Project 1 Milestone 4 (“Release Notes”)

As a Boilermaker pursuing academic excellence, we pledge to be honest and true in all that we do. Accountable together – We are Purdue.

(On group submissions, have each team member type their name).

Type or sign your names: Katie Roberts, Ashwin Senthilkumar, Parker Bushey

Write today’s date: October 4, 2021

Assignment Goal

In this assignment, you will deliver your Project 1 implementation (*software*) and communicate the final status of your Project 1 (*release notes*).

This document provides a template for the release notes. This is a form of documentation that your customer can use to decide (a) whether you've met the contract, and (b) renegotiate the contract based on deviations therefrom.

Relevant Course Outcomes

A student who successfully completes this assignment will have demonstrated the ability to

- *Outcome ii*: the ability to conduct key elements of the software engineering process, including…deployment
- *Outcome iii*: Develop an understanding of the social aspects of software engineering… including…communication [and] teamwork.

Resources

This document is a form of "release notes", albeit on the more technical end of the spectrum.

Definitions:

- Definition 1
- Definition 2

Examples:

- Apache Maven
- Microsoft Windows

Assignment

Fill out each of the following sections.

Location of project

Provide the URL to the GitHub repository containing your team's project.

https://github.com/Purdue-ECE-461/project-1-3.git

Succinct description

In a 5-7 sentence paragraph, describe the system that you have implemented.

The program we designed accepts a list of URLs that refer to NPM packages that are provided by the user. These packages are individually assessed based on a number of metrics including ramp-up time, correctness, bus factor, responsiveness, and license compatibility. These scores are combined into an overall score per package being evaluated and the results are output to stdout. For easy comprehension by the user, the packages are sorted, the highest scoring of them listed first. This allows for the user to determine which packages are best among the set that was input.

Fitness for purpose

In Milestone 1, you identified the requirements for the system.

Fill in this table of the customer's requirements and the degree to which you've met each of them. (If on Milestone 1 you lost points for inadequate requirements, then you should provide a more detailed set in this table).

Requirement	**Is the requirement met? (yes/no)**	**Explanation (2-4 sentences). If met, how did you measure it? If unmet, discuss.**
The system should support input from command line arguments.	Yes	This requirement was met. It cannot be measured quantitatively since it is a feature of the system. When it is run, it can take in input from the command line
The system will print all output to stdout.	Yes	There is no quantitative measurement for this requirement, but all necessary information is printed to stdout during program runtime.
The system will produce an ordered list of repositories starting with the most trustworthy.	Yes	Each repository is ordered by its overall score, the highest scoring being displayed first. The overall score (trustworthiness) is determined by our implementations of all the subscores, so measuring the success of this requirement is

		subjective. However, according to our design, the requirement has been met.
Program should take no longer than 30 minutes to run	Yes	Runtime of "make test" on our own computers was about 10 minutes so the expected runtime of "run (filename)"/"run test" is about 5 minutes each (see note for autograder below). Runtime of "run install" on ECEGrid was about 30 seconds.
In the list of repositories, each entry will include a set of scores (overall score, ramp-up time, correctness, bus factor, responsiveness, license compatibility)	Yes	We successfully were able to implement our design or modified design for each score. When design changes were made, these changes were documented. All changes made to these subscores does not, in our opinion, change the purpose of each subscore.
Any open-source module's licenses that ACME Corporation's service engineers use must be compatible with the LGPLv2.1 license.	Yes	The license subscore always evaluates to 1 if the license is compatible with LGPLv2.1, and always evaluates to 0 if the license is not compatible with LGPLv2.1. The rest of the subscores are multiplied by the license subscore, so if the license is ever incompatible, the package will receive a score of 0.

Final metric designs

In Milestone 1, you described your initial design of the metrics. Please compare your initial design to your final implementation. (This will help us figure out if our auto-grader is not working well for your approach.)

Metric	Initial design	Final implementation ("Same" or whatever the new def. is)	Explain any difference
Ramp-up time	Longer readme, includes links or pictures: GET /repos/{owner}/{repo}/readme Less files (small tree size): GET /repos/{owner}/{repo}/git/trees/{tree_sha} Number of views/downloads/forks/clones: GET /repos/{owner}/{repo}/traffic/views GET /repos/{owner}/{repo}/releases/assets/{asset_id} (includes download count) GET /repos/{owner}/{repo}/forks GET /repos/{owner}/{repo}/traffic/clones	**RAMP_UP_SCORE** = X * (README_SIZE / FILE_COUNT + Y * FORK_COUNT) Where X = 0.25 and Y = 0.0001	Instead of using the download count in this subscore, the fork count was used. The download count wasn't accessible using GitHub API so we substituted in the number of forks.

	RAMP_UP_SCORE = X * (README_SIZE / FILE_COUNT + Y * DOWNLOAD_COUNT) Where X and Y are some constant		
Correctness	Compiles and runs (True/False) Code coverage % Self tests are passing % **CORRECTNESS_SCORE** = COMPILE_RUN * ((COVERAGE > 0.85 ? (COVERAGE - 0.85) / 0.15 : 0) + (SELF_TESTS > 0.7 ? (SELF_TESTS - 0.7) / 0.3 : 0)) / 2	**CORRECTNESS_SCORE** = NUM_TESTFILES / 150 >= 1.0 ? 1.0 : NUM_TESTFILES / 150 where NUM_TESTFILES is the number of js/json files in the repository that include the word "test" in the relative path from cloned root	Running the json code or it's given tests proved to be too costly in both compile time and the hours we would have needed to write the code.
Bus Factor	2-6 collaborators normally Some have single collaborator Some have many collaborators GET /repos/:owner/:repo/stats/contributors **BUS_FACTOR_SCORE** = ((NUM_CONTRIBUTORS) >= 6) ? 1.0 : (NUM_CONTRIBUTORS) / 6	Same	Unlike other score metrics that use the jcabi maven plugin to interface with the GitHub API, bus factor uses direct HTTPUrlConnectors since the jcabi plugin could not handle the Json Array of contributors very well.
Responsiveness	Number of open and closed issues, Issue Date opened compared to date closed, Date of most recent closed issue GET /orgs/{org}/issues Commit activity: GET /repos/{owner}/{repo}/stats/commit_activity GET /repos/{owner}/{repo}/commits/{ref}/status GET /repos/{owner}/{repo}/stats/code_frequency **RESPONSIVE_MAINTAINER_SCORE** = (CLOSED_ISSUES / (OPEN_ISSUES +	Same	The computation of the score remained the same. The definition of an issue changed slightly however. In the initial design, an issue was strictly an issue, but the API made it easier for an issue to be classified as an issue or a pull/merge request. This led to the final design definition of an issue to include pull/merge requests, but we concluded that doing so did not change what the responsiveness score conveyed overall.

	CLOSED_ISSUES) + (MOST_RECENT > 90 ? 0 : 90 - MOST_RECENT) / 90) / 2		
License Compatibility	License scan passed: GET /repos/{owner}/{repo}/community/profile **LICENSE_SCORE** = LGPL21_COMP	Same	There is no computation for this score since it is a boolean variable that is either passed or not. This implementation stayed the same as the requirements we mentioned for license compatibility score.

Notes for the auto-grader

If your submission cannot be automatically parsed by the auto-grader described in the project specification, provide explanatory notes that the course staff can consider while scoring your submission. Be specific. Since this spec was provided well in advance, accommodating any deviations is at the discretion of the staff.

Deviation	Details
code coverage	code coverage analysis is done by the maven build process using the jacoco plugin, run via the "make test" command. This has been done and saved in advance for you so that you don't have to wait for the maven build and testing process (which has to run through the entire program twice). All the autograder run will do is parse the html output from the jacoco directory. If you would like to see detailed results, open the jacoco/index.html file in your browser (you can also look at the code that was present at build time to check it is the same). If you would like to run the "make test" command yourself, change the "MVN :=" line to the one that is commented out with #, and uncomment the tar import commands in run install.
run setup	Since maven has been done in advance, any run command is utilising the jar of the program, not the src java files. "run test" will output to the log file specified in your environment, and the regular "run (filename)" will output to stdout.
log file	If desired, you can see the maven test run by looking at project-1-3.log