**Study Design**

*BSc Thesis:*

*"How Are Different Asynchronous Programming Constructs in JavaScript*

*Related to Software Quality? A Repository Mining Study on GitHub"*

**Research Design**

This research conducts a mining software repository (MSR) study. The source of open-source repositories is GitHub, a network-based version management service for software development projects. The selected language for this study is JavaScript, as the title of this thesis suggests. The GitHub REST API is used to obtain the data from the JavaScript applications (server-side, desktop, and mobile applications).

**Research Questions**

**RQ1: What are common bug patterns of using asynchronous programming constructs in JavaScript?**

* *Atomicity violation (a set of events / asynchronous operations are assumed to be processed atomically without interruption, but the atomicity is not enforced during execution)*
* *Order violation (two or more events / asynchronous operations, which access the same shared resources (e.g., variables, files) are expected to be processed in a certain order. The order is not enforced during execution)*
* *Starvation (tasks take a long time and prevent other events from processing)*

**RQ2: What characteristics influence the frequency of bug patterns of the analyzed asynchronous JavaScript projects? (root causes)**

1. Choice of shared resources. (*Shared variables, databases, files)*
2. Bug-inducing Phase *(Execution, Event triggering, event handling)*
3. Bug-inducing API *(Schedule API, High-level API protocol)*
4. Choice of project domain. *(Native applications, web applications, library, frameworks)*

**RQ3: How do asynchronous programming constructs differ from one another in terms of software quality? / What are the differences between the asynchronous programming constructs as they relate to software quality?**

1. Does using “callbacks” lead to less Functional correctness or Maintainability?
2. Does using “async/await” lead to less Functional correctness or Maintainability?
3. Does using “promises” lead to less Functional correctness or Maintainability?

*RQ4: What best practices should be followed when using asynchronous programming constructs in JavaScript to ensure software quality?*

*RQ5:*

**Sampling Procedures**

Automating the sampling a Python script is created. Using Python’s request library, the data is automatically obtained via the GitHub API. The GitHub Search API is used to set repository requirements, sampling constraints, and to filter asynchronous programming constructs.

*Repository requirements*

In order to obtain only relevant repositories, certain requirements must apply. Since applications always contain content of other languages or additional backends, the percentage value of the primary language, for this study JavaScript, is 60%. Considering only repositories with activity, repositories with more than 5 stars are selected and those that were last modified in the last year at least once. Furthermore, the repositories are not allowed to be a fork, i.e., a copy, but if the repository has many forks, it shows in popularity. Only repositories with more than 30 commits are taken into consideration to ensure sufficient development activity, such as to prevent dead projects or projects where all code was pushed within a few commits. To collect asynchronous bugs, repositories must have closed bug issues that are in English.

*Filter asynchronous code*

To filter asynchronous code from all JavaScript repositories, the GitHub Search API is used, which enables requests to repositories, code, commits, issues, topics, wikis, etc. With topics, it is possible to explore repositories in a particular subject area, find projects to contribute to, and discover new solutions to a specific problem.[[1]](#footnote-1)

With appropriate asynchronous-related keywords, it is possible to find candidate asynchronous repositories and closed bug issues.

Asynchronous-related keywords:

* in GitHub topics, issues, commits, wikis: asynchronous, asynchronicity, asnychrony,
* in GitHub code: promise, callback, async, await,
* concurrent, race, race condition, data race, deadlock, atomic, mutex, starve

*Determine sample size*

Three criteria usually will need to be specified to determine the appropriate sample size: degree of variability, confidence level and precision level.

The degree of variability in the attributes being measured, refers to the distribution of attributes in the population. The more heterogeneous a population, the larger the sample size required to obtain a given level of precision. The less variable (more homogeneous) a population, the smaller the sample size. GitHub has more than 4,5 million JavaScript repositories. With the repository requirements from above this amount decreases to 30 thousand JavaScript repositories. After filtering for asynchronous programming constructs, the amount decreases to 25 thousand JavaScript repositories, which can be assumed to be the size of the population.

The level of precision, sometimes called sampling error, is the range at which the data of the population deviates from the samples. To determine the largest sample size, the value of precision level is set to 5%. The level of confidence, i.e., the probability that the data of the selected samples reflect the data of the population, of 95% is chosen.

There are online tools [[2]](#footnote-2) that enable the determination of the sample size by entering the relevant variables. (After entering the variables, the calculator has calculated a sample size of 379.)

**Data Collection Procedures**

Similar to the Sampling Procedures, a Python script is used to automatically collect the necessary data from the chosen projects. The GitHub API and the static analysis tool SonarQube with its API served as the two main data sources. [[3]](#footnote-3) All repositories are downloaded using the GitHub API because SonarQube requires the source code for analysis.

* Using ESLint or detecting ESLint plugin in projects (lead to fewer code smells)
* Metriken:
  + Code smells per LOC (no sense?)
  + Percentage of bug-fix commits
  + Average time a bug issue is open
  + Cognitive complexity per LOC
  + Maintainability (code smells)

1. Bug proneness: The number of bug-fix commits is used as a key indicator of bug proneness. For evaluation, the entire commit history is scanned to count the bug-fix commits.
2. Bug resolution time: The average time a bug issue is open is determined via the GitHub issue tracking system15.
3. Code quality and understandability: Code smells, which determine code quality, and cognitive complexity, which represent code understandability, are reported by the static code analysis tool SonarQube16.

**Data Analysis Procedures**

**Hypotheses**

1. <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/classifying-your-repository-with-topics> [↑](#footnote-ref-1)
2. <https://www.calculator.net/sample-size-calculator.html> [↑](#footnote-ref-2)
3. <https://next.sonarqube.com/sonarqube/web_api/> [↑](#footnote-ref-3)