PA2579 – ASSIGNMENT 2

Adaptive-lean software testing

Unit testing

Student Name: Uday Jain

Part 1 - Report:

Section 1: Project Description:

Since I am neither a developer or a tester and have experience only in Data Science, I have no experience writing unit tests. As a part of the exercise, I got a chance to speak to a few of the developers in my ex-team. The team provides data as a service (DaaS) and is part of the larger data platform team that owns multiple data and infrastructure products for the wider organization at Spotify. The team in current discussion is 1 squad composed of 5-6 developers, a product manager and an engineering manager. The team owns multiple DaaS products within the space of user-behavior and app instrumentation. These products required the team to not only have close collaboration with downstream squads that consume data products owned by our team but also with upstream squads that own the instrumentation of the app and platform infrastructure that collects and organizes the data for us to build DaaS products.

While the team worked in cycles of sprints and followed SCRUM practices, the development and testing practices were left to individual developers or squads with no central framework or structure. This created dependencies and long term technical debt that reduces overall productivity in the long term at the cost of short term efficiency and innovation gains. While collaboration within the squad was streamlined, cross-squad collaboration became challenging leading to roadblocks and reduced impact when working towards bigger projects involving 5-10 squads.

Section 2: Roles and Responsibilities:

The squad is composed of a mix of 5-6 developers and senior developers along with a product manager and an engineering manager. While the product manager helps develop a roadmap and product pipeline, it is the engineering manager who helps the team break down the pipeline into sprint cycles and tasks. The engineering manager (EM) also optimizes SCRUM cycles to improve productivity and remove bottlenecks. EM as a technical lead also takes part in solution discussions and acts as a go to person for anyone within the team incase of need. The developers and senior developers are expected to plan their tasks during sprint planning and own its completion. This not only includes development but also includes testing and sometimes collaborations with other members within or outside of the squad. Peer review for each of the tasks is also counted towards print planning.

Since the individual developers own their respective tasks , PMs, EMs and other developers are rarely involved in unit testing activities unless they are peer reviewing the work. Absence of a central framework makes it possible for each squad to adopt practices that they feel are significant and helps optimize their

respective squad. Within the squad individual developers own their respective code and associated unit tests. When completed, the code as well as unit tests are well documented and committed following a standard structure, making it easier for peer reviewers to review the work without having to discuss the details of tasks with the developer whose work is being reviewed.

Section 3: Unit Test Activities and Practices:

Within our squad, each developer who develops a task also writes unit tests for it, however, no automatic test generation suits were adopted by the team. This meant that the developers often spent 2x time on writing tests as compared to x time on development during each 2 week sprint. Most tasks ended with a small component that fits into the bigger product roadmap.

While individuals can choose TDD during development, within the squad TDD was not followed as a practice and most ended up writing tests after the development. Writing unit tests however was followed as a practice and is often well documented and published. Standard formats to publish tests helped others not only peer review but also visualize metrics such as code coverage at multiple hierarchical levels of the product apart from other product metrics to gain confidence. Developers typically write enough unit tests to get a code coverage of 80% - 90%.

Section 4: Environment and Tools:

The squad typically uses Scala to develop DaaS products and consequently use scala build tool (SBT) to build the code and tests. Github is used as a code repository and Backstage is used to visualize the product, its components along with relevant metrics such as code coverage, SLA, availability etc.

Backstage is an open source framework for building developer portals, powered by a centralized software catalog, Backstage orders microservices and infrastructure, enabling product teams to ship high-quality code quickly without compromising autonomy. Backstage unifies all infrastructure tooling, services, and documentation to create a streamlined development environment from end to end.

Since the data which the team was working with was enormous 50-60 million rows per hour and had a very high seasonality and variation, SBT was closely integrated with Ratatool, an in-house developed open source tool for random data sampling and generation. This tool helped create a pipeline of sampled data and run unit tests with each build or commit to the product repository.

Section 5: Discussion and Concluding Remarks:

The team follows many lean practices especially SCRUM cycles that helped the team maintain higher levels of productivity. While manually writing unit tests helps the team maintain both internal and external quality as well as catch data quality issues in the incoming data stream, it takes a significant proportion of effort from the team's capacity. One of the key practices I would want to explore is use of automatic test generation suits that can significantly reduce the effort on writing manual unit tests for each of the deliverables.

While code coverage and % tests passed are tracked and monitored both internally and externally, mutation testing can also be explored to monitor the efficiency of unit tests written. This would be a valuable metric to showcase efficiency of unit tests as well as improve unit tests development practices over time.

While automatic test generation tools are useful, tracking and classifying unit tests overtime can also help us create a standardized and configurable list of unit tests that the developers can use, reducing manual effort in writing some of them from scratch.

The code coverage is an important metric to evaluate a product quality, however, including our customers or downstream squads in defining the unit tests or their structure can potentially help catch quality issues and bridge gaps in understanding of the product(s). This could potentially also help improve collaboration and push customer driven product development to a greater level.

Since most products are small and old with new features in development, unit tests often do not generate any waste and contribute towards overall quality of the product. I also wonder how this may change when the product is new and is undergoing massive changes owing to continuously evolving requirements. Would unit tests generate significant waste in such scenarios? Looking at some of the products that are in their early stages of product development within the team's portfolio, I found multiple changes to entire features learning to re-writing of unit tests as well as waste generation. Maybe use of automated unit test generation suits might be more helpful in such situations. Also, talking to some of my friends outside of Spotify but part of development of much bigger products, I found unit tests generating significant wastes over time and consequently extensive use of automated test generation suits.

Given this, I feel efficiency of some of these practices may be dependent on product lifecycle as well as team structure. While the team adopts many lean principles, we can definitely try and evaluate other standardized practices to explore a more optimal process efficiency.

Part 2 – Creating unit tests:

Creating unit tests and using Java as a language has been a new learning experience for me. While I found it extremely insightful, it was challenging at first. The automatic test generation definitely makes it easier to write comprehensive tests for the programs. As a part of the exercise I got a chance to explore automatic test generation using both Randoop and EvoSuite. I found Randoop easier to customize and understand only because of its extensive documentation. I also used MuJava to generate mutations and validate my comprehensiveness of the tests that I wrote as well as those generated automatically using Randoop and EvoSuite.

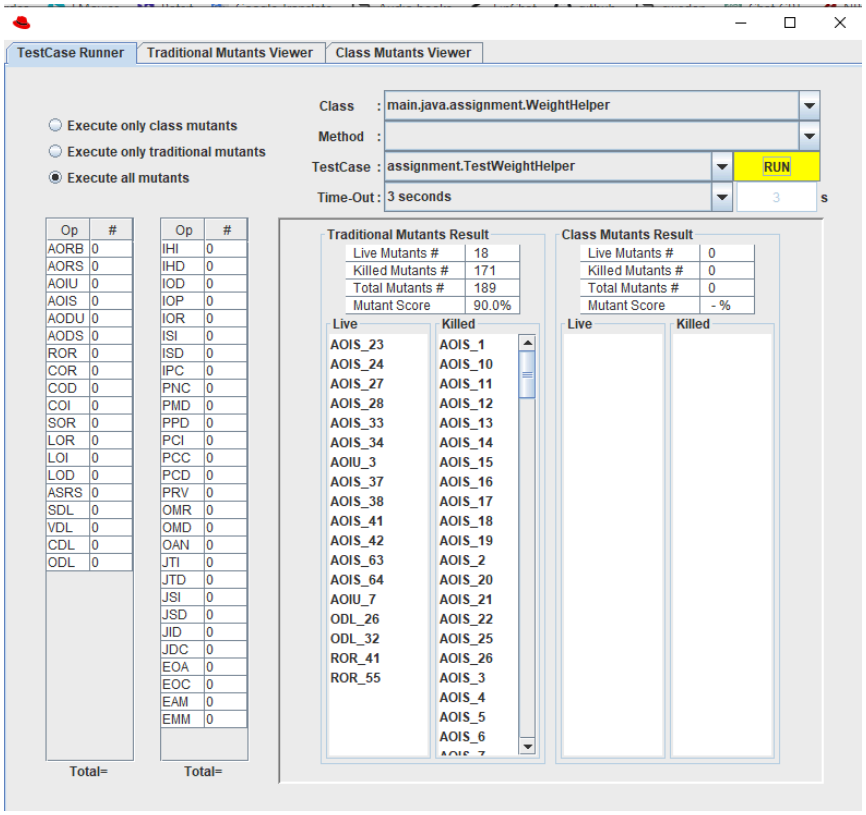
Summary of results : MuJava generated a total of 189 mutants for the Java program to calculate BMI.

*** Summary ***

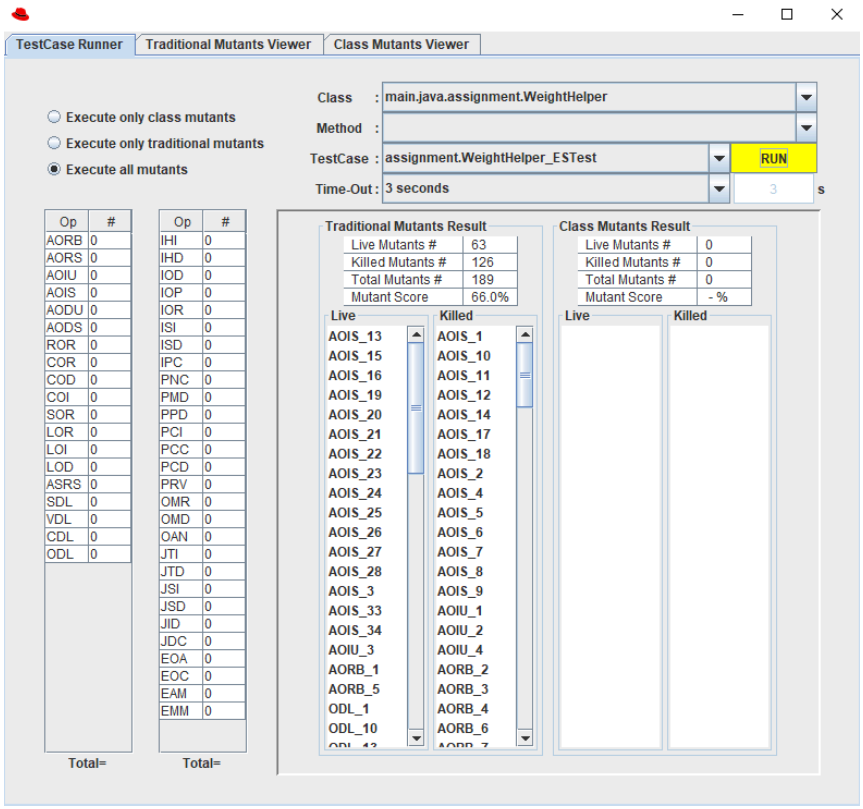
Op	#
AO...	8
AO...	0
AOIU	8
AOIS	60
AO...	0
AO...	0
ROR	63
COR	10
COD	0
COI	14
SOR	0
LOR	0
LOI	0
LOD	0
ASRS	0
SDL	9
VDL	3
CDL	0
ODL	14

Total : 189

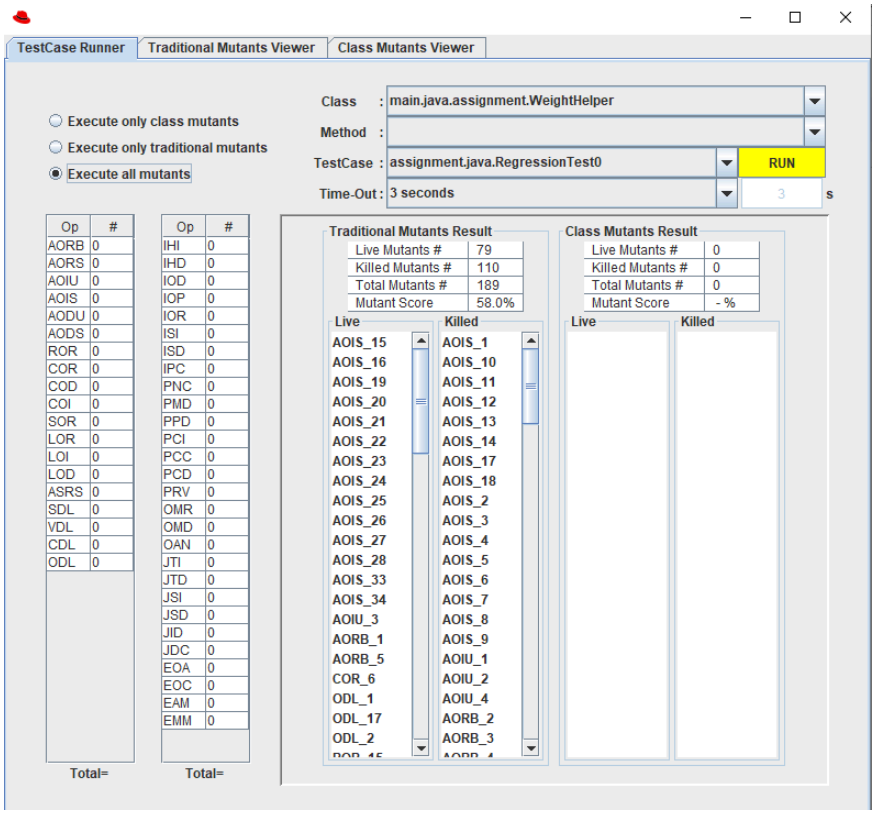
The manual written tests killed **90%** of the mutants with only 18 mutants live after tests simulations.



The automatic tests generated via EvoSuite killed **66%** of the mutants with 63 mutants living after test simulations.



The automatic tests generated via Randoop killed only **58%** of the mutants with 79 mutants living after test simulations. This shows the least performance among 3 test generations done. However with more simulation time, it may be possible to improve coverage.



The tests generated via automatic test generation suits (Randoop and EvoSuite) lack order and I found them difficult to understand at first. However, with experience I see them being more extensive and well structured when compared to manually written tests, whose structure may be influenced by individual testers. Comparing Randoop and EvoSuite, I found EvoSuite easier to understand. EvoSuite was able to achieve higher mutation coverage with much fewer tests compared to Randoop that generated 500 tests. EvoSuite also minimizes lines of code per test making it easier to understand, especially as a beginner.