# WikiMap

Alana Killeen, Dylan Horkin, Gordon Liu, Kimberly Koenig, Liem Dinh, Michael Amorozo, Robert Chu, Steven Kwan

**System Design Specification and Planning Document**

Draft 2.1

3/10/2011

CSE 403 - CSRocks Inc.

Revisions

| Version | Primary Author(s) | Description of Version | Date Completed |
| --- | --- | --- | --- |
| 1.0 | Kimberly Koenig | Added information about the team organization, roles, and responsibilities. Also some extremely preliminary dates, milestones, and deadlines. | 01/27/2011 |
| 1.1 | Kimberly Koenig | Revised some role/responsibility data. Added UML diagram. | 01/29/2011 |
| 1.2 | Kimberly Koenig | Adding some design alternatives information. Revised group responsibility info. | 01/31/2011 |
| 1.3 | Kimberly Koenig | Added new database schema (which will still need to be updated). Added test information. Added updated implementation view. Added updated UI UML Diagram. Added documentation plan. | 02/1/2011 |
| 1.4 | Kimberly Koenig | Added updated database Schema. Added design alternative information. Added elaboration to the “design” area in paragraph form. Added several new diagrams. | 02/2/2011 |
| 1.5 | Kimberly Koenig | Added updated UML diagrams to all relevant sections. Added Customer View. | 02/2/2011 |
| 1.6. | Kimberly Koenig | Added milestones and updated risks section. | 02/3/2011 |
| 1.7 | Kimberly Koenig | UML Sequence diagrams added. Other UML Diagrams updated. Design changes reflected throughout (C# is now Java). | 02/3/2011 |
| 1.8 | Kimberly Koenig | Fixed some UML sequence diagrams. Deleted a design alternative that is no longer relevant. Reorganized some sections and removed all remaining references to SOAP. | 02/4/2011 |
| 1.9 | Kimberly Koenig | Updating the SDS for Beta. | 02/19/2011 |
| 2.0 | Kimberly Koenig | Updated SDS per team feedback on draft changes. Most changes made to Database Retriever section and some diagrams were changed. Changed diagrams are noted by the text (updated) after their captions. | 02/20/2011 |
| 2.1 | Kimberly Koenig | Updating for final release. Adding design pattern data. | 03/10/2011 |

System Architecture

# Introduction

WikiMap is an interactive, web browser-based visualization of Wikipedia articles and their relationships: it allows users to navigate a high-level map of Wikipedia, understand how different articles are related, and preview (or fully view) each article’s content while remaining immersed in the WikiMap experience.

Our target audience consists of knowledge and Wikipedia enthusiasts: individuals who regularly browse Wikipedia for casual fun, research, homework, and more.

This purpose of this document is to explain some of the system architecture, design, documentation, and test decisions that have been made with regard to the WikiMap project, particularly with regard to how they support our main project goals of:

1. Visualizing article relatedness by identifying and representing the strongest article relationships
2. Providing a fun, easy-to-use, simple, and visually appealing way to explore Wikipedia at a high level
3. Enabling users to interact with Wikipedia as a map of interconnected articles
4. Simplifying the process of viewing many related articles’ content and images
5. Providing an extensible software product

Certain features are outside of our scope, including:

1. Create a new Wikipedia search algorithm
2. Exhaustively calculate, map, or otherwise identify all article relationships
3. Host unnecessary Wikipedia content beyond what is required by our project
4. Support or provide editing, login, or other functionality for Wikipedia articles

# System Architecture

* 1. **System Architecture Overview**

The WikiMap implementation is conceptually divided into several layers:

1. **Front End (PHP):** Contains the website/UI, an HTML parser, and the Database Retriever (part of the Database API).
2. **Back End (Java):** Contains the Database Updater (part of the Database API), a caching module, a logic layer, and auxiliary logic to assist the logic layer.
3. **Database (SQL)**  
     
   See the next page for a high-level architecture view of WikiMap.

# 

Figure 1: High Level System Architecture Diagram (updated)

* 1. **Customer View**

The customer view of our system is as follows:

1. **Design View**
   1. **Design Patterns**
      1. Article Vector Singleton  
           
         ArticleVectorSingleton ensures that there is only one instance of an ArticleVector object at any given time by constructing an ArticleVector if its internal pointer to an ArticleVector is null; otherwise, it returns the existing ArticleVector.  
           
         **See: logic/ArticleVectorSingleton.java**
      2. Article Vector Encapsulation  
           
         ArticleVector's fields and information are hidden by getters and setters for String articleName, List<String> links, and boolean redirect.  
           
         **See: logic/ArticleVector.java**
      3. Unit Test Subclassing  
           
         Many of our JUnit tests need to be able to connect to the same test database and extend JUnit's test case class. Instead of repeating this code in a redundant manner or having each JUnit test case inherit from the Test Case superclass, we created an abstract superclass for all of our JUnit tests called WikiMapTestCase, which extends the JUnit Test Case and contains constants for the database connection variables and common setUp() and tearDown() methods.  
           
         **See: test/JUnitTests/WikiMapTestCase.java**

* 1. **Front-end**

As a matter of style (encapsulation and information hiding) we made the decision that the UI components of the front-end do not directly contact the SQL Database. Instead, the Front-End call general Database API methods and receive data in a general format.

# UI Design

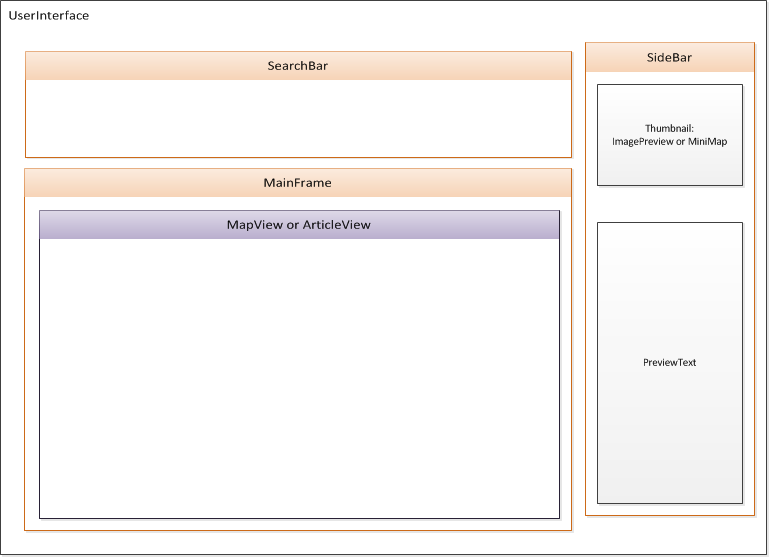


Figure 2: High Level Module view of the UI

The above diagram is a modular way of looking at the UI. Each module is placed as it would be on the actual webpage. The user will see the appropriate components of each module.   
  
Some modules have toggle-able visibilities: when the user is browsing the map, the MapView (with an interactive map and nodes) is visible in the MainFrame. Similarly, an ImagePreview for the main article node (the article node last clicked on) is displayed in the Thumbnail Area.  
  
When the user is browsing an article, the ArticleView is visible in the MainFrame, and an interactive Mini-Map is displayed in the Thumbnail area instead. Clicking on Mini-Map nodes results in articles being displayed in the article view.  
  
In both the MapView and the ArticleView, mousing over article nodes (or article links, in the case of ArticleView) updates the PreviewText area to display a summary of the relevant Wikipedia article.

The preview text area’s text should not be of a fixed length. The length of the preview text should take into account the height of the image in the thumbnail area. Currently we do not handle this as elegantly as we would like – we assume a worst case scenario image height of 300px, as this is the max-height specified for the thumbnail image. Ultimately, we would like to be able to ask the image for its height, determine the user’s resolution, and display enough text based on these attributes.   
  
On the next page is a UML class diagram which details the fields and methods of each module.

# 

Figure 3: UI UML Class Diagram

# As depicted above, the UI will be conceptually divided into several different modules/classes. However, it is worth noting that because the UI will be primarily in HTML, CSS, and JavaScript, this modularity is mostly conceptual as JavaScript does not have a notion of classes, unlike many object-oriented languages. Thus, these diagrams help to conceptually group the different JavaScript functions and regions of the UI, even if the actual implementation will not have a notion of different objects on a page.

# From an Object-Oriented standpoint, here is a description of some of the UI Modules which warrant elaboration:

|  |  |
| --- | --- |
| **Object** | **Description** |
| *Interactive Object* | This interface provides methods for handling user input to the Map and Node objects. |
| *UserInterface* | This is an overall wrapper for the entire UI. It contains one SearchBar, MainFrame, and SideBar. |
| *SearchBar* | Contains a search bar and all functionality for the user to search for an article and have it displayed in the map. A stretch feature is that we might supply some sort of predictive autocomplete. |
| *MainFrame* | The main interactive area on the page, the MainFrame contains a MapView or an ArticleView, whose visibilities will depend on whether the user is in “Full Map Mode” or “Full Article Mode”. |
| *SideBar* | Contains a Thumbnail and PreviewText. |
| *Thumbnail* | Contains an ImagePreview and Mini-Map. |
| *ImagePreview* | Displays a Wikipedia article image for the most recent article node clicked on. This is only visible when the user is in “Full Map Mode” and is viewing the Map in the MainFrame. |
| *MapPreview* | Displays a small, interactive version of the map for the current map orientation, which depends on the current article being viewed. This is only visible when the user is in “Full Article Mode” and is viewing an article in the MainFrame. |

* + 1. **UI Map Implementation**
       1. **Pulling data directly from Wikipedia using client connection**

During the implementation process it has become clear that it is inefficient to attempt to parse all article summary data from the Wikipedia database dump. Similarly, constructing image URLs in the back-end can be fairly difficult: in the database dump, only image names and file extensions (such as *imagename.jpg*) are represented. We would have to determine the URL of the image’s storage location and use an MD5 algorithm to adjust the URL, as this is what Wikipedia does.

Instead, we will utilize the client’s connection (via the UI) to initially download article summary and article image information which we do not currently have. When the user clicks or hovers on an article node, a request will be sent to the back-end asking for the summary and image information.

Any information which is not currently in the back-end will be downloaded via the client’s connection (to avoid over-querying Wikipedia from our back-end). The article summary will be parsed from HTML (instead of the back-end parsing it from XML, which can be time-intensive) and then sent to the back-end to be stored in our database. Similarly, the article image URL (if an image exists) will be scraped from Wikipedia and sent to be stored in our back-end. In doing so, we ensure that we do not store any more data than is necessary, only grabbing and storing data for nodes that some user has attempted to access in the past.

All map data is streamed in the background (per 3.2.2.2.) to reduce the latency when a user hovers over an article. If a user hovers over an article node before the data is fully streamed, there may be a slight delay, but the response time should be much faster than on-demand loading of the nodes only when the user hovers.

* + - 1. **Streaming nodes in the background**

When the user first searches for an article, the front-end UI queries the Database Retriever for the relationship data relating to that article node. Not all relationship data out to infinity should be returned at once, as doing so would be inefficient. Instead, the initial query to the Database Retriever will specify a constant value indicating how many degrees of separation of relationships should initially be returned.  
  
This initial number will be fairly small (4-6 degrees of separation away from the main node) and should return approximately the number of nodes that are able to be displayed onscreen at a given time. The goal of this is to provide a quick way to display the visible part of the graph and provide opportunity for background streaming.  
  
After the initial graph is built, another query will be sent to get further degrees of separation away from the initial node. This data is streamed in the background and these nodes are progressively added to the graph, but with visibility turned off since they are beyond the border of the screen.   
  
Wikipedia allows for batches of 50 articles to be requested at one time. After streaming in a few initial nodes, batches of 50 surrounding/related nodes will be requested at a time from Wikipedia.

* + - 1. **Zoom level and relational strength (cut feature)**

Zoom level and relational strength are tied to each other. When a user is zoomed out all the way, a greater breadth of nodes becomes visible – more degrees of separation away from the main node. But fewer nodes are actually displayed as being related to a given node onscreen – only the very strongest relationships are displayed at this “bird’s eye view”.

If a user zooms in, fewer degrees of separation are shown relative to the main node. However, more nodes are displayed as being related to a given node onscreen – stronger and weaker article relationships (greater granularity) is displayed at this greater magnification.

* + - 1. **Zooming in (cut feature)**

When related article data is first returned for a particular article node, nodes at the default zoom level are automatically made visible to the user as part of the initial graph that is built. Nodes all zoom levels greater than the default zoom level are also returned and added to the graph, but these are not visible.

If the user zooms in, these previously hidden nodes will be displayed. Zoom level and visibility are linked to each other in this manner.

* + - 1. **Zooming out (cut feature)**

Zooming out displays more nodes around the edge of the viewing area, while reducing the density or granularity of the nodes that are displayed onscreen (fewer nodes are shown attached to each node onscreen, but more degrees of separation are visible).  
  
Similar to dragging the map, if the user zooms out, previously invisible nodes around the edge of the screen become visible. Weaker article relationships will disappear, and only the strongest article relationships will be displayed onscreen. If nodes have not been loaded for regions around the boundary of the screen when zoomed out, they will appear as they become available (this should be a quick process).

* + - 1. **Dragging the map**

Nodes beyond the visible area of the map screen are streamed in the background after the initial graph is loaded. If the user clicks and drags away from the default map region that is displayed, previously invisible nodes will become visible as the user drags beyond the initial map area. If nodes have not yet been loaded for a particular map region, they will appear as they become available (this should be a quick process).

We will determine a reasonable maximum for how many peripheral nodes are ever loaded, as it is unlikely that most users will drag past a particular point. For users that are interested in exploring beyond this maximum boundary, clicking on a node at the edge of the map will make that the new “main node” and run a new query, thus streaming in new relationships around that node and allowing the user to continue browsing.

* + 1. **HTML Parser**

As noted in 3.1.2.1., we will be using the front-end/client connection to pull some data directly from Wikipedia, as this is more time and space efficient than parsing the XML and Image data in the back-end. Thus, there will be an HTML parser in the front-end to extract any HTML formatting elements and other tags which we do not want to be included with an article image or an article summary.

* 1. **APIs/Communication Layers**

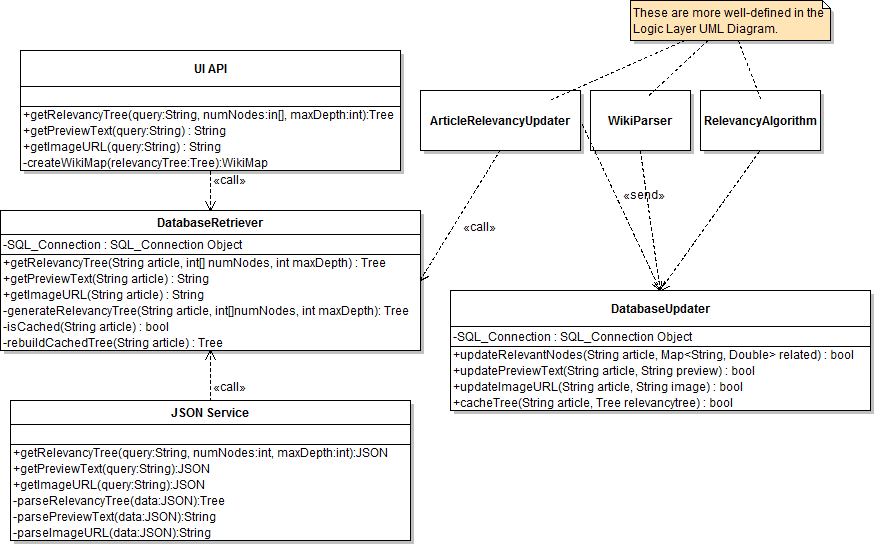


Figure 4: Front- and Back-End API UMLClass Diagram (updated)

* + 1. **UI API (PHP)**

This API is responsible for generating the front-end interface, including the graph, in a general-enough manner that various UI implementations are able to draw it, whether that UI is a website, mobile device, or something else. Thus, the purpose of providing a UI API is so that the client-facing UI may be considered to be modular. It also removes the need for the client’s system to handle graph generation, instead placing this responsibility on our server.   
  
This API will receive requests and queries from the UI when the user searches for an article or performs another action that is equivalent to a search, such as clicking on an article node, which necessitates retrieving updated relational information about that node. These queries are sent to the Communication API.  
  
Relational data will be received by this API in the form of a tree traversal, which is appropriate since WikiMap is an acyclic graph. This tree traversal can be built into the graph which can be displayed by the UI.  
  
This API does not communicate directly with the SQL database. It sends abstract requests for data to the Database Retriever. Article relationships are received back as tree traversals, article preview text is a String, and Image URLs are Strings.

* + 1. **Database API**

The Database API is actually three distinct components (Database Retriever, Database Updater, and JSON Service) which receive requests to read or update the database for the following:

1. Related nodes out to a particular degree of separation from a particular article
2. Article preview text for a particular article
3. Article image URL for a particular article

The Database API is responsible for querying *and* updating the SQL database on behalf of the front end and the back end, regardless of the front or back-end implementations.

It provides a layer of abstraction between the front end, the back end, and the SQL database. The Retrieve/Update functionalities are divided into two parts which “sandwich” the SQL database between the front-end and the back-end.

* + - 1. **JSON Service**

In the case of a mobile device extension, this service is what would provide an API for the mobile device. It is unlikely that we will write any JSON methods for the purposes of the 10-week implementation phase, but it is worth noting that such an option could be made available in future implementations.

* + - 1. **Database Retriever (PHP)**
         1. **Overview**This component is responsible for querying the database, retrieving data, and formatting article relationship data. Relationship data received back from the database needs to be formatted to be more implementation agnostic, as opposed to sending the raw results of the query back to the front-end.
         2. **Caching**Once a tree has been built, it should be cached using the Caching Module (3.2.2.4).  
            After we have implemented relevancy relationship tree caching (see 3.2.2.4.), the Database Retriever will first check the tree cache tables in the database to see if the given article query correlates to a relevancy tree that was recently built and stored into the database.   
              
            If no tree is found, then the Database Retriever will request data from the Article Relationship table in the database.
         3. **Tree Traversal Generation**  
              
            The Database Retriever is responsible for representing article relationships as a tree traversal, which can be re-interpreted and built into an (acyclic) graph by the UI.  
              
            The format of the data passed to the front end is as a tree traversal. The format requirements are that:

1. Trees are serialized as a breadth-first-traversal of the tree.
2. New levels are denoted by "//"
3. Nodes are separated by "|",
4. "||" signifies the next node comes from the next parent

An example tree representation would be:   
  
bill gates//billion|business magnate//Jack Billion|Long and short scales||Bill Gates|petroleum

* + - * 1. **Specifying Tree Depth / Number of Children**The methods getRelevancyTree or generateRelevancyTree both have an integer array parameter followed by an int which specify the number of children at each depth level of the tree, and the number of levels of depth, respectively.  
             
           For example, calling either method with the parameters (articleName, [6, 2], 2) will return a tree traversal of the article relationships for articleName out to 2 degrees of separation away. At the first degree of separation (articleName’s children), 6 nodes will be displayed for that main node. At the second degree of separation (those children’s children), only two nodes will be displayed for each.  
             
           It is also possible to deprecate the command in the instance that many additional degrees of separation have the same number of children. For example, calling either method with the parameters (articleName, [6, 5, 4], 10) will return a tree with 10 degrees of separation from the main article node, articleName. The 1st degree of separation will have 6 children per node (in this case, the main node), the 2nd degree of separation will have 5 children per node, and the 3rd through 10th degrees of separation will have 4 children per node.   
             
           This is essentially a deprecated representation of the full method call with the parameters (articleName, [6, 5, 4, 4, 4, 4, 4, 4, 4, 4], 10) which is unnecessarily long.
      1. **Database Updater (Java)**

The Database Updater is called by the back-end Logic Layer, and is responsible for updating all aspects of the database (relationships, Image URLs, article preview text, or other data as appropriate).

Note: We have stipulated that no front-end component should contact the Database Updater directly for security reasons, thus the caching module (3.2.2.4) will be the intermediary between the front-end and the Database Updater should the front-end or UI need to update the database.

* + - 1. **Caching Module**

The caching module provides several functions:

1. **Tree cache requests:** Receive requests from the Database Retriever to store tree traversals in the database for easy retrieval, so that article relevancy trees do not need to be re-built for relevancy trees which were accessed recently
2. **Flush cached trees:** Periodically flush old relevancy trees which have not been recently requested. How long we keep trees has yet to be determined.
3. **Front-end storage requests:** When the front-end retrieves an image URL or article summary test, the caching module fields those requests, ascertains that the requests are valid (and non-malicious), and passes them to the database updater.
   1. **Logic Layer (Java)**

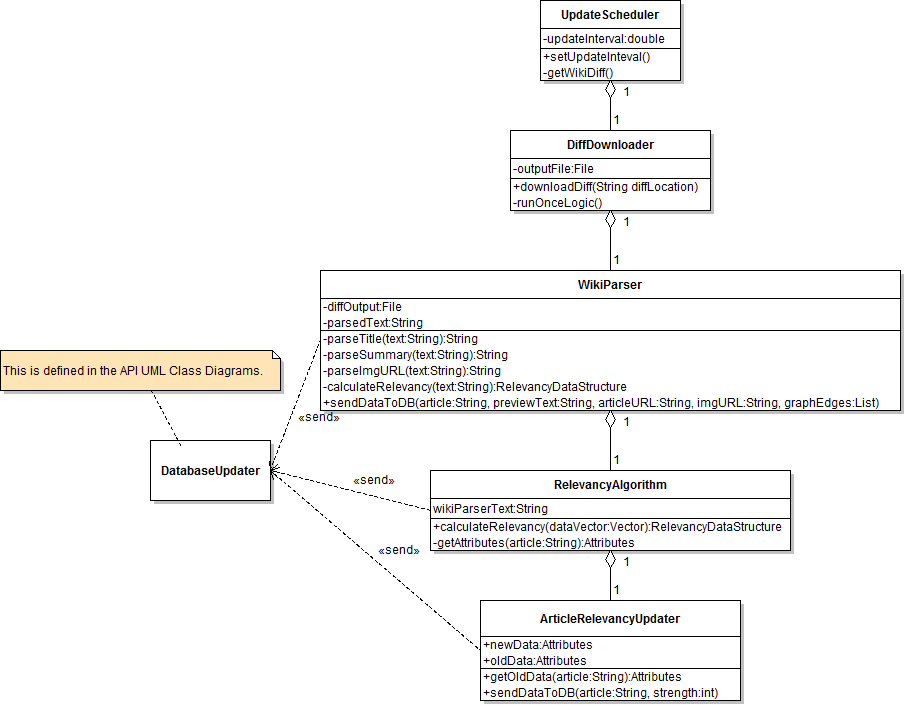
****

Figure 5: Logic Layer UML Diagram

The Logic Layer handles the major downloads of data from Wikipedia, parses that data, and determines article relationships and their strengths.

Updates are kicked off manually or on an automatic schedule by the Update Scheduler. This triggers the Diff Downloader to retrieve the latest Wikipedia Diff file.  
  
Raw Wikipedia data is passed off to the Wikipedia Parser. After parsing, the parser sends Image URL information and Article Summary information directly to the Database Updater (in the Database API).  
  
Parsed data is then sent onward to the Relevancy Algorithm, which determines which articles are most strongly related to other articles.   
  
This data can also be sent to the ArticleRelevancyUpdater, which determines what, of this data, represents a changed relationship based upon the current data in the SQL database. Related article data is then pushed to the database via the Database API.

The “Attributes” referenced in the RelevancyAlgorithm and ArticleRelevancyUpdater are vague at this time: these are attributes of the Wikipedia data that are being used to rank the strength of article relationships. However, since the exact behavior of the algorithm is still being determined, we represent this data as a generic “Attribute”.

1. **Process View *– UML sequence diagrams***
   1. **Use Case 1: Searching for an article**

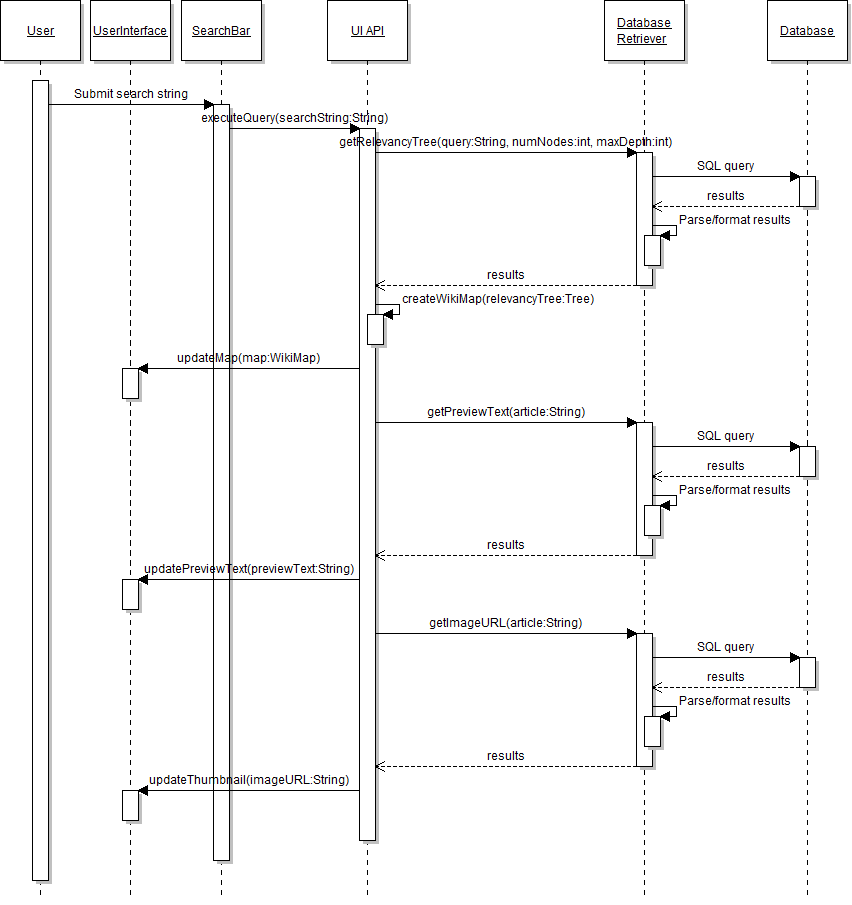


Figure 6: UML Sequence - Searching for an Article

* 1. **Use Case 2: Interacting with the map**
     1. **Single-click on a node**

Note: this addresses the case where insufficient relation information is known about a node and a new query must be run when the user clicks on it. In some cases, single-clicking on a node will not require executing a new query – it will just re-center the map.

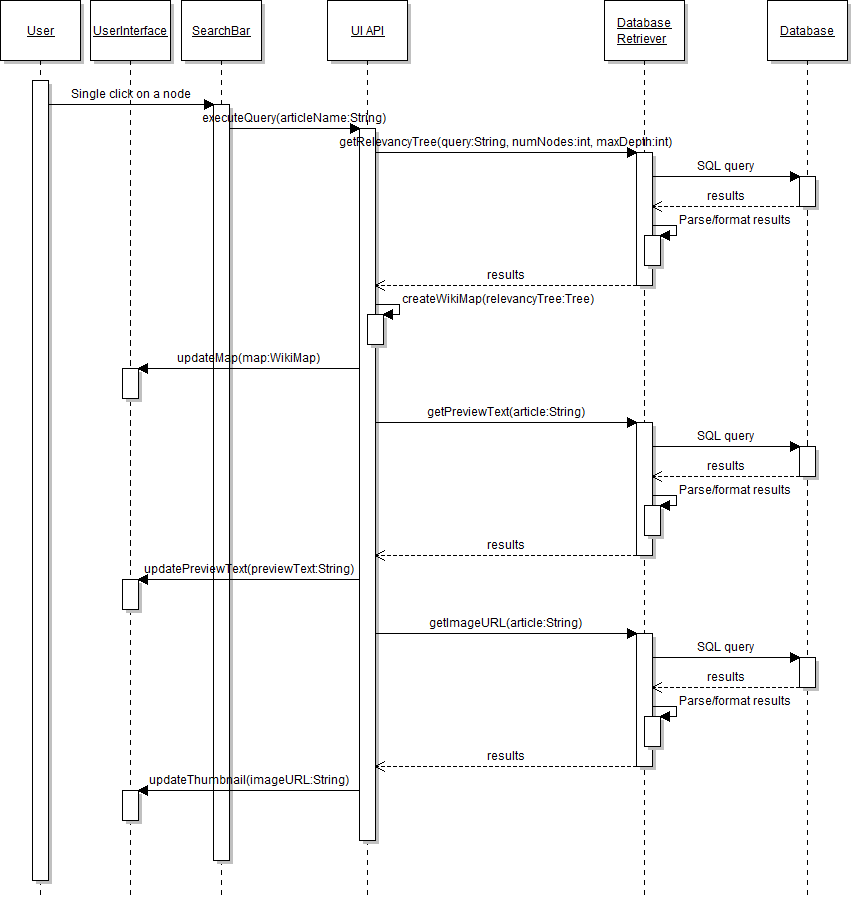
****

Figure 7: UML Sequence - Clicking on a node

* + 1. **Double-click on a node**

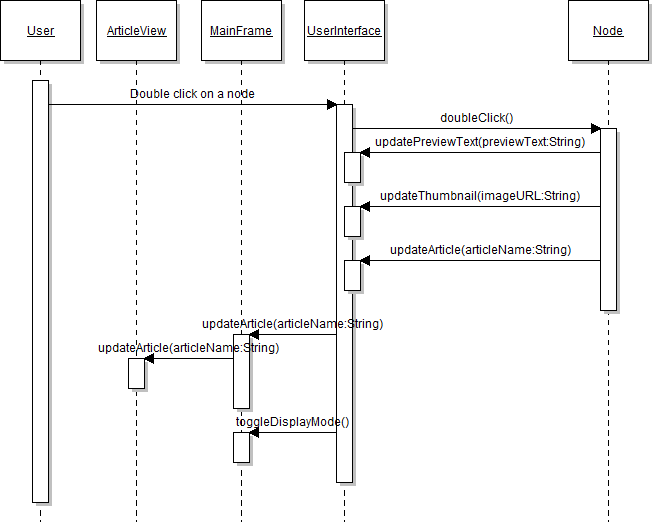
****

Figure 8: UML Sequence - Double Clicking a node

* + 1. **Drag the map**

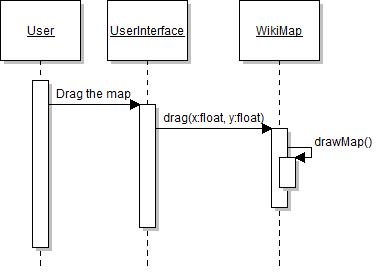
****

Figure 9: UML Sequence - Dragging the Map

* + 1. **Zoom in/out on a node**

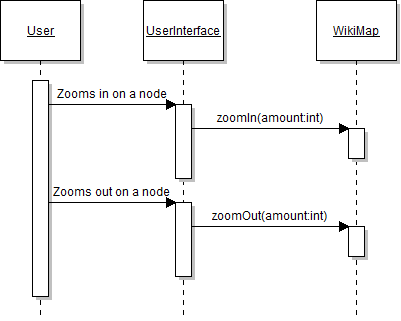
****

Figure 10: UML Sequence - Zooming in/out on the map

1. **Database Schema**
   1. **Overview**

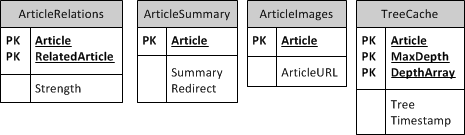


Figure 11: Database Schema Diagram (updated)

The WikiMap SQL database will be used to store and retrieve:

* 1. Related articles for a particular main article, out to as many degrees of separation as desired
  2. Article relationship strength
  3. Article URLs
  4. Image URLs
  5. Preview/summary text for each article
  6. Article relevancy tree caching

Summaries and Image/Article URLs will each be stored in separate tables, ArticleSummary and ArticleImages, respectively.

The table we will likely query most frequently for building the Wikipedia graph is the ArticleRelations table. The intent of this table is to represent edges between nodes in the graph. An example of what the ArticleRelations table might look like:

|  |  |  |
| --- | --- | --- |
| Article | RelatedArticle | Strength |
| A | B | 3 |
| A | C | 2 |
| A | D | 1 |
| B | A | 3 |
| B | E | 2 |
| B | F | 1 |

“Strength” is a rating determined by the relevancy algorithm – it indicates how strong a particular edgewise relationship is, and it should be reflexive (e.g. the edge between A and B has the same strength rating as the edge between B and A). These strengths will be used to determine which nodes are displayed at a given zoom level.

* 1. **Database CREATE TABLE Commands**

**CREATE TABLE `articleimages` (**

**`Article` varchar(100) NOT NULL DEFAULT '',**

**`ArticleURL` varchar(500) DEFAULT NULL,**

**PRIMARY KEY (`Article`)**

**) ENGINE=InnoDB DEFAULT CHARSET=latin1$$**

**delimiter $$**

**CREATE TABLE `articlerelations` (**

**`Article` varchar(100) NOT NULL,**

**`RelatedArticle` varchar(300) NOT NULL,**

**`STRENGTH` int(11) DEFAULT NULL,**

**PRIMARY KEY (`Article`,`RelatedArticle`)**

**) ENGINE=InnoDB DEFAULT CHARSET=latin1$$**

**delimiter $$**

**CREATE TABLE `articlesummary` (**

**`Article` varchar(100) NOT NULL DEFAULT '',**

**`Summary` varchar(10000) DEFAULT NULL,**

**`Redirect` binary(1) DEFAULT NULL,**

**PRIMARY KEY (`Article`)**

**) ENGINE=InnoDB DEFAULT CHARSET=latin1$$**

**delimiter $$**

**CREATE TABLE `articlevector` (**

**`Article` varchar(100) NOT NULL DEFAULT '',**

**`Links` varchar(30000) DEFAULT NULL,**

**`Redirect` binary(1) DEFAULT NULL,**

**PRIMARY KEY (`Article`)**

**) ENGINE=InnoDB DEFAULT CHARSET=latin1$$**

**delimiter $$**

**CREATE TABLE `treecache` (**

**`Article` varchar(100) NOT NULL DEFAULT '',**

**`Tree` varchar(10000) DEFAULT NULL,**

**PRIMARY KEY (`Article`)**

**) ENGINE=InnoDB DEFAULT CHARSET=latin1$$**

* 1. **Database Update Process**

Initial attempts to parse the Wikipedia dump file revealed that the database update process was taking a long time due to the data being unordered, which causes large shifts in the B-trees of the database, as well as due to network latency (we are updating the database over a network).   
  
We are taking the following steps to mitigate/triage this process:

1. **Update logfile:** Logs each batch of 1000 articles that are successfully added to the database by writing the corresponding line number successfully reached in the Wikipedia dump XML file.
2. **Restart/resume logic:** Parser/updater can be told to read from a logfile, starting from the line number last successfully parsed and updated. This is especially useful in the case of a crash during updating.
3. **Halt logic:** Parser/updater be told to cease at a particular line number in the Wikipedia dump XML file. This combined with the previous functionality enable us to parallelize the update process in “chunks” across multiple machines if necessary.
4. **Batch update logic:** Parser/updater generates batches of inserts and stores them to a file before sending these to the database instead of sending one INSERT statement at a time over the network.
5. **Design Alternatives and Assumptions**
   1. **Assumptions**

Some assumptions we are making are:

* Users have some familiarity with search engines
* Users are looking for relationships between articles
* Users are interested in the strongest relationships between articles
* Users have enough basic familiarity with Wikipedia to search for articles of interest
* Users are less likely to read a full article than they are to want an article summary   
  1. **Design Alternatives**
     1. **Location of Wikipedia Full Article View**

Two options exist for how we display a Wikipedia article. We are going with #1, but may re-evaluate based upon usability testing.

***- Option #1: Switchable WikiMap and Article Views***

User can switch between the WikiMap and Wikipedia article by clicking on the “thumbnail” area or double clicking on a node (when in Wikimap View)

**Pros:**

1. Equal screen real estate is given to the current task (browsing Wikipedia vs. browsing the map)
2. Provide value add for Wikipedia articles: being able to see summary text when you hover over a Wikipedia link

**Cons:**

1. May not be immediately obvious to users
2. More technically complex

***- Option #2: Wikipedia Article is always in SideBar***

Whenever the user displays a full Wikipedia article, it’s displayed in the sidebar on the right-side of the screen.

**Pros:**

1. Map and article are displayed side by side

**Cons:**

1. Limited screen space for Wikipedia article
2. Screen may feel cramped on smaller monitors
   * 1. **Sidebar “Thumbnail” Image**

**Overview:** Two options exist for which image should be displayed in the “thumbnail” area of the sidebar when the user is in the “Full Map View”.  
  
**Current choice:** We are going with #1 as we believe that it holds more value for our customers.

***- Option #1:***  ***Article Image Thumbnail***When a user clicks on a node in the map, the image displayed in the thumbnail area is the image associated with that article.  
  
**Pros:**

1. Users can see the image associated with an article
2. The image can be scraped from the Wikipedia API by the client connection

**Cons:**

1. If no image is associated with an article, we need to display something generic in this location.

***- Option #2: Article Screenshot Thumbnail***

When the user clicks on a node in the map, the image displayed in the thumbnail area is a small screenshot of the entire article.  
  
**Pros:**

1. Users can see a high-level view of the Wikipedia article.
2. We can display an image for every article.

**Cons:**

1. We have to provide some kind of functionality for screenshotting an article and downsizing an image.
2. Not as useful to the user as an article image.
   * 1. **Database Table Structure**

**Overview:** Two options exist for how to represent Article Relationships in the SQLdatabase.  
  
**Current choice:** For now we are going with #1, and are viewing #2 as a potential optimization to use instead of (or in concert with) #1.  
***- Option #1: Represent Article Relationships as Edgewise Pairs***  
The proposed representation of article relationships in the ArticleRelations table is as edges, with Articles being paired with RelatedArticles.This results in an entry for every edgewise pair of related articles.   
  
**Pros:**

1. Simple and fast for discovering a few degrees of separation away from a particular node
2. Minimized database size – there are as many tuples in the ArticleRelations table as there are edges in the graph

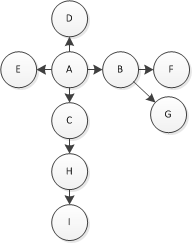
**Cons:**

1. Discovering farther than a few degrees of separation will increase query times
2. Cannot directly specify or query a degree of separation to return in a query. (Querying for *n* degrees of separation away from a particular article requires finding every tuple at *n-1* degrees of separation)

See the database schema [**overview**](#SchemaOverview) for an example of what the ArticleRelations table looks like with this design choice.

***- Option #2: Represent Article Relationships as a “Flattened Tree”***    
  
An alternative representation is to include a distance and parent node with the related article tuples as a means of “flattening” the table and minimizing query time. An example graph and its associated table would be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Article | RelatedArticle | Parent | Distance | Strength |
| A | B | A | 1 | 3 |
| A | D | A | 1 | 3 |
| A | C | A | 1 | 3 |
| A | E | A | 1 | 3 |
| A | F | B | 2 | 2 |
| A | G | B | 2 | 2 |
| A | H | C | 2 | 2 |
| A | I | H | 3 | 1 |
| B | A | B | 1 | 3 |
| B | F | B | 1 | 3 |
| B | G | B | 1 | 3 |
| B | E | A | 2 | 2 |



This table would traverse all of the “main node” possibilities and save the distances for related nodes. In doing so, more up-front calculation is required when building the database, but this is a one-time cost. From thereon out, it is possible to query for all nodes that are distance *x* from a given node Y.  
  
**Pros:**

1. Minimizes query time – can ask the database for all articles with distance *n* from a given node without having to explore all n-1 tuples beforehand
2. Takes advantage of building the database being a one-time (or occasional) cost and optimizes for database queries

**Cons:**

1. Significantly increases database size
2. Increases up-front cost to construct the database

Development Plan

# Team Structure

* 1. **General Organization & Responsibilities**The team is organized into functional groups whose focus is a particular piece of the product’s technology. Each team has a “tech lead”, and there is an overall PM, or “program manager”.

Individuals were assigned to groups based upon their expressed interests early on in the course, taking into account their other commitments as well as what they desired to get out of the class and project.  
  
Each person was assigned to at least two groups. Everyone who had test experience was placed on the test team, which resulted in a few individuals having three roles.   
  
These group assignments are meant to be fluid, meaning that individuals may be moved away from or to a different functional group depending on how many people/resources are required at a particular point in the development cycle.

* 1. **Responsibilities by Role**

|  |  |
| --- | --- |
| *Role* | *Responsibilities* |
| Tech Lead | * “Go-to” person for team members’ (technical) questions * Delegate tasks within group * Provide status reports to the PM * Provides vision and direction for their team * Brings blocking issues and other concerns to the attention of the PM or course staff, as necessary * Track team progress against schedule |
| Team Member | * Become a subject-matter expert * Work with other team members to make difficult technical decisions * Communicate with Tech Lead and PM with regard to status and workload appropriateness * Bring blocking issues, workload issues, and other concerns to the attention of the tech lead and/or PM * Track individual progress against overall schedule |
| Program Manager | * Provide decision-making authority when needed * Facilitate communication between group members, and disseminate information to all individuals in group * Delegate and redistribute workloads * Track all members’ progress against overall schedule * Ensure product design is consistent with SRS and SDS * Update SRS and SDS as necessary * Maintain a high-level view of the product design |

* 1. **Responsibilities by Functional Team**

|  |  |
| --- | --- |
| *Team* | *Responsibilities* |
| Front-End | * UI design, layout, and rendering * Graph algorithms * Handling user interactions * Website layout and content |
| Communication/APIs | * Designing and implementing communication layers between front-end, back-end, and data sources * Ensuring that all intra-program modules talk to each other and have adequate information hiding * Determining the optimal methods of abstraction and encapsulation between different program components * Providing abstraction and sound coding practices amongst all program modules |
| Database | * Design and maintenance of a database schema * Optimization of the database where possible * Working closely with logic team and communications team to determine input/output requirements for the database, logic layers, and APIs |
| Logic/Algorithms | * Designing, implementing, and tweaking any necessary algorithms, including the relevancy algorithm * Designing, implementing, and managing components of the logic layer, including the update scheduler, diff downloader, Wikipedia parser, and other components in/alongside the program logic |
| Test | * Setup/maintenance of bug tracking system * Setup/maintenance of version control syste * Automated and manual testing, both black-box and white-box, of all code as changes occur * Ensuring adherence to use cases and program security * Compilation scripts (as necessary) |

* 1. **Teamwork Ground-rules**

1. **Timely Communication:** In order to ensure that all emails are answered in a timely manner, group members are expected to check and respond to WikiMap-related emails at least once daily.
2. **Mutual Respect:** Remember that all ideas came from someone. Be respectful of others’ ideas. Be sure to acknowledge the pros and cons of each idea and use constructive criticism.
3. **Report Absences:** Provide team leads and the PM with advance notification if you will be unable to make a meeting. If you will be missing multiple class sessions, also notify the PM and team leads of your status.
4. **Workload Responsibility:** It is each group member’s responsibility to let their lead and PM know if they do not think they will be able to complete their work on time. Similarly, if an individual does not feel they have enough work, it is their responsibility to ask for a greater workload beyond what they’ve already been given.
   1. **Weekly Meetings**  
        
      The team will communicate during weekly meetings on **Tuesdays and Thursdays at 3:30 pm**. These meetings are in addition to the meetings during section on Thursdays and are held in the **5th floor Allen Building breakout area**, unless otherwise noted.  
        
      An additional “as-needed” meeting time is provided on Wednesdays at **2:30 pm** for those who want an extra meeting time or cannot attend the other meeting(s). and are held in the 006 lab, unless otherwise noted.
   2. **Status Reports**Individual and functional group status reports will be presented verbally (and recorded in notes) during Thursday meetings.
   3. **Coding Sessions**Team coding sessions are being provided throughout the implementation process, particularly leading up to major deadlines. Team members are encouraged to all attend and work together in the same room, and food and drink is provided.
   4. **Other Communication**

|  |  |
| --- | --- |
| **Mode of Communication** | **Link** |
| Team Email | [cse-403-wikimap@googlegroups.com](mailto:cse-403-wikimap@googlegroups.com) |
| UI (Front-end) Email | [cse-403-wikimap-ui@googlegroups.com](mailto:cse-403-wikimap-ui@googlegroups.com) |
| APIs (Communication Layer) Email | [cse-403-wikimap-api@googlegroups.com](mailto:cse-403-wikimap-api@googlegroups.com) |
| Algorithms (Logic Layer) Email | [cse-403-wikimap-algos@googlegroups.com](mailto:cse-403-wikimap-algos@googlegroups.com) |
| Test Email | [Cse-403-wikimap-test@googlegroups.com](mailto:Cse-403-wikimap-test@googlegroups.com) |
| Website/Wiki | <http://code.google.com/p/cse403-wi11-wikimap/> |
| Group Calendar | [Group Milestone Calendar](http://www.google.com/calendar/embed?src=g3cbepv7fh476l7btp00uhhnd4%40group.calendar.google.com&ctz=America/Los_Angeles) |

# Project Schedule and Features

# Project Schedule

\* Note: the “Resources” named are the group leads responsible for delegating tasks, and thus is used to indicate group ownership. The Resource field is not necessarily the individual directly working on a feature at this time.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Task/Milestone*** | ***Estimated Effort*** | ***Start Date*** | ***Due Date*** | ***Status*** | ***Team*** | ***Resources\**** | ***Details*** |
| *Setup Mercurial Repositories* | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *ALL* | *Steven* |  |
| *Mercurial Documentation* | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *Test* | *Steven* |  |
| *Setup bug tracking* | *1 day* | *Jan 31* | *Feb 2* | ***Complete*** | *Test* | *Kimberly, Steven* |  |
| *Set up bug tracking labels/categories* | *1 day* | *Jan 31* | *Feb 2* | ***Complete*** | *Test* | *Steven, Kimberly* |  |
| *Assign all alpha tasks* | *2 days* | *Feb 2* | *Feb 4* | ***Complete*** | *PM* | *Kimberly* |  |
| *Test plan* | *2 days* | *Jan 31* | *Feb 4* | ***Complete*** | *Test* | *Test team* |  |
| *Set up web server* | *1 day* | *Jan 31* | *Feb 4* | ***Complete*** | *UI* | *Liem* |  |
| *Set up barebones website* | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *UI* | *Liem* | Contains modular regions and search box |
| *Set up web URL to access site* | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *UI* | *Liem* | Must be a public URL |
| *Set up barebones front-end API (PHP)* | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *UI* | *Liem* | Must talk to Communications API and/or back-end |
| Set up front-end Communication API (PHP) | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *API* | *Dylan* |  |
| Set up back-end Database API | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *API* | *Dylan* |  |
| Basic communication between DB API and DB | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *API* | *Dylan* | 1. DB Updater updates DB 2. DB Receiver queries DB |
| Basic communication between DB API and Logic Layer | *1 day* | *Feb 2* | *Feb 4* | ***Complete*** | *API* | *Dylan* | 1. DB Updater receives requests from Logic Layer 2. DB Receiver receives requests from Logic Layer |
| Basic communication between front-end API and DB API |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Dylan* | 1. Front-end sends requests to back-end 2. Back-end receives request from front-end 3. Back-end returns data to front-end |
| Set up initial DB tables |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Liem* |  |
| Set up basic communication between DB API and DB |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Liem* | Coordinate with communications team to do this |
| Registering with Wikipedia for batch queries |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Michael* | Need to be able to query the Wikipedia API in batch mode if necessary. |
| Set up update scheduler |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Michael* | 1. RunOnce logic to kick off an update 2. Communicates with diff downloader to start update |
| Set up diff downloader |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Michael* | 1. Receives request from update scheduler 2. Connects to Wikipedia and gets data 3. Writes data to file 4. Returns foo (junk data, generated internally) |
| Set up Wiki Parser |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Michael* | 1. Receives foo from diff downloader 2. Calls relevancy algorithm with parsed data 3. Calls DB Updater with parsed data (article summary and ImgURL) |
| Set up barebones Relevancy Algorithm |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Michael* | 1. Receive some pseudo data from Wiki Parser 2. Calls attribute grabber 3. Calls DB updater with pseudo data (manually generated edge data and strengths?) |
| Set up barebones Attribute Grabber |  | *Feb 2* | *Feb 4* | ***Complete*** |  | *Michael* | 1. Receives calls from Relevancy Algorithm 2. Calls the DB Retriever |
| *Alpha Release* |  |  | *Feb 7* | ***Complete*** |  |  |  |
| DB Alpha 1 |  | Feb 7 | Feb 10 | ***Complete*** |  | *Liem* | Optimizations and adjustments |
| UI Alpha 1 |  | Feb 7 | Feb 10 | ***Complete*** |  | *Liem* | Graph generation and UI interactivity: basic clicks, drags, and more functionality if time |
| Search queries Alpha 1 |  | Feb 7 | Feb 10 | ***Complete*** |  | *Robert/Michael* | Basic search functionality (searches that return one result) |
| Logic Alpha 1 |  | Feb 7 | Feb 10 | ***Complete*** |  | *Michael* | Basic algorithm functionality (begin ranking articles in a simple manner) |
| APIs Alpha |  | Feb 7 | Feb 10 | ***Complete*** |  | *Dylan* | Any necessary adjustments/changes to the APIs (TBD) |
| Testing and automation Alpha 1 |  | Feb 7 | Feb 10 | ***Complete*** |  | *Steven/Kimberly* | First pass on browser test automation, database test queries, and PHP Unit Tests (time permitting) |
| SRS/SDS/Documentation Rev 1 |  | Feb 7 | Feb 10 | ***Complete*** |  | *Kimberly* | Any necessary updates to SRS, SDS, and admin docs |
| *Code Review 1 – Bugfix 1* |  |  | *Feb 10* | ***Complete*** |  |  | Critical bugs from Alpha 1 should be triaged, resolved, or escalated before Alpha 2 |
| DB **Complete** |  | Feb 10 | Feb 15 | ***Complete*** |  | *Liem* | Little to no DB changes after this point |
| UI Alpha 2 |  | Feb 10 | Feb 15 | ***Complete*** |  | *Liem/Robert* | Adding more UI interactivity/ functionality |
| Search queries Alpha 2 |  | Feb 10 | Feb 15 | ***Complete*** |  | *Robert/Michael* | Adding more search functionality |
| Logic Alpha |  | Feb 10 | Feb 15 | ***Complete*** |  | *Michael* | Iterate on algorithm logic to have greater complexity and more accurate strength ratings |
| APIs Alpha 2 |  | Feb 10 | Feb 15 | ***Complete*** |  | *Dylan* |  |
| Testing and automation Alpha 2 |  | Feb 10 | Feb 15 | ***Complete*** |  | *Steven/Kimberly* | Second pass on test work items |
| *Code Review 2 – Bugfix 2* |  |  | *Feb 15* | ***Complete*** |  |  | Critical bugs from Alpha 2 should be triaged, resolved, or escalated before Alpha 3 |
| UI Alpha 3 |  | Feb 15 | Feb 17 | ***Complete*** |  | *Liem* |  |
| Search queries Alpha 3 |  | Feb 15 | Feb 17 | ***Complete*** |  | *Robert/Michael* |  |
| Logic Alpha 3 |  | Feb 15 | Feb 17 | ***Complete*** |  | *Michael* |  |
| APIs Alpha 3 |  | Feb 15 | Feb 17 | ***Complete*** |  | *Dylan* |  |
| Testing and automation Alpha 3 |  | Feb 15 | Feb 17 | ***Complete*** |  | *Steven/Kimberly* |  |
| SRS/SDS/Documentation Rev 2 |  | Feb 15 | Feb 17 | ***Complete*** |  | *Kimberly* |  |
| *Code Review 3 – Bugfix 3* |  |  | *Feb 17* | ***Complete*** |  |  | Critical bugs from Alpha 3 should be triaged, resolved, or escalated before Alpha 4. |
| UI Alpha 4 |  | Feb 17 | Feb 19 | ***Complete*** |  | *Liem* |  |
| Search queries Alpha 4 |  | Feb 17 | Feb 19 | ***Complete*** |  | *Robert/Michael* |  |
| Logic Alpha 4 |  | Feb 17 | Feb 19 | ***Complete*** |  | *Michael* |  |
| API Alpha 4 |  | Feb 17 | Feb 19 | ***Complete*** |  | *Dylan* |  |
| Testing and automation Alpha 4 |  | Feb 17 | Feb 19 | ***Complete*** |  | *Steven/Kimberly* |  |
| *Code Review 4 – Bugfix 4* |  |  | *Feb 19* | ***Complete*** |  |  |  |
| Bugfix all and buffer time |  | Feb 19 | Feb 21 | ***Complete*** |  | *All* |  |
| UI Alpha 4 |  | Feb 19 | Feb 21 | ***Complete*** |  | *Liem* |  |
| Search queries Alpha 4 |  | Feb 19 | Feb 21 | ***Complete*** |  | *Robert/Michael* |  |
| Logic Alpha 4 |  | Feb 19 | Feb 21 | ***Complete*** |  | *Michael* |  |
| API Alpha 4 |  | Feb 19 | Feb 21 | ***Complete*** |  | *Dylan* |  |
| Testing and automation Alpha 4 |  | Feb 19 | Feb 21 | ***Complete*** |  | *Steven/Kimberly* |  |
| SRS/SDS/Documentation Rev 3 |  | Feb 19 | Feb 21 | ***Complete*** |  | *Kimberly* |  |
| *Beta Release* |  |  | *Feb 21* | ***Complete*** |  |  | All Beta features in place |
| *Customer Exposure* |  |  | *Feb 28* | ***Complete*** |  |  |  |
| *Coding Session* |  |  | *Mar 3* | ***Complete*** |  |  |  |
| *Coding Session* |  |  | *Mar 8* | ***Complete*** |  |  |  |
| *Final Release* |  |  | *Mar 11* | ***Complete*** |  |  | All Final features in place |

# Feature List

|  |  |
| --- | --- |
| **Feature** | **Target** |
| Search for an article | Beta |
| Display article connections | Beta |
| Display article preview (on hover) | Beta |
| Display article thumbnail (on hover and on-click) | Beta |
| Click on article node to center it | Beta |
| Switch between full article and full map views | Beta |
| Initial iteration on moving the map | Beta |
| Initial parsing and relevancy ranking pass | Beta |
| Relevancy tree caching first iteration | Beta |
| Move the map to explore more relationships | Final |
|  | Cut Feature |
| Return to the initial search location | Final |
|  | Delayed feature |
| Relevancy tree caching second iterationq | Final |
| “Help”, “About”, and “Contact” sections on the website | Final |
| Article traversal history/breadcrumb trail | Stretch |
| Support multiple languages | Stretch |
| Search for and display a random Wikipedia article | Stretch |
| Support multiple Wiki formats with an API | Stretch |
| Support multiple front-ends with an API | Stretch |
| Right/middle click opens an article in a new browser tab | Stretch |
| Resizable viewing panes | Stretch |

***Zooming functionality:*** *The zooming functionality has been cut at this time. After an implementation and usability test, our customers expressed that the behavior was confusing. We would rather not release a feature for the sake of releasing one and wait until we can revisit its design and ensure that it is fully polished.****Data ranking with the relevancy algorithm:*** *Due to the amount of data we are dealing with (90 million Wikipedia entries), the relevancy algorithm will not finish on time. The impact to our system is low because WikiMap functions with or without relevancy data. Without relevancy data, linked articles are grabbed alphabetically from a particular main article entry in the database. This has been determined to be the least impactful decision to make at this point in time, as we would rather have complete breadth of data (a complete database) than complete relevancy data (non-vital to the functionality of our product, though ultimately invaluable as we do put great stock in our relevancy algorithm). Note that all data is parsed, the strength/relevancy ranking is what is incomplete at this time.*

# Risk Assessment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Risk** | **Chance of occurring** | **Impact if it occurs** | **Steps taking to increase chance it won’t occur** | **Mitigation plan should it occur** |
| This is now a non-issue due to ramping up on PHP. |  |  |  |  |
| Unable to rely on Wikipedia Search for article search results | Med | Med | 1. Do more research to ascertain if this is feasible 2. Do not make implementation decisions assuming that we will be able to rely on the Wikipedia search engine. 3. Design the database with the intent to be able to search for and disambiguate Wikipedia articles (as a backup plan). | 1. Use database for article lookup instead of Wikipedia 2. Implement basic disambiguation logic alongside the database |
| Handling queries with more than 1 result on Wikipedia | Low | Med | 1. Design UI with the intent to handle these cases 2. Do not assume that search results will always return one article result exactly | 1. Re-evaluate query logic 2. Determine the source of the issue, such as the Wikipedia search or our code 3. Prioritize fixing the issue before continuing other implementation |
| Poor performance of graph and relevancy algorithms | High | HIgh | 1. Plan algorithms and logic with large data sets and high performance in mind 2. Research and implement common graph and relevancy ranking algorithms where possible | 1. Identify performance “hotspots” and prioritize optimization of these areas 2. Re-evaluate algorithms and simplify wherever possible |
| Falling behind schedule | Med | Med | 1. Provide reasonable time estimates for tasks and buffer with extra time to allow for bugs and other issues 2. Track team member status closely and adjust schedule as needed along the way 3. Adjust team member assignments to reflect highest priority milestones and highest cost feature areas | 1. Reevaluate current priorities and milestones 2. Prioritize major deliverables and reassign team members as necessary to focus on these 3. Cut minor features in order of priority in the case of major setbacks |

We have altered the risk level of poor performance of relevancy algorithm to high, as we are currently experiencing about a one week delay in populating all of our relational strength data due to the amount of data that must be ranked. The ETA on this delivery is approximately 1 week from final release (1 week from March 10, 2011.)

Test and Documentation Plan

# Test Plan

# Overview

Testing work will be divided amongst the members of the testing group.  We will assign members to focus on each test component as they are developed and as necessary so that there are multiple perspectives involved in assessing functionality.

Most tests will be manual tests rather than automated, but those measuring and assessing stress behaviors as well as dealing with the database will be ideally implemented through automation.  We believe in striking a balance between manual and automated testing, and wish to avoid relying too much on either.  
  
We wish to explore some automated testing, particular unit tests, with the use of Unit Testing tools such as:

* Java Unit Testing: JUnit
* PHP Unit Testing Framework
  + [SimpleTest](http://www.simpletest.org/)
  + [PHPUnit](https://github.com/sebastianbergmann/phpunit/)
* Browser-Based Test Automation
  + [Selenium](http://seleniumhq.org/)

If necessary, we will provide separate scripts for running tests on Windows and for Linux in our final release.

* 1. **Test Adequacy**

These below test strategies represent a best approximation of what we can do at this time with a limited understanding of our actual implementation choices.  The intent is to combine black- and white-box testing, as well as manual and automated testing, to have a full suite of ever-evolving tests.  We have attempted to include stress tests as well as functionality tests in order to test the scalability of our system.

* 1. **Bug Tracking and Plan of Use**

We will use Google code’s inbuilt bug tracking.  This will be used to assign tasks and bugs to individuals, which will then be reassigned (if necessary) and resolved.    
  
All tasks/bugs are assigned a priority and deadline.  When tasks are created, the PM will triage tasks out to various team members.  The team members will receive email notifications and are responsible for starting the tasks and resolving the tasks.    
  
Some tasks/bugs will be assigned to team leads with the intent of being delegated and reassigned. In these instances, the leads are responsible for updating the ownership of the tasks as necessary.

# Unit Test Strategy Unit tests will attempt to isolate different aspects of the WikiMap framework for testing, such as the PHP front-end, the UI, and the database. The ultimate goal will be to mix black- and white-box testing strategies for optimal coverage.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Name** | **Purpose** | **Coverage** | **Test Actions** | **Development** | **Frequency** | **Type** |
| **UI Gestures** | Test various gestures on the UI in isolation to make sure they behave as specified in the SRS. | UI: WikiMap | 1. Dragging the map (clicking and dragging on whitespace) 2. Dragging the map (clicking and dragging on a node) 3. Clicking on a node 4. Double clicking on a node 5. Hovering over a node 6. Zooming in/Zooming out 7. Zooming in/out while attempting to drag the map 8. Zooming in/out while clicking | No development required | Bi-weekly, or as-needed | Manual |
| **Search queries** | Test various kinds of searches to ensure that the behavior is correct | UI: Search | 1. Misspelled query 2. Blank query 3. Non-alphanumeric query 4. Search that returns a disambiguation (on Wikipedia) 5. Search that returns a specific article on Wikipedia 6. Search that returns many articles on Wikipedia 7. Search that returns no articles 8. Search containing SQL query injections | Development by one group member | Bi-weekly, or as-needed | Auto |
| **Swapping focus** | Test the behavior of swapping between full WikiMap view and full article view | UI: Full WikiMap View and Full Article View | 1. Attempt to swap immediately after running a search 2. Swap after clicking on an in-article link 3. Swap after clicking on a node 4. Swap after switching between article and map view several times 5. Swap after dragging and/or  zooming in/out on the map | Development by one group member | Biweekly, or as-needed | Manual and auto |
| **Basic Database Functionality** | Test the behavior of the database | Database | 1. Query for a nonexistent node 2. Query for an existent node 3. Do joins between some/all tables 4. Attempt to insert invalidly formatted data into all tables 5. Attempt to insert validly formatted data into all tables 6. Attempt to remove/change data from tables with foreign key dependencies | A battery of tests that can be run automatically should be developed by one group member with database experience | Bi-weekly, or whenever the database is updated | Auto |

* 1. **System Test Strategy**   
       
     Many system tests will take the form of simultaneous tests, as a mode of stress-testing our system. System tests will typically be run biweekly as part of an ever-evolving automated test suite. Though these tests may be developed by individual group members, they should be reviewed by the test team.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Name** | **Purpose** | **Coverage** | **Test Actions** | **Development** | **Frequency** | **Type** |
| **Simultaneous Queries** | Test various methods of triggering multiple queries from different client connections. | Front-end and back-end, especially DB scalability | 1. Searches 2. Clicking on links in article view 3. Clicking on nodes in WikiMap view 4. Hovering over many nodes in the graph | Development by one group member | Biweekly | Auto |
| **Simultaneous UI Interactions** | Test various map and UI interactions from many client connections. | Front-end/UI and back-end scalability | TBD | Development by one group member | Biweekly | Auto |
| **Whitebox Communications Tests** | Test communications (inputs/outputs) between different layers of the infrastructure. | Communications/APIs | Valid, invalid, and null inputs to each of the APIs, modules, and other standalone (back-end) components | Development by one group member | Biweekly | Auto |
| **Security tests** | Ensure that there is no way that a user can alter database data or do malicious things to the back-end from the front-end. | Database, back-end, security | TBD | Development by one group member | Biweekly | Auto |

* 1. **Usability Test Strategy**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Name** | **Purpose** | **Coverage** | **Test Actions** | **Development** | **Frequency** | **Type** |
| **Use Case Tests** | Test the steps of each use case in order. | Use case | 1. The normal path of the use case should be the primary test 2. Any alternatives or extensions to the use case should also be tested | Development by one group member | Biweekly | Manual and/or auto |
| **Paper UI Prototype** | Meet with customers/staff to do a UI paper prototype | Front-end/UI usability |  | Development/participation by entire test team | Per every UI change | Manual |
| **Beta Testing** | Determine what aspects of the product are usable and what aspects are extraneous to determine what our focuses should be. | Front-end/UI usability | Meet with customers from our customer group, one or two at a time, and have them use our product.  Test should begin with a survey of what they expect to see and interact with based on our product descriptions.  We will then observer them through their motions, asking them questions and watching how they naturally go through Wikimaps. | Development/participation by entire test team  * Create a schedule for meeting with each customer/customer pair * Create a survey to determine their understanding of the product based of SRS and SDS * Create a series of tasks to guide the customer’s interaction with WikiMaps. * Arrange meeting times where one or two members of the test team sit with the customer to do the usability test | Once with each member/member pair of the customer group at least once after the beta release.  Add more individuals and sessions as time and iterations permit. | Auto |
| **Security tests** | Ensure that there is no way that a user can alter database data or do malicious things to the back-end from the front-end. | Database, back-end, security | TBD | Development by one group member | Biweekly | Auto |

# 2. Documentation Plan

# Administrative Documentation

1. Mercurial repository: setup and usage
2. Bug tracking system: setup and usage
3. (Database setup beyond providing a schema is out-of-scope)
4. (web server setup beyond installation of WikiMap is out-of-scope)Web UI – setup, access, and updating
5. Back-end APIs and logic – overview, setup, access, usage, and updating
6. How to run tests and add tests to our suite
7. How to read-only access our database

# Main (Search) Page, WikiMap View, Article View:

1. Documentation should be in the form of graphical or interactive cues
2. Search box – Search button has magnifying glass or other recognizable “search” icon
3. Language selection drop-down (stretch feature)
4. Logo, icon, watermark, or other graphic to indicate what the WikiMap looks like on the main (index) page  
     
   1. **About Us Page:**

Provides information about our group, the purpose of the site, and the purpose/inspiration behind WikiMap.

* + 1. What is WikiMap? (“About WikiMap”)
    2. What are WikiMap’s goals/vision? (“Our Mission”)
    3. Who worked on WikiMap? (“Our Beginning” on the About page and “Team” on the Contact page)
  1. **FAQ Page:**

This provides a simple overview of the various navigation abilities and interactions with WikiMap. This area should be largely unnecessary if we strive to make a user-friendly UI. The goal is that the user would never actually need this information, but it’s provided as a “just in case”.

* + 1. What is WikiMap?
    2. Why did you make WikiMap?
    3. What browsers are supported?
    4. Why is WikiMap running slowly?
    5. How do I search for an article?
    6. How do I navigate the map?
    7. How do I get more information on an article?
    8. What does hovering on an article bubble do?How can I view a full Wikipedia article?
    9. Why is your article different than Wikipedia?
    10. I found a bug – how do I report it?
    11. Where can I find your source code?
    12. What if I’d like to contribute to the WikiMap project?
  1. Contact Page
     1. WikiMap Team
     2. Contact Information