Homework #3

CECS 550

Dr. Ragade

March 20, 2014

Team Wikiviz

Ashley Revlett

Everett Rush

Sarah Mullins

**1)** *Define a Fishbone Diagram (Ishikawa diagram, Cause-Effect diagram)*

*appropriate to your project. Mark on this diagram priorities for different use cases.*

*Create operational profiles for software components or modules or subsystems.*

*Also identify where normal use, misuse and abuse cases could arise and hence enable*

*test case planning.*

As defined in the Software Requirements Specification and expanded here, use cases include:

**Table 1. Use Cases**

|  |  |  |
| --- | --- | --- |
| **ID** | **Use Case** | **Priority** |
| 1 | User searches for Wikipedia article | 1 |
| 2 | User browses related Wikipedia articles | 2 |
| 3 | User views more detailed information about Wikipedia article | 3 |
| 4 | User shares Wikipedia article on social media | 4 |

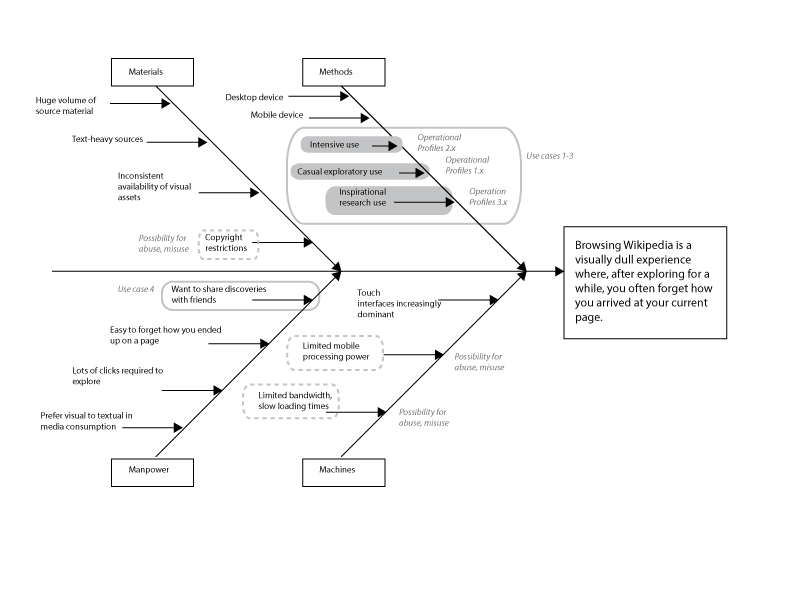
Based on the Customer, System Model, and Functional Profiles from Homework 2 and expanded here, the Operation Profile is as below:

**Table 2. Operational Profile**

|  |  |  |
| --- | --- | --- |
| **ID** | **Operation** | **Occurrence Probability** |
| 1.0 | Casual, First search | 0.13 |
| 2.0 | Enthusiast, First search | 0.05 |
| 3.0 | Researcher, First search | 0.02 |
| 1.1 | Casual, Viewing Graph | 0.26 |
| 2.1 | Enthusiast, Viewing Graph | 0.10 |
| 3.1 | Researcher, Viewing Graph | 0.04 |
| 1.2 | Casual, Exploring further | 0.26 |
| 2.2 | Enthusiast, Exploring further | 0.10 |
| 3.2 | Researcher, Exploring further | 0.04 |
|  | **TOTAL** | **1.0** |

The fishbone diagram below illustrates the core problem the Wikiviz app seeks to solve. It incorporates use cases and operational profiles while also pointing out potential spots for abuse and misuse.

**Diagram 1. Wikiviz Fishbone Diagram**



**2)** *Write out your test plans in terms of documented test suites, appropriate test data sets. Identify these plans for unit, integration and system tests. Mark the Most Important*

*Tests. Consider the usage based strategies, as Google uses. Also note usage corresponding to Configuration Management concerns.*

**Unit Tests:**

A unit test suite is to be created for each module to verify that all methods meet the design requirements. We will have test suites for:

* Display
  + create window
  + accept user search terms
  + display nodes
  + button interactions
* Controller
  + pass messages between display, model, and network
* Model
  + record data from network
  + dispatch notifications on model update
* Parser
  + return most important links on page
  + extract image from page correctly
* Network
  + sanitize query
  + retrieve page data
  + handle network errors

Each test suite will include at least one test case for each method in the module being tested. Each test case will test the functionality of a specific method, ensuring it produces correct output given known inputs, and that it handles errors and malformed input gracefully.

Test data must be created to test against known good and bad inputs. Test data required will include:

* List of possible keywords users could enter, including typos, bad characters, intentional malformed input
* Sample Wikipedia pages, images and list of correct relevant links

Any functions which make use of the network, rely on outside resources, or require user interaction are not unit testable as-is. These functions will also require the use of test data to simulate part of their functionality. These functions include:

* All Network functions, including on\_success(), on\_error(), get\_page()
* Model functions that make use of the Network, including on\_update()
* Display module’s Node.image property

These functions will require the use of test data, such as locally stored HTML files that are copies of Wikipedia pages and locally stored images.

**Integration Tests:**

Integration test suites will be created to test the interaction between different modules. Integration tests will also simulate user interaction when necessary to complete a test. Network access may be used during integration tests.

We will have integration test suites for interactions between:

* Model and Display (notifying Display of Model update)
* Display and Controller (notifying Controller of keyword request)
* Network and Display (notifying Display of network error)

**System Tests:**

System tests will be created to test the execution of all Use Cases. These tests will make use of scaffold code to simulate user interaction so that they can be run automatically. These tests will cover the functionality defined in the Wikiviz Use Cases from the SRS:

1. User searches for Wikipedia article
2. User browses related Wikipedia articles
3. User views more detailed information about Wikipedia article
4. User shares Wikipedia article on social media

Performance testing will be also performed during the System Tests to determine whether changes to the network module code improve network performance. The application’s performance will also be profiled to discover bottlenecks. Poor performance will be simulated to test how error conditions are handled.

In addition, manual end-user testing will be performed to ensure that the application operates well “in the wild” under various network speeds and computing hardware.

**Most Important Tests:**

The most important tests (MITs) are those that cover that the most vital and common functions, and the most high-risk functions. These can be identified by looking at the Operational Profiles, Use Cases, and the complexity of each node in the UML design graph. Tests that ensure basic functionality is met, such as drawing the nodes and extracting links from page data, are MITs. Tests that cover the network functionality also qualify as MITs, because the network is more likely to encounter errors and is also more open to abuse than other functionality.

**Change Management:**

The Test Plan will need to evolve alongside development. As such, configuration management is vital. If the change is extensive, such as a new type of test or a change in testing strategy or tools, the change must be requested using the Software Change Request form (see Homework 1).

If the change is a minor change that occurs naturally alongside development, it may be recorded as a git commit. If the test is created in response to a bug or issue, a comment should be added to that issue indicating which new test and git commit fixes that bug.

**3)** .

1. **Library Limitations**
2. **Architectural Design**
3. **UI Requirement Change**
4. **Module Miscommunication**
5. **Network Communication**
6. **Security Vulnerabilities**
7. **Legal Responsibilities**
8. **Hardware Limitations**
9. **Performance**
10. **Lack of Communication between Teams**
11. **Implementation Errors**

Defect Submission: UI Architecture doesn’t make sense.

Error Category: 2

Description: The UI design as stated in the design document is incapable of producing the UI as defined in the SRS and sketches.

Location: Display Module

Estimation of time to fix: 50 hours

Defect Submission: UI Team broke Network’s working code

Error Code: 10

Description: The display team altered the model module. In the process working code was deleted. The code communicated back and forth between the controller.

Estimation of time to fix: 20 minutes

Defect Submission: Parent-Child relationship was left out of architecture

Error Category: 2

Description: The UI and controller were not designed to represent the parent-child relationship. So the controller doesn’t accept the proper parameters. The model is not designed to include the parent relationship.

Estimation of time to fix: 2hrs

Defect Submission: Parser returns incorrect set

Error Category: 11

Description: The parser does not return the correct set of information after it parses a page.

Estimation of time to fix: 45min

Defect Submission: UI Node Bug

Error Category: 2

Description: Fast mouse movements and clicks cause the scatter widget (Node) to break. It will rotate around and scale up and down unexpectedly.

Estimation of time to fix: Included in Architecture Redesign

Defect Submission: UI Controls Translation

Error Code: 2

Description: The UI controls for searching, making a new search, and sharing the graph do not translate as the scatterplane pans.

Estimation of time to fix: Included in Redesign

Defect Submission: UI Control Scale

Error Code: 2

Description: The UI Controls do not scale properly as the scatter plane zooms in and out.

Estimation of time to fix: Included in Redesign

Defect Submission: Cannot Remove Non Type UI pop up

Error Code: 2

Description: The UI Pop Up tries to remove the search bar twice when the yes option for new search is selected.

Estimation of Time to fix: Included in Redesign.

Defect Submission: X already has a parent

Error Code: 2

Description: A widget will try to add X twice. The first add succeeds, but the second throws an exception.

Estimation of time to fix: Included in Redesign

Defect Submission: The Picture moves faster than background

Error Code: 2

Description: The nested widget Node has a background and an Async Image attached to it. The stencil for the Async Image will translate faster than the parent as the scatterplane is panned.

Estimation of time to fix: Included in Redesign

Defect Submission: Summary Pop Up doesn’t appear and throws exception

Error Code: 2

Description: When a node is clicked, the summary popup will not display. The Pop Up class is designed to hold a single widget. So to get around that I nested multiple widgets inside the contents of the Pop Up. Kivy did not like that.

Estimation of time to Fix: Included in Redesign

Defect Submission: RstDocument is an inappropriate choice for Text Summary Pop Up

Error Code: 2

Description: The RstDocument is classified as an experimental Widget. It does not have a robust implementation. Consequently the UI design that used a RstDocument to display the Node Summary failed.

Estimated time to fix: Included in Redesign

Defect Submission: The Page Layout causes the scatter plane to reset to (0,0)

Error Code: 2

Description: When a user clicks a node, the scatterplane will move to the initial position before the text summary appears.

Estimated time to fix: Included in Redesign

Defect Submission: The text box distorts text

Error Code: 2

Description: Once the user performs a search and chooses yes for new search, the textbox that appears distorts the text.

Estimated time to fix: 45 min

Total Number of Defects to date: 13

**4)** *Calculate the McCabe complexity. From the Program graphs (object diagrams, dynamic models, functional models, CRC graphs etc.), generate the minimal number of test cases for each use case. Identify these test cases as functional or structural, black box, white box etc. For white box testing identify the paths through the modules. Relate the number of these test cases to the design complexity. Document your testing plans and estimate the testing effort. Based on your Most Important Tests, declare a target coverage for the testing effort. Use McCabe Software or equivalent from open sources.*

The process of deriving the McCabe complexity of Wikiviz is reproduced here from Homework 2:

The equation for deriving the McCabe’s Complexity of an application is based off its state diagram (Braude, Bernstein):

Complexity = # of graph edges - # of nodes in graph + 2

We can use the state diagram from part 1c to compute the complexity of Wikiviz:

Complexity = 30 - 12 + 2 = 20

As the textbook states, this result represents the least number of paths that can be combined to generate every possible path of execution, and therefore, the number of paths that should be tested to ensure adequate coverage.

We will now generate the minimal number of test cases for each use case using the class diagrams and sequence diagram. We begin with Use Case 1, “User searches for Wikipedia article,” and use the strategy found in Chapter 26 of the “Software Engineering - Modern Approaches” textbook to find a set of basis paths. The steps are as follows:

1) Start with the straight path through the code (all conditions true).

2) Set the first condition to false, keeping all the rest true.

3) Reset the first condition to true, set the second condition to false, keeping all the rest true.

4) Continue, setting the next condition false with all the rest true.

5) Stop after the last condition has been set false.

For Use Case 1, we look first to the sequence diagram to see what steps are taken in the program. First, the program starts and the display is initialized. Then, the search button is pushed and the controller gets the page from the network, using the appropriate URL from the Wikipedia server. If successful, control of the program is sent to the controller module. Next, the controller sends the page data to the parser in order to extract links, images, and a text summarization. The controller sends this data to the graph module, so that a node containing this data can be shown on the screen. The display module then updates the screen display, and gets further nodes from the graph module. This completes the processes needed for Use Case 1, and represents the basic functionality of Wikiviz. From starting the application to successfully reaching the page data, six sequential steps occur, meaning that 6 test cases will be required. The parser’s actions take place in parallel, so we can now utilize the steps given earlier by Braude and Bernstein. Setting the get links, get images, and get text summarization functions to false one by one, we obtain 4 possible test cases for full coverage (T-T-T, F-T-T, T-F-T, T-T-F).

We add this to our previous 6 to obtain a running total of 10 test cases. Next, the node adding and display updating functions also run in parallel; setting each to false in succession yields a result of 3 test cases (T-T, F-T, T-F). Finally, the get node function has only one possible control flow, so adding this single test case to the previous total gives us 14 necessary test cases for Use Case 1. All of the test cases given are structural and white-box; each statement and logical branch point must be tested to ensure that successful results are produced.

Next we analyze Use Case 2, “User browses related Wikipedia articles.” This use case largely mimics Use Case 1, excluding only opening the application, initializing the display, and pushing the search button. We exclude these 3 steps from the number of test cases required, to get a total of 11 necessary test cases. Like Use Case 1, these test cases are structural and white-box.

We turn now to the class diagrams of Wikiviz to analyze Use Case 3, “User views more detailed information about Wikipedia article.” In order to perform this action, the parser must retrieve the previously extracted HTML file and send it to the display. The display will then create a window containing the full contents of the current active article. With only two functions needed, we derive 3 test cases (T-T, F-T, T-F). These test cases are functional and black-box, since we are only concerned about whether the complete Wikipedia article is displayed or not, and pay no attention to the inner workings of the functions used.

The final Use Case is “User shares Wikipedia article on social media.” We look to the behavioral requirements given in the Software Requirements Specification to analyze the test cases required for this use case. First, upon clicking the social network logo, the display creates an event that is recognized by the network. Next, a request is sent to the social network by the network to retrieve the page necessary to post a summary of the graph. The network will then send the necessary graph data to the posting box, and the social network will then take control of operations. This results in a total of four functions, and thus, 4 test cases (T-T-T, F-T-T, T-F-T, T-T-F). These test cases are again all functional and black-box, as only the success or failure of sharing the graph data with the social network is notable.

Overview:

Use Case 1: 14 test cases

Use Case 2: 11 test cases

Use Case 3: 3 test cases

Use Case 4: 4 test cases

**Total: 27 test cases**

The number of test cases required (27) is surprisingly close to the number produced by the McCabe Complexity calculations (20). This confirms the fact that Wikiviz is relatively complex and will require extensive testing. However, we do not believe that our application is *overly* complex, as all the functions included are absolutely necessary to meet the specifications given in the SRS.

Our testing plan will be carried out using the unit tests described in both this question and 2). The tests must be both written and executed. Following are predictions of testing effort:

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case** |  | **Hours Required** | **Total** |
| **1** | **Model** | 10 |  |
|  | **Controller** | 15 |  |
|  | **View** | 8 |  |
|  |  |  | 33 |
| **2** | **Model** | 2 |  |
|  | **Controller** | 2 |  |
|  | **View** | 2 |  |
|  |  |  | 6 |
| **3** | **Model** | 0 |  |
|  | **Controller** | 3 |  |
|  | **View** | 2 |  |
|  |  |  | 5 |
| **4** | **Model** | 10 |  |
|  | **Controller** | 1 |  |
|  | **View** | 0 |  |
|  |  |  | 11 |
| **Total Hours** |  |  | 55 |

From Question 2 above, our MITs are those that ensure basic functionality, including node creation, link extraction, and successful network operation. We will define test coverage as number of test coverage items exercised divided by total number of test coverage items (equation from istqbexamcertification.com), and strive to cover as many of these MITs as possible. We will remove test cases from Use Case 3 and 4, as their functionality is not completely essential to the operation of Wiviz. These leaves us with a minimum of 20 out of 27 possible test cases, or approximately 75% test coverage. Certainly we will aim for 100% test coverage, but we will use this metric as a baseline.

**5)** *Design and note all test-stub drivers that you wrote for test instrumentation. Write the extent to which you have test coverage. What types of coverage do you have for the object oriented part of your design? Give an overall measure based on these measurements for the quality of testing. Discuss which tool you used, such as JUNIT, or NUNIT or some other testing tool.*

We used PyUnit, the unit tester packaged with Python, to create test stubs for Wikiviz. This module allows the user to test another program by mimicking its functions with the string “test\_” as prefixes. PyUnit runs each function as a test and reports back the number of errors that occurred overall.

Each function in the parser module is tested using PyUnit: initialization, extracting the article links from the Wikipedia page returned by the network module, and prioritizing the relevancy of the links (not including images). The initialization function has full test coverage, as every object attribute is initialized (link url, link name and link priority), there are no decision points, and there is only one code path to be executed.

For the link extraction test, both Wikipedia pages with and without internal links or images are addressed. Pages with internal links provided full statement coverage, but those without (so-called “dead-end pages”) ensure that every decision point in the function is executed.

In regards to path coverage, the link extraction test performs well. If the link extracted from the article page has a type, it may or may not be an internal link. If not, the loop exits immediately. If so, only internal links to other articles are considered; links to meta-pages are irrelevant for our purposes. Next, we address the keyword filtering function. If an undesirable keyword is found in the page URL (e.g. ‘Category’, ‘Talk’, which signify meta-pages), the page object which contains the page url, name, and priority level is removed from the relevant list. If not, the loop simply exits.

Finally, the link prioritization test also provides relatively good test coverage. Stub objects with only the link name attribute are tested for priority level: one link name that includes the search term (higher priority) and one that does not (lower priority). This ensures that every line of the prioritization function is executed, the defining metric of statement coverage. Another important method of prioritizing links is counting how many times a link word appears on a given page, and whether this exceeds the average number of times link words appear. To ensure adequate branch coverage in this area, search terms with below average, average, and above average occurrences are tested.

In terms of path coverage, the link prioritization test also performs well. If the search term is contained within the link name, its priority level is increased; if not, its priority level stays static. Similarly, if the link name occurs more times than the average number of times that link names occur, its priority level increases; if not, its priority level decreases. All distinct code paths are therefore tested.

Moving on to the network module, determining test coverage is relatively easy, for reasons we will soon discover. Beginning with the initialization test, the single attribute of the Network object (headers) is initialized each time and tested for its Singleton status. No decision points or independent paths exist. The next two functions, on success and on error, also have no decision points or independent paths; each statement is performed no matter what data is input. Finally, the function to retrieve the relevant Wikipedia page is similarly straightforward. The unit test will send both valid and invalid keywords to the function to ensure that the “try” and “except” statements will both execute. The rest of the statements in the function, setting and requesting the relevant URL from the Wikipedia server, will execute regardless of the results of this decision point.

Unit tests for both the display and model modules are currently in progress. In the display module, we currently only have unit tests for creating and implementing the search button. Extensive tests will need to be performed for the following functions: creating objects to represent nodes (widgets), add nodes themselves, connect nodes to edges, confirm new searches and move the screen upon mouse or finger touch.

Model module tests are also not yet complete, but we will need to analyze whether this module can successfully call the display to add nodes and edges, print a given graph and send a message on success. The nodes and edges, in turn, must successfully initialize.

Next we will analyze the current level of test coverage. Wikiviz is still in the early development stages, so Use Case 3 and 4 from 4) have not yet been addressed. Six test cases have coverage for Use Case 1 and Use Case 2 both (as they are very similar), giving a result of 12/27 possible test cases covered, or about 44%. From this analysis, we can also tell that, in terms of object-oriented programming, we still need test coverage for two out of the three main modules (display and controller). We must continue creating unit tests to have some level of coverage for all three modules and 75% test coverage overall, but we are still making progress.