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:
Alexander Lill
Shirin Pirouzkhah
Dr. Bruno Rodrigues

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 come up with its own personal solution. A strict plagiarism policy is going to 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., CSV files):
  - The explanations must be written in a single PDF file.
  - The names of the students in your group, source code, data files, and explanations must be uploaded to OLAT as a single ZIP file by Nov 27, 2023 @ 23:55.

Background

Apache Kafka[1] is a distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Not only is it on the topic of data-intensive applications, but it also has a large and active codebase,[2] making it an interesting candidate for analysis in the context of software systems and quality.

Task 1: Codebase Overview

- Create a list of all the entities in the latest stable release of Apache Kafka (i.e., version 3.6.0). Consider an entity to be equivalent to a file (i.e., file-level granularity).
- For each entity from the list above, consider the period between the last two major releases (i.e., 3.5.1 and 3.6.0), and extract:
  - the number of revisions each entity went through (i.e., the number of commits on each file),
  - which authors worked on these entities and how many lines they added/removed/changed.

Save your results in a JSON-formatted file, as in the following:

```
{
  "FILENAME1": {
    "num of revisions": INTEGER,
    "authors": {
      "AUTHOR1_NAME": {"added": INTEGER, "removed":
         INTEGER, "total": INTEGER},
      "AUTHOR2_NAME": {...},
      ... }
    "revisions:": {
      "COMMIT1_HASH": "changed lines": INTEGER,
      "COMMIT2_HASH": "changed lines": INTEGER
      ... }
  },
  "FILENAME2": {
     ...
  },
  ...
}
```

[1]https://kafka.apache.org/
[2]https://github.com/apache/kafka

Task 2: Complexity hotspots

Measuring complexity is a non-trivial task and a number of metrics exist:

Metric	**Description**	**How To Measure**
Cyclomatic Complexity (CC)	Quantifies the number of linearly independent paths through a program's source code by measuring the control flow within the program.	Using python libraries such as lizard, pydriller, radon, or McCabe
Lines of code (LoC)	Measures the number of lines of code in a file.	Using python libraries such as lizard, cloc, or radon
Number of code changes (NCC)	Measures the number of times each file has been changed in a certain timeframe or between two releases.	Using the json file you already generated in the previous exercise
Indentation-based complexity (IC)	Analyses the visual shape of the code, such as the level of indentation. This approach is language-neutral and can be applied to different programming languages.	Counting whitespace "blocks" at the beginning of each line. A block is defined by a set number of spaces, i.e., 4 or a tab character for indentation.

Table 1: Complexity Metrics

Complete the following analysis:

- Considering the last major release (i.e., 3.6.0), collect a list of all Java files (you can use `glob` to find all `.java` files).
- Measure the complexity of all collected Java files using all four metrics.[3]
- Visualize the hotspots. The visualization should effectively convey which parts of the code are more complex or change more frequently. You can use heatmaps, bubble charts or both.
- Complexity Trends Analysis: Identify which files are likely complexity hotspots (areas of the code that might be problematic due to their complexity), also using your visualization. For ten of these files conduct a more in-depth analysis about how the complexity of these files has changed over time to identify trends.

[3]For the NCC metric, consider the commits between the last two major releases (i.e., 3.5.1 and 3.6.0).

Task 3: Logical and Temporal Coupling

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.

- Temporal coupling analysis: To analyze coupling, look at files that were changed in a specific time window. Analyse which files are changed together within a time window of 24, 48, 72, and 168 hours of each other.
- Determine the degree of coupling based on the number of joint changes within the specified time windows. For instance, a higher number of changes in close succession may indicate a stronger coupling. For each pair of coupled files, track the number of versions over time to see if the rate of change is consistent, increasing, or decreasing. This can give insights into the evolving interdependence of files.

Your results about *temporal coupling* should contain the following information:

```
[
    {
      "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. Your result should contain the following information:

```
[
    {
      "file_pair": ["FILENAME1", "FILENAME2"],
      "logical_coupling": {
        "Joint": "number_of_joint_commits_file1_file2",
        "FILENAME1": "number_of_joint_commits_with_others",
        "FILENAME2": "number_of_joint_commits_with_others"
      },
    }
    ...
]
```

Select three sets of strongly coupled files (by either temporal or logical coupling) and perform an in-depth analysis. Try to understand the reasons for their coupling, which could include feature development, bug fixes, refactoring, or architectural dependencies.