Homework #2

CECS 550

Dr. Ragade

February 20, 2014

Team Wikiviz

Ashley Revlett

Everett Rush

Sarah Mullins

**1a) Software Architectures**

We are using the Python framework Kivy as a base for our project. Kivy makes use of several architectures that we’ll also use:

* **Component-based** - separate modules provide isolated functionality, but depend on each other for the larger application.
* **Event-driven** - the GUI operates based on an event loop that reacts to user input.
* **Layered** - layered modules provide different levels of abstraction.

Our requirements include a GUI that responds to user events by displaying different information. The event-driven model is a good match for that requirement. The component-based architecture should allow us to easily adapt to any requirements changes that occur during the agile lifecycle, as components are independent and easily replaced. The layered architecture allow us to code at a higher level of abstraction than using low-level APIs, which will increase the likelihood of us reaching our deadline for this project.

We will use a **Model-View-Controller** architecture as a base for our overall application design. In this architecture, the View gets information from the Model, and the Controller manipulates the Model. This will allow us to decouple our application functionality and work independently on the modules.

**1b) 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.or). 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.

**1c)** See attached Software Design Document

**1d)** The Voice of the Customer is heard in the user interface requirements. The user interface design is what is driving the functional requirements and necessary system architecture. This can be seen in any of the UML diagrams that include the Display package. The Use Case diagram also specifically speaks to the Voice of the Customer, as it is the definition of the Customer’s wants and needs.

**1e)** UML Software Design – See attached Software Design Document

**1f)** All of these diagrams are useful for testing. The UML Class diagrams show which classes are most heavily interconnected and which areas of the design are most complex. Any essential functions or complex behavior should be heavily tested.

The Use Case diagrams show the “big picture” use cases we need to test in integration testing. They also describe functionality that we can break down into testable units.

The Sequence Diagram is useful because it shows at which points the communication between objects can fail. These are the points at which we want to test heavily to make sure all possible error conditions are handled gracefully.

The State Diagram is useful because it shows the relative complexity of states. More complex states are more prone to bugs and should be more heavily tested.

**2)**

* **Library Failures**
* **UI Requirement Change**
* **Network Communication**
* **Security Vulnerabilities**
* **Legal Responsibilities**
* **Hardware Limitations**
* **Performance**

**2a)** Our application at worst would result in the loss of comfort. However, we still have design risks that threaten the project. Our application is designed around many libraries. We risk performance degradation and portability issues by relying so heavily on libraries. It is also possible that those libraries will no longer offer support and updates. That could seriously damage the longevity of the application.

**2b)** The Configuration Control Board, explained below, will oversee any changes to the project’s artifacts.

Each member of WikiViz is responsible for submitting error reports for their respective domain of responsibility.

**2c)** Any changes to any existing artifact belonging to the WikiViz project must follow the WikiViz change procedure. Elaboration and extension of design patterns, architecture, diagrams, or requirements is not considered a change. A change is limited to modification of existing documentation but not new additions.

Once an item has been identified as needing change, a Software Change Request must be submitted. We will be using an SCR form based off the template provided by energy.gov (http://energy.gov/sites/prod/files/cioprod/documents/scr-form.pdf). Before an investigation into the problem is started, the SCR will be evaluated for completeness, clarity, and relevance. After a short preliminary investigation lasting no longer than half a sprint, the Configuration Control Board will review the request. The CCB is made up of the project leader and both team leaders. Upon review by the CCB, a request will be declined or designed, implemented, and tested.

**3) Operational Profiles**

Table 1. Customer Profiles

|  |  |
| --- | --- |
| **Customer** | **Occurrence Probability** |
| Casual Wikipedia users | 0.65 |
| Wikipedia Enthusiasts | 0.25 |
| Researchers and Students | 0.10 |
| **Total** | **1.00** |

Table 2. System Model Profile

|  |  |
| --- | --- |
| **System Mode** | **Occurrence Probability** |
| Single-User Mode | 1.0 |

Table 3. Input Parameter Profile

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Values** | **Occurrence Probability** |
| Keyword | (Search result,  No search result) | (.85, .15) |

Table 4. Functional Profile

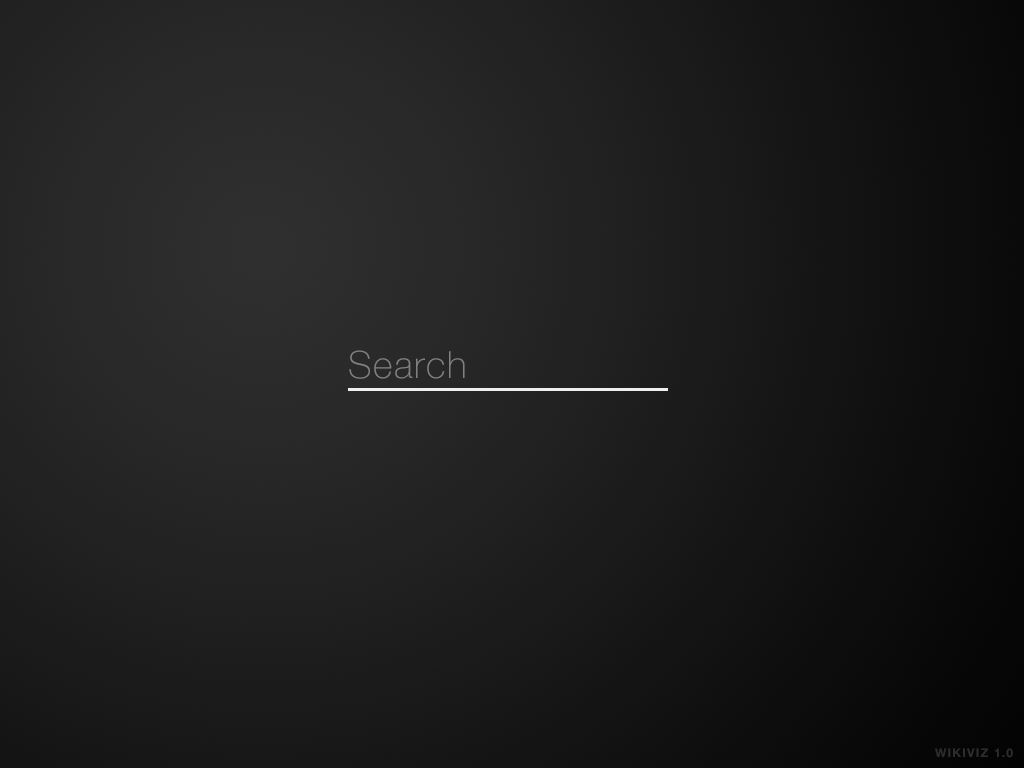
|  |  |
| --- | --- |
| **Functions** | **Occurrence Probability** |
| First search | 0.2 |
| Viewing Graph | 0.4 |
| Exploring further nodes | 0.4 |
| **Total** | **1.0** |

Table 5. Operational Profile

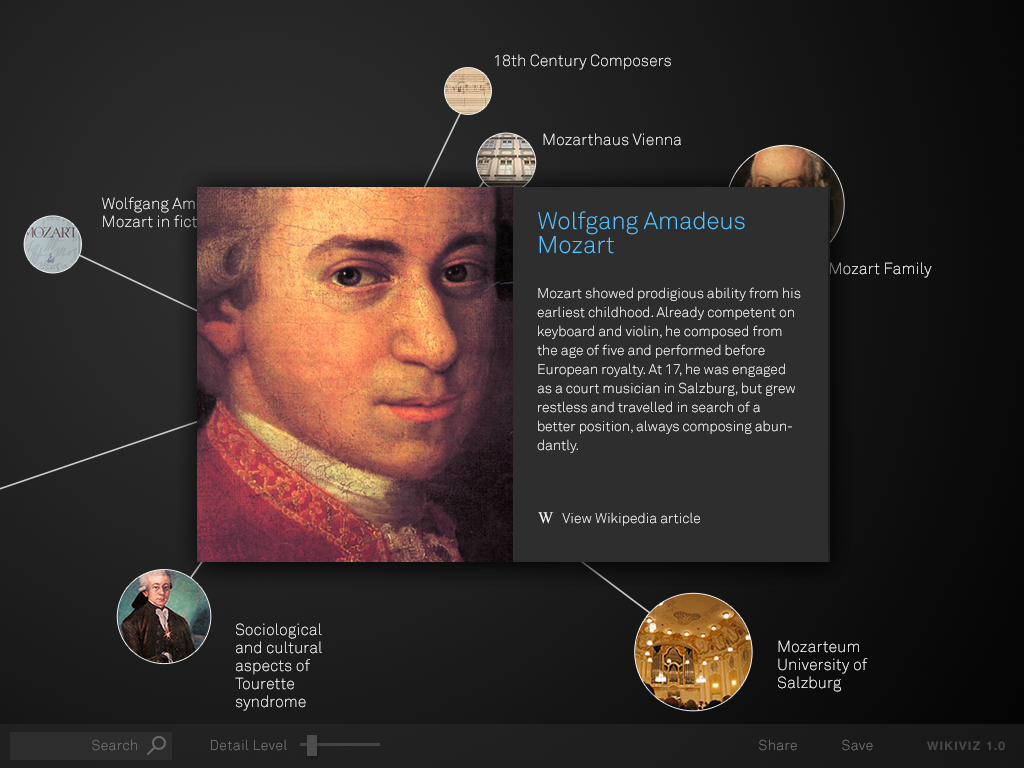
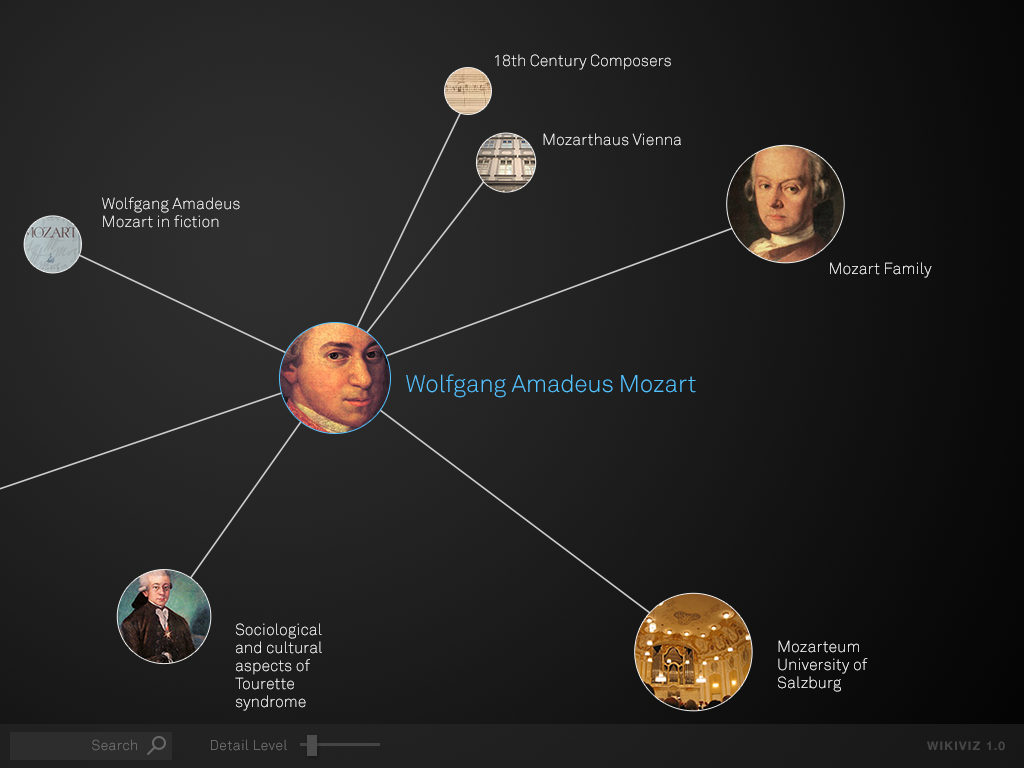
|  |  |  |
| --- | --- | --- |
| **Operation** | **Occurrence Probability** | **Number of Tests** |
| Casual, First search | 0.13 | 5 |
| Enthusiast, First search | 0.05 | 2 |
| Researcher, First search | 0.02 | 1 |
| Casual, Viewing Graph | 0.26 | 11 |
| Enthusiast, Viewing Graph | 0.10 | 4 |
| Researcher, Viewing Graph | 0.04 | 2 |
| Casual, Exploring further | 0.26 | 11 |
| Enthusiast, Exploring further | 0.10 | 4 |
| Researcher, Exploring further | 0.04 | 2 |
| **TOTAL** | **1.0** | **42** |

Assume it takes 1 hour to design and run one test, 5% of tests reveal faults, and it takes 4 hours to correct each fault. Our test budget allows up to 45 hours. The total number of tests to perform is: T + (0.05 \* 4T) = 50, T = approx. 42. The table above uses the Occurrence Probability to calculate the number of tests that should be performed for each operation based on total testing hours available.

**4)** Our UI will only have one stakeholder layout. The UI is designed to be visually pleasing and easy-to-use. Our guiding principle is to make searching through WikiPedia simple, interactive, and sexy.



The UI is very simple. To explore a topic the user simply clicks the search box, types in a word, and presses ENTER. Then the user can tap a node to explore that topic. The node will expand and give a summary of the topic. Then the user can collapse the node with a single click and continue exploring other related topics.



The UI is interactive. With just the swipe of your finger, you can explore different aspects of your topic.

The UI is sexy. Currently when a user explores Wikipedia they see a website will blue hyperlinks and a white background. That is not a visually pleasing experience. WikiViz allows users to visually explore their topic.

**Complexities:**

* If a user only wanted to read a single Wikipedia page, it would be very time consuming to use this app. First the user would start the app. The user would have to search for their topic. Then they would click the central node. Then they would have to click view Wikipedia article.
* It is often the case that the user would not want to restrict themselves to a single information source like Wikipedia. Many users will would prefer Wikipedia supplemented with other sources like YouTube.

**Abuse and Misuse:**

* If a user repeatedly hit the go button, the program would send many requests to Wikipedia. Our design will prevent this by checking a page cache before sending a URL request.
* It might be possible to crash the device that the application is running on. This could result in information loss or damage to the device.
* There are security risks using network protocols that we have not discussed yet.

**5)**

*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 Wikiviz:

Complexity = 30 - 12 + 2 = 20

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

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. Wikiviz’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 Wikiviz 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 Wikiviz 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, Wikiviz 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, Wikiviz 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 Wikiviz 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, Wikiviz 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, Wikiviz currently meets 3.5/6 or 58% of the original software quality attributes.

*Robustness*

Wikiviz 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 Wikiviz 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 Wikiviz 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
  + Wikiviz 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 Wikiviz 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

*Efficiency*

We can calculate the total relative speed of Wikiviz using a table of methods, as found in the textbook:

|  |  |  |
| --- | --- | --- |
| **Step** | **Function** | **Relative Speed**  0 = negligible  1 = neither  2 = significant |
| 1 | Display: init() | 0 |
| 2 | Display: search\_query(string keyword) | 0 |
| 3 | Controller: init() | 0 |
| 4 | Controller: initDisplay | 1 |
| 5 | Graph: init() | 0 |
| 6 | Graph: add\_node(Node) | 2 |
| 7 | Graph: add\_edge(Edge) | 2 |
| 8 | Graph: get\_nodes() | 1 |
| 9 | Graph: get\_edges() | 1 |
| 10 | Node: Node init(string keyword) | 1 |
| 11 | Edge: Edge init(source, destination) | 1 |
| 12 | TextParser: TextParser init() | 0 |
| 13 | TextParser: get\_instance() | 2 |
| 14 | TextParser: get\_links(Page page) | 1 |
| 15 | TextParser: get\_image(Page page) | 1 |
| 16 | TextParser: get\_excerpt(Page page) | 1 |
| 17 | Network: init() | 0 |
| 18 | Network: on\_success() | 0 |
| 19 | Network: on\_error() | 0 |
| 20 | Network: Page fetch\_page(string keyword) | 2 |
| 21 | Network: Network get\_instance() | 1 |
| 22 | Network: Page get\_page(string keyword) | 2 |
| 23 | Page: init() | 0 |
|  | **TOTAL** | **19/46** |

With all the methods together taking less than half of the maximum run time to execute, our application is predicted to be very time efficient. In addition, as can be seen in the Sequence Diagram in the Software Design Specification, only two methods will be *not* be running in parallel with any other method, which vastly improves efficiency.

At the moment, we do not have enough specific information to create a full space efficiency table, but at the very least we can identify that only exporting the graph to an image file would require permanent memory storage.

*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.

Wikiviz 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 Wikiviz 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.

*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 Wikiviz 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.

**6)** While our tasks have been re-arranged significantly due to changes in our personal schedules, the contents and timing of our schedule have not changed much. We did include several more research and design tasks, which added about 22 hours of effort.

Many tasks within the same user story have been separated and re-located to different sprints, depending on what we have already completed as part of our design process and time restraints for each two week period. One other change is spreading out our portability goal throughout the rest of the sprints: Ashley is using a Mac, and will already need to make programming and design adjustments on her system, so progress toward this goal can be made throughout the remainder of the project.