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

Winter 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.

The Postal extension 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

B. Testing Status

III. Remaining Work

IV. Problems and Solutions

A. Fall Term

B. Winter Term

C. Spring Term

V. User Testing

A. Testing Description

B. Measuring Results

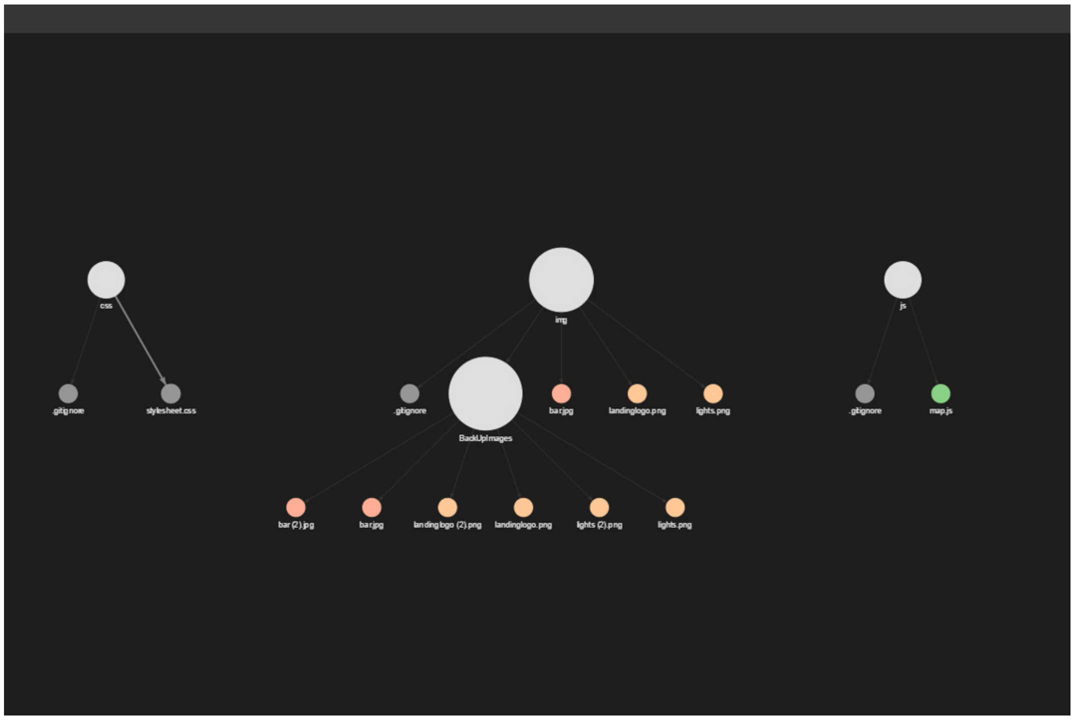

Figure 1. Current User Interface

VI. Images

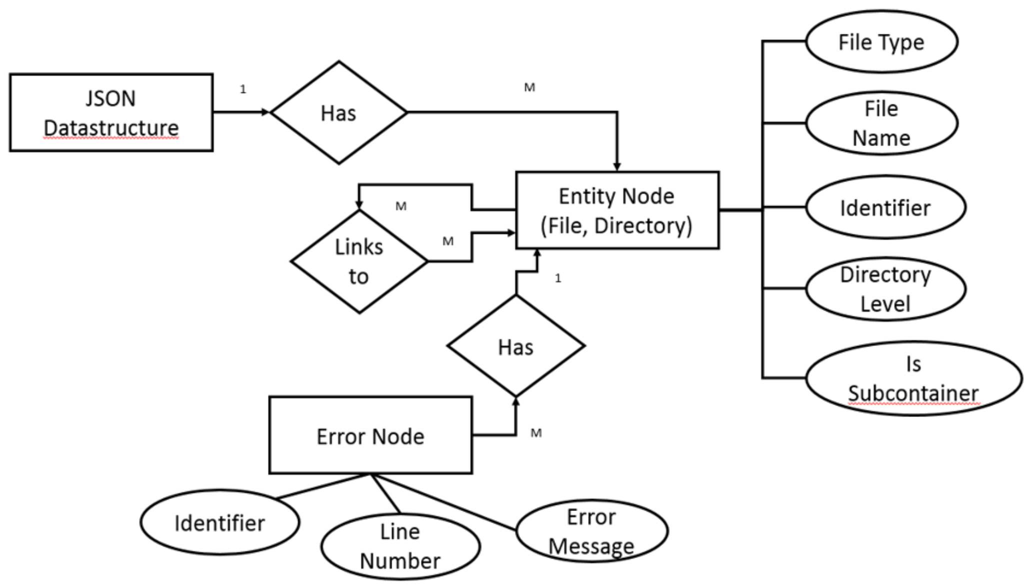

Figure 2. Updated Data Structure

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;
}
```