implementations made during this project were not related to the Test smells ore exporting graphs.

The new application required an extra textbox for the user to input the path to the directory containing their resource files. Changes were made to the method implementing the parse button. It now gets the resource path and the Gauge file path, passes it to the generator class and populates a list of checkboxes with each of the parsed files.

The draw graph button has been left unchanged, it calls the combine and render methods of the graph generator for each of the checked file names, this then generates the graph, it is then opened. The graph is produced as an HTML file, which is then displayed in the QWebEngineView, which is essentially a web browser window.The options button has



[illegible]

Figure 23- UI when application is loaded

[illegible] to the configuration class, when the save button on the form is clicked the configuration class is updated with the values from the form and a pop-up message appears telling the user that they will need to draw the graph again for them to see their changes to take effect. These changes can be seen in the *figures 23* above.

5 Testing

Testing was done in two ways for this application, mixing automated unit tests and manual integration testing, using a mixture of purpose made and real-world specification, concepts, and resources.

5.1 Manual Integration testing

Many spec files and concept files were used for manual integration testing of the application. As mentioned previously a test suite was developed for pet clinic specifically to test the application. A test suite for an application called Testinium [12] and an android application called btcturk [16] was also used during manual testing. These where extremely useful as their test suites were at a much larger scale than pet clinic and

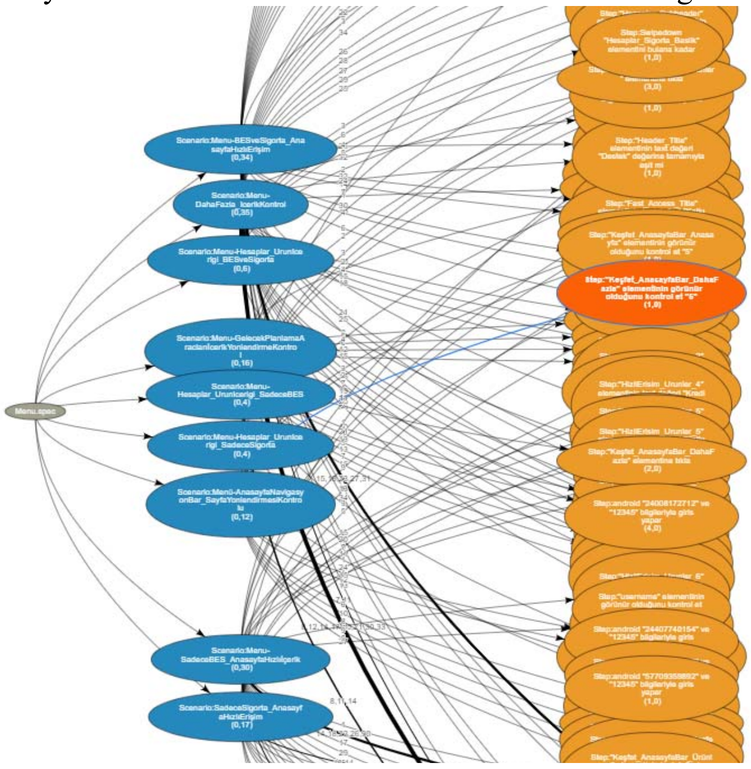


Figure 24- Overlapping step nodes error

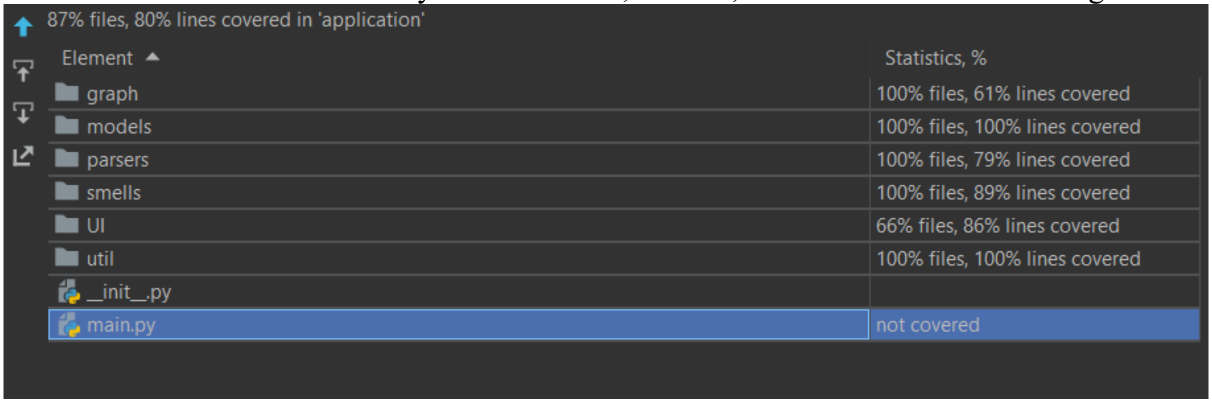
Testinium already had resource json files describing client-side objects and their html paths. An older version of Testinium that had been translated into English from Turkish was also used, but it did not have resource files.

The manual testing was done using a trial-and-error process. The folder containing the spec and concept files were processed through the application and the errors were patched and collected as they were found. Manual testing helped fined errors caused by bad code practices written in the specification files, such as forgetting to close quotation marks around variables, or having comments in the middle of steps. Such things would not have been picked up by the automated test suits. This was also very useful when testing visual parts of the application. Some errors only occurred when specification had a large number of steps, shown in *figure 24* above, although large numbers of steps would be considered a test smell, so this was considered a lesser error. Manual testing was needed for visual errors as the solution was often a very small change in code and correct visuals would be hard to quantify for assertion testing. This also would have been unnecessary and tedious to apply as automatic testing.

Manual testing was needed when developing new code. Tt was useful to immediately test a method that had been changed or a new method that had been added without having to write a separate method to automate the testing of the method. This then saved time in the development stage. After manual testing, then the time was taken to develop automated testing so if any further changes were made, functionality could easily be tested.

5.2 Automatic unit testing

The unit tests for this application are implemented using pythons built in unit testing library and contains a total of 58 automated unit tests with 80% code coverage according to the build in the test coverage functionality of IntellJ, shown in *figure 25*. The remaining 20% of uncovered code is mainly main methods, ui code, and html lines for rendering.

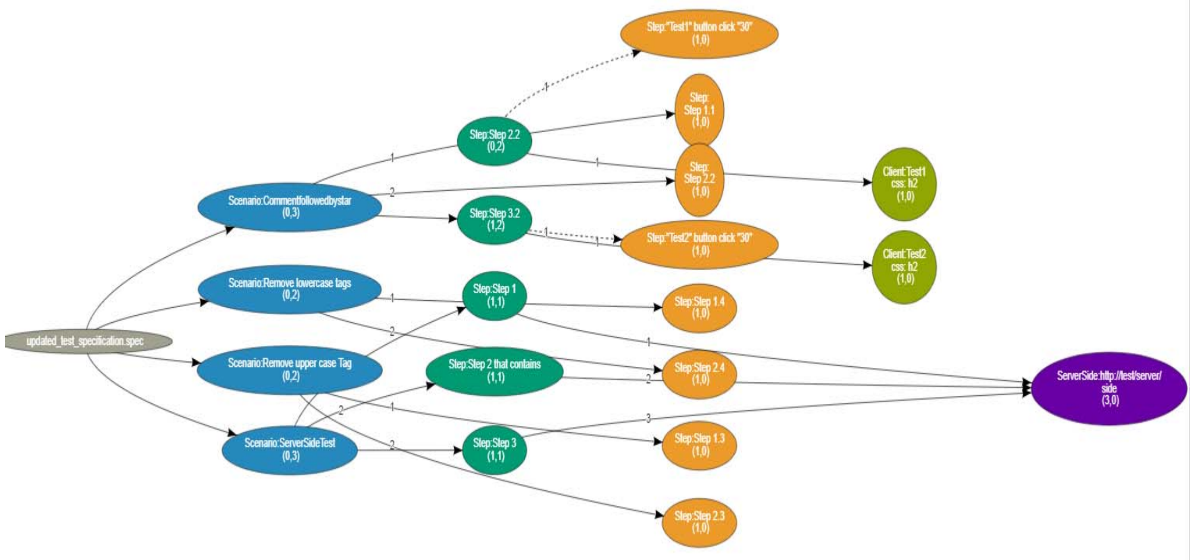


87% files, 80% lines covered in 'application'

Element	Statistics, %
graph	100% files, 61% lines covered
models	100% files, 100% lines covered
parsers	100% files, 79% lines covered
smells	100% files, 89% lines covered
UI	66% files, 86% lines covered
util	100% files, 100% lines covered
__init__.py	
main.py	not covered

Figure 25- IntellJ code coverage statistics

The unit tests make use of several sample Gauge files that were previously created as well as new Gauge files designed to test new features that were not covered by the older specs. These files do not contain any meaningful steps and have no step implementation, they were made specifically to test code and are not examples of good test specifications. An example of a graph generated by the new specification file is shown below in *figure 26.*



updated_test_specification.spec
Scenario:Commentfollowedbystar (0,3)
Scenario:Remove lowercase tags (0,2)
Scenario:Remove upper case Tag (0,2)
Scenario:ServerSideTest (0,3)
Step:Step 2.2 (0,2)
Step:Step 3.2 (1,2)
Step:Step 1 (1,1)
Step:Step 2 that contains (1,1)
Step:Step 3 (1,1)
Step:"Test1" button click "30" (1,0)
Step: Step 1.1 (1,0)
Step: Step 2.2 (1,0)
Step:"Test2" button click "30" (1,0)
Step:Step 1.4 (1,0)
Step:Step 2.4 (1,0)
Step:Step 1.3 (1,0)
Step:Step 2.3 (1,0)
Client:Test1 css: h2 (1,0)
Client:Test2 css: h2 (1,0)
ServerSide:http://test/server/side (3,0)

Figure 26- Sample test suite graph generation

The Unit tests were added to test a number of newly added features. The new unit tests were added by going through the flow of the application, starting at the markdown class, and ending at graph generation. The first new unit test was testing how the markdown class separated http(s) lines into sperate lines, as well as how some of the regex patterns were updated to be more specific. Next was to check if server-side steps were added to their own list under the scenarios model while parsing, and if resource files were parsed into the appropriate model. Finally, the last tests needed where to check the generation of specification and concept graphs contained the appropriate nodes, including the new features. Old unit tests were then modified to run with the new code. This last step was important to make sure that other features, like detecting smells, had not been affected by the new implementations.

.

6 System Evaluation and Evidence

6.1 SUTs used for evaluation

The application has been evaluated using some SUTs of varying sizes. Petclinic which is a small web application used to show springboot, Testinium a test automation management tool, and an android application, btcturk.

As mentioned above Petclinic's specifications and concept files were made from scratch specifically for testing the application. Petclinic generated an average sized graph with 74 nodes. This was useful for building the application and getting the main features working.

Testinium and the btcturk were two real-world examples of industrial code bases. Testinium generating 928 individual nodes and btcturk generating 768 individual nodes. These graphs are reliably generating at a reasonable pace, but as they were very large, they did not scale well, this has been taken into consideration, but extremely large graphs are usually a sign of bad test smells, so this has been left untouched.

6.2 How GaugeDependExtended helped assess test suites

When running the application with Testinium, it showed some syntax errors within the spec files. Some files had random open brackets and quotes left unclosed which in turn made some nodes appear with unusual text that they should not have had.

The application also helped correct test smells. Using the GaugeDepend smell report feature will help identify smells, such as a scenario with too many steps, but now with the added client-side nodes this can help the user break the steps up into different scenarios based on each client-side object. For example, in *figure 27*, a sample step from a Testinium dependency graph, the step "Step: Check the Reports page" (translated from Turkish) has many sub steps and is linked to many client-side objects.

We can then use the client-side nodes to break the step into multiple steps based around each object, such as, “showOnlyFailedTestsInReports”, or “suitsInReports”, or “tableCheckboxInReports”.

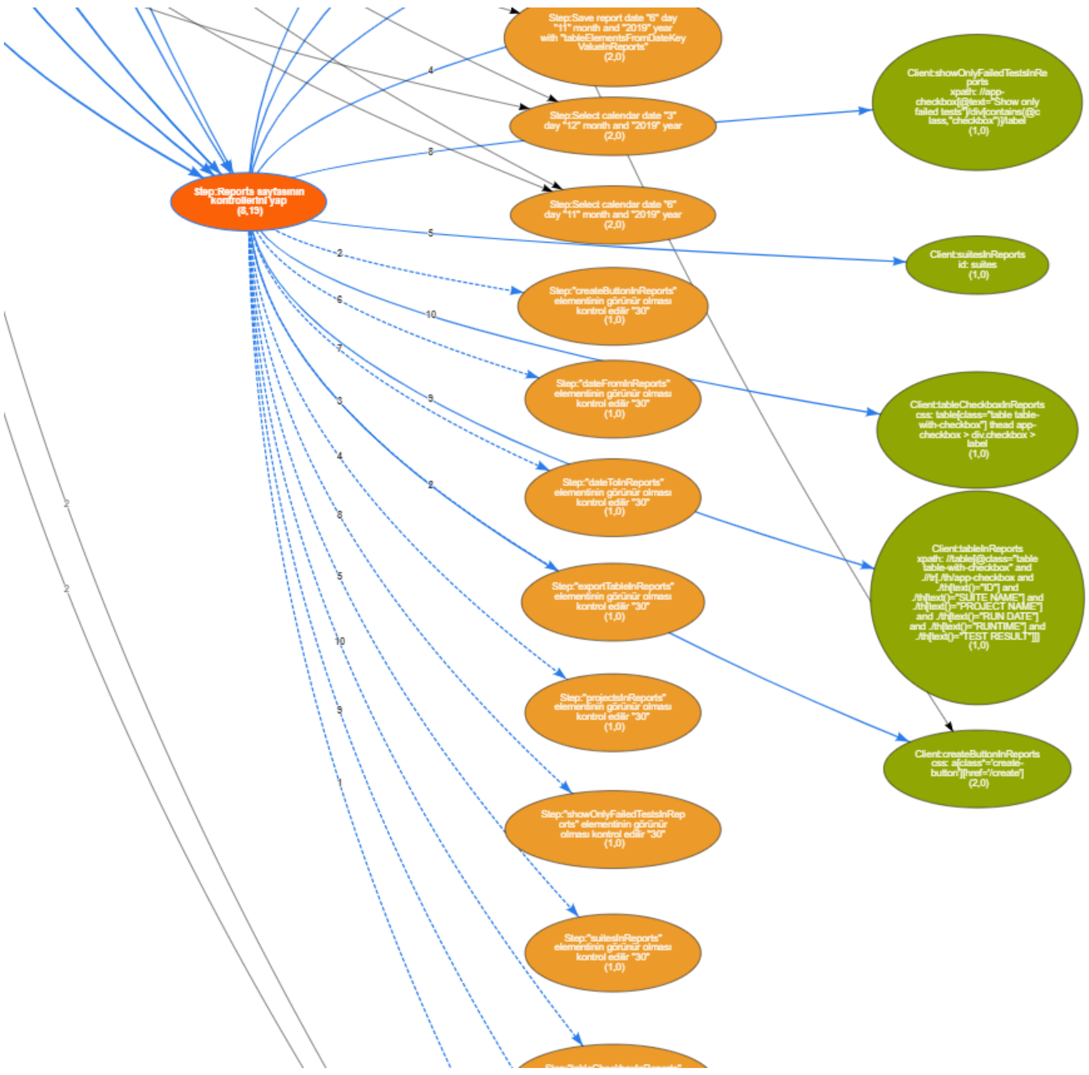

Step:Save report date "6" day "11" month and "2019" year with "tableElementsFromDateKeyValueInReports" (2,0)
Step:Select calendar date "3" day "12" month and "2019" year (2,0)
Step:Select calendar date "6" day "11" month and "2019" year (2,0)
Step:Reports sayfasının kontrollerini yap (8,19)
Step:"createButtonInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Step:"dateFromInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Step:"dateToInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Step:"exportTableInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Step:"projectsInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Step:"showOnlyFailedTestsInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Step:"suitesInReports" elementinin görünür olması kontrol edilir "30" (1,0)
Client:showOnlyFailedTestsInReports xpath: //app-checkbox[@text="Show only failed tests"]/div[contains(@class,"checkbox")]/label (1,0)
Client:suitesInReports id: suites (1,0)
Client:tableCheckboxInReports css: table[class="table table-with-checkbox"] thead app-checkbox > div.checkbox > label (1,0)
Client:tableInReports xpath: //table[@class="table table-with-checkbox" and .//tr[./th/app-checkbox and ./th[text()="ID"] and ./th[text()="SUITE NAME"] and ./th[text()="PROJECT NAME"] and ./th[text()="RUN DATE"] and ./th[text()="RUNTIME"] and ./th[text()="TEST RESULT"]]] (1,0)
Client:createButtonInReports css: a[class*='create-button'][href='/create'] (2,0)
4
8
5
2
6
10
7
9
3
2
4
8
5
10
9
1
2
2

Figure 27- Sample Step from Testinium

When writing Gauge specifications, the test engineer will follow the flow of the SUT to help develop the scenario and test. The added feature of showing server-side nodes is a great benefit for this as users can see how many tests call a specific web page. As shown in *figure 28*, we can see there are 2 steps calling “https://testinium.io/scenario”. This will help the engineer keep track of how many tests are calling each page and if they need to develop more tests for a page.

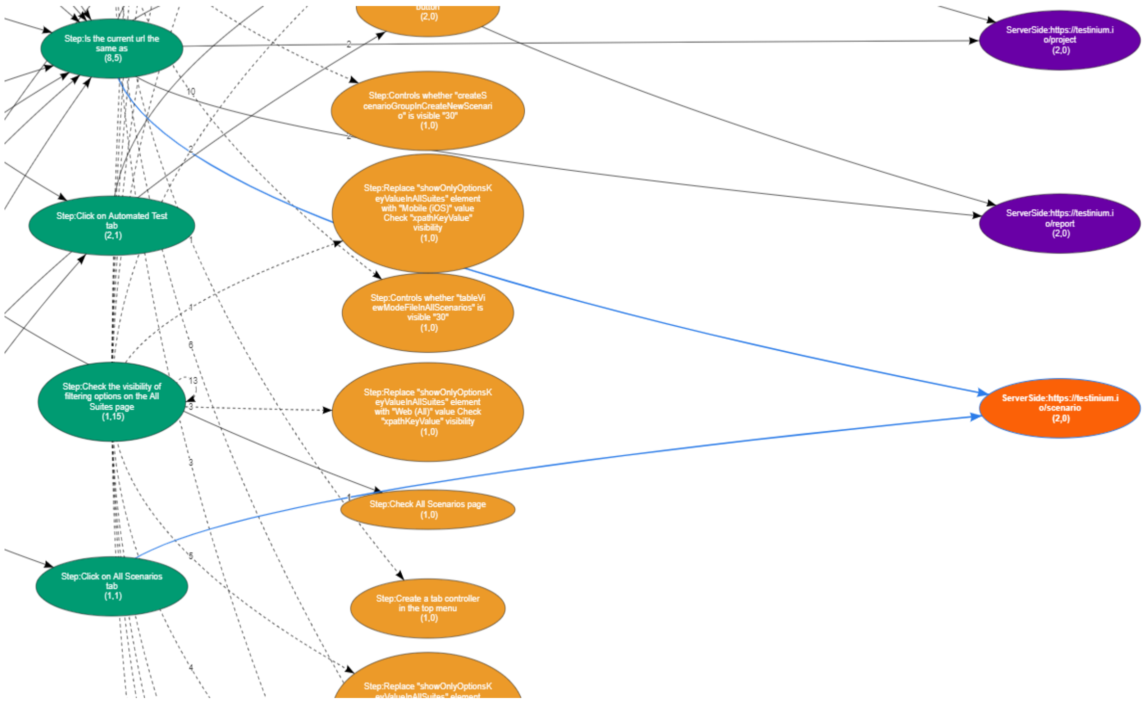
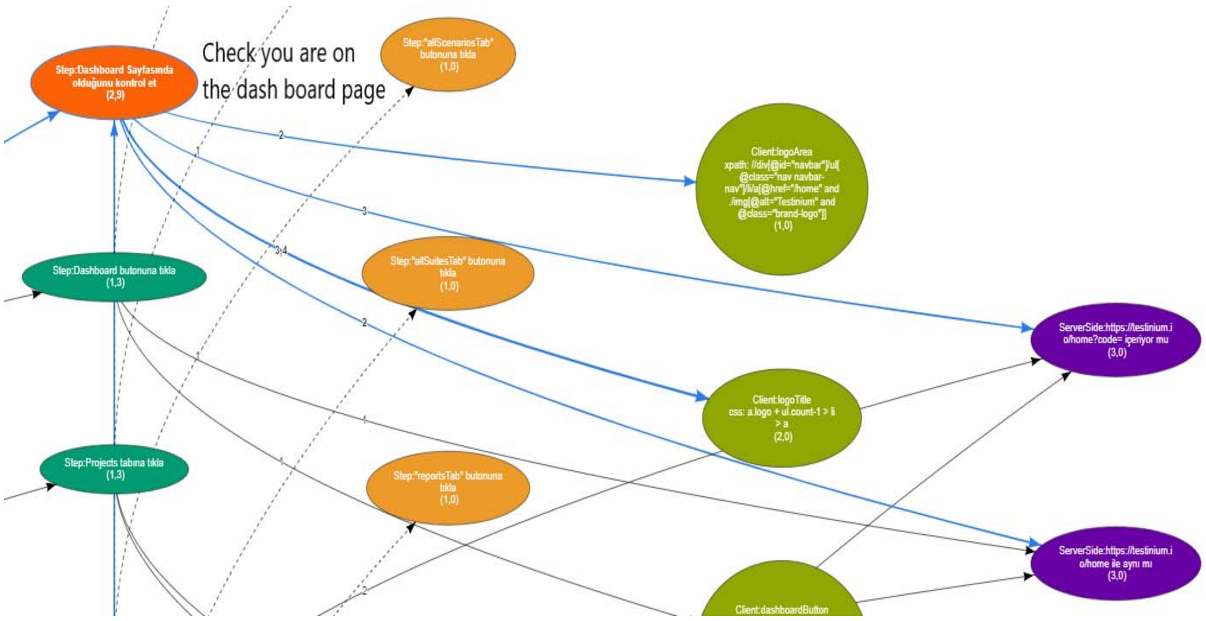

Step:Is the current url the same as (8,5)
Step:Controls whether "createScenarioGroupInCreateNewScenario" is visible "30" (1,0)
Step:Replace "showOnlyOptionsKeyValueInAllSuites" element with "Mobile (iOS)" value Check "xpathKeyValue" visibility (1,0)
Step:Click on Automated Test tab (2,1)
Step:Controls whether "tableViewModeFileInAllScenarios" is visible "30" (1,0)
Step:Check the visibility of filtering options on the All Suites page (1,15)
Step:Replace "showOnlyOptionsKeyValueInAllSuites" element with "Web (All)" value Check "xpathKeyValue" visibility (1,0)
Step:Check All Scenarios page (1,0)
Step:Click on All Scenarios tab (1,1)
Step:Create a tab controller in the top menu (1,0)
ServerSide:https://testinium.io/project (2,0)
ServerSide:https://testinium.io/report (2,0)
ServerSide:https://testinium.io/scenario (2,0)

Figure 28- Server-Side node example

As mentioned in section 2.2 the application helps when following MBT. An example of this can be seen when putting the Testinium test suite through the application. Shown in *figure 29*


Step:Dashboard Sayfasında olduğunu kontrol et (2,9)
Check you are on the dash board page
Step:"allScenariosTab" butonuna tıkla (1,0)
Client:logoArea xpath: //div[@id="navbar"]/ul[@class="nav navbar-nav"]/li/a[@href="/home" and ./img[@alt="Testinium" and @class="brand-logo"]] (1,0)
Step:Dashboard butonuna tıkla (1,3)
Step:"allSuitesTab" butonuna tıkla (1,0)
ServerSide:https://testinium.io/home?code= içeriyor mu (3,0)
Client:logoTitle css: a.logo + ul.count-1 > li > a (2,0)
Step:Projects tabına tıkla (1,3)
Step:"reportsTab" butonuna tıkla (1,0)
ServerSide:https://testinium.io/home ile aynı mı (3,0)
Client:dashboardButton

We can see that the step node highlighted (translated from Turkish) is connected to a client-side node that is called "logoArea" that also has a xpath that references "home". The same step is linked to the server-side node that calls testinium home. This test suite

Figure 29- A step node linking the related client-side and server-side nodes

is well into development, but we can still see that this shows there is a test that encompasses the flow from the navigation bar to the homepage.

6.3 Meeting Requirements

I have produced evidence to support my claim that this application has met all requirements outlined in section 2.2.

- Aesthetic requirements: The below screen shot, *figure 30*, is an example of all aesthetic requirements asked for in section 2.2. It is generating graphs using specification and concept files, all steps are colour coded, each dependency is easily seen from the edges of the graph, and the graph is pleasing to look at without any loss of information.

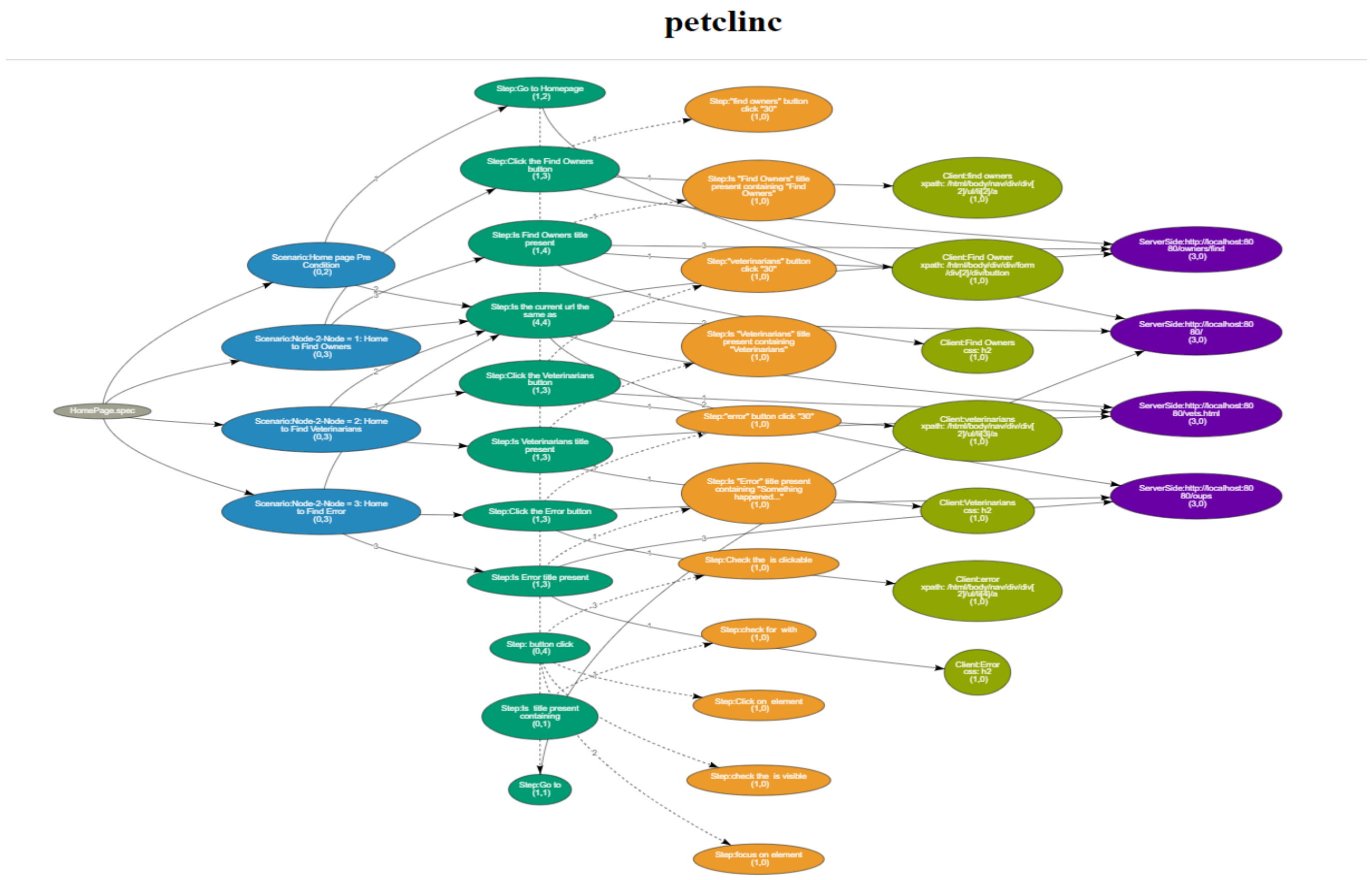

petclinc
HomePage.spec
Scenario:Home page Pre Condition (0,2)
Scenario:Node-2-Node = 1: Home to Find Owners (0,3)
Scenario:Node-2-Node = 2: Home to Find Veterinarians (0,3)
Scenario:Node-2-Node = 3: Home to Find Error (0,3)
Step:Go to Homepage (1,2)
Step:Click the Find Owners button (1,3)
Step:Is Find Owners title present (1,4)
Step:Is the current url the same as (4,4)
Step:Click the Veterinarians button (1,3)
Step:Is Veterinarians title present (1,3)
Step:Click the Error button (1,3)
Step:Is Error title present (1,3)
Step: button click (0,4)
Step:Is title present containing (0,1)
Step:Go to (1,1)
Step:"find owners" button click "30" (1,0)
Step:Is "Find Owners" title present containing "Find Owners" (1,0)
Step:"veterinarians" button click "30" (1,0)
Step:Is "Veterinarians" title present containing "Veterinarians" (1,0)
Step:"error" button click "30" (1,0)
Step:Is "Error" title present containing "Something happened..." (1,0)
Step:Check the is clickable (1,0)
Step:check for with (1,0)
Step:Click on element (1,0)
Step:check the is visible (1,0)
Step:focus on element (1,0)
Client:find owners xpath: /html/body/nav/div/div[2]/ul/li[2]/a (1,0)
Client:Find Owner xpath: /html/body/div/div/form/div[2]/div/button (1,0)
Client:Find Owners css: h2 (1,0)
Client:veterinarians xpath: /html/body/nav/div/div[2]/ul/li[3]/a (1,0)
Client:Veterinarians css: h2 (1,0)
Client:error xpath: /html/body/nav/div/div[2]/ul/li[4]/a (1,0)
Client:Error css: h2 (1,0)
ServerSide:http://localhost:8080/owners/find (3,0)
ServerSide:http://localhost:8080/ (3,0)
ServerSide:http://localhost:8080/vets.html (3,0)
ServerSide:http://localhost:8080/oups (3,0)

Figure 30- Aesthetic requirements

- Graphical Requirements: Each requirement under graphical requirements from section 2.2 is also met. Each vertex representation is seen, as shown in *figure 31*. We can also see each Form starts with a grey root node that represents the source file. Client-side nodes show the object location and type (*figure 32*).

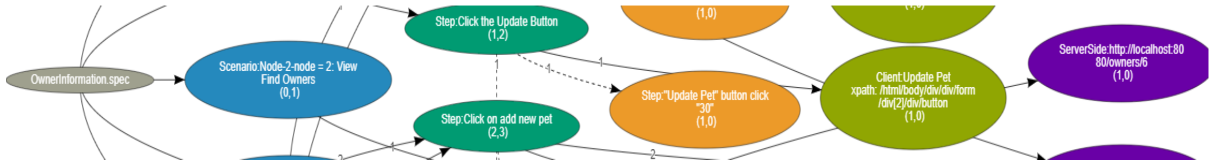
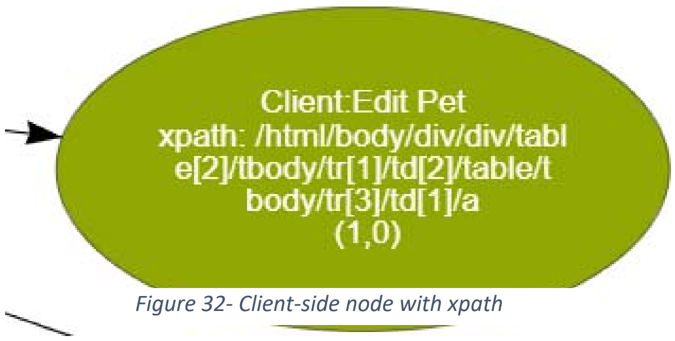
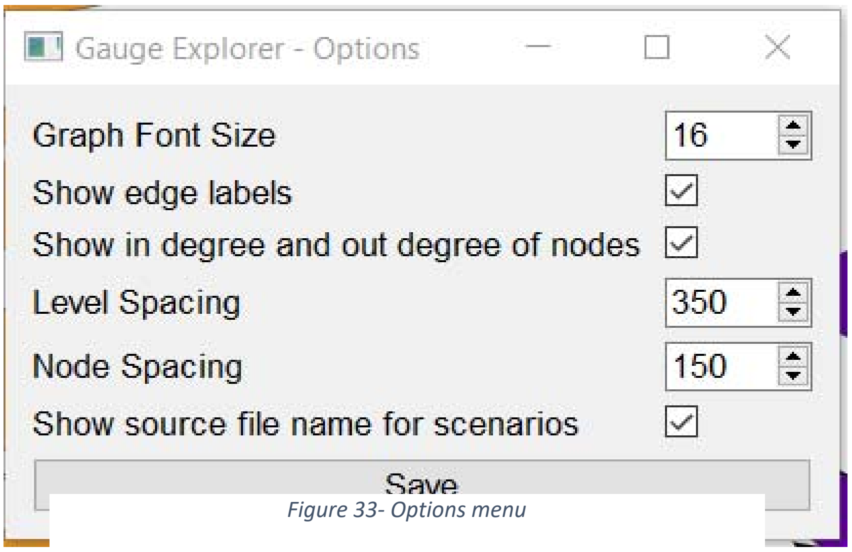

OwnerInformation.spec
Scenario:Node-2-node = 2: View Find Owners (0,1)
Step:Click the Update Button (1,2)
Step:Click on add new pet (2,3)
Step:"Update Pet" button click "30" (1,0)
Client:Update Pet xpath: /html/body/div/div/form/div[2]/div/button (1,0)
ServerSide:http://localhost:8080/owners/6 (1,0)

Figure 31- Colour coded nodes. Source file, Scenario, spec step, cpt step, client-side, server-side (left to right)


Client:Edit Pet
xpath: /html/body/div/div/table[2]/tbody/tr[1]/td[2]/table/tbody/tr[3]/td[1]/a
(1,0)
Figure 32- Client-side node with xpath

- Customisation requirements: The user can make several different custom changes to the graph using an options ui. The customisable variables are, font size, level spacing, node spacing, toggle labels, toggle edge degrees, toggle if the source file node should be shown. Evidence of this is shown in *figure 33*.

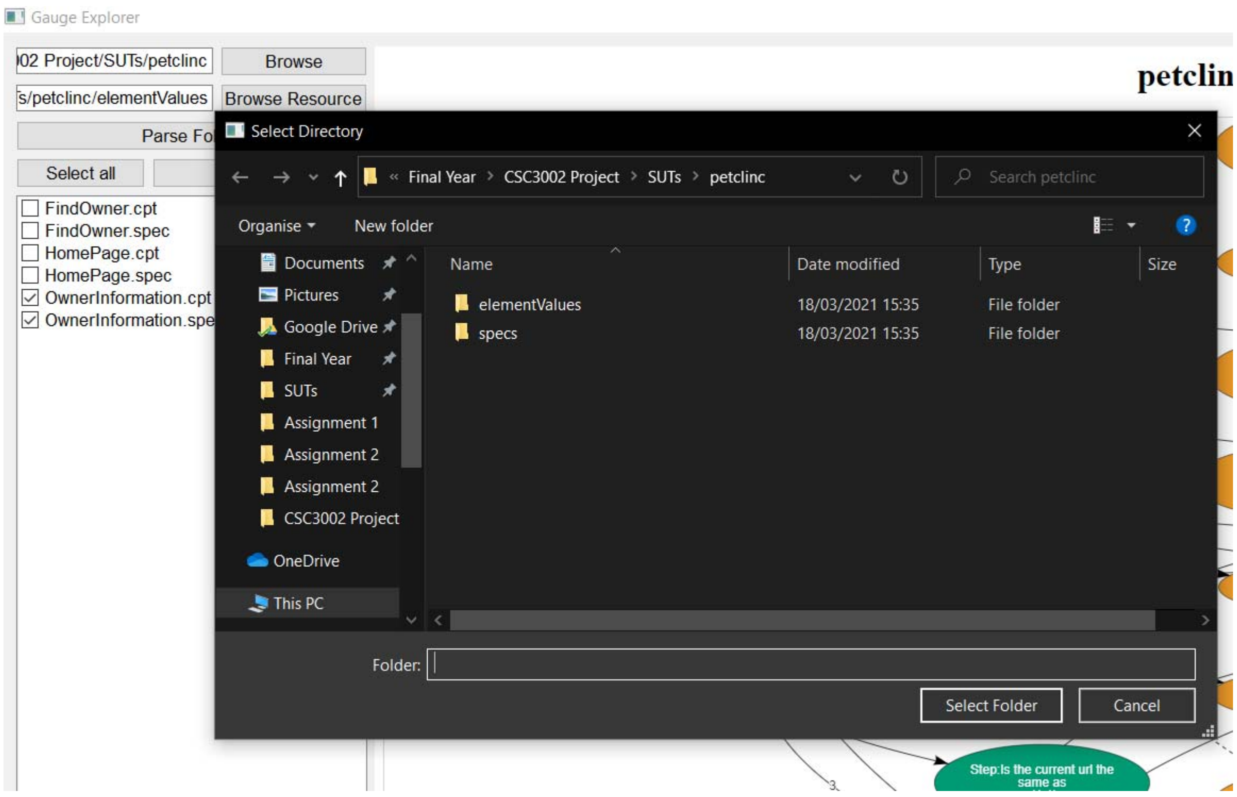

Gauge Explorer - Options

Graph Font Size 16
Show edge labels ✓
Show in degree and out degree of nodes ✓
Level Spacing 350
Node Spacing 150
Show source file name for scenarios ✓

Figure 33- Options menu

- As mentioned in section 2.2 the user will be able to easily import resource folders to implement object paths to client objects using the applications integration with windows, where they can easily navigate their folders and directories. *Figure 34* shows this being used.

Figure 34- Windows explorer navigation



Name	Date modified	Type	Size
elementValues	18/03/2021 15:35	File folder	
specs	18/03/2021 15:35	File folder	

- Scalability: The application does meet the requirement that it must be able to handle larger specification, although some spacing must be changed by the user to make larger graphs more readable. This was discussed in more detail in section 6.1. The user is still able to break the graph down into smaller groups of specifications and concepts for better visibility. An example of a usual large graph after the appropriate resizing has been done is shown in *figure 35* below.

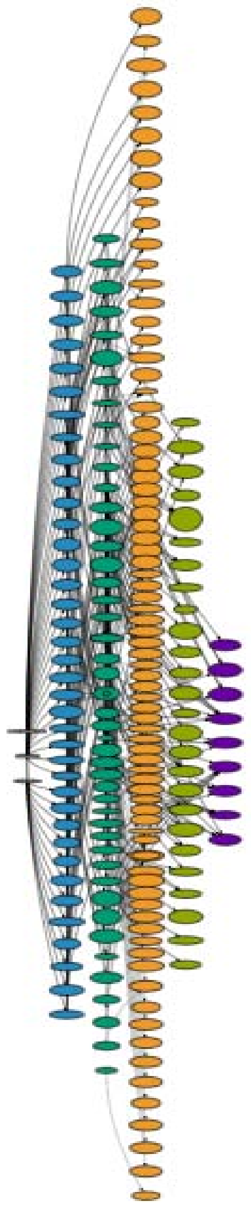


Figure 35- Example of a larger graph

- Extendibility: As each type of node is implemented as a model, new nodes can easily be added with a small amount of code change. This project was already an extension of GaugeDepend, so it still stands that it can be extended again if a different aspect is needed.

7 Improvements

The application does achieve all the required scope of the project, however, there is obviously always room for improvement. One of the major requirements for the project was to allow for extendibility, so this implies that there can be additional code added to the application to improve its functions for the user's needs.

The features added in this project could be extended further to provide additional information from simply looking at the nodes. For example, adding further sub-nodes for client-side and server-side nodes to divide them into categories. Client-side nodes could be categorised by type of object, for example buttons, textboxes, or checkboxes. While server-side nodes could be categorised based on what they are calling e.g., a database record or a webpage. These could then be identified in a few ways, perhaps by changing node shape or colour or splitting the nodes into separate child nodes.

One improvement seen would be to change how graphs are rendered and displayed. Currently graphs are generated, rendered, and then saved behind the scenes. The html files that have been created are then displayed via the UI. If this were changed so that the graphs were live rendered this would allow the users much more freedom when it comes to customising graphs. Allowing them to change the graphs physics models quicker without the need to reload the graphs. Unfortunately, this change would require an overhaul of the current rendering code and could increase memory consumption of the application.

A feature that would allow users to customise the colours of each type of node would be useful. For example, if a company already had a colour scheme in use for test dependencies, they could change this applications colour scheme to match what they are used to. This feature was not implemented as it is not required for the core functionality but would be nice to have. This concept design shown below in *figure 36* shows how a menu could be implemented.

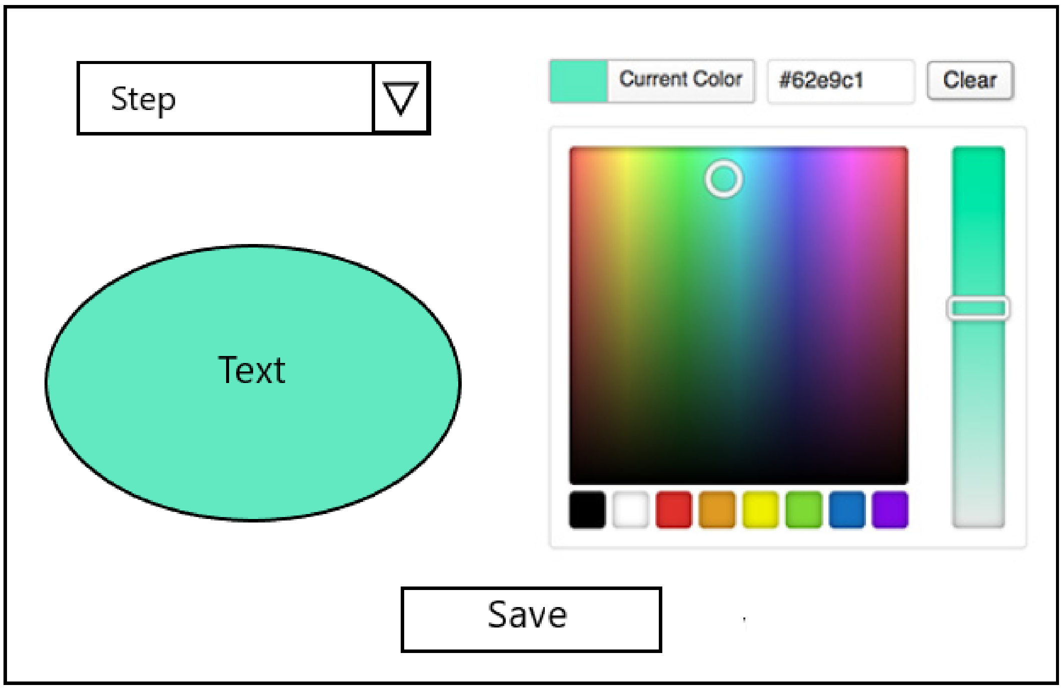


Figure 36- Concept design for colour changing a node

Currently there is an extremely simple UI design for the application. This was to allow ease of use for new users. A framework like Angular could be used to make a more modern looking UI but this would require adding JavaScript and html to the application and in its current state this could overcomplicate the project.
A feature that was simply out of scope for this project that would be useful would be the ability to directly link step nodes to their code snippets. This would allow engineers to quickly navigate their code base.

8 Appendices

I have added screen shots and a link to the code base [10] just to give examples of what the web page looks like that the example test suite has been made for. The project does not require the user to run Pet Clinic.

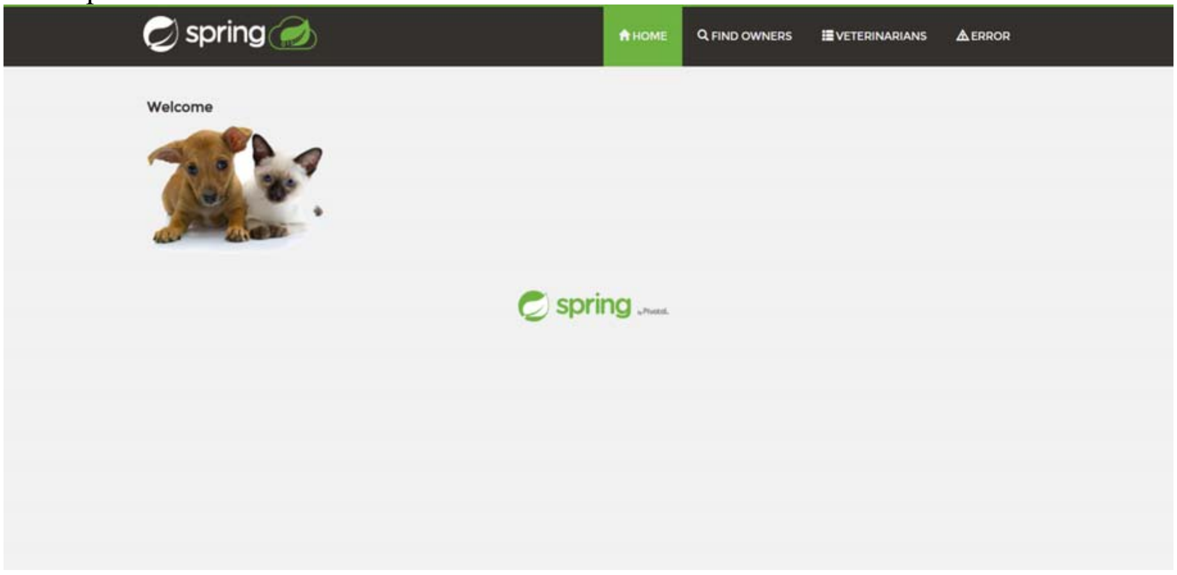
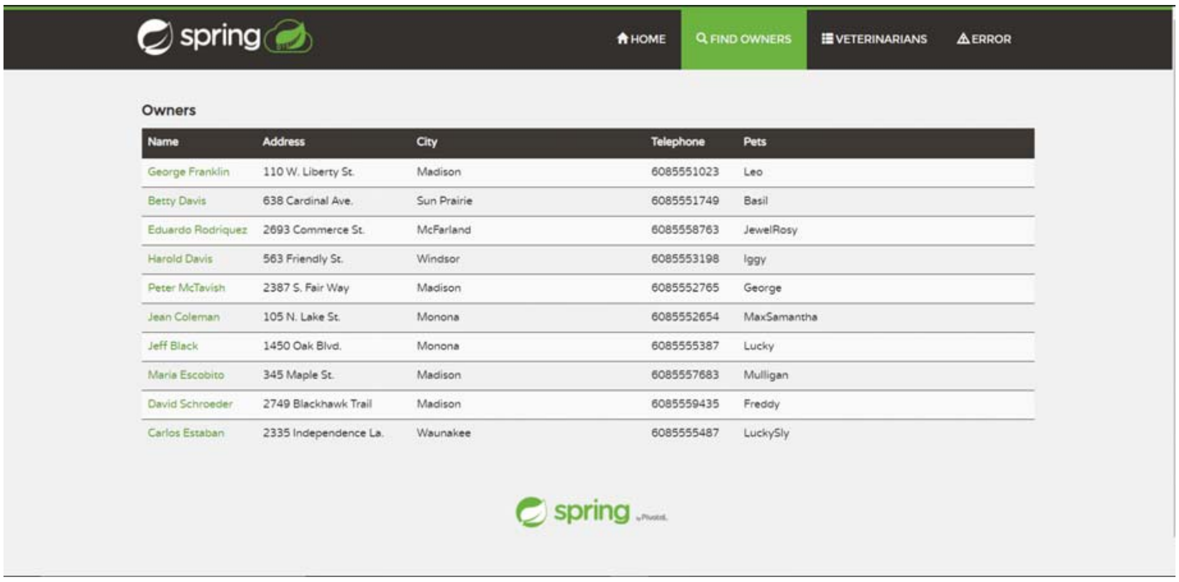


Figure 37 - Pet Clinic Home Screen



Owners

Name	Address	City	Telephone	Pets
George Franklin	110 W. Liberty St.	Madison	6085551023	Leo
Betty Davis	638 Cardinal Ave.	Sun Prairie	6085551749	Basil
Eduardo Rodriquez	2693 Commerce St.	McFarland	6085558763	JewelRosy
Harold Davis	563 Friendly St.	Windsor	6085553198	Iggy
Peter McTavish	2387 S. Fair Way	Madison	6085552765	George
Jean Coleman	105 N. Lake St.	Monona	6085552654	MaxSamantha
Jeff Black	1450 Oak Blvd.	Monona	6085555387	Lucky
Maria Escobito	345 Maple St.	Madison	6085557683	Mulligan
David Schroeder	2749 Blackhawk Trail	Madison	6085559435	Freddy
Carlos Estaban	2335 Independence La.	Waunakee	6085555487	LuckySly

Figure 38 - Pet Clinic Find Owners Screen

9 References

Project source code can be found here

- https://gitlab2.eeecs.qub.ac.uk/40206673/vg03-csc3002-40206673

[1] G. Di Lucca and A. Fasolino, "Web Application Testing", Web Engineering, pp. 219-260, 2006. Available: 10.1007/3-540-28218-1_7 [Accessed 10 February 2021].

[2] Microsoft Technology Licensing LLC, "TECHNIQUES FOR PRIORITIZING TEST DEPENDENCES", 7,840,844, 2010.

[3] ProfessionalQA, “WebService Testing”, https://www.professionalqa.com/web-service-testing (retrieved 22 October 2020)

[4] J. A. Jones, M. J. Harrold and J. Stasko, "Visualization of test information to assist fault localization," Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, Orlando, FL, USA, 2002, pp. 467-477, doi: 10.1145/581396.581397.

[5] James Whittaker, http://blogs.msdn.com/james_whittaker/archive/2008/09/19/the-future-ofsoftware-testing- part-5.aspx, Last accessed: 22 October 2020.

[6] "Kizceral: A Dynamic Realtime Architecture Diagram", Medium, 2020. [Online]. Available: https://medium.com/knerd/kizceral-a-dynamic-realtime-architecture-diagram-7198f07235d2. [Accessed: 22- Oct- 2020]

[7] "Gauge Documentation", Docs.gauge.org, 2020. [Online]. Available: https://docs.gauge.org/overview.html?os=null&language=null&ide=null. [Accessed: 28-Dec- 2020].

[8] V. Garousi, A. Keleş, Y. Balaman and Z. Güler, "Test Automation with the Gauge Framework: Experience and Best Practices", Computational Science and Its Applications

– ICCSA 2020, pp. 458-470, 2020. Available: 10.1007/978-3-030-58802-1_33 [Accessed 28 December 2020].

[9]N. Koochakzadeh and V. Garousi, "TeCReVis: A Tool for Test Coverage and Test Redundancy Visualization", Testing – Practice and Research Techniques, pp. 129-136, 2010. Available: 10.1007/978-3-642-15585-7_12 [Accessed 28 December 2020]

[10] "spring-projects/spring-petclinic", GitHub, 2020. [Online]. Available: https://github.com/spring-projects/spring-petclinic. [Accessed: 28- Dec- 2020].

[11] GitHub. 2021. vgarousi/GaugeDepend. [online] Available at: <https://github.com/vgarousi/GaugeDepend> [Accessed 16 February 2021].

[12] Testinium.com. 2021. Testinium - Complete Test Automation Management Tool.. [online] Available at: <https://testinium.com/> [Accessed 21 March 2021].

[13] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca and P. Tonella, "Web test dependency detection", Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p. 154, 2019. Available: 10.1145/3338906.3338948 [Accessed 30 March 2021].

[14] H. Achkar, Model Based Testing Of Web Applications. Sydney, Australia, 2010, p. 7.

[15] F. Dib, "regex101: build, test, and debug regex", regex101, 2021. [Online]. Available: https://regex101.com/. [Accessed: 13- Apr- 2021].

[16] "BtcTurk.com | Bitcoin ve Kripto Para Alım Satım Platformu", Btcturk.com, 2021. [Online]. Available: https://www.btcturk.com/. [Accessed: 20- Apr- 2021].