University of Zurich[UZH]

Department of Informatics IfI
University of Zurich
Binzmühlestrasse 14
CH—8050 Zürich
Switzerland
URL: https://www.ifi.uzh.ch

Instructors:
Prof. Dr. Alberto Bacchelli
Prof. Dr. Burkhard Stiller
Prof. Dr. Christoph Lofi
Dr. Marco D'Ambros
Assistants:
Shirin Pirouzkhah

Fundamentals of Software Systems (FSS) 03SM22MI0002

Software Evolution - Part I Assignment

Submission Guidelines

To correctly complete this assignment you **must**:

- Carry out the assignment in a team of three students.
- Carry out the assignment with your team only. You are allowed to discuss solutions with other teams, but each team should develop its own personal solution. A strict plagiarism policy will be applied to all the artifacts submitted for evaluation.
- Provide solutions to the exercises. Each solution will consist of source code, explanations (e.g., of decisions taken), and data files (e.g., JSON files):
  - The explanations must be written in a single PDF file.
  - The students' names in your group, source code, data files, and explanations must be uploaded to OLAT as a single ZIP file by Nov 25, 2024 @ 23:55.

AACSB ACCREDITED
EFMD EQUIS ACCREDITED
Research Evaluated by INFORMATICS EUROPE

Background

React[1] is a declarative, efficient, and flexible JavaScript library for building user interfaces. It's maintained by Facebook and a community of individual developers and companies. React has a large and active codebase[2], making it an interesting candidate for analysis in the context of software systems and quality.

Task 1: Component-Level Codebase Overview

A component in React is a self-contained unit or module that encapsulates a set of related functionalities. Each component manages its own state. Components allow developers to break down complex UIs into simpler, reusable pieces. They can also be imported or exported to and from different parts of a React application, making them highly modular and maintainable.

Checkout

- Using the latest stable release of React, v18.3.1, list all the components. In React, there are two main types of components: Class-based Components and Functional Components. Class-based components are defined where there is a class declaration that extends 'React.Component' or 'React.PureComponent'. Function-based components are JavaScript functions that return JSX elements (JavaScript XML, return elements enclosed in brackets `(return <...>)`). To find these components in the React codebase, look for any '.js' file, and scan the file content for patterns mentioned above (hint: use regrex to find the patterns)

- Use Madge tool[3] to detect dependencies between React files. Madge is a tool that analyzes dependencies in software projects making it easier to understand the complexities of file interactions within a codebase. Report all the dependencies in a JSON file. Use this JSON file and find and report the top 3 files with the highest number of dependencies. Provide the data in a JSON file with the following format:

Listing 1: Example of JSON records for dependencies

```
{ "FILENAME1.js": [],
  "FILENAME2.js": [
    "FILENAME3.js"
  ],
  "FILENAME5.js": [
    "FILENAME6.js",
    "FILENAME7.js"
    "FILENAME8.js"
  ]}
```

[1] https://opensource.fb.com/projects/react/
[2] https://github.com/apache/kafka
[3] https://github.com/pahen/madge

- Identify the commit that resulted in the most substantial change between v17.0.1 and v17.0.2 versions of React. Use Git commands to explore the commit history and quantify changes based on the number of files affected. Document the commit hash, number of files changed, number of insertions and number of deletions.
- Check out the repository at the commit hash identified in the previous step. This allows you to view the state of the codebase at the time of that commit. After checking out, use Madge to analyze the dependencies at this specific commit point. To trace how dependencies changed and evolved over time compare these dependencies to those detected before the checkout (i.e., in the current latest version). Document any new dependencies introduced, or dependencies removed.

Task 2: Temporal and Logical Coupling Analysis

Logical coupling often occurs because two seemingly separate parts of the code are functionally related. It can be detected by mining software repositories to see which files or modules tend to be committed together frequently over time. Temporal coupling can arise from development practices, such as when features are developed in parallel and merged at similar times. This is observed by looking at commits and/or their timestamps to identify files that are often changed together in commits or in close temporal proximity to one another.

Locigal Coupling: Files commited together
Temporal: Files commited in the same timewindow

- To analyze coupling, look at files that were changed in a specific time window. Analyse which files are changed together (Not necessarily the same commit) within a time window of 24, 48, and 72 hours of each other. Report your result in a JSON file with the following format. For each time window, report the top 3 file pairs with the highest degree of temporal coupling.

Json with All files

"Report" top 3 in pdf

Listing 2: Example of JSON records for temporal coupling

```
{"file_pair": ["FILENAME1", "FILENAME2"],
"coupled_commits":
{
    "time_window": 24,
    "commit_count": number_of_commits_on_same_day
},
{
    "time_window": 48,
     "commit_count": number_of_commits_within_2_days
},
// ... other time windows
}
```

- Identify cases of logical coupling by finding files frequently committed together. Report your findings in a JSON file and report the top 3 file pairs with the highest degree of logical coupling.
- Compare the top 3 pairs of logical coupling and temporal coupling with the dependencies you found in Task 1 and explain whether or not these couplings align with dependencies.

Task 3: Analyzing Commit Messages for Defect Indicators

Defective hotspots are areas in the codebase where defects are frequently found. This task utilizes a simple yet effective approach to identify these hotspots by analyzing commit messages for keywords that indicate defect resolutions, such as ”fix”, ”bug”, ”error”, or ”issue”. This analysis will provide insights into the frequency and distribution of defect-related activities within the project.

“go ahead and use the keyword you like”

- Use Git to extract all commit messages from the repository. Analyze these messages to detect the presence of specific keywords (For example bug fix) related to defect fixes.
- Compile a list of all commits that include the identified keywords. Quantify these results to determine which files are most frequently associated with these commits. Provide the data in a JSON file with the following format:

Listing 3: Example of JSON records for commits with identified keywords

```
[
        { "FILENAME1.js",10},
        { "FILENAME2.js",  7},
        ...
]
```

- Calculate the cyclomatic complexity for each file identified in this JSON file. Record these complexity scores and visualize the correlation between cyclomatic complexity and the frequency of defects. This analysis will help assess whether higher complexity correlates with a higher frequency of defects. Consider using tools like McCabe or Lizard for complexity analysis.