CS 190 Final Design Document
Amy Simpson (asimpson)
March 6, 2006
GeoEvents

Revised Specifications

1. **Title: GeoEvents**

2. **Description**
   GeoEvents is designed for Rhode Islanders looking for something to do. Often people want to go out and do something but lack the motivation, time, or energy to put in the effort to actually find that something. Newspaper listings lack the ability to adapt to any of the specific needs of the user and can be hard to use. GeoEvents will be able to mesh the current events listings from different sources (turnto10.com and others) with the Google Maps API to provide a much more interactive and intuitive way to look for interesting activities. One current example of a similar project can be found at http://api.local.yahoo.com/eb/demo/. GeoEvents will go beyond, however, and provide additional useful features (specified in section 3 below).

3. **Feature Set/Priorities** *(Rated 1-6: 1 being the highest priority)*
   (The priority rating is a result of responses to questions from possible users and an attempt to mirror similar existing interfaces in order to increase the intuitiveness of the project's use)
   a. Google Maps API with flags marking the location of local events that qualify for the filter parameters provided. [1]
      i. This will be a frame on the page containing a Google Map of the area. When GeoEvents is first opened up, the default map should be of Providence and all filter types of events should be included on the map. All events are marked by a flags which can be clicked for details (more details in section c below.)
      ii. With the addition of new parsers this will easily expand in the future to a larger base area.
   b. Be able to enter in a 'base' address (the address the user will be starting from, for example their home address or the address of their office if they are leaving from work [2]
   c. Clicking on an event:
      i. an event's flag should expand to show the brief details of the event in a concise, readable, and consistent manner.
         1. Display the event location (address) [2]
         2. Display the event time [2]
         3. Display the price of the event [3]

4. Show the distance from specified start (home/work) address (if there is one, other wise this field should not be included in the flag information) [3]
5. Display a link to the original listing [2]
6. Display an event related phone number if relevant [3]

d. An interface for choosing the search dates [2]
e. Ability to filter event type (e.g. choose only Theatre & Museum) [2]
f. Be able to get directions to a specific event by linking to Google Maps direction capabilities directly [3]
g. Display below the map window a list of the filtered events [1]
    i. This contains all information included in the event tag with the addition to a link to obtain directions.
h. There should be a link to basic user instructions [2]
i. Display weather below the events filtering for the specified date [3]

Optional Features (Priority four and over)

(a) Have an interface that will learn from past usage of specific users and be able to 'suggest' events [6].
    (i) Using past inputs (searches and events the directions were found for) from the user, GeoEvents would be able to find and 'suggest' similar types of events that the user might enjoy
    (ii) The GUI would call this component when the user requests suggested events. The component would, in turn, query the Events Database to find events. Note: this component would most likely contain a database to store information about the user.
(b) Be able to select more than one date at a time [4]
(c) Have a calendar with saved or selected events marked on it [6]

The first priority (priority one) is having a map that shows and lists the events around Rhode Island. This is absolutely essential. For this project to have real usefulness, all items in priority two should be implemented. This includes event details including time, location, and price. The third set of priorities are extremely useful and what would make this project more useful than what is currently available since it would integrate the direction creating capabilities of Google Maps into the project, but is not essential to the overall functionality of the project. Beyond that, other items are helpful and added bonuses, but not what truly define the project.

4. System Model Diagram

(See below for the actual diagram)

This is the HTTP server that users will connect to via their web browser. This server will host the GUI code that generates HTML pages in response to user requests. The database(s) might also be stored on the same machine.

The GUI is responsible for formulating responses to user requests. This means that the GUI not only has to be able to take the input and decipher some meaning from it, but it also has to decide what to query for from the database(s) based on that meaning and then it has to use Google Maps to display the info it gets from the database(s). It also links to Google Maps to provided direction to and from events. The GUI is essentially the glue of the whole project that ties all the separate pieces together.

The Weather component is in control of gathering information about the weather from weather.com. Minimal functionality would be to just provide a link, more extensive functionality would be to actually return the weather to the GUI.

The Events Parser Manager is in control of deciding when to gather new information about events. When the time comes, the manager uses one or more parsers to collect events data and update the Events Database.

The parser, when called upon, parses its respective website (most likely turnto10.com) and collects data about local events.

Events Database

The Events Database stores the most important information, the actual events. Locations should already be resolved and latitude/longitude coordinates already computed and stored in the database. It does, however, still contain the original address as well (this is primarily for user display purposes).

Location Resolution Genie

The Location Resolution Genie is the component that takes a "location" and returns both an address and a pair of latitude/longitude coordinates corresponding to that address. The first version of this should be trivial. Any location that is passed in that isn't just an address that can be geocoded should just return null which will cause the Events Parser Manager to drop the event instead of add it to the database. Once this simple functionality is in place, if there are sufficient resources to do something cleverer, then a new strategy may be devised to replace this first one. (The actual creation of the latitude and longitude is in the Geocoder).

Geocoder

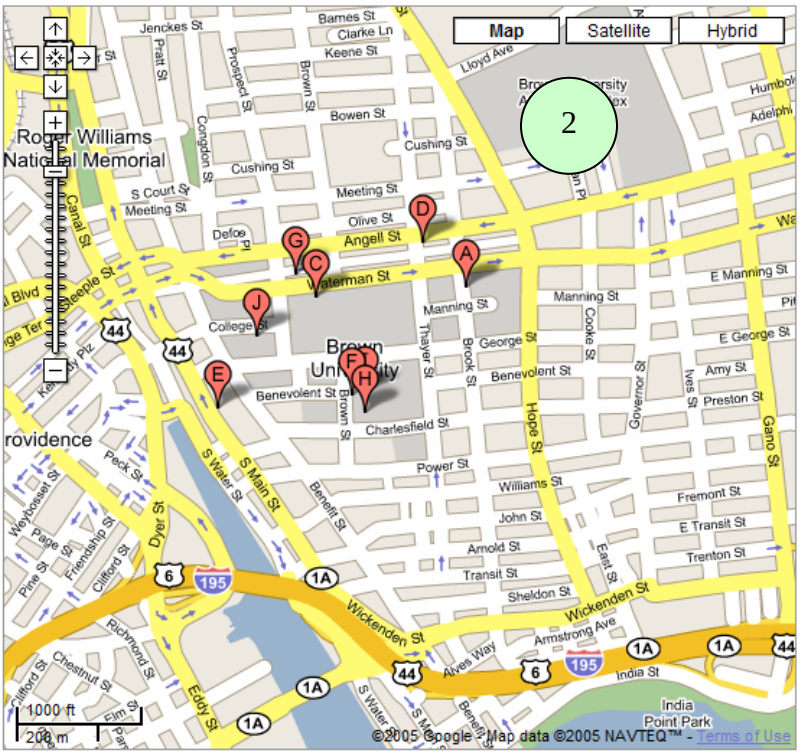
The Geocoder takes the address provided by the parser and will return the associated latitude and longitude.

Concierge

The Concierge is spawned by the WebServer and depending on the length of time since the last update to the Events Database, it will do one of three things. If it has been updated recently, it will only run the GUI. If it has been updated within a week, but more than a day, the GUI will be run in parallel with the EventManager. Any longer than a week and the GUI will not be run until the EventManager has completed its update.

5. GUI Diagram

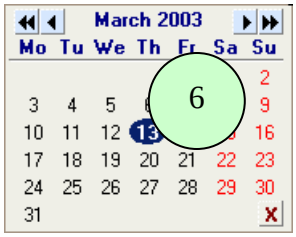
GeoEvents 1



Address:
City:
State:
Zip Code:
3

- ☐ Theatre
- ☐ Music
- ☐ Arts & Museums
- ☐ Etc…

4



A Event #1
Time and Place
Cost
Phone-number
Event type Information
Link to original information

B Event #2
Time and Place
Cost
Phone-number
Event type Information
Link to original information

5

Friday Night: A passing evening shower; otherwise, partly cloudy and breezy.

7

Friday Night: A passing evening shower; otherwise, partly cloudy and breezy.

About Us 8 User Information

The GUI is organized so that the most noticeable item is the map itself. Easily accessible on the right are the options for changing and filtering the events. They are in a standard clean format so that they do not distract from the map. Below this main part of the GUI, the list of events is displayed.

The items in the map, corresponding to the green number tag, are:

1. **Title:** The page title.
2. **Map:** This is the map that will display the event locations with markers. Users can pan around the map and zoom in or out. Clicking on a tag will provide the user with more complete information about the event including time, location, cost, and distance (see details in the features section above)
3. **Address:** This area is where the user inputs the address to center around and base directions from. After entering the address data, the user can click "set address" set this option.
4. **Filter:** This box contains all the filter options for event types.
5. **Date:** Allows the user to select a date to look at events for.
6. **Results:** All events and their corresponding details are displayed here.
7. **Weather:** Weather for the specified date is shown here and also in the popup for each event.
8. **Important Links:** There will be links here to "About us" and "User Information" which will be documentation detailing any important usage information

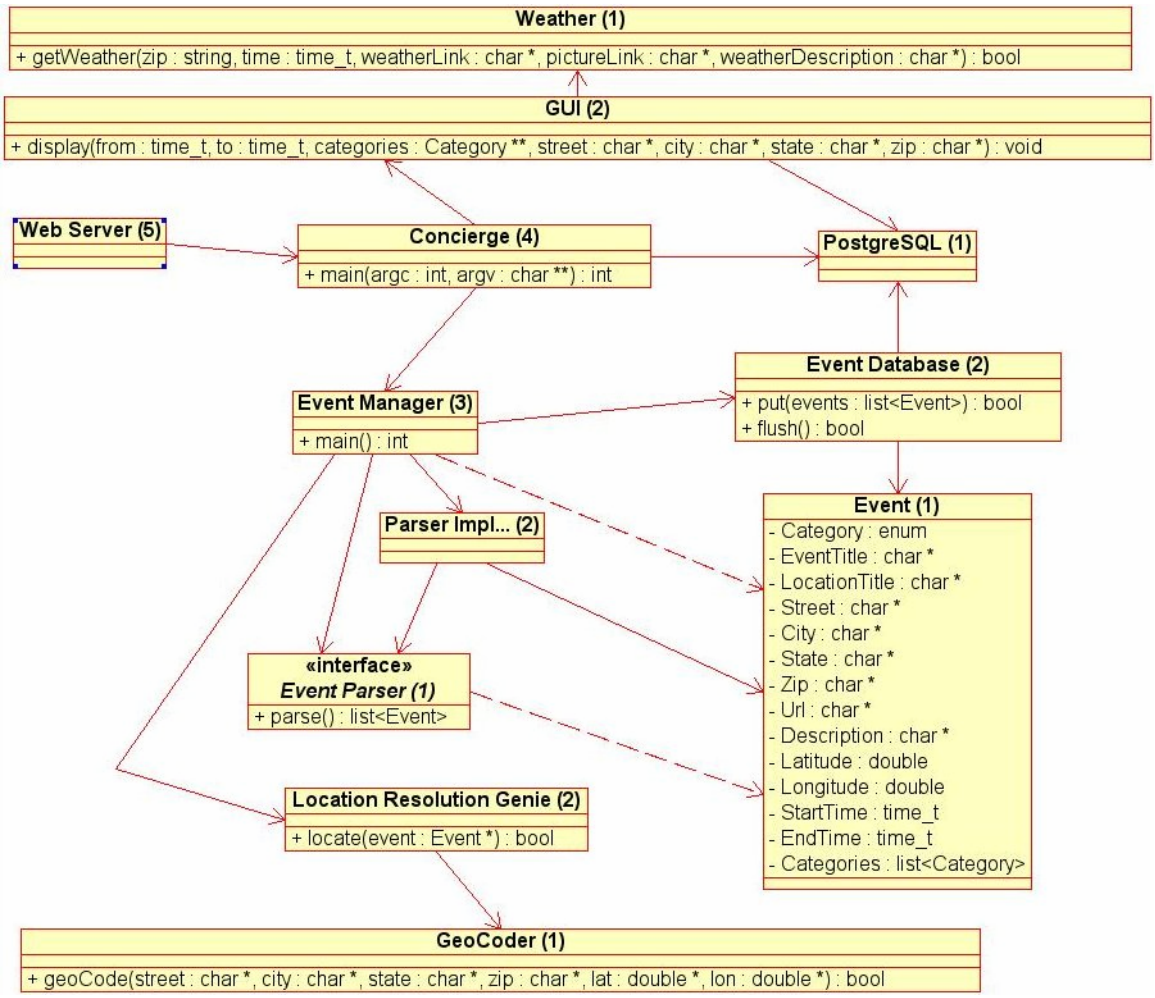
6. **Usage Requirements**
    a. Since this product would be of public interest it would need to be reliable under a heavy usage (500).
    b. Any longer of a response time than 5 seconds would be inadequate since multiple searches or refinements of searches (different types of events, the zoom factor on the map) could be made in short succession.
    c. Depending on connection speed, loading should take less than 20 seconds.
    d. GeoEvents must be consistent.
        i. If a user searches around a start address and certain type filters and finds an event they like, if they come back an hour later to get directions, they should easily be able to find it with the same search parameters.
        ii. Essentially, searches with the same parameters should return the same results.
        iii. If a new event is added to the database, however, it should come up in the search. Essentially, events must continue showing up consistently in a search until removed from the database.
        iv. Events are removed when the EventManager runs and events are past.
    e. Must obtain at least 90% of events on the page successfully.

7. **Non-Functional Requirements**
    a. Testing
        i. Besides testing to ensure that the project responds in the specified amount of time, it is essential that it continues to do so as the user usage increases.
        ii. Data displayed must match events included in the database
            1. This should be tested under a high load (500 simultaneous) users for reliability
            2. Compare located events to actual HTML sources; ensure that addresses do not get confused and that all events claimed by parser really exist.
        iii. The program will be tested on multiple computer platforms on Linux, Mac and Windows computers – this entails running the website.
        iv. The program will also be tested with multiple browsers including Internet Explorer, Firefox, and Safari.
        v. Error handling will be extensively tested, and scenario testing will be used to ensure a good user experience from start to finish. (see testing section below for more information).
    b. GeoEvents must be accessible from any computer through the web
    c. Must be written and ready to demo by the end of April 2006
    d. Must have thorough documentation on the use of the program and possible extensions for the future that were not implemented in this version. Includes:
        i. Help links from the GUI
        ii. Extendibility documents
        iii. A brief user README
    e. Ease of Use
        i. This project should be usable for anyone capable of surfing the internet.
8. **Divisibility**
    a. This project should end up being sufficiently divisible for the projected size group for this year's class. Different elements include
        i. Overall GUI
        ii. Google Maps API interaction
        iii. Parsing of the www.turnto10.com information page and other events listings
        iv. Filtering and searching of the event information once parsed
9. **Specific Challenges/Issues for the Project**
    a. External Dependencies
        i. Neither the Google Maps API nor the Yahoo Maps API interfaces with their directions-creating capability.

   ii. Google forces the use of an external geocoder (switches an address to longitude/latitude that can be used in the API).
b. The genie at the moment is fairly vague. Defining and creating the genie will be a challenge.

Levelized Component Diagram

Architect: Sam Cates



Testing Strategy

Tester: Toby Muresianu

Test will occur in three phases, component testing, integrated testing, and final stress/scenario testing.

Component Testing

- As people are coding, they will find bugs and should fix them as they find them. They should also use dummy classes (described below) to help them run automatic tests. Having a mainline in your class and doing simple tests of each method as they are written will allow you to catch simple internal errors while they are still simple, internal and not hard to track down as none of the causes of the bug are hidden.
- A class with a main function which incorporates all tests and functional verification will be written. This will be run every time code is checked in via CVS to ensure that updates have not broken older functionality. All tests that are run should be added to the main function prior to check in; they should also print out appropriate debug information when they fail.
- As a general rule, very method should have an assert in it to verify the input is non-null and within the acceptable range of values, and a printline to track that it is being called. This will both allow you and everyone who touches your code to identify bugs much more quickly and speed up time spent working out bugs that occur in integration.
- After all of the pieces of a class are together and basic functionality is verified, dummy classes which make and receive appropriate function calls should be used to simulate interactions with other classes, and the test cases detailed below should be run, as should any other methods of potentially breaking your code (prior to integration, the dummy classes should be checked in for use when code is checked in to CVS, but afterwards references to the actual classes should be used).
- A good rule to remember is the best way to ensure your code works is to try and break it in as quickly and in many ways as possible; doing so will tell you where the weakest links in your program are.
- Those components with external dependencies should test their code with them.

Integration Testing

- After component testing is finished, we integrate, and certain integration tests need to be run on the subcomponents, specifically the front end (GUI, Weather, Google Maps) and back end (Event Manager, Parser Manager/Implementations, Geocoder, Event Database/Event).
- The back end should be able to interact with through a simple text interface, allowing it to print out all the events that are coming up and search for events given certain criteria, etc. It should be able to parse all websites and the output checked by hand against the content of those sites. Addresses should be complete and correct, and longitude and latitude checked against Google Maps. Error testing should include asking it to parse long documents of data, web pages it is not equipped to (to prepare for redesigns on the sites we use) , and asking it to connect to the server when it is not connected to the internet and making sure that it doesn't crash and passes on appropriate error messages or codes to the front end. The databases should also be rigorously tested, by adding lots of events, deleting them off the webpage and making sure they are updated on when the pages is parsed again, changing the system clock to simulate time passing and making sure old events are removed, and making sure that new events are added for the appropriate days ahead. It should also be observed whether it can be made to display so many events are displayed that it slows down the computer or overwhelms the user (performance should be monitored here in general, and printouts tracked whenever methods are entered and exited so that slowdowns can be pinpointed).
- The front end should be passed fake events and run online, so that it can be ensured that it is capable of displaying everything correctly. Selecting criteria and searching should update an output window with the request being sent; accesses to PostgreSQL should be tracked and reported to ensure they are coming in at the correct time. Strenuous combinations of criteria should be requested, such as requesting a very narrow or very wide date range (it should also make sure the start date cannot be set after the end date).
- The Web Server should be connected with the concierge and PostgreSQL and tested independently, ideally before the front end and back end so that it can be connected to both without fear of significant new bugs; dummy front ends and back ends will need to be written for this.
- After subunit integration tests are complete, we will integrate the whole thing and perform scenario and stress tests on it after functional verification and the individual component tests are passed. Stress tests occur by scripting a battery of actions (search requests, filter changes, location changes, event page changes) overnight or over several days and ensuring that the program still works afterwards. Scenario tests occur by simulating start to finish expected user interactions (e.g. connect to page, enter criteria, change criteria, click on a

few icons, quit program) and ensure that they work and do not leave any loose ends. This should take two weeks.

- Tests should be run by coding the instructions into a mainline and having it return the program to the initial state when they are completed. In this way, multiple tests can be run in series automatically once they are finished, so that it can be verified that updates to the code have not broken previous functionality.

Test Cases

- Web Server
  - Different web browsers on different platforms; test from home/CIT cluster
  - Slow connection (either simulate by using a modem if we can find one or by accessing while downloading files); should load properly, just more slowly and not time out.
  - Should be able to handle multiple requests simultaneously; test by having several people use software at the same time.
  - Should be able to handle filter requests made before page is fully loaded, since some users will make requests while program is still sending data.
  - Should generate working html if one dependency (say Weather) is broken and include a placeholder (say "Weather not currently available") where it should appear. This could potentially be handled in the GUI--you should talk to Peter about how he wants to handle the case where one component is broken.
  - When a person closes the connection, that should be recognized by the web server and they should not allocate resources to that thread.
  - It should be determined whether page refreshes should be handled by re-querying the database or by simply resending the last generated html. Then it should be tested that this is actually occurring, and that when there are multiple page refreshes from one user before a page is sent, only the first or last one goes to completion and the rest are killed.
  - When a user clicks 'back' or forward, the information should be cached. This is probably handled by the web browser, but the web server should not use any scripts which require it to reprocess events whenever a page is accessed.
  - Test both when the Database is on the same computer and when it is on a different computer than the web server.
- GUI
  - Should display acceptably when one component is missing (weather, maps, etc); layout should not be broken as a result.

    - A quick program should be used to automatically verify that the selections for the events filter reflect what is actually selected. This should be tested when a search is in progress and another is requested with different criteria, after tests are completed and others are requested with different criteria, and when a user selects and deselects various items.
    - Display should be correct when there are no events and when there are many (test 50) events). If an event location cannot be resolved, it should be not be put on the map.
    - The user's start location should always be on the map. If the start location is invalid (cannot be resolved by google), it should indicate it as such and either display no events or use Providence center as the start location so as to show events around the area where most users will live.
    - Either in the GUI or the events database, the amount of events able to be displayed should be capped at a certain amount (say 100), and the program tested by attempting to display that amount and one over that amount. There should be an indication that more events are available but cannot be displayed.
    - When events are clicked, their flags should display correctly even if the price, time, or other information is not available or a null/empty string.
    - The place and time of events should display correctly even if the address, url or name of the event is long (say 100 characters).
- Weather
    - If new weather is unavailable, that should be indicated in the GUI rather than having outdated information passed to the user.
    - If the weather is updated while the user is using geoevents, the weather on the page should either be updated instantly or after any search query (for a retrieval of events, filter criteria changes, address change, etc)
    - If for whatever reason it returns data nonstandard in format, the gui should requery.
    - The parser should be tested with all types of weather and all types of information which may appear (% chance of rain, temp, wind chill, lightning risk, etc). Weather warnings should be handled correctly when they appear.
    - Timeouts should be reported to the gui. It should be tested without a connection to the internet, with invalid URLs and URLs to non-weather websites to make sure it doesn't stall or crash in these cases (otherwise it will be a big problem if weather.com changes its site layout).
    - A dummy class should be made which will contain a verifier to confirm the correct formatting of the data.

    - The parser should be tested on different zip codes and for different areas of the state, as well as invalid ones. If there is an invalid zip code, it should return an appropriate message to the GUI
- EVENTS PARSER MANAGER
    - Could use more detail about this class. How does it talk to the location genie?
    - Test to make sure that it has consistently timed calls, and assuming it uses a timer, doesn't break if the system clock changes in either direction (check to ensure it still updates in that case).
    - If the event parsers have not all returned by the time the update timer expires, make sure it waits until they have all finished until it updates again; if it senses one has timed out, it should stop it and get data from the other ones.
    - Manager should be tested with large amounts of data that it has to insert into the events db.
    - Should be tested with lots of parser instances returning data to ensure that it is able to handle all of them.
    - Make sure that if it receives an error from the Events DB it maintains operations and continues to attempt updates.
    - Should verify data returned from parsers is in the correct format, and be tested by returning invalid data.
    - If a location cannot be resolved, should query the genie. Test that this works correctly and that if the genie is unable to resolve it the event is simply discarded (or check with Amy/Peter to see if it is still possible to display it without a location).
    - Check with parser authors to see if there are any special event formats that may need special handling--e.g. if some events may not have locations--and test whatever implementation you set up to deal with them.
    - Make sure that if the location of the event is returned as null it deletes that event.
    - Enforce that all locations are in a standard format when returned from the parsers, so that the location genie does not have to unscramble them.
- GENIE
    - Test with zero-length string input, legitimate input, malformed (out of order input), and that it returns appropriately in these cases.
    - Test with locations which return multiple matches from Google.
    - Test with full length and five digit zip codes, intersections, city centers, beach and park names as input.
- EVENTS PARSERS
    - Ensure that data is parsed into a standard format for all websites; the event manager should enforce this. A dummy class should be made

which will contain a verifier to confirm the correct formatting of the data.
    - Should be tested with all types of events which have different formats (movies, outdoor festivals, etc) and locations (schools, city centers/areas (e.g. 'kennedy plaza'), parks...any data type you encounter from looking up past events)
    - Timeouts should be reported to the manager. Should be tested without a connection to the internet, with invalid URLs and URLs to non-weather websites to make sure it doesn't stall or crash in these cases (otherwise it will be a big problem if events sites change their site layout).
    - Make sure that all events are retrieved when there are a lot of them on the page, and that if there are no events the parser returns correctly.

- EVENTS DATABASE
    - Same tests as for the weather database; make sure events which are in the past are deleted.
    - As people are coding, they will find bugs and should fix them as they find them. They should also use dummy classes (described below) to help them run automatic tests. Having a mainline in your class and doing simple tests of each method as they are written will allow you to catch simple internal errors while they are still simple, internal and not hard to track down as none of the causes of the bug are hidden. Also, every method should have an assert in it to verify the input is non-null and within the acceptable range of values, and a printline to track that it is being called. This will both allow you to identify bugs much more quickly and speed up time spent working out bugs that occur in integration.
    - After all of the pieces of a class are together and basic functionality is verified, dummy classes which make appropriate function calls should be used to simulate interactions with other classes, and the test cases detailed below should be run, as should any other methods of potentially breaking your code. Remember, the best way to ensure your code works is to try to break it in as many ways as possible. Those components with external dependencies should test their code with them.
    - After component testing is finished, we integrate, and certain integration tests need to be run on the subcomponents. This should take about a week.
    - After subunit integration tests are complete, we will integrate the whole thing and perform scenario and stress tests on it as more specific ones. Stress tests occur by scripting a battery of actions (search requests, filter changes, location changes, event page changes) overnight or over several days and ensuring that the program still works afterwards.

Scenario tests occur by simulating start to finish expected user interactions (e.g. connect to page, enter criteria, change criteria, click on a few icons, quit program) and ensure that they work and do not leave any loose ends. This should take two weeks.

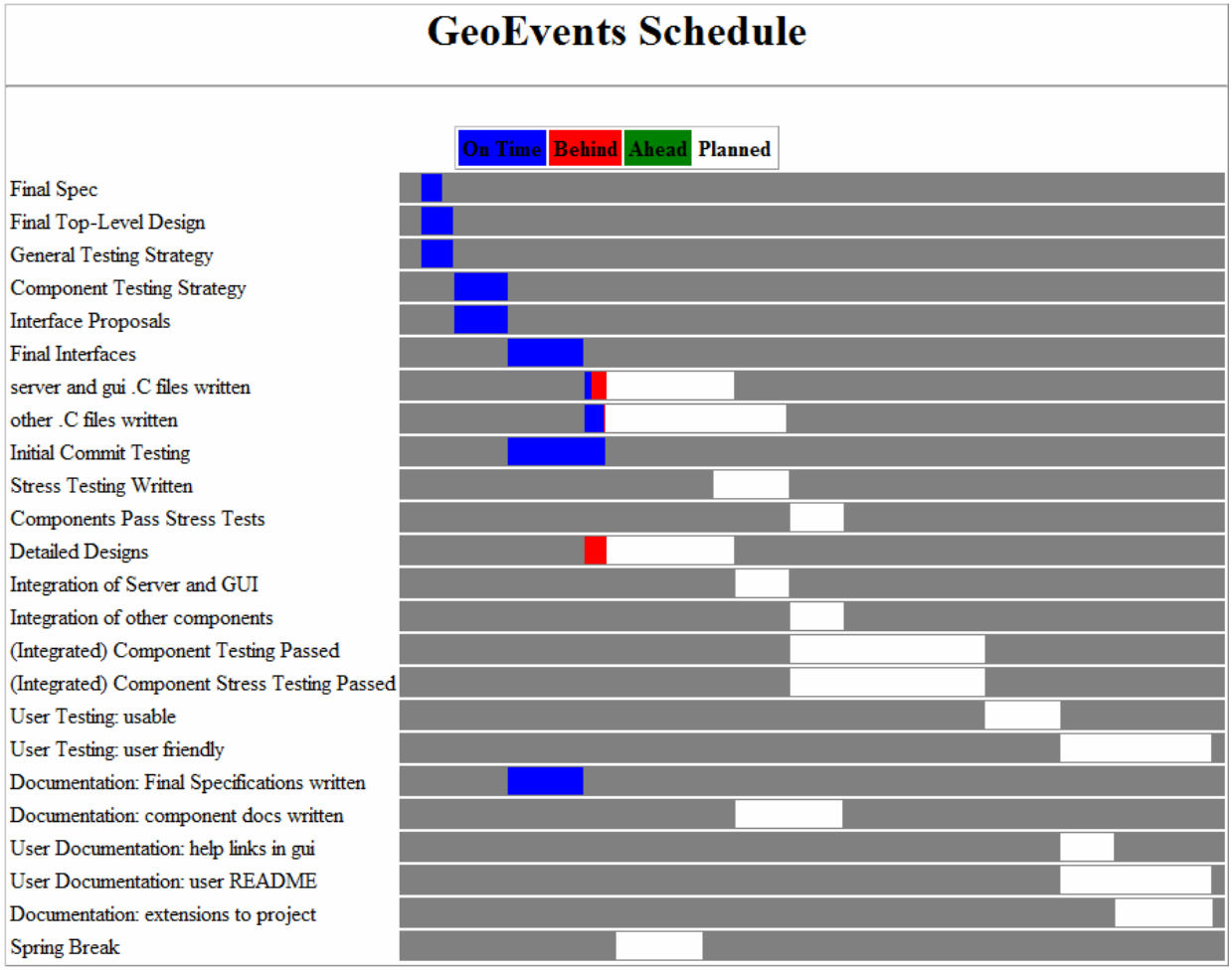
External Dependencies

Event Sources: Since we are dealing with a variety of sources with different formats and types of information, we should design our components in a way so that they can eloquently deal with missing information. We should, to the extent permitted by the functionality requirements of our project, avoid assuming any reliability on the part of the sources providing the events.

Google Maps: We have already implemented minimal functionality with Google Maps so this helps to minimize the risk. Our site, however, will not work if Google Maps goes down.

Weather.com: It may be relatively easy to add a source for our weather information. In addition to Weather.com, we can have a secondary source to contact in case we are missing information. Most weather sites make it very trivial for users to parse their information.

Task Breakdown and Schedule



Group Roles & Organization

Task Maser:	Amy
Vice Task Maser:	Haley
Documentation:	Haley
Architect:	Sam
Testing:	Toby
Parsing:	Andy
Databases:	Yotsawan
Server:	Kaveh, Sam
Genie:	Amy, Haley
Sheriff:	Sam
GUI:	Peter
Parsing Managers:	Kaveh
Integration:	Amy, Sam
Tools:	Toby
Weather:	Haley