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

Spring Midterm Update

Team Postal — Group #38

Cramer Smith, Sam Lichlyter, Eric Winkler, Zach Schneider

Abstract: Developer tools are often complex pieces of software. Gathering and manipulating useful information for a programmer can often be a slow and costly process. By implementing Information Foraging Theory design patterns in the creation of these tools, the information collected may be more useful or produced faster. Information Foraging Theory is the theory and math behind the choices people make to maximize the value of the information they find versus the cost of getting that information. The aim of this project is to develop a tool that will act as a proof of concept to this idea and increase developer efficiency. Through the implementation of multiple IFT design patterns, the Postal team will create a developer tool that helps enforce and maintain code structure.

CONTENTS

I. Project Purpose and Goals

Developer tools are often complex pieces of software. Gathering and manipulating useful information for a programmer can often be a slow and costly process. By implementing Information Foraging Theory design patterns in the creation of these tools, the information collected may be more useful or obtained faster. Information Foraging Theory (IFT) is the theory and math behind the choices people make to maximize the value of the information they find versus the cost of getting that information. The aim of this project is to develop a tool that will act as a proof of concept for IFT and increase developer efficiency.

Our project (code-named ”Postal”) is an extension for Visual Studio Code. It is being designed to allow developers to more quickly search through and better visualize their projects. This extension will also help developers create clearer and cleaner code structure by offering reminders and suggestions about the best coding practices in the programming language they are currently using. Any major errors or incompatibilities within the project or its files will be reported to the developer as well.

II. Current Status

A. Project Status

As of the end of week 6 of Spring term, all requirements of our project have been met, the extension itself is fully functional and most bugs in the codebase have be either resolved or negated. Our extension is available for download and installation from the Microsoft Visual Studio Marketplace and is labeled as our 1.0 release. Previous alpha and beta versions had been available on the Marketplace but lacked some of the final functionality or had significant bugs present. Project Postal works and has been tested on all 3 major OS platforms, Windows 7/10, MacOS and Ubuntu specifically. The extension can parse and visualize most major programming languages, having been tested primarily on C, C++, HTML, CSS, JavaScript, TypeScript and PHP. Default grammars (a series of regular expressions that search for language-specific keywords or tags) that allow the parser to process and visualize most of these languages have been included with the extension installation. The program is highly extensible, allowing any user to add to existing language grammars or create new languages grammars themselves. To this effect, the team intends to demonstrate Project Postal at the Engineering Expo with grammars not included in the default installation, so as to show how grammars can be designed toward specific codebases.

Two deficiencies of note remain in our project that were not able to be resolved by the week 5 code freeze: a separate Node.js installation and minor visualization bugs. At this time, the Postal Extension requires Node to be independently installed on the user’s machine in order to work. The reason for this additional installation requirement is that our display and window system, Electron, is too large of a package to be bundled in with a Visual Studio Code extension. As such, the user has to use Node’s ’npm install’ command to acquire Electron separately. The other remaining bug of note in our project is an occasional inconsistency in how a given codebase is visualized. Postal uses the Vis.js package to create a network of interactive visual nodes that represent files in the user’s codebase. Occasionally, Vis will generate a network that has some file nodes set to display in a location far from the rest of the network. This visualization bug can typically be overcome by either redrawing the network (a button

available to the user) or closing and reopening the Electron window. The team was not able to determine the cause of this problem, but we feel it does not significantly detract from the overall functionality of the program.

B. Testing Status

According to our agreement with our client, Prof. Chris Scaffidi, the team will conduct user testing to demonstrate whether the IFT principles with which the program was designed are effective. The team has both written up a series of tasks for a set of users to perform with and without the Postal Extension, as well as a post-testing questionnaire that will gauge the whether the users felt the extension helped them achieve the tasks faster or more effectively. These tasks and the questionnaire were both approved by our client and have been submitted to the university's Institutional Review Board (IRB) for human subject research approval. As of the end of week 6 of Spring term, the team is still waiting for response from the IRB. When approval is received, the team and our client will set up a selection of users to test our program. The actual testing procedures have been defined in the User Testing section later in this document.

III. Remaining Work

IV. Problems and Solutions

A. Fall Term

B. Winter Term

C. Spring Term

V. User Testing

A. Testing Description

The test in its current state contains two sets of tasks that go with two specific software projects. One is a C++ project and the other is a HTML and PHP project. Both of these projects are about the same size and are of similar complexity. The test subjects will go through the projects completing the tasks that we have designed to measure the effectiveness of the use of IFT.

The tasks are as follows:

For the **C++ project**

1) Locate the line at which void RenderingDirector::rdRender() is defined.
2) Find the number of functions in the snake.cpp file.
3) There is a function that is defined in Snake.cpp, but is never used. Name that function.
4) Find all the TODO comments in the program.
5) Identify the return type of the Snake::Turn function.
6) Find the number of times the Food class referenced in the gameplay-director.cpp file.
7) Find the number of .h files included in main.cpp.
8) Locate the file in which the Punish() function is defined.

For the **HTML and PHP project**

1) ?index.php? is the homepage for this web application. Over time, various features and pages were added and removed. This resulted in some files that exist within the directory that have been deprecated (they are not used within the index page or any of the pages linked to index.php). Locate all the files that are deprecated. (select all that apply check list)
2) Find the number of ?div?s in header.php
3) Some website applications make use of several external references like jQuery and Bootstrap. Find all external references used in the project directory. (External references are links to outside resources that are not included within the project directory, and are hosted on external servers)
4) What file and line number is ?site-menu? declared in?
5) How many references are there to ?stylesheet.css? in the project?
6) How many images are linked in the entire project?

B. Measuring Results

We will measure how long it takes the tester to complete the tasks as well as how well they answer the questions as a percentage. For example if a user were to find all the TODOs they would receive a 100% for that question. If they missed one of the ten they would get a 90%. If the subject failed to find any of the TODOs they would receive a 0% for that problem.

We will also do a post test questionnaire. The questionnaire will pose the subject with a number of questions that the subject can answer on a spectrum from strongly agree to strongly disagree.

The questionaire is as follows:

1) I could effectively complete the tasks and scenarios using the Postal tools.
2) I was able to complete the tasks and scenarios quickly using the Postal tools.
3) The organization of information and interface elements of the Postal tools within Visual Studio Code was clear.
4) It was easy to use this tool.
5) I would use Visual Studio Code again.
6) I could think of multiple scenarios for using for the Postal tool.
7) I would use the Postal tools within Visual Studio Code again.
8) Seeing a project visually was useful.
9) I felt the file link feature[1] allowed me to complete the tasks more quickly.
10) I felt the file link feature[1] allowed me to answer the questions with more confidence.
11) I felt the expand feature[2] allowed me to complete the task more quickly.
12) I felt the expand feature[2] allowed me to answer the questions with more confidence.
13) I felt the notification feature[3] allowed me to complete the task more quickly.
14) I felt the notification feature[3] allowed me to answer the questions with more confidence.

[1] File link feature: Internal references to other files displayed as red lines within the node network.

[2] Expand feature: A right-click on any file node expanded the node network to include all sub-nodes (classes, functions, divs, etc.) of the indicated node.
[3] Notification feature: Notifications (regarding malformed code, incorrect styling, etc.) were indicated by a red circle attached to the associated files/nodes and described in the notification window on the right-hand side of the screen.
The questionnaire will help us to measure how the user feels about the usefulness and general feel of the extension. The more that they strongly agree with the statements that will mean that the users enjoy and feel that the extension is useful.

C. Report Rough Draft: Information Foraging Theory Improving the Rate of Information Gain in Software Development

1) Information Foraging Theory: Information Theory is the idea of turning information into weighted directed graphs and then implementing these graphs to better access the data
2) The Postal Extension: Postal is a proof of concept using IFT design patterns to present data to the user will improve the rate that the user can locate and navigate the information.
3) : We hypothesize that the use of IFT in the Postal Extension will prove to increase the productivity and efficiency of programmers navigating around code and locating different parts of the code.
4) Experiment Procedure:
5) Measuring Results:

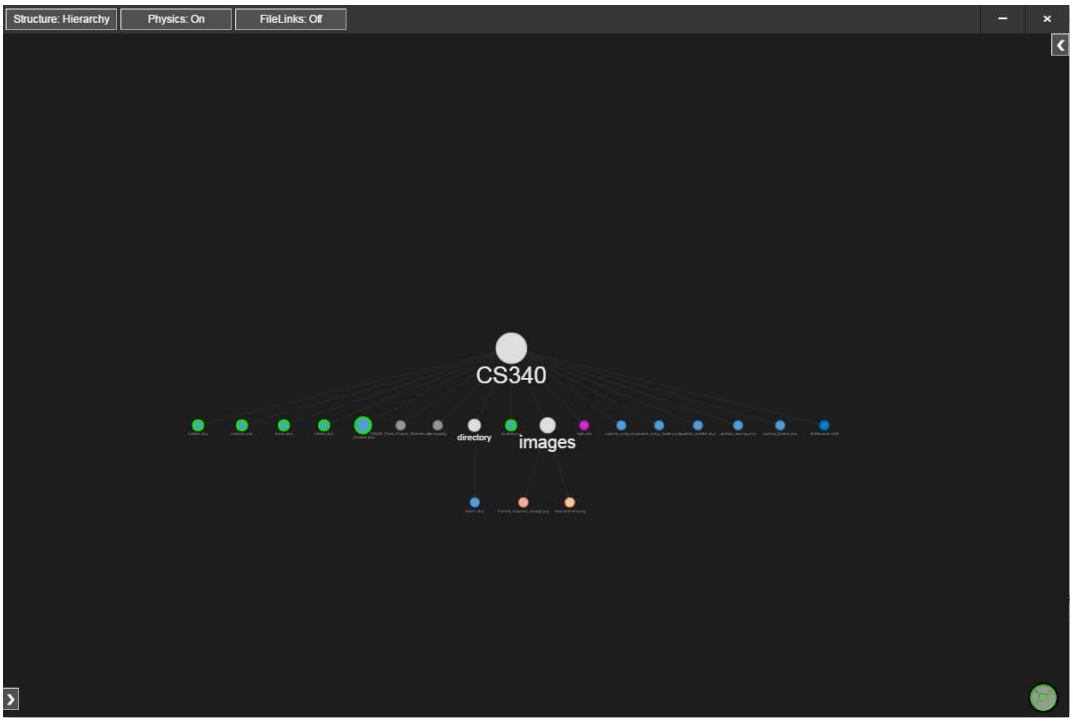

Figure 1. Visualization Interface

6) Conclusion:

VI. Images

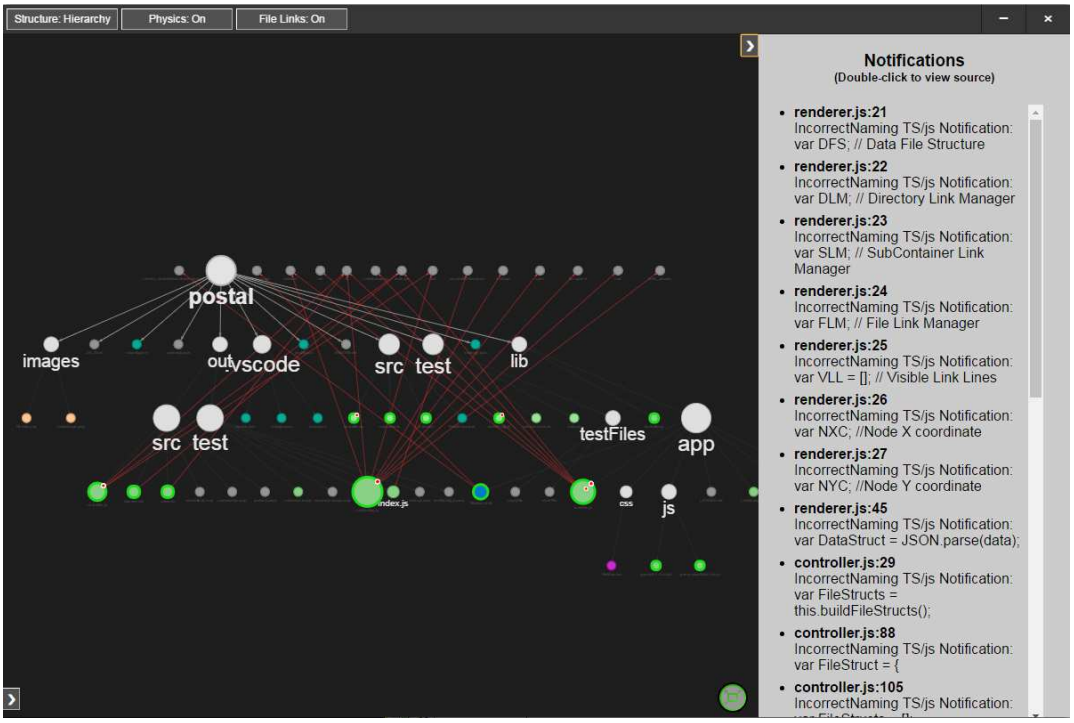

Figure 2. Notification Interface

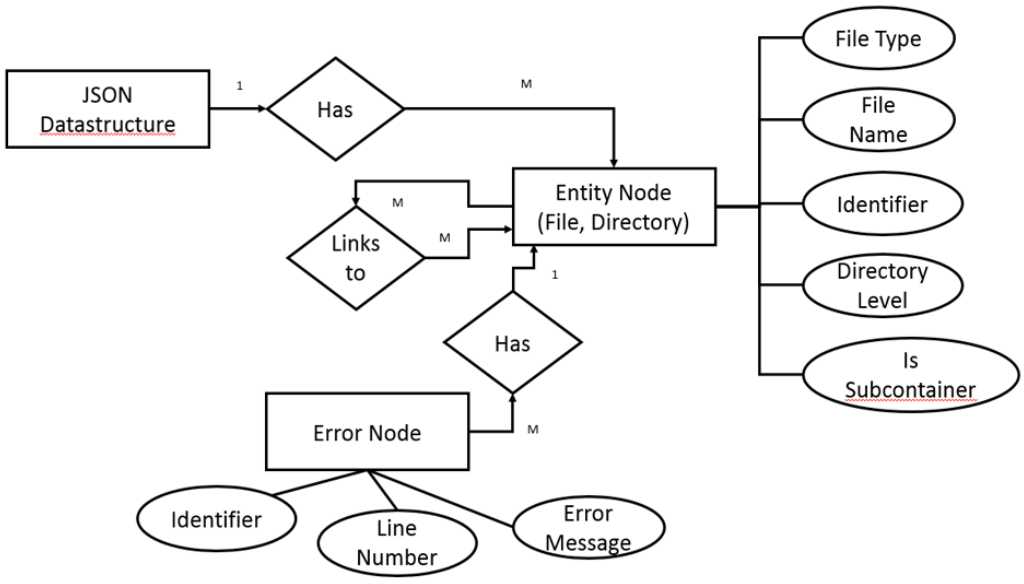

Figure 3. Updated Data Structure

VII. Code Samples

A. Example Grammar

This is the default grammar that parses HTML and PHP files for divs and links.

```
{
   "id" : 0,
   "title" : "html",
   "filetypes" : ["html", "php"],
   "rules" : [{
         "title": "div",
         "type": "tagged",
         "options" : {
            "tagStart": "<div",
            "namedOption" : "id=\"(.+?)\"",
            "tagEnd": ">",
            "closingTag": "</div>",
            "nodeColor": "blue"
         }
      }, {
         "title": "href link",
         "type" : "link",
         "options" : {
            "link": "href=[\"](.+?)[\"]",
            "nodeColor": "blue"
         }
      }, {
         "title": "includes link",
         "type": "link",
         "options": {
            "link": "include=[\"](.+?)[\"]",
            "nodeColor": "blue"
         }
      }, {
         "title": "body",
         "type": "tagged",
         "options" : {
            "tagStart": "<body",
            "tagEnd": ">",
            "closingTag": "</body>",
            "nodeColor": "blue"
         }
      }
   ]
}
```

B. *Recursive Get All Links*

This function grabs all the links from the data structure of a specified file struct and it's children.

```
// Recursive function to get all links from this and children
function getAllLinksFromFileStructRecursive(FileStructID) {
   var links = [];

   // check parent
   if (DFS[FileStructID].links.length > 0) {
      for (var i = 0; i < DFS[FileStructID].links.length; i++) {
         var link = DFS[FileStructID].links[i];
         links.push(link);
      }
   }

   // check children
   if (DFS[FileStructID].subContainers.length > 0) {
      var childLinks = [];
      for (var i = 0; i < DFS[FileStructID].subContainers.length; i++) {
         var childFileStructID = DFS[DFS[FileStructID].subContainers[i].toFileStructid].id;
         childLinks = getAllLinksFromFileStructRecursive(childFileStructID);

         // push what we found to parents link list
         for (var j = 0; j < childLinks.length; j++) {
            links.push(childLinks[j]);
         }

      }
   }

   return links;
}
```