A Tool to Automatically Organize the Structure of a Codebase Using Information Foraging Theory Design Patterns

Technology Review and Implementation Plan

Team Postal — Group #38

Cramer Smith, Sam Lichlyter, Eric Winkler, Zach Schneider

February 17, 2017

CONTENTS

I. Introduction

The following document will compare already available tools we have considered and our decisions on the best tool for our product. Sam will be looking at the HTML Parser and the various tools required to perform those actions. Zach will be looking at various ways to store our data structure as well as its serialization and the comparison between the two. Cramer will be comparing the various text editors we have to build our product around as well as the best way to handle events inside the text editor and what language we should build our tool with to best utilize the text editor's provided functionality. Eric will compare various visualization libraries, as well as the best way to display the rules the user has broken, and how to display the file map.

II. Sam Lichlyter

A. Parser Class

This is the class we will use to actually parse the files in the directory and translate it into the data structure we decide to use.

1) PEG.js: PEG.js is a parser generator in JavaScript. Its sole function is to generate a parser which is saved as a JavaScript Object that can be interacted with. It also generates a small API. This would reduce the work needed to create a parser, which would allow for more time to be dedicated to other areas of the project. The problem with this is that it would only generate a simple API. Theoretically the API it generated could be altered, but this would then require more time dedicated to learning how the generated parser was built and how to successfully build in the required changes. [**?**]

2) GOLD: GOLD is also a parser generator. It however does not generate to JavaScript. GOLD takes in a grammar, which we would come up with that reflected the various web technologies we would support and then it outputs an engine in which we would use to parse the files we were given and output our data structure. This would require us to figure out how to use this engine once we get it generated and test it against the various scenarios we expect to come against. It could however catch various corner cases we had not expected, but this is unlikely since we would have to come up with the grammars. [**?**]

3) Custom Parser: The pros of a custom parser are that we would only include what we needed to. This would reduce a massive overhead in interacting with any API we would have to use with a third-party tool. It would also mean we could optimize the algorithms we use for it specifically so it could be faster than the third-party tools. This would also reduce the number of dependencies our extension would rely on greatly meaning there is fundamentally less to go wrong should one of the dependencies change. For these reasons I think this is the route we will take.

B. Parsing Language

This is the language we will choose to use for the parser. This was fairly dependent on the parser class we chose to use, but since we are planning on writing our own, we have a little bit more flexibility to choose which language would be best.

1) Perl: Perl from the beginning isn't that great of a choice because not very many of our team members have experience with it. However it does do really well as a parsing language. There are multiple examples of excellent parsers coming out of Perl. For example Markdown[**?**] was originally written in Perl. Perl was written for parsing in mind. There are many articles explaining why regular expressions will not be sufficient enough to parse HTML.[**?**] For these reasons, at least part of our parser will be written in Perl.

2) Python: Python includes an HTML parser built-in, but it is not nearly as robust or as extensive as the ones written for Perl. The Python library only looks for starting and ending tags. It has a small amount of error reporting, but for our tool, these will not be sufficient. [**?**]

3) JavaScript: Writing a parser in JavaScript would be ideal since most of our tool will also be written in JavaScript. In the end the parser may be required to be JavaScript because of the restrictions placed on us by VSCode. This is fine because there are many tools included in Node.js and npm that also parse HTML similar to some of the Perl tools. [**?**][**?**]

C. Perl Parsing Tool

This is the Perl tool we will be using to parse the HTML which our users want analyzed. This tool must be able to parse complicated HTML in a reasonable manner and give some form of error feedback.

1) HTML-Parser: The HTML-Parser is the canonical Perl HTML parser. It will recognize the various forms of markup on an HTML page as well as separating it from plain text. This tool also parses HTML similar to how web browsers, which is not always how the W3C specifies it should be done. The HTML-Parser will also allow us to turn this feature off, which will allow us to check against the standards of the W3C which is what our tool is mostly about. [**?**]

2) HTML-TokeParser-Simple: The TokeParser parses HTML and tokenizes the HTML page. This is another popular way to parse HTML pages, but for our intents, it may not be the best solution. We don't care much about tokenizing the input as much as verifying the input is standards compliant as well as throwing it into our data structure. Another problem with this tool is it says the tokens are not always intuitive to parse, meaning there could come extra overhead to get this to work. [**?**]

3) Parse-RecDescent: The RecDescent parser isn't really a parser, it's a parser generator similar to GOLD mentioned above. It has most of the same pros and cons that GOLD did. We would have to pass it a grammar, which again probably wouldn't catch many corner cases because we would be creating the grammar. [**?**]

4) Decision: The HTML-Parser is our best stand-alone option. However, we are not limited to just one of these options. Using the HTML-Parser to parse our HTML and check if it is standards compliant would be the best use case. We could also use the TokeParser to tokenize our HTML input which might be extremely useful for building our data structure.

III. Zach Schneider

A. Data Structure Storage Medium

The text editor extension created for this project functions in part by parsing an existing code base. The parsed code base will be converted into a sort of data structure that can be more easily utilized and manipulated for helping the user, while not interfering with their actual files. The user's code base will be parsed periodically, but that parse function may be resource intensive, and as such, serialization of the parsed data into a more accessible form will be required. The following sections will evaluate SQL vs. NoSQL data storage options, as well as the method for converting the parsed data into a storable format.

1) SQL Databases: SQL Databases have historically been the go-to method for data storage for many computing applications. SQL has been around for many years, resulting in great familiarity with it and its resources among developers and our team. SQL databases are often based on tables and predefined schemas, contrasted with NoSQL databases which often lack a predefined structure and consist of key-value pairs. The primary issue with a SQL database in our project is the overhead required in installing and maintaining a local database. Databases may be hundreds of megabytes just to install, and setting up a database connection may not be natively supported in or application. For these reasons, our extension will opt not to use a SQL database. [**?**]

2) NoSQL - JSON: In contract to a more structured SQL database, NoSQL databases may provide the flexibility, native support and light weight file structure our team needs to efficiently store parsed user data. NoSQL databases often store data in the JSON format, a human and machine readable file that is ideal for temporary data storage and exchange. JSON is built into JavaScript, one of the languages that will be used in our extension, meaning no additional setup will be required to send parser data to this storage format. Additionally, JSON is natively supported by most modern text editors and browsers, including the one being considered for this project. JSON can store complex JavaScript objects and variables, making it the ideal choice for containing the data structure utilized by our extension. [**?**]

3) NoSQL - BSON: BSON is a binary-encoded JSON data format utilized by MongoDB, a NoSQL database. BSON provides additional data types to the database for "efficient encoding and decoding within different languages." [**?**] MongoDB is able to manipulate BSON files to create additional structures inside the file, even after object hierarchies have been created. MongoDB is a free and open-source database solution that uses BSON documents for storing data. It is commonly used in big-data or real-time applications, often used in conjunction with node.js and other JavaScript technologies our development team is familiar with. Two main problems exist with BSON: it is not human readable, making it harder to debug, and it requires a full installation of MongoDB, which brings the same space and overhead concerns mentioned before. These additional challenges have convinced the team to stick with a purely JSON solution. [**?**]

B. Data Structure Storage Serialization

1) JSON.stringify: Since the text editor will be written in JavaScript or a superset of it, using JSON as the method of storing objects seems the most obvious. JSON text is valid JavaScript code and is supported in many modern languages and IDEs. JSON is also plaintext, which requires significantly less overhead and space than a full database. As of the ECMAScript 5.1 standard (2011), the JSON object in JavaScript has a built in stringify and parse function, which will serialize and deserialize JavaScript variables and object respectively. As this function has been officially supported for many years and has multiple sources of documentation and example usage online, it will be the primary choice for how the extension stores files related to user projects. [**?**]

2) serialize-javascript: A shortcoming of JSON.stringify is that it does not allow actual JavaScript functions to be serialized into JSON, nor does it allow regular expression statements. The serialize-javascript npm package serves as a superset of JSON.stringify while also including the aforementioned functionality. Additionally, serialize-javascript will auto escape HTML code, making the JSON safe to display on webpages in raw form. Npm is supported by dozens of IDEs and text editors, so compatibility will not likely be a problem. The largest drawback of using serialize-javascript is its lack of documentation, despite its mild popularity. There are fewer examples of its usage online than JSON.stringify, and it adds another external dependency to our extension. For these reasons, serialize-javascript will not be a part of the storage system in the extension. [**?**]

3) JSON-js: JSON-js, also known as JSON in JavaScript is the former implementation of JSON support for JavaScript in web browsers. It provides conversion of JavaScript data to the JSON format, as well as the reverse parsing functionality. It is backwards compatible for browsers all the way back to Internet Explorer 8 with its json2.js file. However, these files have been superseded by native JavaScript support of JSON since ECMAScript 5 was made the web development standard in 2009. Even Douglas Crockford, the author of this utility now recommends against its use. For this reason, our team will opt for the natively supported JSON.stringify function. [**?**]

C. Comparing the Data Structure to the Serialized JSON

Once the parsed user code base has been serialized into a JSON file, the extension will then need to periodically check if the data objects in memory are different than the objects saved to the disk in the JSON file. There are dozens of methods to compare objects in JavaScript, some prioritizing speed with others focusing on functionality. Our team opted for somewhat of a middle ground, while leaning towards functionality and ease of development when comparing the Lodash library, the JSON.stringify comparison function, and the deepEqual npm module.

1) Lodash: Lodash is a JavaScript library the provides a wide range of functionality while focusing on performance. Lodash abstracts iteration through arrays and objects to maintain speed while simplifying the experience for the developer. The library contains functions such as .cloneDeep() and .isEqual() which allow for deep object comparison (instead of the native ===), as is needed for this extension's circumstances. Lodash has continuing developer support and sufficient documentation throughout web. The main detraction for using Lodash is that it creates a new dependency on an entire library, something our developers would have liked to avoid. However, the benefits of Lodash seem to have outweighed that primary flaw enough for this library to be our choice method of data structure comparison. [**?**]

2) JSON.stringify: JSON.stringify, as previously mentioned, is native to JavaScript and will be our primary method of serialization to JSON files. The usage JSON.stringify(a) === JSON.stringify(b) can compare more deeply than === alone, caring about the contents of the objects it serializes rather than the structure or referential equality. JSON.stringify can also compare more efficiently than the deepEqual package according to this article. [**?**] The main downside in relying on JSON.stringify to compare our data structure to existing JSON files is that the rest of the legwork in finding what was different would still be up to our team to develop. Lodash, instead provides much of this functionality, leading to it being chosen over JSON.stringify. [**?**]

3) DeepEqual: The deepEqual deep object compare function is a free library available via npm. It is quite popular among JavaScript developers and has a decent amount of documentation online. Like JSON.stringify, it does a deep comparison on two JavaScript objects checking for differences. However, it is not only slower than JSON.stringify, but it also has the same lack of functionality that JSON.stringfy suffers from. For these reasons, our team opted not to use deepEqual. [**?**]

IV. Cramer Smith

A. The Best Base Integrated Development Environment

It is necessary to look all the possible integrated development environments (IDE) for the postal project to see what would be the best fit for this specific extension, both development wise and release wise. To this purpose this review will examine three similar IDEs that could all be used as the base of the extension that is planned to be implemented. Those IDEs are Brackets, Atom, and Visual Studio Code (VSCode) all of which have there advantages and disadvantages.

1) Brackets: Brackets is a text editor owned by Adobe, and is currently in development as an open source project. [**?**] Brackets is made specifically for web development, and offers tools such as the Chrome debugger, inline editor, and live website previewer built in. The research into the Brackets IDE has brought to attention many features that were good, but also some that were not as good. Adobe started the development in 2011 making Brackets the oldest of the IDEs research in this paper meaning it is the most established and it has had more time to become a more stable build. The longer life of Brackets could also explain the more extensive documentation that it has when compared to other IDEs extension development documentation. Another nice feature of Brackets is that it has dedicated API functionality that allows for extensions to safely modify the underlying Document Object Map (DOM) which would be very useful for the Postal extension. While there are several benefits to Brackets there are some drawbacks specifically being the extension debugging and lack of usability in the extension manager. The extension debugging consists of having another development environment open with the extensions code and restarting Brackets with every change. This restarting will get tedious after prolonged development. The other issue is the extension manager in brackets is not user friendly, it is basically a list of extensions and a search bar. With the target audience Postal is trying to reach is one of rather new web developers, people who do not want to sift through extensive lists of confusing extensions. Brackets would make it more difficult for our product to get to the users.

2) Atom: Atom very similar to Brackets in that it is a text editor developed by GitHub that is currently in development as an open source project. It features cross platform editing, a built in package manager, and smart auto completion. Atom advertises itself as the 'hackable' text editor meaning that it is made using HTML, CSS and JavaScript in such a way that anyone is using the text editor for web development then they should be able to develop for Atom. [**?**] This is a commonality between all of the possible IDEs. Atom is actually what VSCode from, but VSCode added TypeScript to the languages that it is built in. While atom is a good has a editor and auto completion it does not stand out when compared to the other IDEs. In fact it seems as though Atom uses extensions as a crutch not implementing built in functionality requiring the user to get extensions in order to complete their tasks. While this does make the initial learning curve of using Atom a bit larger it does keep the editor light and running very fast. What Atom gains in speed it loses in ease of use as a new user has to do a good amount of initial set up and for the audience that Postal is targeting would be put off by the confusing initial set up of Atom.

3) Visual Studio Code: Visual Studio Code, like the other IDEs, is a text editor by Microsoft, that is currently in development as an open source project. [**?**] It is the editor that was initially proposed as the base of the Postal extension. This initial idea to using VSCode was a product of the team having worked with Visual Studio and enjoying the experience. Now that the team has tried working with Visual Studio Code there has been some benefits and drawbacks identified. The benefits were that VSCode has justifiably better extension debugging than the other IDEs available. VSCode seems to build around the idea that people will be making extensions for Visual Studio Code so there is a development window that is created when debugging extension code within VSCode. The other IDEs seem to be less approachable with a system that makes the developer reinitialize the IDE every time the extension's code is changed. The other benefit of using VSCode was that the extension can be written using TypeScript, but that being said none of the team members have used TypeScript meaning it would be additional learning curve added on to the obstacle of learning more JavaScript. The first of the drawbacks of VSCode become evident when working with the documentation and finding that Visual Code is fairly new, and does not have a lot of documentation or examples of extensions to readily examine. As this team is fairly new to creating extensions and not efficient at writing in JavaScript or TypeScript this is going to be a problem. Visual Studio Code also had no built in API for modifying underlying DOM that make up the main user interface which the postal extension possibly could do.

4) Decision: All this being said the team has decided to use Visual Studio Code based on several major benefits. Visual Studio Code's use of TypeScript and being able to import .NET libraries. All of the team members have at least some experience with .NET libraries, and these could be used to greatly improve the project. The Visual Studio Code also has the best Extension Debugger than the other IDEs and a more defined extension creation process. This is important for the integrity of the development of the extension in the long run. Visual Studio Code has a extension creation guides the developer through the process of making the extension and creates all the helper files that the developer will need in their extension. The other benefit is that the built in extension manager will be easy for the user to manage than the other IDEs.

B. Event Handling Within A Separate Window of the IDE

Visual Studio Code is going to interface with the user and our extension will need to know what the user is doing. The specific instance that this portion focuses on the the event handler that the IDE is actively listening for from it's extension. The Postal extension needs a way of getting information from the extension, this section will explore the options that the VSCode extension API offers developers for this kind of interaction. The events that the IDE will be listening

1) HTML Preview Links: The first way that VSCode makes it possible to have events is through the HTML Previewer. [**?**] This method would require that the extension makes an HTML page and then display that page either in a web browser or using the built in command previewHtml using the vscode.executeCommand() and passing it an Uniform Resource Identifier (URI). The URI that would be past to the command would have a HTML DOM that would be rendered in a separate panel on top of the VSCode's HTML and CSS. This seems like an easy way of creating a user interphase, but the only intractable element would be links. Links would only be able to change the the view to other HTML files. The idea of making every possible use case a separate HTML file would be difficult, and extremely tedious.

2) IDEs Built In Events: Visual Studio Code has a number of already built in events that the extension could listen for but these events fire at specific timing such as when the user saves a file or edits a document. The Extension could fire these events that are built in to Visual Studio Code, and have handlers set up to listen to those events using the already built in listeners. This method would be extremely problematic. These built in events are meant to be used at very specific times and have set listeners that parts of the IDE itself, and other extensions are expecting at very specific times. Using the events would be an incorrect solution as these events happen at every specific instances and using these events for any purpose other than the purpose that they are currently made for would be irresponsible and would not work properly.

3) EventEmitter: The better option would be to use the VSCodes EventEmitter to create and manage for other to subscribe to. With an eventEmitter the extension can create listeners that can then be listen for when the event fires signifying to the listeners that the event has occurred. This is the intended way that exertions are supposed to create and the event handlers. There are also built in events that the these event listeners will need to be initialized before the user actually interfaces with the extension and to do that the extension will need to use the activation events. An activation event is set in the package.json of the extension and these activation events are sent from the IDE to the extensions. These events can be onLanguage, onCommand, onDebug, workspaceContains or a star (*) signifying that the extension should always be running. When the extension receives one of these events that it is listening for it will start the extension operations. Setting these custom eventEmitters and event listeners should be the first thing that the extension sets up as they will be very important for the interface that the user will interacet with. [**?**]

4) Process Bridge: The process bridge is a way of connecting VSC to the electron GUI that we use to draw the nodes. With the process bridge we will be able to set up event listeners for the user interacting with the GUI elements. This way we will be able to have the extension and VSCode react to the users actions in the electron window. The process bridge will be able to communicate those actions to the IDE.

5) Decision: For the purpose of this project the eventEmmiter and Listener will be used for anything that is contained within VSCode for obvious reasons, but we will also need a process bridge that will send messages back and forth to the GUI and VSCode. The first reason that eventEmmiter was chosen was that it is the standard of Visual Studio extensions, and this is what extensions are expected to implement. The HTML preview method would be extremely limited, tedious, and impractical, and the built in events would go against the design of the IDE and would cause conflicts that would be extremely problematic. Make custom eventEmitters will allow for greater flexibility and better design in general. The reason for the process bridge is that it is the only way for the GUI and VSCode to properly communicate users actions taken on the GUI. Having the process bridge will make the GUI actually respond like a proper GUI.

C. The Language The Extensions Written In

All the possible IDEs that could be used for this project are written in HTML, CSS, and JavaScript, but the bulk of the logic is JavaScript. They all use these languages because they are the best for creating beautiful and consistant cross platform applications. This means that the extensions use mainly Javascript. With Visual Studio Code there is the option to use either JavaScript or TypeScript. This portion will look at JavaScript and TypeScript looking at each language's pros and cons in terms of the development of the Postal extension.

1) JavaScript: JavaSctipt was created in 10 days in 1995.[**?**] It is a well known, and well documented language. With all the possible IDEs the extension can be written in JavaScript. It can be very confusing at times as it is an asynchronous language and not compiled inline. This compounded with the fact that all the extension documentation for VSCode is written with TypeScript in mind making the learning curve for making an extension in JavaScript that much steeper. The upside would be that there are in more people programming with JavaScript and that it has a larger community to gather information from when compared to TypeScript. This would give the Postal team more resources when looking for solutions to the inevitable problems that will come up during the development process.

2) TypeScript: TypeScript is a language created by Microsoft with the purpose of being 'JavaScript that scales.' [**?**] In fact TypeScript compiles to JavaScript and its syntax is almost identical to JavaScript. The main advantage of using TypeScript is that it adds types to JavaScript allowing for static checking and code refactoring when developing JavaScript applications. [**?**] Another major benefit of using TypeScript is the ability to import .NET libraries. This allows for developers to use more powerful APIs then the some of the less robust JavaScript APIs. For this project that will allows the the extension access to APIs that the team has experience using. Another major advantage of using TypeScript is that all the VSCode examples are already written in type script making understanding the documentation easier to understand. TypeScript also allows for the use of JavaScript within TypeScript files and that flexibility will be nice to have while developing an extension.

3) Decision: For the postal extension the development will be done with TypeScript as its main programming language. The main reason for this decision is that the documentation for VSCode extension development is written with TypeScript examples making for a smoother development experience. The other reasons for choosing TypeScript over JavaScript is that TypeScript will allow for the use of both JavaScript and TypeScript. Have the flexibility to chose either depending on the circumstance will make for faster development with more possible solutions to some problems that the developers may face.

V. Eric Winkler

A. Rendering the File Map

The file map is the visual representation of the user's current project. It will most closely resemble a web of nodes where the nodes are individual files and the edges are confirmation of some sort of link or dependency between two files. Whatever technology we use for this must be capable of running the VS Code environment (JavaScript).

1) vis.js: Vis.js is a “dynamic, browser based visualization library”. It features several modules for visualizing data in a JavaScript application. One of these modules, the Network module could serve to visualize the file map in a web like format. The Network modules features the ability to click and drag the map around which will be useful for navigation. Even better is the modules ability to zoom further into the map and dynamically change text size of the nodes depending on the zoom level. This is an extremely beneficial feature as it allows the system to draw many smaller nodes without having to deal with the concern of the user being able to clearly read each node at once. Additionally, the module appears to be highly customizable in terms of aesthetic.

Vis.js claims to “run fine on Chrome, Opera, Safari and IE9+.” However as the project is being developed with the intent to within visual studio code, Its ability to run in that environment is questionable. This is currently the primary concern with this option.

Vis.js is open source and is dual licensed under both Apache 2.0 and MIT.

2) d3js: Similar to vis.js, d3js is another javascript library designed to visualize arbitrary data. Unlike vis.js, it allows the user to bind processed data directly to a Document Object Model (DOM). This essentially means that it is more flexible in its ability to read in data. It also features an output to an SVG file type, which could be useful as we know Visual Studio Code is capable of displaying SVGs within the environment.

D3js features a large massive number of what vis.js referred to as modules. There are several suitable modules that D3js offers, but the most promising appears to be either the force-Directed Graphs or the Curved link graphs. Like vis.js, D3.js features zooming and panning. The API reference appears to be a bit less straight forward than vis.js and the library descriptions claim to prioritize making the library light weight and efficient. This option has the potential benefit of being more efficient, but will likely be slightly more difficult to integrate.

D3.js supports “modern browsers” (Firefox, Chrome, Safari, Opera, IE9+, etc.) and also runs on node. The library is available under the BSD License.

3) Custom Code: Our final option is to write custom own code for generating the file map. This option is massively more work than the first two but still has some benefits. Writing custom code allows us more flexibility in how the code runs and allows us to keep the size of the extension to a minimum. Every feature implemented in the custom code will be out of necessity. Additionally, it adds the benefit of not having to worry about properly citing the code according to the licenses above. This is a relatively minor benefit, but also means that copy write will never be an issue for this corner of the project.

B. Displaying Broken Rules

In addition to displaying the file map, the UI will also feature a list of broken rules to one side. These broken rules are pseudo-error our parse detects within the project directory and should be displayed alongside the map. The displays should be as aesthetically pleasing as possible and will ideally provide an API a hook for triggering an event when the user clicks on an object in the display. The errors will be stored as strings, so there should be relatively little complexity in passing the data to an API.

1) JQuery Advanced News Ticker: The JQuery Advanced News Ticker is a JavaScript based JQuery plugin. It claims to provide several callbacks and methods to allow maximum flexibility and implementation. It is called through CSS classes and does feature an API hook for triggering a JavaScript event on clicking an object. It being a JQuery plug in should allow it to work within Visual Studio Code. It features a relatively through API guide and install instructions.

The example news tickers it displays an aesthetically pleasing and varied, indicating that customizing the displays is possible. It features a standard vertical list which will suit this projects needs but also an interesting call out version. The version has on expanded item in the list while the rest are collapsed, saving a decent amount of screen space. This may be something to look more into if the project uses this API.

JQuery Advanced news ticker is available under the GNU Public License.

2) nanoScroller.js: NanoScroller.js is similar to the JQuery Advanced News Ticker in that it too is a JQuery plugin. Its goal is to offer a very simple and uncluttered way to create scrollable divs that are visually pleasant. As such it features far less in terms of animation and visual 'coolness' but is far smaller in size. It is likely a much more efficient option in comparison to JQuery Advanced News Ticker as well. It will accomplish the minimum in terms of displaying errors within the UI. A downside of the plugin is that it doesn't feature explicit hooks for triggering a JavaScript event when something is clicked. This is to be expected as it is basically just a div class but it will still be something we will have to implement ourselves if we go with this option.

Documentation is fairly thorough and the website offers a website compression file which will help keep the extension size to a minimum.

NanoScroller.js is available under the MIT license.

3) Custom Code: The final option is to create our own div class. As discussed above in the file map technologies section, this will be quite a bit more works but has quite a few benefits. Again, the project team will have more control over design, efficiency and features than if an API was used. Additionally, code created would belong to the project and the need to cite could would not be a concern.

C. Technologies for displaying the file map

The file map will be created in JavaScript but there are multiple options for choosing where to actually display it in a desktop environment. Any technologies used in this process should be capable of handling a HTML or JavaScript UI application.

1) Visual Studio Code HTML Preview: In the initial plan for the project, the file map was to render within the Visual Studio Code environment. This option is still valid and the team has already demonstrated from a code perspective in the last prototype that this can be accomplished. The actual process for doing this is slightly unusual however. The code to do this is HTML Preview which is a built in functionality to vs code and is typically used to preview a website. By dynamically creating an HTML page, we can simulate a GUI within VS Code. A downside to this is that the team has already noticed some slightly unusual behavior which may need to be tweaked. Additionally, the size of the window that is used to render the HTML is slightly difficult to control. There is also a major advantage of using the HTML Preview however. The first is that, by using built in functionality, the extension can remain more compact and not have to support itself.

2) In Browser Window: Through the use of some Visual Studio Code API Calls, we can have the IDE launch the user's default browser and load our rendered file map into it. The advantage of this methodology is that the UI can be constructed entirely as a webpage. This provides us with many simple to implement options for creating an interactive UI. The down side however is that the web application UI will have to support multiple browsers and additionally the use of an external program for the extension will make usability inconvenient.

3) Electron: Electron is a framework for creating desktop applications through the use of HTML, CSS and JavaScript. It is kind of like a lightweight browser that doesn't browse but instead servers as a container for applications made with web based technologies. Interestingly enough, Electron was used to create visual studio code. Using Electron would allow us some freedoms for the UI that might otherwise be restricted by Visual Studio Code's preview HTML Option. As mentioned above, testing the preview option led to some slightly bizarre behavior. Electron could potentially fix those issues. The primary disadvantage would again be that the extension has to include the entirety of the electron framework, making it significantly larger.

VI. Revisions

A. *Sam Lichlyter*

1) Parser Class: The parser class we ended up using was a custom implementation.

2) Parsing Language: Because we used a custom parser class, it was easiest to use JavaScript because the entire extension is written in JavaScript using NodeJS.

3) Perl Parsing Class: This section is now irrelevant since we are using JavaScript for our custom parser.

B. *Zach Schneider*

1) Data Structure Storage Medium: The team has continued to use JSON files as the single storage medium for the data structure. This is unchanged from the original justifications above.

2) Data Structure Storage Serialization: The project has exclusively used JSON.stringify as the method of serialization from the data structure to our JSON file storage. This is unchanged from the original justifications above.

3) Comparing the Data Structure to the Serialized JSON: Throughout the development of this extension, the team has realized that reading and writing a relatively small JSON file in its entirety causes minimal if any delay in the function of the extension. As such, the decision was made to remove to compare functionality from our parser and data structure. The write to the JSON file by the parser and the read of the file by the UI cause very little taxation of the CPU/RAM of each of our test systems. Therefore, both components will just read/write the entire file each time a node is changed, instead of only reading/writing the changed portion of the file. Subsequently, the Lodash library has been removed from this project.

4) Switch to ECMAScript 7 from ECMAScript 6: The team decided to update the version of JavaScript that the extension will use. The version is described by the ECMAScript 2016 specification and is known both as ES7 and ES2016. The previous version we used was ES6/ES2015. This will not affect the end user in any way to our knowledge. The reason behind the switch is that ES7 includes extended array manipulation functions built in to the language, helping simplify some of the logic between the parser and the data structure. The switch was executed by changed an extension dependency from ES6 to ES7 in a configuration file. [**?**]

C. *Cramer Smith*

1) The IDE: The team has decided to continue with the use of the Visual Studio Code Integrated Development Environment. This is unchanged from the original justification.

2) The Event Handlers: The team has added the process bridge to the way that the GUI handles events. The need for this process bridge along with eventEmmiters comes from the way that the GUI is drawn separately from the Visual Studio Code and to have these two pieces communicate we need the process bridge. The eventEmmiters will be used if there are any events that happen within any of GUI built into Visual Studio Code.

3) The Languages: The team is still using mainly Typescript. This has allowed us to use both typescript and javascript in the development process, which was stated in the original justification for the use of Typesrcipt. This also allows the development team to use NodeJS and its many useful libraries.

D. Eric Winkler