ECSE 429 Software Validation
Term Project
Part C

Group 17
Weige Qian 260763075
Erdong Luo 260778475
Yudong Zhou 260721223
Hanwen Wang 260778557

1. Deliverable Summary

In part C of this project, we did two non-functional testing of Rest API, which are performance testing and static analysis. For performance testing,we wrote performance testing scripts and used windows perfmon to generate the graph of memory use and cpu use changing. We also analyzed the runtime for each type of transaction (add, delete and change) when the number of existing objects varies. For static analysis, we used SonarCube to generate a report about the code quality. These results will be contained in our report.

Apart from those two testings, a video is also included to show the execution of all the performance tests.

2. Performance test suite implementation

In this project, we continue to use python as our programming language. The entire test suite consists of 12 python scripts. The three named with ‘populate’ contain ‘populate’ functions to add a specific number of different objects into our server. These three files are implemented as libraries, so that the ‘populate’ functions can be used in the unit tests.

Each of the other 9 python scripts defines a test scenario which can be described by the filename. For example, the script ‘add_todo.py’ contains the test cases for the scenario when we add a new todo object to the system, given a specific number of existing todo objects. Within each test script, several test cases are implemented and the number of existing objects varies for different test cases.

In every test scenario, we use a setup function to start timing of T1, boot the server and call the ‘populate’ functions to add a specific number of objects into the server. In every unit test function (test case), we first start timing of T2, send the transaction request, calculate T2 and then assess the correctness. After that we use a teardown function to calculate the T1, and shut down the server. Since every time when a test case is finished, the program will restart the server, so the server will go back to the original status. Therefore, we do not need to worry about restoring the server. To ensure that the server has been running successfully in each setup function, we use a try-except to catch the exception if the server is not running.

Each test script will continuously execute all its test cases by just running the script. There is no need to manually start or stop the server. Each test case in a test script will log the times T1 and T2, the number of existing objects for that test case, and the current sample time. Except the first test case in each script, for all the remaining test cases, the number of existing objects for that test case is defined by its previous test case, using a global variable ‘num’. The initial value of ‘num’ will be used for the first test case in a test script. For example, if the initial value of the global variable ‘num’ in a test script is 1000, then the first test case in that script will be executed after adding 1000 random objects into the system.

```
num = 1000 # number of objects
```

Then, at the end of the first test case, the value of ‘num’ will be changed to get prepared for the second test case. Therefore, in the second test case in this script, the server will initially have 2000 random objects of the given type before the transaction is executed.

```
def test_case_example_1(self):
    ...
    global num
    ...
    num = 2000
```

By using this way, after a test script is finished, we will have all the information for the specified object and transaction type, with multiple values for the number of initially existing objects. Furthermore, we can easily extend this script to have more test cases.

After we get the transaction time T2, and the total time T1. We put them into an Excel and draw graphs about them.

3. Transaction time vs sample time

Todo:

Chart 1: Transaction time vs sample time for Add, Change and Delete Todo

Project:

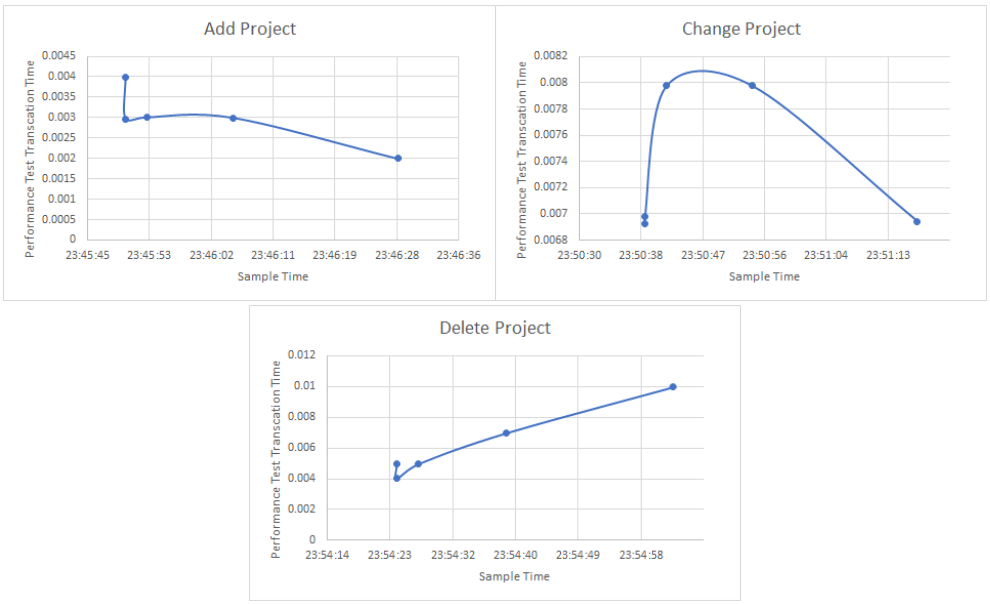

Chart 2: Transaction time vs sample time for Add, Change and Delete Project

Category:

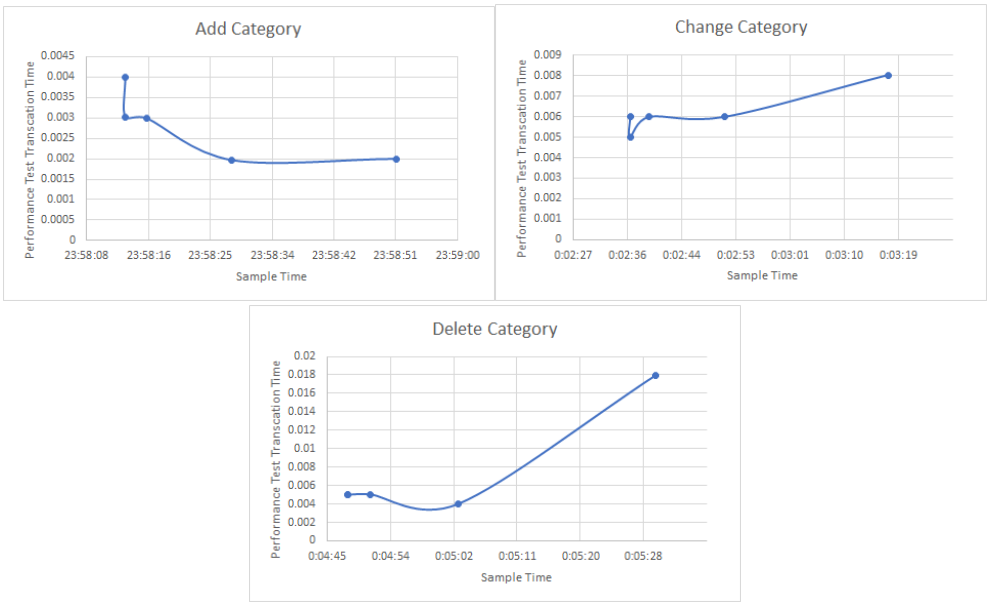

Chart 3: Transaction time vs sample time for Add, Change and Delete Category

For all the add functions, transaction time has a trend of decreasing. For all the change and delete functions, transaction time has a trend of increasing, except for change project function.

4. **Transaction time vs object number**

Todo:

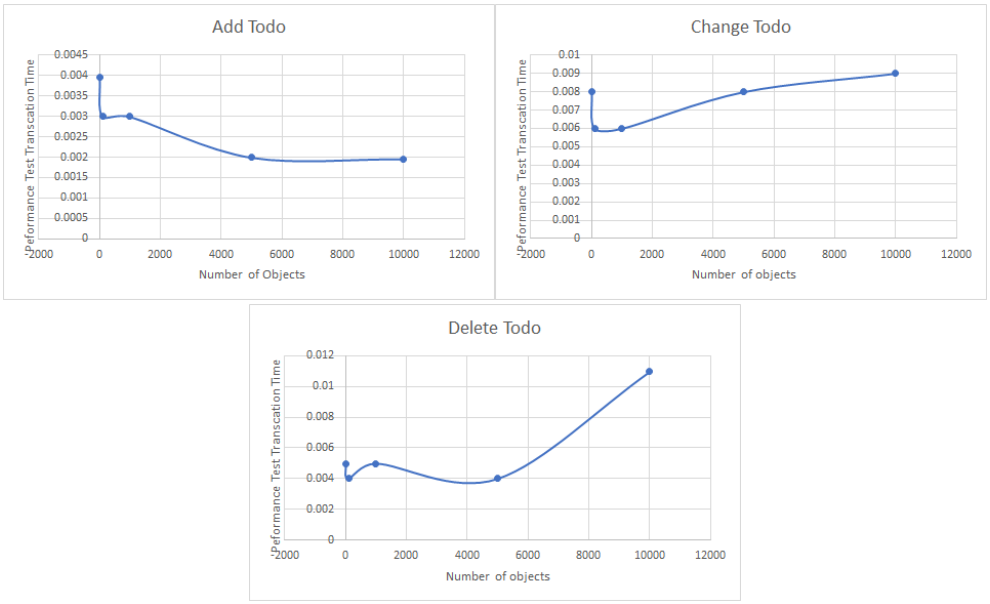

Chart 4: Transaction time vs number of objects for Add, Change and Delete Todo

Project:

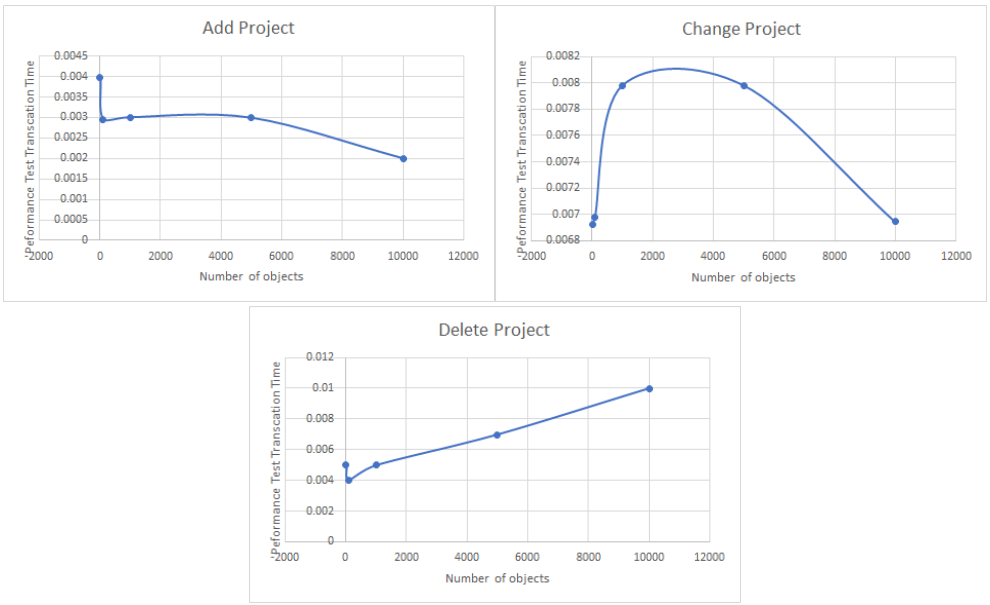

Chart 5: Transaction time vs number of objects for Add, Change and Delete Project

Category:

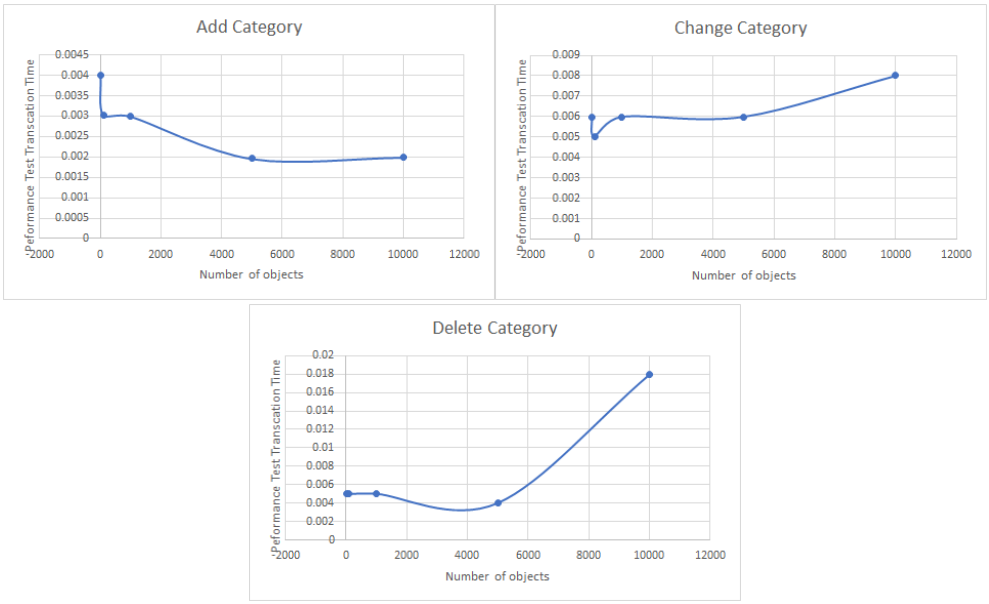

Chart 6: Transaction time vs number of objects for Add, Change and Delete Category

The general trend is similar to that in the former question, even if the x-axis switches from sample time to number of objects.

5. Memory available vs sample time

Todo:

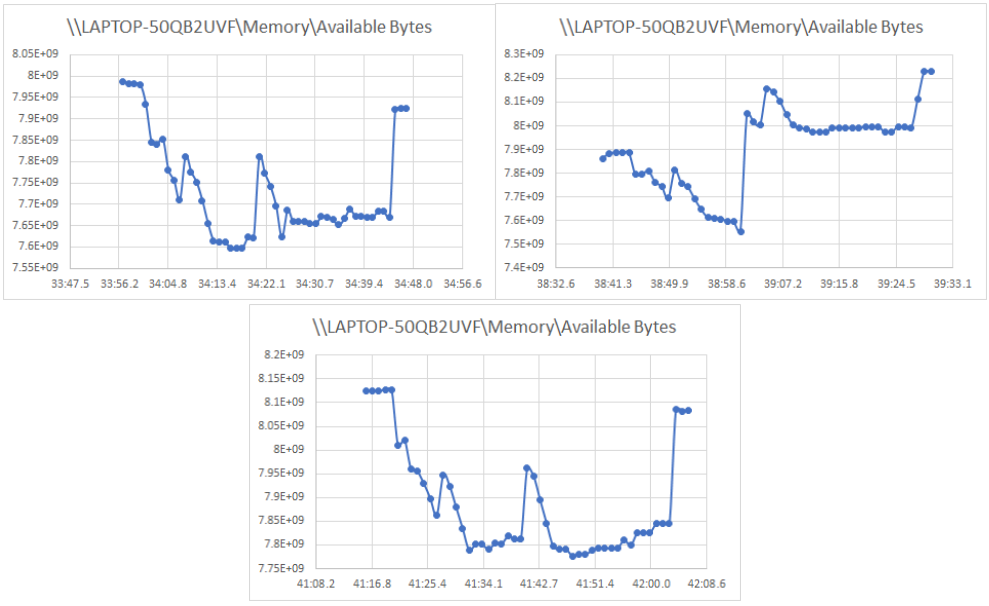

Chart 7: Memory available vs sample time for Add, Change and Delete Todo

Project:

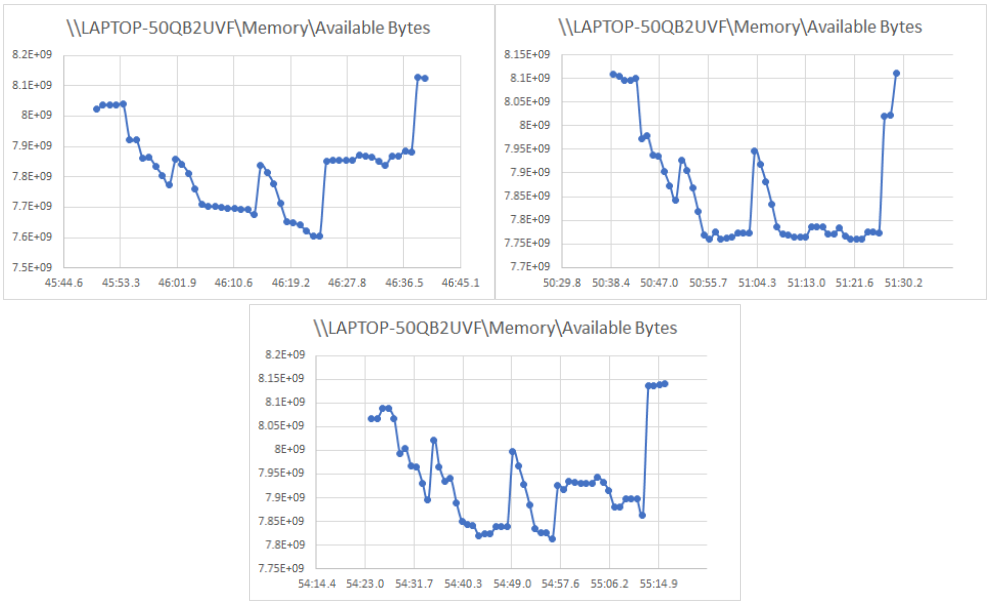

Chart 8: Memory available vs sample time for Add, Change and Delete Project

Category:

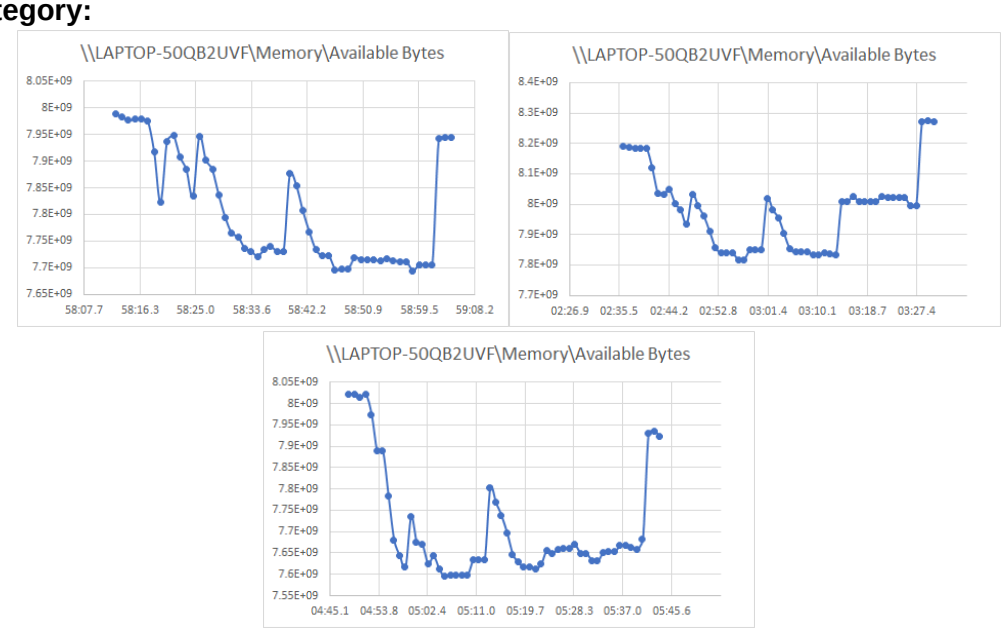

Chart 9: Memory available vs sample time for Add, Change and Delete Category

6. Memory available vs object number

Todo:

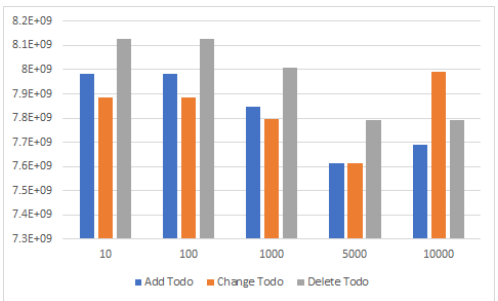

Chart 10: Memory available vs object number for Add, Change and Delete Todo

Project:

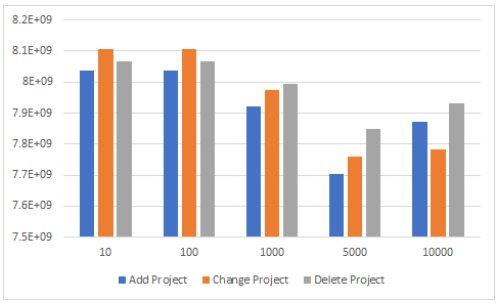

Chart 11: Memory available vs object number for Add, Change and Delete Project

Category:

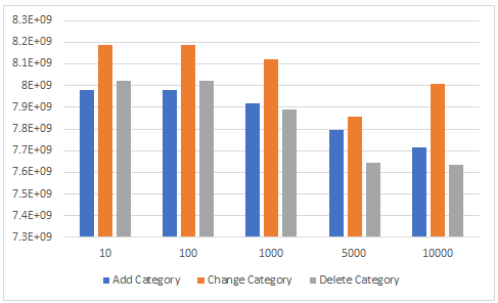

Chart 12: Memory available vs object number for Add, Change and Delete Category

All the values have a general trend of decreasing, it makes sense because when more object numbers are required, more memory will be used to create those objects and thus available free memory decreases.

7. CPU use vs sample time

Todo:

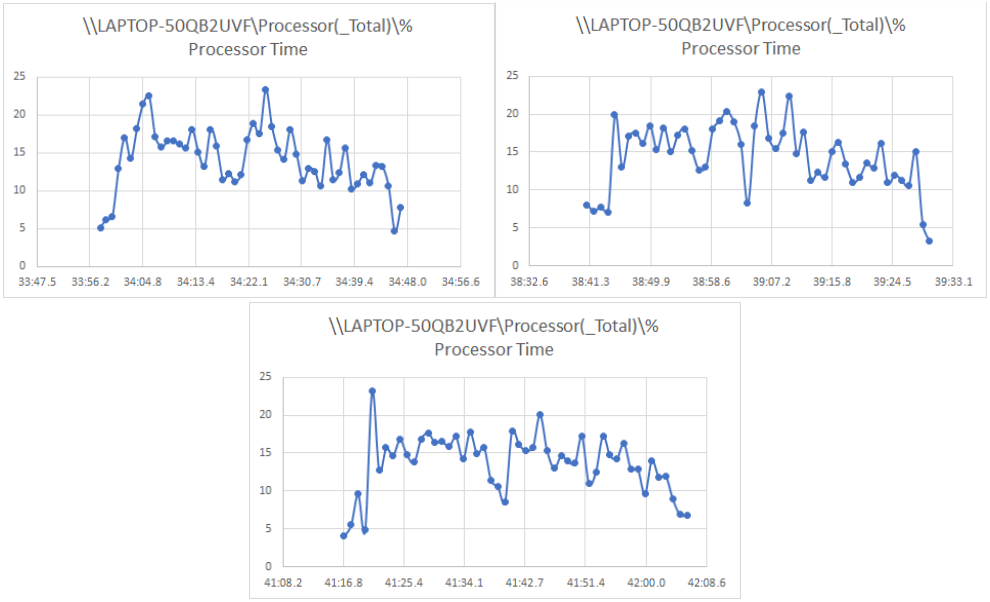

Chart 13: CPU use vs sample time for Add, Change and Delete Todo

Project:

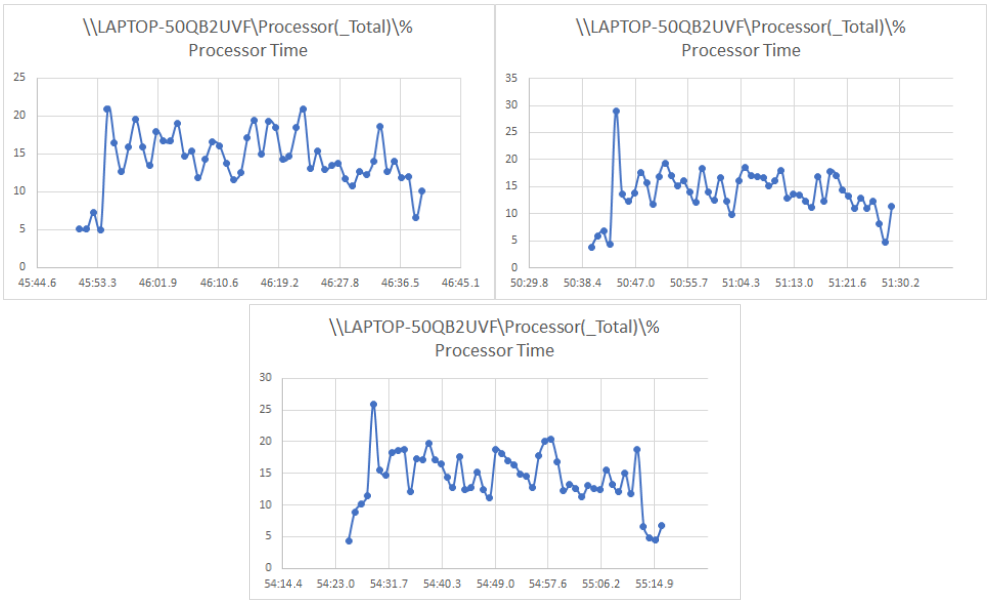

Chart 14: CPU use vs sample time for Add, Change and Delete Project

Category:

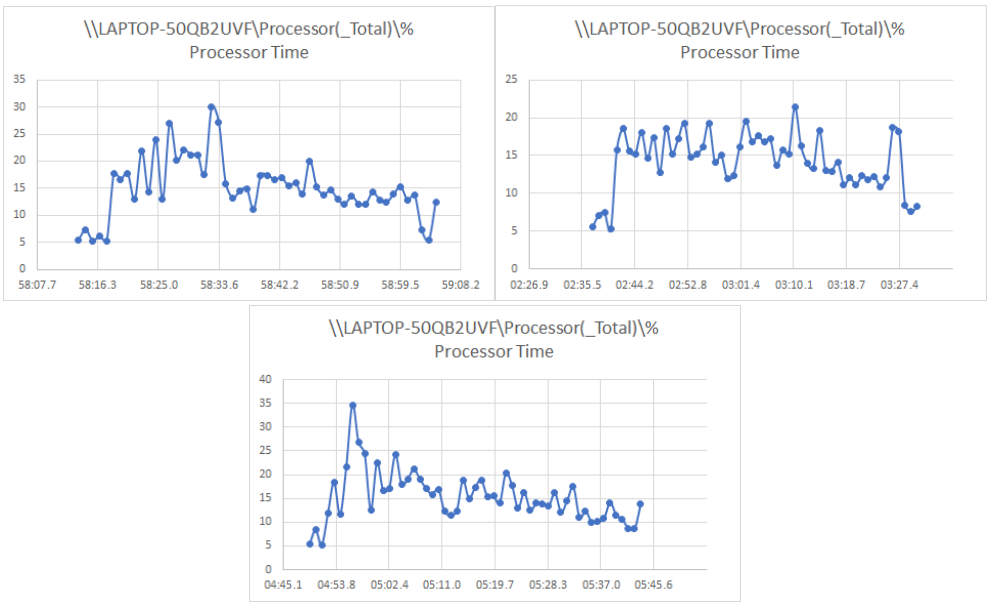

Chart 15: CPU use vs sample time for Add, Change and Delete Category

8. CPU use vs object number

Todo:

Chart 16: CPU use vs object number for Add, Change and Delete Todo

Project:

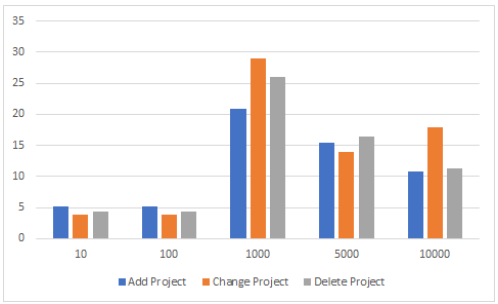

Chart 17: CPU use vs object number for Add, Change and Delete Project

Category:

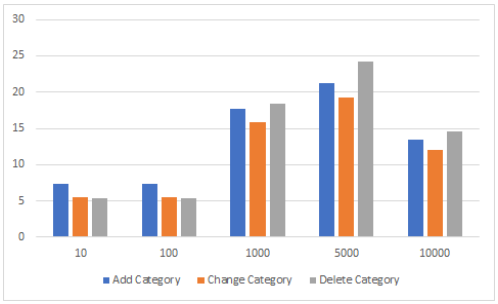

Chart 18: CPU use vs object number for Add, Change and Delete Category

The general trend is first increase and then decrease. Since we took 10 and 100 objects at first, they are done in less than a second, and this dynamic analysis tool records once per second, their values have no difference. And all the values change in CPU use will be shown after one second, from when the program starts running. As a result, the CPU use is high when the object number is 1000. Later since our program continuously restarts the server instead of restarting the program, the CPU usage will become relatively stable and will only change a bit. That is why CPU use in 5000 and 10000 are close. If we want to see the exact trend, more tests should be done and more data should be collected.

9. Performance risks and code change recommendations

The most risk we have met during our performance test, if we populate more than 10000 items into the server, we found the program will meet a socket error and the program will fail, this may be caused by the imperfect memory handling algorithm. The number of items exceeded the limitation and the function failed. This is also shown in the graph of memory use in the previous sections. We could find that the decreasing rate of available bytes in memory is very high against the increasing population number.

The team recommended changing the algorithm of memory handling, maybe using different data structures. Maybe it could improve the performance of the whole system.

10. Static Analysis

In this project, we use SonarQube to do the static analysis to implement the static analysis of the source code. In general, the source code receives a grade of D on reliability. The tool finds 8 bugs: 1 critical bug, 6 major bugs and 1 minor bug. Obviously D grade is not a quite good score on anything. But, this source code gets quite a good score on Security, an A grade. 0 Vulnerabilities found. In contrast, this source code receives an extremely low score on security review, an E grade. 0.0 % of the code has been reviewed. 8 security hotspots have been found, 6 in medium review priority and 2 in low review medium. But the source code receives an A grade on maintainability, 8 days 1 hour on debt and 952 code smells. 952 code smells are distributed in 6 on blocker level, 72 on a critical level, 211 on the major level, 122 on the minor level and 541 on info level. Besides this data, we could also find 0.0% code coverage with 572 unit tests. This is a pretty low percentage. In contrast, the source code has achieved an excellent score on Duplications. Only 0.2% of code and 2 duplicated blocks has been found.

The report provided by SonarQube mentioned two conditions failed: Coverage on new code is less than 80% (0 %) and Security hotspot reviewed is less than 100% (0%).

SonarQube also provides information about code size, the whole project has more than 9000 lines of code. Three major subprojects are thingifier, ercoremodel and challenger with 4761,1965 and 1954 lines of code.

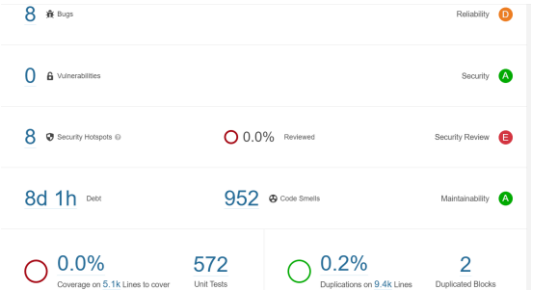

Figure 1: Screenshot of SonarQube

11. Static analysis code change recommendations

Since the source code gets a pretty low score on reliability, security review and code coverage on the unit tests. The recommended change is on these parts. For code coverage, more unit tests should be added to the source code. The test staff could use Eclemma to check the coverage of the test case. For reliability, we got 8 Bugs, we have 1 critical bug, 6 major bugs and 1minor bug. In ChallengerTrackingRoutes.java, line 31 and line 63 have a problem of using “==” to compare String, an equals method should be used. There are also 3 places where a null pointer exception could be thrown: line 117 and line 131 in file RelationshipCreation.java and line 189 in InstanceFields.java. A try-catch statement should be added. Also, in line 362 of MainImplementation.java, the return value of the format method should be used. In line 341 of

RestApiDocumentationGenerator.java, the Random object should be saved and reused.

Also, the problems of security review are very important. Even though the tool has found 8 Security Hotspots, there are only 2 types of problems. First, the program should be used as a secured way to generate a random number, such as java.security.SecureRandom. Second, the program should use a logger method to print the exception message instead of e.printStackTrace().