**Problem Statement**

WikiMapper allows a user to search and explore Wikipedia in a visually-driven, graph-based way that clearly illustrates the connections between related articles.

**References**

[1] Wikipedia contributors. (2013, December 20). Wikipedia: Version 1.0 Editorial Team/Assessment [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia:GRADE

**Similar Works**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name and Link** | **Description** | **Release Date** | **Platform** | **Pros** | **Cons** |
| [**WikiNodes**](http://www.idea.org/WikiNodes.htm) | Visually-driven, interactive graph-based Wikipedia | 2011 | iOS (iPad) | Well-designed and currently active. Full featured (multilingual, annotations, bookmarks, sharing) | iPad only.  Occasional slow response times in the UI. |
| [**Wikistalker**](http://sepans.com/sp/works/wikistalker/) | Visualization of the meta-structure of Wikipedia articles | 2011 | Web (HTML5/JS) | Novel visualization of relationships between articles. | Interesting way to explore, but not very useful as article content isn’t shown, only meta-structure. |
| [**The Web Stalker**](http://www.visualcomplexity.com/vc/project_details.cfm?index=7&id=7&domain) | Experimental web browser that uses graph-based representation of web pages and links | 1997 | Downloadable Program (Windows, Mac) | Interesting visualization of information. One of the first in this area. | Outdated. Non-wikipedia related. |
| [**Ask Ken**](http://datavisualization.ch/tools/ask-ken/) | Node-Link diagram that allows to visually navigate through interconnected topics provided by the Freebase Service | 2010 | Web (Ruby, JS) | Unique “ring chart” visualizations. Beautiful design. | Buggy. Not usable on Mac. Not based on Wikipedia content. |
| [**IndyWiki**](http://indywiki.sourceforge.net/index.html) | Visual Wikipedia Browser | 2008 | EXE (Python, QT) | Good concept. Runs on all OSes. | Terrible design. Looks outdated. |
| [**WikiMindMap**](http://www.wikimindmap.org/viewmap.php?wiki=en.wikipedia.org&topic=Wolfgang_Amadeus_Mozart) | Interactive mind map visualization of Wikipedia | 2007 | Web | Mind map visual is useful. | No graphics, no pictures. Just text. Limited ability to click down into articles. |
| [**Liveplasma**](http://www.liveplasma.com/#/artist/Wolfgang_Amadeus_Mozart/255b884c-25e9-4da2-99e9-b104ed637eae/US) | Graph-based visually-driven search engine | 2004 | Web (Flash/Flex) | Good use of animation, audio, and interactivity. | Can search only music. Non wikipedia-based. |

**Requirements and Analysis**

*External Interface Requirements*

User Interfaces

The main elements of the GUI will be a search bar, graph display, and node manipulation. The search bar will consist of a text area for entering search terms and buttons to reset or initiate the search process. The graph display will contain an active article, including a text summary and a representative image from the Wikipedia page, if available. From this central point, the names of related articles will be displayed as link branches. Clicking on the name of a related article will spawn the branching nodes of that article; double-clicking a node reveals the text summary overlay. The user can also manipulate the node positions by clicking and dragging.

Hardware Interfaces

None.

Software Interfaces

Wikipedia Mapper will be built for Windows systems, though compatibility with Macs is also of interest. HTTP will be used to connect to Wikipedia. Python and the Kivy framework will be used for the application logic and GUI. Kivy also uses Cython for resource-intensive modules. The Beautiful Soup and textwrap library will be used for text parsing. For target platform, please see WikiMapper Software Requirements Specification.

Communications Interfaces

Wikipedia Mapper relies on a network connection with Wikipedia. In order to avoid over-taxing Wikipedia servers, throttle processes will be implemented to control the rate of connection requests. These processes may be customized for the network connection speed of individual systems (faster systems would be allowed more requests).

*Functional Requirements*

* Enter search query

User will type their search term into a text area

* Clear search

Clear display area and reset search request

* Send queries to Wikipedia upon button click

Send contents of search text area to Wikipedia

* Throttle network requests

Control number of requests sent according to network connection speed

* Parse images, text, and links from corresponding Wikipedia article

Categorize article elements as image, text, or hyperlink

Extract groups of elements

Determine relationships between active article and related articles based on

hyperlinks

* Record query results as part of graph

Save elements in model

* Summarize article text

Use Python library to extract introductory paragraphs

* Draw graph structure in display area

Draw active article node with text summary

Draw branching related article nodes with their titles

* Show text summary and representative image upon double-clicking a node
* Change active article node upon clicking a related article node

Shrink active article node to title

Pan screen to center on relevant related article node

Render new related article nodes

* Reposition nodes

Alter position of nodes in display area upon clicking and dragging them

* Preload media elements

Save as many parsed elements and relationships to cache as possible for later

retrieval

* Future functionality

Import/export graphs

Connect and post to social media sites

Use Google as a data source

Apply same data parsing strategies to Google search queries

Switch between Google and Wikipedia search

*Behavior Requirements*

### *3.3.1 Search* Use Case

1. User enters term in search box
2. User clicks “Search” button
3. Network connects to Wikipedia and retrieves the relevant article page
4. Parser finds related articles based on hyperlinks within page
5. Parser prioritizes related articles by relevancy
6. Model and UI create first active article node with number of related article nodes as specified by user
7. System stores current graph structure in model

**3.3.2 *Reset Search* Use Case**

1. User clicks on “Reset” button
2. Model deletes current graph structure
3. UI clears display area

**3.3.3 *Change Active Node* Use Case**

1. User clicks on a related article node
2. UI centers node on screen and sets corresponding article as the active article node
3. System performs Steps 5-8 under *Search* Use Case

**3.3.4 *Show Text Summary* Use Case**

1. Users double-clicks on the active node
2. Parser extracts introductory paragraphs from Wikipedia article
3. UI displays text summary overlay and representative image

//change for how you actually zoom

**3.3.9 *Zoom* Use Case**

## Performance Requirements

## Connections to Wikipedia and graph render time will not exceed 10 seconds

## Need to ensure network connection does not time out, and that the system does not try to render the graph for an indefinite period of time

Security

Several of the security metrics in Braude and Bernstein’s list do not apply to Wikiviz, as no user-sensitive data will be sent to the Wikipedia server. We will still address the relevant areas:

* Confidentiality
  + We currently do not have measures in place to ensure that search terms will not become visible to unauthorized persons.
* Integrity
  + We currently do not have measures in place to ensure that search terms are not altered in transit to the Wikipedia server.
* Availability
  + If WikiMapper sent more network requests than Wikipedia’s server could handle (in the case of an article with many, many links), the program could potentially freeze and be unable to create a graph from all the information.

**Design and Architecture**

*Design Goals or Tasks and Related Software Patterns*

Being event-driven requires the application to listen for events dispatched by objects. An **Observer** pattern can be used here. Kivy already uses the Observer Pattern within its EventDispatcher class.

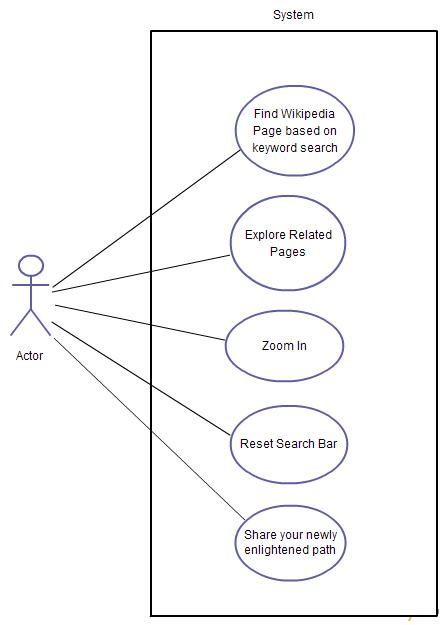
The **Singleton** pattern will be used for major modules that only need one instance, like the main Model, Controller, Network, Parser, and Display.

For finding related node content, we plan to first support Wikipedia as a data source, and at a later time add support for Google or other sites. To allow that flexibility, we can use the **Adapter** pattern to easily change data providers in the future. Or, if we wish to allow users to change the data source at run-time, we could use the **Strategy** pattern instead.

The application requires a lot of network traffic to a single host (Wikipedia.org). Creating a socket connection is an expensive operation that can take a relatively long time. The **Object Pool** pattern improves performance when initializing new class instances is high (for example, if the initialization relies on a network connection). We could use the Object Pool pattern to create a set of initialized objects and keep them ready for use on demand, allowing a more predictable time frame for object creation.

*Design Overview*

Our challenge is to depict the relationship between content on Wikipedia in a visually stimulating manner, and to allow users to discover new information in a novel way, and to share that information with their friends. The following figure depicts the major use cases covered by the application:



Because our data comes from a network connection to Wikipedia, our application’s performance is reliant upon the network’s performance. To maximize the responsiveness of our application, we will use asynchronous network calls to retrieve the data, and continue the UI event loop in a separate thread. New nodes are to be displayed incrementally, as they arrive via the network.

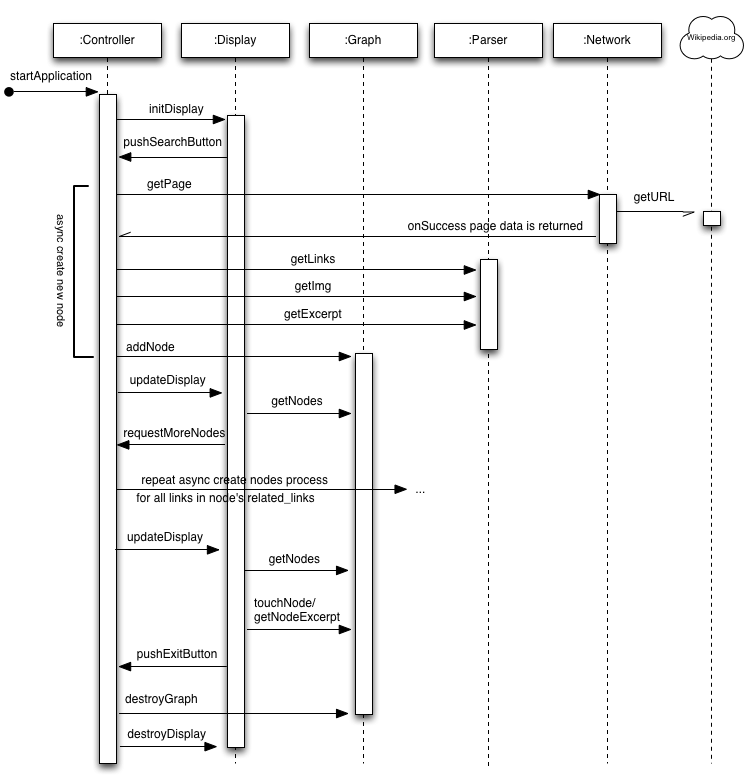
*System Architecture*

The system is constructed from three top-level packages: Display, Model, and Controller. Those packages contain the classes below:

* **Display** – The UI for the sytem; draws the GUI and the Graph representation
* **Controller** – The main business logic for the system
  + **Network** – An interface for interacting with network resources
    - **Page** – Class used by the Network to represent page data
  + **TextParser** – A module to encapsulate text parsing functionality
* **Model** – The model for the data we show
  + **Node** - Vertice in the models’ graph
  + **Edge** - connection between 2 edges in graph

System Operation

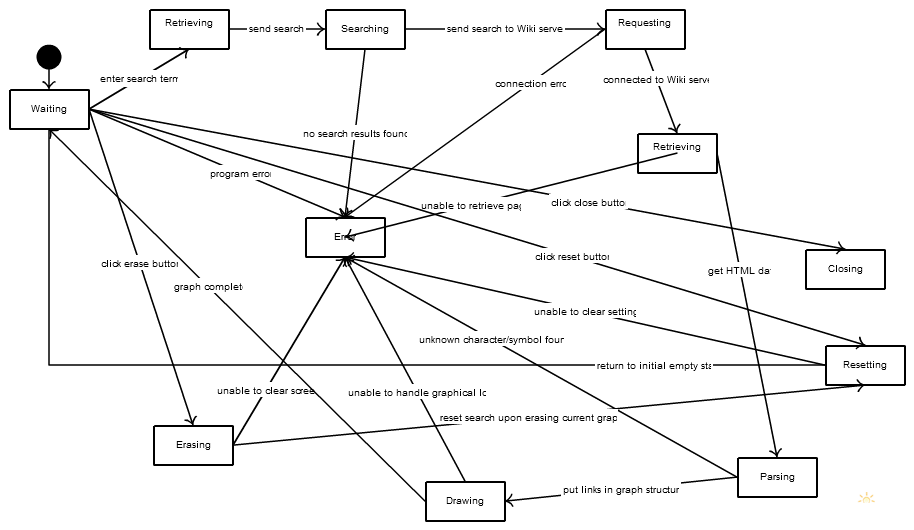
The figure below is the typical sequence of events that occur during an initial WikiMapper session. Asynchronous network communication is used to update the model’s Graph. The Display is notified when the update occurs, at which time the Display updates its drawing of the nodes.



During this session, the application can enter several states:

* Waiting
* Retrieving
* Searching
* Requesting
* Error
* Erasing
* Drawing
* Parsing
* Closing
* Resetting

The following figure depicts the states and transitions that occur during a WikiMapper session.



***Detailed Design***

## Display Module

The Display Module’s responsibility is to:

1. Draw the user interface
2. React to user events (taps, button presses) by notifying controller
3. Read data from the Model and display it on screen

The top-level Display class is a singleton which encapsulates all Display logic. It manages several Kivy-specific UI classes, along with custom Node classes for the application.

Controller Module

The Controller Module’s responsibility is to:

1. Manage the application lifecycle
2. Coordinate the function of the Network and Parser
3. Update the Model

The top-level Controller class is a singleton which encapsulates all Controller logic. It owns the Parser and Network class. The Network class owns the Page class, which is used to manage the data returned from the network before it has been parsed. The Parser class uses the BeautifulSoup library to parse HTML.

Model Module

The Model Module’s responsibility is to:

1. Store and provide a consistent interface for our application data
2. Protect data from unintended alteration

The Model package encapsulates the Graph structure we’re using. It includes classes for a Graph singleton, Edges and Vertices. The graph is maintained with 2 structures, a list of vertices, and a list of edges as vertex tuples. The Node structure contains all information required to display and interact with the node.

**Coding and Inspection Guidelines**

*Testing Plans*

Unit Tests:

A unit test suite was created for each module to verify that all methods meet the design requirements. We 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 includes 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 was created to test against known good and bad inputs. Test data required includes:

* 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 requires 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 have been 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 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 analyze the execution of all Use Cases. These tests make use of scaffold code to simulate user interaction so that they can be run automatically. These tests cover the functionality defined in the WikiMapper 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

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

In addition, manual end-user testing was 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.

**Product and Process Metrics**

*McCabe’s Complexity*

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 WikiMapper:

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.

McCabe and the National Institute of Standards and Technology (NIST) proposed that generally, application modules should be decomposed into smaller modules if the complexity exceeds 10 (or in extreme cases, 15. WikiMapper’s complexity score of 20 clearly exceeds both of these figures, so finding ways to reduce the number of possible paths will be of relative importance as we continue program design.

*Understandability*

Braude and Bernstein define the metric of understandability as:

½ \* percentage of strongly cohesive modules + ½ \* percentage of modules connected to very few other

We will use the formula for graph-theoretic complexity as found in the textbook and our class diagram to determine the level to which classes are highly connected to others:

Connectability = # of modules in the architecture -  # of modules having at least one connection + 1

Connectability = 3 - 3 + 1 = 1

A low complexity value indicates a high degree of connectedness, so we immediately see that the level of coupling in WikiMapper could be an issue. The maximum complexity value would be 4 (if no modules had at least one connection to one another), so the percentage of modules connected to very few others would be 1/4, or approximately  25%.

In order to determine which modules are strongly cohesive, we turn to the metrics of TCC and LCC, or Tight and Loose Class Cohesion (from aivosto.com, “Project Analyzer v10.2”). From this website:

NP = maximum # of possible connections = N \* (N - 1) / 2, where N is the # of methods

NDC = # of direct connections

NID = # of indirect connections

Tight class cohesion TCC = NDC/NP

Loose class cohesion LCC = (NDC + NID) / NP

We strive for the highest values of TCC and LCC possible, with maximums of 1. We calculate each metric for each module:

**Display**

NP = (2 \* 1) / 2 = 1

NDC = 0

NID = 1

TCC = 0

LCC = 1

**Controller**

NP = (3 \* 2) / 2 = 3

NDC = 2

NID = 0

TCC = 2/3

LCC = 0

**Model**

NP = (1 \* 0) / 2 = 0

NDC = 0

NID = 0

TCC = 0

LCC = 0

These metrics show  that while WikiMapper is not very cohesive, it is not *un*cohesive. The Model class has no cohesion, but the Display class and Controller class have strong loose class cohesion and tight class cohesion, respectively. For the purposes of the usability metric, we will count both the Display and Controller class as cohesive. This gives us a percentage of highly cohesive modules as 2/3 or 66%

Therefore, returning to the original metric for usability, we substitute the 66% and 25% from earlier:

Usability = ½ \* 66% + ½ \* 25% = 0.455

With an optimal value being 1, WikiMapper has an average score in regards to usability. We still have room for improvement in regards to increasing cohesion and decreasing coupling in our project.

*Sufficiency*

The sufficiency metric is defined as the percentage of detailed requirements clearly accommodated by a given design. We will use the Software Requirements Specification document as a guide, and divide the analysis into functional, behavioral, performance, safety and security, and software quality.

* **Functional**: We have determined that throttling network requests will be largely unnecessary, as testing has suggested that no Wikipedia page has enough links and relations to cause issues. We also decided to remove the user ability to change the types of articles displayed, and instead integrate prioritizing quality articles into the application itself. Finally, time and security restraints have led us to remove connecting to social networks from our initial list of requirements. Therefore, our project design currently meet 19/22 or 86% of the original functional requirements.
* **Behavioral:** As stated above, we removed the article quality filter and export directly to social network site, so in terms of behavioral requirements, WikiMapper meets 10/12 or 83% of those initially proposed.
* **Performance**: After preliminary network testing, we predict that timing out will not be an issue, as we simply could not make enough requests to cause this. However, we have yet to see if graph rendering will be attempted for an indefinite period of time, so WikiMapper currently meets ½ or 50% of the performance requirements.
* **Safety and Security**: Since we will not be connecting directly to social networks to export data, we have eliminated the single security requirement; therefore, WikiMapper meets 0% of the requirements in this area.
* **Software Quality Attributes**: Again, the security of social networking site information is no longer an issue. Scalability is still yet to be proven, as we have not necessarily tested Wikipedia articles with the greatest number of links. Performance has only been partially met, since we don’t yet know how quickly dense graphs can be rendered. With this in mind, WikiMapper currently meets 3.5/6 or 58% of the original software quality attributes.

*Robustness*

WikiMapper does not have many opportunities for user input, so its robustness is a fairly easy to measure. If the user initially enters a search term that is not found in the list of Wikipedia articles, an error message is displayed and the search is cleared. If the user chooses a number of nodes to display that is larger than the available number of nodes, the application will simply display as many as possible.

*Flexibility*

We can measure the flexibility of WikiMapper by looking at its design patterns and levels of class inheritance. The class diagram within the Software Design Specification shows a few examples of  class inheritance: TextParser and Network (and therefore Page) inherit the Controller class, which inherits the Display class. Our application also uses variations of at least six different design patterns.

In addition, one of our future goals is to extend the capabilities of WikiMapper to visually exploring Google search results. As extensibility is strongly connected to flexibility, achieving our goal would further improve this metric.

*Reusability*

In terms of reusability, we will look at the following five areas, as defined by Braude and Bernstein**:**

* Abstract enough to get wide coverage
  + WikiMapper is being developed for Wikipedia, but our future goal is to expand the visualization techniques to Google searches. Certain classes related to the text parser are currently using Wikipedia standards to sort links, and the main network class is being constructed to request data from the Wikipedia server, but none of the other classes or methods are website-specific.
* Specific enough to be useful
  + As explained above, the classes used in WikiMapper are general enough to eventually be applied to Google searches, but are specific enough to address the requirements for visualizing Wikipedia searches.
* Parameterized methods
  + Currently, many of our class methods do not have parameters. Those that do have only one parameter, which is less than the maximum of six recommended by Braude and Bernstein.
* Degree of coverage (0 through 2: negligible coverage of different applications, as wide as can be expected)
  + 2: covers any Wikipedia search term and displays an error message if term not found
* Degree of content (0 through 2: negligible content or substance, very rich content or associations
  + 2: searches yield text summary and image and allows user to view entire
  + Wikipedia article.
* Parameterization of methods - allows method reuse (0 through 2: very restrictive methods and very narrow scope, widely applicable methods)
  + 2: methods are very flexible, as they have few parameters

*Reliability*

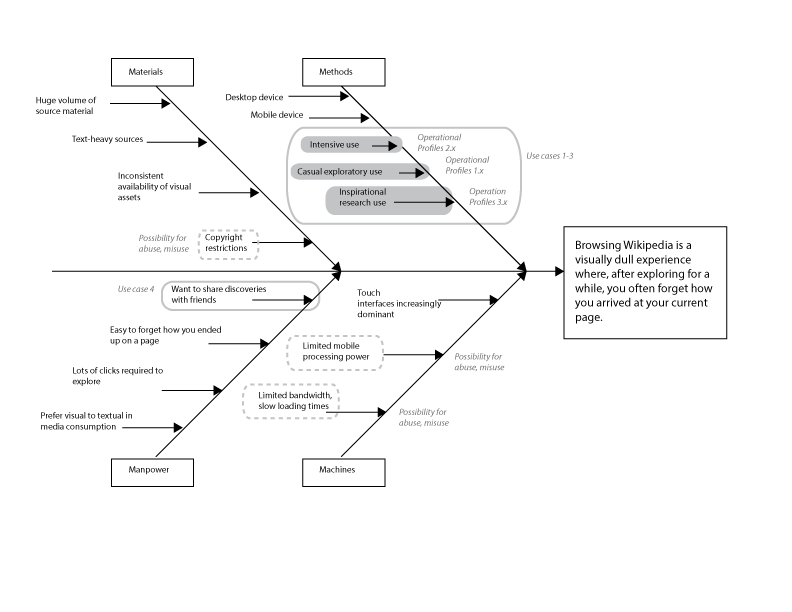
The reliability of an application can be measured by looking at two main types of failure, choke points and deep inheritance. A choke point refers to a class which relates to many other classes, while deep inheritance occurs when a class has three or more levels of inheritance.

WikiMapper has very few potential choke points, as the greatest number of related classes is only two (Graph, Node and Edge). Deep inheritance is currently not an issue, as the Display class only has 2 levels of inheritance, but this could become a potential zone of unreliability.

We can also look at the state diagram of WikiMapper to identify potential bottlenecks; states involving the most transitions could signal unreliability. The Waiting state and Error state contain the most transitions (6 each). If necessary, we could break up these states into sub-states, e.g. specify particular Waiting states depending on what state immediately preceded it.

**Software Quality Assurance**

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



**Management and Process**

The most critical stage in the process was planning and design. By choosing Kivy for a UI framework, we received both benefits and disadvantages that affected the entire project. As this was the first time any of our team had used Kivy, those disadvantages were not well understood until they were directly encountered during implementation. Kivy uses its own Python interpreter, which caused problems when all team members tried to set up their development environments. This could be solved in the future by using a tool designed for synchronizing development environments, like a shared virtual machine or Vagrant.

Github and email proved to be a satisfactory way for us to manage the project, after some team members overcame an initial lack of familiarity with git. Github’s issue tracking and commenting system were helpful for group communication and documentation. There was occasional confusion over meeting schedules, which could be easily remedied by a shared calendar in the future.

Potential maintenance of the project should be minimal, assuming the Wikipedia API remains stable. To support new touchscreen devices, a new version of Kivy may eventually need to be merged into the source, and certain code updated. How much effort this would require depends on how much the Wikipedia API and/or Kivy changes, which could be significant or trivial.

We do have unfinished work that could be taken over by another team. Our documentation is supportive of this task, but it needs to be updated to reflect the architectural changes we’ve made during development. Our discussions and issues list on Github will provide supporting information that is more up-to-date than the SDS. The SRS, however, would still be very beneficial as it is still current.

We are treating this project as open-source software, rather than packaged, proprietary software. The source code is available on Github for anyone interested in reviewing, modifying, or maintaining the codebase. Because realistically our team members are not going to be able to continue development on this project after this semester, users will have to patch any bugs and address any maintenance issues themselves. If anyone does so, we will merge their patch into the source so that new users may benefit. To upgrade, users will have to visit the Github page and download the latest package.