Algorithms and Software Engineering Questions (Part 1 and Part 2)

Student’s Name

Institutional Affiliation

Algorithms and Software Engineering Questions (Part 1 and Part 2)

**Part I: Question 1**

1. **Draw the control flowchart for the function**
2. **Give the gen sets for each of the building blocks**

**c. How is the incoming and the outgoing data flow (in and out) of a basic block computed?**

This can be computed using a scenario below, first we need to the ***transfer function,*** which can be given by ***fb of a basic block b:***

***OUTB[b]=fb(IN[b])***

Incoming reaching definitions/ declaration can be described as follows, which are part of block computation:

Generate definitions =Gen[b]- this capture locally available definitions in block b

Propagation definitions= in[b] -kill[b]- in this case, kill[b] is set of defs[in the rest of the program].

***out[b]=Gen[b] U (In[b]-kill[b]).***

The problem is a ***may data flow problem*** as it provides the details on what may be true at each program point such as variable analysis, among other important details.

**d. Compute the final values for the in and out sets of each building block.**

In computing the final values for the in in and outs of each building block, the flowing flow chart can summarize how computation can be achieved.

In the above flowchart, outl[b]=fb(IN[b])

The join node in the above diagram is ***f1 ,*** which is a node with multiple predecessors

In this case, in[b]=out[p1] U out[p2] U out[p3] …. U[pn], where p1 to pn are the predecessor of b.

**f. Which problem in the source code can be identified from the in sets?**

 The problem generated by the live variable in[b], is locally expressed and uses *b*

From the source code, this can be simplified to as follows,

f=1

if input () exit;

f=f \* n

n= n – 1

**g. Give the definition of McCabe’s cyclomatic complexity and show how it is computed on the above program.**

McCabe’s cyclomatic complexity is a metric for quantifying the complexity of a software program or code.

The code complexity of a program can be calculated using the formula below

V (G)=E -N + 2, where E is considered to be the number of edges, while Nis the Number of Nodes

The formula can be extended as follows, V (G) =P + 1.

Based on the problem given, the problem can be computed as follows

// description of how McCabe’s cyclomatic complexity is computed.

f=1

input n

while (n>=0){

f =f \* n

n=n – 1

 return n

return f

}

**h. Consider branch coverage.**

Branch coverage involves test coverage criteria that require adequate test cases, so that each condition within a decision consider all possible outcomes at least once in every point of entry and in the program or sub-routine.

1. **How is McCabe’s cyclomatic complexity related to branch coverage for a single function?**

Both McCabe’s cyclomatic complexity and branch coverage relates in the sense that they are used in building and computing the control flow graph of the codes that measure the quantity of directly independent ways through a program module within a single function.

The McCabe’s cyclomatic complexity can be described using an example below

**Measure=Number of Errors**

Measurements – Number of Errors identified per individual

Features of cyclomatics complexity testing

The above parameters are useful in in branch coverage, when developing the testing strategy,; hence this express their relationship with McCabe’s cyclomatic complexity

1. **Compute the branch coverage for the following four calls:**

 • fac(-1)

• fac(0)

 • fac(1)

• fac(2)

This problem can be solved as follows based on branch coverage for the above four calls

int f (int n) {

int ans=1;

if (n > 0)

ans=n \*

f (n--1) // f (n+1)

f (n-0)

f (n-1)

f (n-2)

return ans;

}

1. **Give a combination of calls that achieves full branch coverage.**

The four calls above achieve full branch coverage, because they measure the decisions calls that have been tested within the given condition, as shown above

**Part II: Software Engineering Practices**

In this section, the goal is to provide a description of 10 software engineering practices, as presented in the article, Software Engineering at Google. The practice are also compared with the lessons learned in the lectures. The practices can be presented as follows:

1. **The Source Repository**

Source code repository is essential in storing the code adopted during software development, to ensure the code is secured, and can be used for future developments or reference. With a source repository, it creates a platform where automated tests can be run frequently, which helps the author and reviewers to be notified about the changes done during the software development. Additionally, with this practice, it helps in identifying the integration problems by minimizing the amount of work needed, which makes it easier and faster to fix the security issues. The practice is diverse and can also help in creating subtree, which is associated with the different subdirectories within the repository, where the code is stored (Henderson, 2017). The above software engineering practice is consistent with what learned in the lectures, and this has an advantage that it has provided more insights on. Another positive lesson learned that, all the users in the repository are always listed by the primary owner, and any change such as deletion of the repository, directories, or subdirectories, cannot be done without the authorization from the owner.

1. **The Build System**

The practice involves compilation, linking of software, and running the tests for the software, which is a common start used by Google in the whole repository, to ensure that the software does not comprise of any bugs. At google software developers use blaze, which is a distributed build system, that has a capacity to compile and run the tests. One of the important things to note about this practice is that is done Google’s distributed computing infrastructure that can help in running the code in several machines simultaneously (Henderson, 2017). The key positive thing about this practice that is different from other practices is that it makes the work easier to run, test and execute the code due to the integrative functionalities. Additionally, it is typically easier and simple for the Google engineer when building and testing the software in the repository. The only negative thing that can be pointed out with this practice is that, it that it is complex in nature as compared to using other tools, and this requires understanding by the software engineering on how to build software dependencies, especially when effecting the changes in the software and interacting with the platforms when compiling and running the tests. With the build system approach, the incremental rebuilds are fast and the results can be cached in the cloud platforms, unlike other platforms, where cloud only resides in the repository.

1. **Code Review**

In this practice, it provides a platform where authors can perform the review of code with excellent coding schemes such as colors. The reviewers are notified via email platforms when code is review and comments written by the software engineers. This makes it easier for the reviewers to understand the changes done in the software. In ensuring that the code written meets the standards, the main source code must be reviewed from the repository by more than one other software engineer (Henderson, 2017). The code author must commit any urgent changes needed in the software before the review is done. The code reviews can also be shared with different reviewers, and once the approval has been done, such changes can be committed to the repository with the immediate effect. The positive thing that appears to be different in what is learned so far, is the idea of extensive review to ensure that the code is consistent and meet the threshold of being committed to the repository. On the other hand, the negativity is based on the time taken to review the code as well as all other processes.

1. **Testing**

Testing is a platform for ensuring that the code is free from bugs, and that it is working accordingly. At Google, the most used type of testing is unit testing, where all code written must undergo unit tests, to ensure the software is okay. Some of the testing include load testing which is done by benchmarking to check on the resource utilization and optimization of the software. In any additional functionality done, the reviews are done extensively to cover such functionalities (Henderson, 2017). Apart from unit testing, other forms of testing done include integration testing and regression testing, which ensures that the code is working according before the actual installation or hosting of the application. Based on the lessons learned, this practice is crucial and has added more insights on the processes of deploying a software. The positivity of this practice is that it involves load testing during the deployment, which is not common practice in other platforms.

1. **Bug Tracking**

The software engineering practice at Google, involves using bug tracking system to identify possible issues and processes, to ensure that there are no bugs within the system. The Google engineers always look at all the open issues, to ensure they are updated accordingly. During the development process, specific teams are given the responsibility for bug triage, which is done frequently or within the timeline stipulated (Henderson, 2017). The common approach used by the teams include putting labels on bugs, to ensure that all bugs have been fixed accordingly. The positive thing that differentiate this practice form others is the fact that it involves a thorough scrutiny of the code to ensure that no bugs are left unfixed. In terms of negativity the practice appears to be so complex than other practices, even though Google use bug tracking system.

1. **Debugging and Profile Tools**

This practice is based on using Google servers that are linked with libraries, that are meant to provide various debugging tools, to ensure any bug is correctly fixed. The availability of log file provides a platform where the developers or those involved in testing the system, can easily identity and fix the errors. The most important thing is to identify the root cause of the bugs, whether it is as a result of server crashing or any other form of problems (Henderson, 2017). If the issue is a server crash, the event handler will automatically be displaced in the log file, to depict the actual error and its cause. Apart from the using the log files as the debugging tools, Google uses other web interface tools that are used to examine both outgoing and incoming Request Procedure Calls (RPCs), which capture the error rates, etc. The positivity of this tool as compared to other tools is its capacity to increase the overall ease of debugging. These tools have several other advantages as compared to the traditional GNU Debugger (GDB) tool.

1. **Release Engineering**

Release engineering is a concept that deals with the deployment of software, to make them high-quality and more reliable. At Google as described in the article, there is inclusivity of the release engineers which is mostly done by the regular software engineers. This software engineering practice is geared towards developing software products that meets the requirements and are of high quality. Notably, releases of software is a practice that is done frequently, which is made possible, through the automation of the normal software engineering tasks. With this practice, the advantage is that it keeps the engineers to be more active and motivated (Henderson, 2017). Any release of the software is done by creating a new workspace in the repository and updating the code, where all the automatic tests must be passed to ensure that the new release is free from any bugs, and is of high quality. When the release is done, it is copied to the production server and then from the production traffic to the staging server for actual processing. When all the changes done are verified, the processing that involves rolling out the release to all the data centers. The advantage of this Google software engineering practice as compare to other available platforms is that it involves extensive review to ensure the reliability, high quality and optimization of the software products. The only negative thing that can be pointed out is the complexity of the release engineering process.

1. **Launch Approval**

For any significant change of the system to be done, it requires approvals from several stakeholders, who are bestowed with the implementation of the required changes. All the changes done are subject to the approval or review, which must be based or assessing on the code compliance and legal requirements. Other approvals are subjected to the privacy requirements, security requirements, reliability requirements to ensure that the software products meet all the requirements. The whole process must guarantee automatic monitoring and detection of any outages that can be registered when the software is operating (Henderson, 2017). The whole launch process is also geared towards notifying people within the company, about any significant feature being launched or any new product release. The positivity of this practice at Google as compared to other platforms, is the existence of internal launch approval tool that is necessitated in tracking the required approvals and reviews and ensuring that there is compliance in legal terms, reliability and other aspects.

1. **Post-mortems**

This practice is necessary in a situation where there are changes or outages in the production systems or any other issue with the systems. All aspects are documented on the post-mortems document which documents the incident, summary of the issues, timeline, root causes, what worked and what failed to work, and the action taken when escalating on the issues (Henderson, 2017). The main goal is to focus on the problems within the system and prevent them from happening in future. With this practice, the goal is to quantify on the whole incident in terms of the duration taken for the outage, failed RPCs, timeline, and the revenue lost. In the sections such as what worked on didn’t work section, present the lessons learnt, and the practices used to detect, escalate and resolve on the issues identified. Other things documented include what caused the issue, and the concrete actions taken to provide the solution. This practice is ideal and provides a comprehensive report on all the issues identified and the formula used to resolve on such issues.\

1. **Frequent Rewrites**

The fundamental objective of this practice is to have the software rewritten, which is a common phenomenon in Google, which is implemented after every few years. This means that after significant years, the requirements are subjected to change significantly. The idea of rewriting the code is to cut-off any complexity that could be in existence and transferring knowledge from the owner to other team members. Rewriting of code is a paramount practice to ensure that the productivity is enhanced (Henderson, 2017). Additionally, with frequent rewriting, it encourages mobility of software engineers in various projects, which is important in consolidating different ideas; hence, increasing the productivity of software engineers. Another positivity of this practice with respect to what is learned is that rewriting helps in adopting the modern technology and methodologies in rewriting the code.

References

Al-Fedaghi, S. (2015). On Collaboration in Software: Annotation vs. Flow-Based Diagrams. *JSW*, *10*(11), 1327-1335.

Henderson, F. (2017). Software engineering at google. *arXiv preprint arXiv:1702.01715*.

Ikerionwu, C. (2010). Cyclomatic Complexity as a Software Metric. *International Journal of Academic Research*, *2*(3).

Kosower, D. A., Lopez-Villarejo, J. J., & Roubtsov, S. (2014, September). Flowgen: Flowchart-based documentation framework for c++. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* (pp. 59-64). IEEE.

Wei, Y., Meyer, B., & Oriol, M. (2010). Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification* (pp. 194-212). Springer, Berlin, Heidelberg.