Study of Assertions: Understanding Assertion Use in Java Projects on GitHub

Student Name: Bhavya Chopra
Roll Number: 2018333

BTP report submitted in fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Engineering

on May 11, 2022

BTP Track: Research

BTP Advisor
Dr. Rahul Purandare

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

I hereby declare that the work presented in the report entitled **"Study of Assertions: Understanding Assertion Use in Java Projects on GitHub"** submitted by me for the fulfillment of the requirements for the degree of *Bachelor of Technology* in *Computer Science & Engineering* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of my work carried out under guidance of **Dr. Rahul Purandare**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

.... **Place & Date:** ***IIIT-Delhi, 11-May-2022***
Bhavya Chopra

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

.... **Place & Date: ....**
Dr. Rahul Purandare

Abstract

Assertions are often used by programmers to test assumptions that they have about the program. Meaningful assertions often help in early detection of bugs, and also help developers in understanding the working of their code better. Formal verification is an important field with increasing studies in the domain focusing on it. However, most developers do not have the background for formal verification, and they use assertions to test their understanding about the code. Assertions used by developers are generally much weaker than inductive verification conditions. Understanding assertions itself is an important problem and might be a good first step toward generating inductive invariants and verification conditions as a study. Our work aims to understand the use of assertions across Java projects on GitHub. In later stages, we aim to provide an IDE based tool to generate meaningful candidate assertions at suitable program points using program analysis and deep learning techniques for software development based Java programs. This report discusses our experiment design, implementation and results for few research questions.

Keywords: Assertions, Java, GitHub, Spoon, Program Points, Code Complexity, Domains; Hurdle Regression, Reactive; Proactive; Program Analysis.

Acknowledgments

I sincerely thank my BTP advisor, Dr. Rahul Purandare, for providing guidance, valuable insights and support throughout this project. I would like to thank Anjali Singh (MTech student) and Lakshya A Agrawal (BTech student) for working on the project with me. I extend my gratitude towards Nikita Mehrotra (PhD student at Program Analysis Group) who constantly guided and encouraged us. Lastly, I would also like to acknowledge IIIT-Delhi for providing me the opportunity to work on this project as my BTech thesis.

Work Distribution

The work presented in this report has been completed over the Winter 2022 Semester (January 2022 - May 2022). Based on the work done in the previous semester, we continued the implementation and analysis of the study regarding the use of assertions across projects (Chapters 4 and 5). We present a literature review for the proposed tool and the study (Chapter 2). Chapter 6 elaborates results for the research questions. Chapter 7 describes the ongoing and future work for the project.

Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Motivation 1

2 Literature Review **3**

2.1 Assert Use in Java and C/C++ 3

2.2 Assertion and Unit Test Generation 4

2.3 Categorizing Projects into Domains 5

3 Study of Assertions across Programming Languages: Motivation **6**

3.1 Reviewing the Qualitative Study 6

3.1.1 Approach 6

3.1.2 Learnings 7

3.2 Additional Observations 8

3.3 Motivation for Study of Assertion Use 9

4 Experiment Design and Methodology **10**

4.1 Study Subjects 10

4.2 Research Questions and Methodology 11

5 Implementation and Qualitative Analysis **15**

5.1 RQ 1: Is there a correlation between the code complexity and use of assertions? . 15

5.1.1 Data Filtering 15

5.1.2 Computing SLOC and Cylomatic Complexity 16

5.1.3 Statistical Methods 16

5.2 RQ 2: Does the theme/domain of the project influence the use of assertions? . . 17

5.2.1 Previous Attempts 17

5.2.2 Categorizing Projects into Domains 18

5.2.3 Categorizing Assertions into Types 18

5.3 RQ 3: Are assertions added by developers proactively or reactively in projects? . 21

5.3.1 Data Collection 21
5.4 RQ 4: At what program points in a method are assertions most frequently seen? 21
5.4.1 Previous Attempts 21
5.4.2 Qualitative Analysis 23

6 Findings: The Use of Assertions in Java Projects 24
6.1 Correlation between SLOC, CC, Number of comments and use of Assertions . . . 24
6.2 Varying Use of Assertions with change in Project Domains 25
6.3 Reactive and Proactive use of Assertions 26
6.4 Program-points where Assertions frequently occur 27

7 Ongoing and Future Work 28
7.1 Ongoing work 28
7.2 Future Work 28

Chapter 1

Introduction

Assertions are boolean expressions connected to a program point, that need to evaluate to true at the program point during the execution of the program. If the boolean expression (predicate) evaluates to false during the execution of a Java program, JVM will throw an Assertion Error and the program will crash. Assertion statements in Java are executed when they are enabled. Disabling assertions would prevent evaluation of the predicate during runtime.

Assertions can be used to test assumptions that developers have about their code. Each successful execution of the program provides confirmation to the developer about their assumptions and helps in understanding if the code has any errors. Thus, assertions can help in early detection and correction of bugs. They can also act as documentation, helping the developer and co-developers in understanding the intended functionality of the method better. Assertions are often described as unit tests associated to a program point, that are performed with real data during code execution.

Studies have been conducted to report that developers can detect up to 80% of the bugs with the use of assertions [4]. However, assertions are generally missing in practice, or are poorly authored. Assertions are generally much weaker than inductive verification conditions and are used by programmers to test their understanding of the program, and generally not for formal verification.

1.1 Problem Statement

We aim to develop an in-depth understanding of how developers author assertions in Java projects spanning various domains. With this understanding, we aim to develop a novel tool based on program analysis and deep learning techniques, to automatically generate meaningful candidate assertions for Java programs spanning across various software development domains.

1.2 Motivation

Automatic generation of non-trivial assertions will involve (1) identification of appropriate program points, (2) identification of variables associated to the identified program point to be used in the assertion predicate, and (3) identifying the relation to be tested by the assertion. By answering these questions and generating candidate assertions, our tool can prompt and aid developers to think about the intended functionality of their code and understand it better while programming. This will help developers in writing error free programs, and promote best software development practices for the use of assertions. An in-depth understanding of the use of assertions will guide us in generating inductive verification conditions and invariants.

We are pursuing this project in two stages. The first stage is a detailed study of assertions, and the second stage is the automated generation of candidate assertions. The outcome of the first stage is going to be the input for the second. The first stage is itself an independent project and we plan to submit this work to a conference.

Chapter 2

Literature Review

As of December 2021, Java is among the most popular programming languages according to the TIOBE programming community index[1]. Java has also been ranked as one of the most popular programming languages by IEEE Spectrum[2] in 2020.

2.1 Assert Use in Java and C/C++

A study was conducted by Casalnuovo et al. that analyses 69 C and C++ opensource projects to show that assertions have a small effect on reducing the density of bugs and developers often add asserts to methods of which they have have prior knowledge of and larger ownership [5]. Their study also compares the frequency of assertions in projects belonging to different domains, but report that the domain of the project didn't significantly affect the number of assertions used.

Pavneet Singh and David Lo partially replicate the above study for Java projects and revisit the use of assertions [6]. Upon analysing 185 open-source Apache GitHub projects, they find that adding assertions to a method has a small, yet significant relationship with defect occurrence. They also report that developers who add an assertion to a method have greater experience and ownership of that method, as compared to developers who do not add assertions. Further, they also conduct an open card-sort study with 575 methods to understand the different types of assertions added by developers and the motivation behind adding them. They identify that developers often use assertions to check for null deferences, initialisation, process state, resource lock, implausible conditions, data length, and minimum and maximum value constraints.

Further, the study conducted by Baudry et al. reports that the use of assertions eases debugging for the developers, and that developers value the quality of assertions over their quantity [1].

We leverage the studies in this domain to gather insights about the significance of assertions for Java developers, and understanding the desired quality and characteristics of meaningful assertions. These studies will help us in developing a tool that matches the programmers' expectations and add value to the development process.

[1]TIOBE programming community index: `https://www.tiobe.com/tiobe-index/`

[2]IEEE Spectrum Ranking: `https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020`

2.2 Assertion and Unit Test Generation

Assertion generation using active learning

A study to automatically generate candidate assertions for a given program point using active learning has been conducted by Pham et al., considering complex Java programs as their target [8]. They utilise test cases already defined by the programmer and generate additional test cases via dynamic analysis (diakon-like[3] approach) for obtaining the feature vectors for the pre-defined/selected program point. They use SVM-based classifier for template based assertion generation and improve the recommended assertion via active learning. They conduct experiments for 425 methods and generate assertions at the beginning of the method (as a pre-condition), and are able to generate 211 assertions correctly (necessary and sufficient).

Assertion generation for formal verification

Wang et al. conduct a study for assertion recommendation for formal program verification for C programs [11]. Their approach involves developing a modified KNN Classifier using AST node properties as feature vectors, which predicts if a given method needs an assertion or not. They propose an algorithm to calculate the score for each variable based on the AST. The variable with the highest score is selected for being used in the assertion. The assertion checks if the variable is within the range calculated using CProver[4]. Their range finding algorithm interacts with CProver to check if the program can be formally verified for the default range, and based on the prover's feedback, the range is narrowed down. Evaluating the tool for SV-COMP benchmarks provides 92% accuracy rate for assertion necessity, with 84% precision rate and 86% recall rate.

Related studies

Recent studies have also focused on automated unit test assertion generation using program analysis and deep learning techniques, exactly matching 31% of the unit tests written by developers manually [12]. Their tool, ATLAS, can also match 50% of the unit tests written by developers when considering top-5 predicted tests.

AssertJ[5] is a Java library that generates `Assertions` class for focus methods and generates template based assertions with appropriate error messages.

These studies hint towards the usefulness of automated testing for software development and also provide reliable results. Automation in testing and assertion generation can support developers by helping them understand the code better. We aim to contribute to the domain of software engineering and practices by developing a reliable tool for automated assertion generation with high precision. To the best of our knowledge, this work is the first to focus on automated detection of program points and subsequent generation of meaningful assertions for software development oriented Java projects spanning across various domains.

[3]Diakon: `https://plse.cs.washington.edu/daikon/`
[4]CProver: `https://www.cprover.org/cbmc/`
[5]AssertJ: `https://assertj.github.io/doc/`

2.3 Categorizing Projects into Domains

Baishakhi Ray et al. study the effect of programming languages on software quality [9]. They use text analysis and clustering methods to group projects into domains, and categorize types of bugs to study the relation between project types and defect types across programming languages. They use Latent Dirichlet Allocation(LDA), a topic-modelling algorithm for the categorization of projects and defects. In their subsequent study of assertions in C and C++ projects, they use the same domain categorization for projects as derived earlier using LDA [5].

According to Borges et al., GitHub projects could be classified into six domains (i.e., Application Software, System Software, Web Libraries and Frameworks, Non-web Libraries and Frameworks, Software Tool, and Documentation) [34]. The first author along with another student annotator manually checked the domains of selected GitHub projects in this study. They propose the following categorization [3]:

- Applied Software: systems that provide functionalities to end-users (browsers; text editors)
- System software: systems that provide services and infrastructure to other systems, like operating systems, middleware, servers and databases.
- Web libraries and frameworks: These include frameworks such as angular.js and bootstrap
- Non-web libraries and frameworks: Such as Google/guava and Bitcoinj/bitcoinj
- Software tools: systems that support software development tasks, like IDEs, package managers, compilers.
- Documentation: repositories with documentation, tutorials, source code examples, etc.

This categorization has also been leveraged by Tingting Bi et al. in exploring accessibility issues in popular GitHub projects [2].

We plan on leveraging the above described techniqies such as LDA, or devised categorizations for our dataset via manual annotation.

Chapter 3

Study of Assertions across Programming Languages: Motivation

3.1 Reviewing the Qualitative Study

A brief qualitative study was performed in the Winter 2021 semester, to understand the use of assertions by Java developers. The findings of this study have been helpful in motivating and defining research questions for the subsequent study on assertion use being pursued this semester. It also provided us insights into techniques for generation of candidate assertions. Following is a brief review of the approach and findings.

3.1.1 Approach

To understand the use of assertions by Java developers, we collected 132 most popular Java repositories, hosted on GitHub[1], on the basis of their popularity (using star count). The cloned repositories spanned across various software development categories, such as algorithms, program meta-analysis and development tools, web utility, database management, data processing, authentication, and visualisation applications. Following this, I extracted all classes containing at least one assertion in the method block using Spoon[2]. 32 projects among 129 parsed projects contained at least one assertion.

Table 3.1 summarises the statistics for the extracted assertions. The JSON files for the projects can be found here: `https://github.com/BhavyaC16/Assertion-Generation/tree/master/utils/assertion_parser/extracted_asserts_using_spoon/json`.

For understanding the use of assertions, we manually went through every class with assertions and tried to understand the programmer's intent behind adding the assertion statement, as well as the functioning of the code. We observed patterns in the use of assertions across projects to corroborate findings presented in section 3.1.2.

[1]List of collected repositories: Link *(Curated on 24-March-2021)*

[2]Spoon: `https://spoon.gforge.inria.fr/`

Total Projects	129
Projects with Assertions	32
Total Assertions	**2279**
Total Methods	1649033
Total Methods with Assertions	1612
Total Classes	62945
Total Classes with Assertions	708

Table 3.1: Summarising results of assertion extraction using our parser

3.1.2 Learnings

Expected use of Assertions

- Use of assertions to check method preconditions, postconditions and invariants.
- Checking if an object has been instantiated before use in subsequent operations/checking specific properties of an object via null deference checks.
- Checking the bounds of input arguments in private methods.
- Testing algorithm implementations with specific input values in driver methods.

Unintended use of Assertions

- Use of assertions in public methods to validate/check input arguments as the first statement.
- Using an assertion as a mandatory check for subsequent operations.
- Using assertions to do work that the program needs for correct operation.

New insights about assertion use

- The developers use `assert false;` or `throw new java.lang.AssertionError();` at points where they expect execution not to reach.
- Assigning the return value of a function call to a variable, and performing assertion checks on this variable. In other words, checking if the function has returned the expected value.
- Multiple assertions involving variables that are related to one another, or whose values will hold a relation at given program points.
- Use of methods and object properties that return values with simple data types (boolean, short int, and enum).
- Lastly, we observe that assertions are most frequently used as the first statements in a method to validate the values of input arguments, or, as the last statements, just before the returning, to check if the value being returned is appropriate.

3.2 Additional Observations

We noticed that there was a huge variation in the number of assertions used in each of the 32 projects:

1. **Minimum assertions:** 1, **Maximum assertions:** 1108
2. **Median:** 6 assertions, **Mean:** 71.22 assertions, **Standard Deviation:** 196.3 assertions
3. **Quartiles:**
   (a) 1st quartile (25th Percentile): 2 assertions
   (b) 2nd quartile (50th Percentile/Median): 6 assertions
   (c) 3rd quartile (75th Percentile): 64 assertions

Through manual analysis, we also identified that several repositories make extensive use of only few types of assertions, as categorized by Pavneet Singh and David Lo [6]. For instance, repositories like netty/netty[3] (an event-driven asynchronous network application framework) and radsz/jacop[4] (A Java Constraint Programming solver) make extensive use of assertions, 168 and 1108 assertions respectively, with varying use cases (types) or structure of assertions.

The netty/netty repository majorly performed only length checks for buffer arrays and resource lock checks using assertions since it is an asynchronous network based application. *(Derived from netty-netty.json)*:

```
 // Checking if destination buffer has space to hold encrypted content in
     bioReadCopyBuf
    assert bioReadCopyBuf.readableBytes() <= dst.remaining()
// Resource Lock Check
    assert java.lang.Thread.holdsLock(this);
```

Similarly, radsz/jacop repository mostly performed only maximum and minimum variable constraint checks and null condition checks, since it is a constraint solver project. *(Derived from radsz-jacop.json)*:

```
// first pass
        firstPass();
// Good ranges for match1 and match1XOrder
         assert checkFirstPass();
         secondPass();
         assert checkSecondPass();
         thirdPass();
         assert checkThirdPass();
```

This motivated us to explore if there exists a relation between the domain of the project, and the type of assertions primarily used by developers for such projects.

[3]netty/netty: `https://github.com/netty/netty`
[4]radsz/jacop: `https://github.com/radsz/jacop`

3.3 Motivation for Study of Assertion Use

The observations from the above explorations motivated us to conduct an in-depth study of assertions across programming languages for a better understanding of how developers use assertions in practice.

The large standard deviation in the number of assertions used in projects motivated us to explore the dependence of assertion use (frequency, as well as type of assertions) on code size, complexity, and the programming language or paradigm. The increased use of specific types of assertions in projects with different themes motivated us to understand the dependence of a project's domain or theme on the types of assertions. If such a relation can be observed, this shall enable us in generating context or domain-aware candidate assertions. Lastly, we aim to understand where (at what program points) are assertions added and which variables and properties are checked for in an assertion predicate at a specific location. These explorations will help us answer how an assertion predicate can be structured, and inserted in most appropriate and meaningful locations.

The following chapters discuss the research questions for the study, and the initial explorations and findings.

Chapter 4

Experiment Design and Methodology

The objective of this study is to analyse the use of assertions in software development projects. Java has been ranked among the top 3 most popular programming languages on GitHub since 2014 by Octoverse[1]. As of December 2021, Java is one of the most popular programming languages according to the TIOBE programming community index[2]. Java has also been ranked as one of the most popular programming languages by IEEE Spectrum[3] in 2020. The study of assertions Java GitHub projects will help us understand how assertions are used across project domains. The findings of the study can then be used to automatically generate candidate assertions that align with the best software development practices.

4.1 Study Subjects

I collected 1000 most popular repositories for Java, hosted on GitHub. We used stars as a metric for project popularity since it represents the number of people who are aware about the project and maybe found it to be useful for bookmarking, or starred it as a token of appreciation. I extracted information about these repositories using the PyGitHub Wrapper[4], and cloned the primary development branch for each of these repositories for analysing the use of assertions, while storing the latest commit id.

Further, to clean the dataset, I identified if the cloned repositories were forks or duplicates of each other, and removed such instances for each language. Very few repositories belong to 'archived' organisations or users, and have not been removed from the dataset yet. Following this filtering process, we obtained 750 repositories for Java. The cloned repositories spanned across various software development categories, such as algorithms, program meta-analysis and development tools, web utility, database management, data processing, authentication, and visualisation applications.

Next, we implemented programs to parse each repository and check if it contains at least one

[1]GitHub Octoverse Top Languages: `https://octoverse.github.com/#top-languages-over-the-years`
[2]TIOBE programming community index: `https://www.tiobe.com/tiobe-index/`
[3]IEEE Spectrum Ranking: `https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020`
[4]PyGitHub Wrapper: `https://pygithub.readthedocs.io/en/latest/`

assertion. I used Spoon [5] [7] for Java programs, and found that 152 projects from the dataset of 750 projects contained at least one assertion. Further, only 95 repositories contained at least one assertion not belonging to test files. I used the keyword ‘test’ to filter out the test files, as assertions used in test cases are different from the ones used in the production code. All assertions present in test files were discarded from the dataset. This allowed us to filter out projects that do not contain any assertions in production code. We will be using only the projects that contain at least 1 assertion for the study. Following are some metrics for Java repositories containing assertions:

- **Minimum assertions:** 1, **Maximum assertions:** 2813
- **Median:** 5 assertions, **Mean:** 66.611842 assertions, **Standard Deviation:** 267.948543 assertions
- **Quartiles:**
  - 1st quartile (25th Percentile): 2 assertions
  - 2nd quartile (50th Percentile/Median): 5 assertions
  - 3rd quartile (75th Percentile): 19.5 assertions

4.2 Research Questions and Methodology

The general observation is that certain projects make excessive use of assertions, whereas other projects make extremely limited or no use of assertions. Further, Java is an object oriented programming language, Python is a scripting language, while C is a procedural programming language. Python is appreciated for its brevity and their expressiveness. On initial attempts, we also noticed that the number of assertions in Python are substantially higher than those observed in Java and C projects. Despite the fact that Python programs are often 3-5 times shorter than equivalent Java programs, Python projects seem to utilise more assertions. The high variance and standard deviation in frequency of assertions across the projects leads us to ask research questions 1 and 2 below.

RQ 1: Is there a correlation between the code complexity and use of assertions?

To explore this question, we will perform multiple regression analysis, where the dependent variable is the frequency of assertions, and the independent variables for analysis could be code size, code complexity, number of contributors, number of bugs fixed, and the domain of the project.

We aim to use Source Lines of Code (SLOC) and Cyclomatic Complexity as the metrics to gauge code complexity. SLOC is an appropriate metric since it is easy to compute, and gives us the number of executable lines of code. Cyclomatic complexity measures the number of linearly independent paths through the source code of a program, and can be computed on the basis of the control flow graph. It takes into account nested blocks of statements and the number of paths through them.

[5] A metaprogramming library to analyze and transform Java source code

*Cyclomatic Complexity = E - N + 2*P*

Here, E denotes the number of edges in the control flow graph, N denotes the number of nodes, and P denotes the number of nodes with exit points. We utilize this metric as code size might not be the only reason for adding assertions. They may be added for reducing the number of software bugs and make the programmer's assumptions clear in a complex piece of code. Code complexity can be one of the reasons for adding assertions, for instance, a program may have multiple execution paths. As a result, a mistake in one procedure might impact the entire code.

Understanding how the usage of assertions is affected by the Lines of code, and the code complexity will help us answer the large variation in use of assertions across languages and projects.

RQ 2: Does the theme/domain of the project influence the use of assertions?

As observed previously, projects in different domains seem to make use of different types of assertions primarily. Exploring this question will help us check if such patterns exist, and also generate candidate assertions that are domain-aware. It will enable us to understand the different types of assertions being used in context of the project domain.

For adressing this question, we plan on using LDA (Latent Dirichlet Allocation) algorithm to categorize domains, followed by manual inspection to verify domains. Finding a consistent set of domains across languages is a challenge, since different languages support the development of projects in certain specific domains.

Following this classification, we aim to perform a semi-automated analysis (rule based checking, accompanied by manual analysis) to categorize type of assertions being used in projects, as proposed by Pavneet Singh and David Lo [6]: Null Condition Check, Process State Check, Initialisation Check, etc.

RQ 3: Are assertions added by developers proactively or reactively in projects?

We wish to understand the intention with which a developer adds an assertion to a piece of code, and whether it was added proactively, or reactively. An assertion is considered to be added proactively if it was added or modified irrespective of bug occurrence, and reactively if it is added or modified as a reaction to a bug detection or bug fix. Proactively added assertions would likely reflect the programmer's assumptions about the code, including preconditions and postconditions, whereas reactively added assertions will likely behave as checks.

To address this question, we shall write a parser to collect the history of all methods of a project, and analyse the git logs and associated commit messages. We will consider all commits where an assertion has either been added, modified or deleted. We can perform automated classification of proactive and reactive assertions on the basis of the corresponding commit messages. If the message indicates a bug fix, with the use of keywords such as 'error', 'defect', 'flaw', 'bug', 'fix', 'issue', etc, then we can assume the assertion to be reactive. However, if the commit message indicates no such bug fix, then the assertion can be classified to be proactive.

RQ 4: At what program points in a method are assertions most frequently seen?

This question will help us understand at what points in a program is an assertion most needed or helpful. For this, we aim to identify patterns in paths from the function entry point up to the assertion statement, and from the assertion up to the end of the method body. For instance, consider the following code in Java:

```
int foo(){
    int i = 10;
    i++;
    assert(i>5);
    while(i>0){
        assert(i<20);
        i--;
    }
    return(i)
}
```

We aim to obtain the following sequences/sentences for the above code:

```
// Types of statements leading up to the assertion(s):
    CtLocalVariable CtUnaryOperator CtAssert
    CtLocalVariable CtUnaryOperator CtAssert CtWhile nest

// Types of statements following the assertion(s):
    CtWhile CtReturn
    CtUnaryOperator nest-end CtReturn
```

We plan on using clustering algorithms to generate clusters on the basis of these path sequences as features. Following this, a manual inspection of clusters will be conducted to develop a hypothesis about why certain patterns form a cluster.

The following research questions aim to explore which variables, and their properties are checked by the assertion, and how the assertion predicate is structured. Further, we wish to explore where the variable was last updated or accessed, and if it has any relation with the assertion.

RQ 5(a): Which variables are evaluated by the assertion predicate at these program points?

We plan to consider the method containing the assertion as the scope for the analysis.

1. We shall analyse the type of variable(s) being checked in the assertion predicate: int, boolean, string, character (primitive types), or objects of a class, or literals.
2. Analysis of the scope of the variable (input argument, or local variable, or class property, or global variable).
3. Mapping the location of the assertion with respect to the variables' declaration, updation and use.
4. Following these automated analyses, we will consider a subset of methods and develop understanding of the reason behind choice of variable.

5. Analysing the logical role of a variable, as understood by a programmer, in a method (For instance, counter, iterator, memoization array, buffer, etc.) will be helpful in understanding if it must be checked for in an assertion or not.

RQ 5(b): What types of expressions, operators and variables are evaluated in assertion predicates?

With this research question, we want to understand patterns in assertion predicates. We aim to convert assertion predicates to ASTs and exploring the structures and use of operators in assertion predicates. Further, we could categorise the assertions based on their intended use by developers, findings of which can be used as validation for existing studies, in addition to proposing new templates or types of assertions.

Chapter 5

Implementation and Qualitative Analysis

This chapter describes the implementation details for each research question, and description of qualitative analysis performed for the dataset. We performed qualitative analysis for RQs 2, 4, and 5, where three annotators manually annotated a representative sample size of methods with assertions (Population Size: 2146, Confidence Level: 95%, Confidence Interval: 5, Representative Sample Size[1] 326 methods). We conducted discussions amongst ourselves to form a common understanding of the annotation tasks. This was followed by independently rating a randomly selected subset of methods with assertions and computing the inter-rater reliability score. We compute the inter-rater reliability scores for each rating activity as described by Anthony J. Viera et al. [10]. Upon obtaining scores between 0.81 to 0.95, indicating **"Almost Perfect Agreement"**, for all annotation tasks, we proceed with splitting the annotation rows and marking them individually. For each annotation row, the raters indicated their level of confidence as 'High', 'Medium' and 'Low'. All rows marked as 'Medium' were validated by another annotator, and all rows marked as 'Low' were discussed collectively and then annotated.

5.1 RQ 1: Is there a correlation between the code complexity and use of assertions?

5.1.1 Data Filtering

The data reflects that very few methods in the repositories contain assertions. To understand the dependence between code complexity and the use of assertions, we developed a three-step filter to mitigate confounding factors such as developer unawareness about the use of assertions. In this two-step filter, we first extracted all files and line numbers where assertions are present, using the Visitor Design Pattern in Spoon to traverse the AST. Next, I fetched information about all authors who have authored at least one assertion. In the second step, we filtered out all methods that have been authored by any of the developers identified in the first step.

I developed this tool using the PyGitHub wrapper and parsing the commit messages obtained using the following command: **git log -L <line_number>,<line_number>:file_path**

The second step helped identify 248 unique developers who have authored at least one assertion. In the third pass, I filter all methods (with and without assertions) on the basis of whether or not

[1]Calculated using: `https://www.surveysystem.com/sscalc.htm`

have they been authored (fully or partially) by any of the identified authors. This methodology will help us in understanding how developers who are aware of assertions decide whether to add or not to add an assertion to a method. We obtain approximately 1 lakh out of 4 lakh total methods that have been authored by the identified developers. This also reflects the finding reported by Pavneet Singh and David Lo's paper, that developers who add assertions often have high ownership of the code.

5.1.2 Computing SLOC and Cylomatic Complexity

I worked on implementing AST traversers by extending the CtScanner class in Spoon, and implemented the calculation of Source Lines of Code (SLOC) and Cyclomatic Complexity (CC) from scratch as follows:

- SLOC implementation:
  - For each method obtain the starting and ending line numbers.
  - Subtract the starting line number from the ending line number to obtain SLOC for the method.
- CC Calculation:
  - Start with CC = 1 for each method
  - Add 1 if the operators && or || are used in any condition
  - Add 1 for break statement
  - Add 1 for each case (in switch-case construct)
  - Add 1 for catch
  - Add 1 for conditional (? and :)
  - Add 1 for continue statement
  - Add 1 for Do loop
  - Add 1 for For loop
  - Add 1 for ForEach loop
  - Add 1 for if construct
  - Add 1 for throw statement
  - Add 1 for finally block
  - Add 1 for While loop

5.1.3 Statistical Methods

The derived data contained a large number of zeroes and it became difficult to analyze the relationship between code complexity and use of assertions. We employed the Hurdle Regression Poisson Model and Hurdle Negative Binomial Regression Model to obtain findings. This model has two components. The hurdle component models overcoming a hurdle, in this case, the effect of going from 0 assertions to 1 assertion. The count component models the effect of going from a non-zero value to another non-zero value. We perform this statistical analysis for three independent variables: SLOC, CC and Number of Comments; where we consider the frequency of assertions to be the dependent variable.

5.2 RQ 2: Does the theme/domain of the project influence the use of assertions?

5.2.1 Previous Attempts

This section discusses an initial attempt to categorise projects into domains for RQ 2. Most GitHub projects have one line project descriptions, tags, and readme files. These features can be treated as documents that describe the project. Initially, I have considered the one-line project descriptions from 152 Java projects as the documents. I then used Latent Dirichlet Allocation (LDA) for these documents, which is a topic-modelling algorithm. I used the Google Translate API to filter out 19 project documents that were written in languages like Mandarin and Spanish, to consider only documents written in English for running LDA for **10 topics**.

Given these documents, LDA identified a set of topics where each topic is represented as probability of generating different words by the bag-of-words model, and also using TF-IDF vectors. Following is the probability of words for 10 topics using TF-IDF:

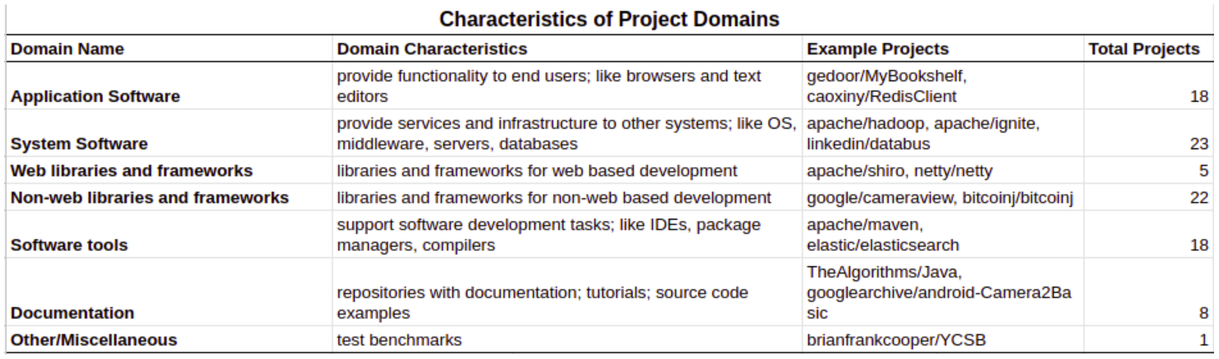
```
// Topic 1
Words: 0.082*"android" + 0.055*"library" + 0.054*"high" + 0.052*"framework" +
    0.042*"build" + 0.036*"java" + 0.033*"process" + 0.028*"performance" +
    0.027*"test" + 0.027*"fast"

// Topic 2
Words: 0.090*"tool" + 0.058*"material" + 0.055*"design" + 0.055*"android" +
    0.046*"spring" + 0.040*"library" + 0.039*"application" + 0.037*"java" +
    0.031*"platform" + 0.029*"implement"

// Topic 3
Words: 0.142*"android" + 0.075*"library" + 0.070*"native" + 0.055*"image" +
    0.053*"simple" + 0.047*"project" + 0.038*"video" + 0.037*"powerful" +
    0.032*"flexible" + 0.030*"develop"

// Topic 4
Words: 0.158*"java" + 0.124*"android" + 0.055*"library" + 0.032*"code" +
    0.027*"client" + 0.023*"support" + 0.023*"framework" + 0.021*"provide" +
    0.020*"performance" + 0.018*"design"

// Topic 5
Words: 0.152*"source" + 0.115*"open" + 0.075*"android" + 0.069*"data" +
    0.044*"dynamic" + 0.039*"platform" + 0.034*"distribute" + 0.032*"google" +
    0.031*"message" + 0.023*"version"

// Topic 6
Words: 0.088*"support" + 0.065*"android" + 0.057*"spring" + 0.051*"project" +
    0.045*"recyclerview" + 0.042*"boot" + 0.038*"design" + 0.037*"load" +
    0.034*"screen" + 0.030*"cloud"

// Topic 7
Words: 0.322*"android" + 0.044*"library" + 0.041*"custom" + 0.036*"effect" +
    0.031*"support" + 0.026*"style" + 0.025*"view" + 0.023*"video" +
    0.021*"deprecate" + 0.020*"time"

// Topic 8
```

```
Words: 0.098*"android" + 0.096*"spring" + 0.088*"base" + 0.065*"file" +
    0.044*"library" + 0.040*"cloud" + 0.031*"background" + 0.027*"mybatis" +
    0.023*"boot" + 0.021*"architecture"

// Topic 9
Words: 0.063*"java" + 0.055*"code" + 0.044*"data" + 0.044*"management" +
    0.030*"support" + 0.030*"engine" + 0.030*"distribute" + 0.029*"redis" +
    0.029*"spring" + 0.027*"service"

// Topic 10
Words: 0.100*"view" + 0.071*"android" + 0.071*"apache" + 0.068*"library" +
    0.064*"animation" + 0.056*"support" + 0.038*"easy" + 0.036*"feature" +
    0.028*"wechat" + 0.027*"components"
```

Further, for each document, LDA also estimates the probability of assigning a particular document to each topic. Estimating the appropriate bucket size for LDA and performing manual annotation for consolidating results obtained via LDA was challenging. We thus decided to leverage the project domain categorization proposed by Borges et al. and described in the related work section.

5.2.2 Categorizing Projects into Domains

We leverage the categorization proposed by Borges et al. and we manually annotated the 95 repositories into the provided 6 domains. Of these, only 1 repository could not be added to any of the previously proposed domains since it was a benchmarking repository. The following table shows the project distribution:



Characteristics of Project Domains

Domain Name	Domain Characteristics	Example Projects	Total Projects
Application Software	provide functionality to end users; like browsers and text editors	gedoor/MyBookshelf, caoxiny/RedisClient	18
System Software	provide services and infrastructure to other systems; like OS, middleware, servers, databases	apache/hadoop, apache/ignite, linkedin/databus	23
Web libraries and frameworks	libraries and frameworks for web based development	apache/shiro, netty/netty	5
Non-web libraries and frameworks	libraries and frameworks for non-web based development	google/cameraview, bitcoinj/bitcoinj	22
Software tools	support software development tasks; like IDEs, package managers, compilers	apache/maven, elastic/elasticsearch	18
Documentation	repositories with documentation; tutorials; source code examples	TheAlgorithms/Java, googlearchive/android-Camera2Basic	8
Other/Miscellaneous	test benchmarks	brianfrankcooper/YCSB	1

Figure 5.1

5.2.3 Categorizing Assertions into Types

David Lo and Pavneet Singh's paper described the following types of assertions:

- Null Condition Check
- Process State Check
- Initialization Check
- Resource Check

- Resource Lock Check
- Minimum and Maximum Value Constraint Check
- Collection Data and Length Check
- Implausible Condition Check

We studied and discussed this section from the paper in detail and conducted a session to sort 20 randomly selected assertions into the provided 8 categories. Upon observing that 3 assertions did not belong to any of the previously mentioned categories, we decided to group them under "OTHER" for an open-card sort of these assertions following the annotation of all assertions in the selected methods that formed a representative sample. Next, we conducted an inter-rater reliability test, and the calculated kappa score was 0.89, indicating high agreement.

Following the closed-card sorting method, we categorized 112 assertions under the "OTHER" category. We removed 6 assertions (all were instances of "assert true") from our dataset as they did not perform any checks and were authored trivially. We conducted discussions and three iterations of open-card sorting among the annotators to either expand/adapt/modify the existing categorization, or introduce new categories. As an outcome of this activity, we propose three new categories and modification of two previously defined categories:

- **Type Check:** We observed frequent type checks with the use of 'instanceof' operator in Java, and checking values with the use of enum data type. The instanceof operator is used by developers in assertion predicates to verify instances of a class or super-class, and interfaces. The enum data type is also frequently used to check for values held by certain input variables or class properties. The following examples illustrate the use of enum as well as instanceof:

```java
// apache/rocketmq-externals
public static MqttMessage getMqttPubcompMessage(MqttMessage message) {
        assert message.fixedHeader().messageType() == MqttMessageType.PUBREL;
        MqttFixedHeader fixedHeader = new MqttFixedHeader(
            MqttMessageType.PUBCOMP,
            message.fixedHeader().isDup(),
            message.fixedHeader().qosLevel(),
            message.fixedHeader().isRetain(),
            message.fixedHeader().remainingLength()
        );
        return new MqttMessage(fixedHeader);
    }
```

```java
// mockito/mockito
  private HashSet<HashCodeAndEqualsMockWrapper> asWrappedMocks(Collection<?>
      mocks) {
        Checks.checkNotNull(mocks, "Passed collection should notify() be null");
        HashSet<HashCodeAndEqualsMockWrapper> hashSet = new HashSet<>();
        for (Object mock : mocks) {
            assert !(mock instanceof HashCodeAndEqualsMockWrapper) : "WRONG";
            hashSet.add(HashCodeAndEqualsMockWrapper.of(mock));
        }
        return hashSet;
    }
```

- **Arithmetic or Logic Comparison Check:** We observed several algorithmic, arithmetic and logic operations being performed on ordered data and strings, that assert invariants and post-conditions. These could not be classified under minimum and maximum value constraint checks as these were defined to check if a given value is "above a certain minimum limit or less than a certain maximum value". We observed equality checks, string comparisons, validation of data structures post operations and checks on their properties as important invariants or post-conditions to be categorized under arithmetic and logic comparisons.

```java
// aistrate/AlgorithmsSedgewick
 /**
    * Delete and return the smallest key on the priority queue.
    * Throw an exception if no such key exists because the priority queue is
        empty.
    */
   public Key delMin() {
       if (N == 0) throw new RuntimeException("Priority queue underflow");
       exch(1, N);
       Key min = pq[N--];
       sink(1);
       pq[N+1] = null;       // avoid loitering and help with garbage collection
       if ((N > 0) && (N == (pq.length - 1) / 4)) resize(pq.length / 2);
       assert isMinHeap();
       return min;
   }
```

```java
// TheAlgorithms/Java
assert isFibonacciNumber(1);
public static boolean isFibonacciNumber(int number) {
       return isPerfectSquare(5 * number * number + 4) || isPerfectSquare(5 *
           number * number - 4);
   }
```

- **Compound Disjunction Check:** We observed few (5) assertions that make use of disjunctions and the assertion predicates belong to more than one of the previously identified categories. We handled conjunctions by counting them separately in the respective assertion categories, since they were equivalent to being split and authored as two assertions one after the other in the code. However, disjunctions posed a challenge as the predicate has to be seen as a whole. Hence, we propose this new category.

```java
// We either expect to have no Channel in the map with the same FD or that the
    FD of the old Channel is already
// closed.
assert old == null || !old.isOpen();
```

```java
assert keys.isEmpty() || keys.iterator().next() instanceof String;
```

We also propose modifications to previously defined categories by David Lo and Pavneet Singh:

- **Null Condition Check:** We propose expanding the previous definition and adding assert null conditions to this same category.

```
// oracle/opengrok
for (String name : factory.getFileNames()) {
        AnalyzerFactory old = FILE_NAMES.put(name, factory);
        assert old == null :
                "name '" + name + USED_IN_MULTIPLE_MSG;
}
```

- **Minimum and Maximum Value Constraint Check:** We propose adding instances of equality check in numeric data types as pre-conditions to this category, as they can be viewed as a strict case of minimum and maximum value constraint checks.

```
// h2oai/h2o-2
public final void scoreCrossValidation(Job.ValidatedJob job, Frame source,
    Vec response, Frame[] cv_preds, long[] offsets) {
    assert(offsets[0] == 0);
}
```

5.3 RQ 3: Are assertions added by developers proactively or reactively in projects?

5.3.1 Data Collection

Similar to RQ 1, I wrote a parser using PyGitHub wrapper to identify all commits that add or modify an assertion. Upon obtaining 846 commits, I parse each commit to obtain the commit message, commit description, and the diff files. We use these to annotate whether an assertion has been added proactively or reactively. The presence of words such as 'error', 'flaw', 'bug', 'fix', 'problem', etc indicate that the assertion might be added as a consequence of bug identification. We perform qualitative analysis to understand reasons for assertions being added reactively or proactively by developers, and analyze the distribution of such commits.

5.4 RQ 4: At what program points in a method are assertions most frequently seen?

5.4.1 Previous Attempts

For RQ 4, I implemented a **custom scanner class**[2] which extends the *CtScanner* Class and overrides the Visitor Pattern provided by Spoon for all Statement types and Blocks in Java: CtMethod, CtAssert, CtLambda (currently ignores lambda within a block), CtAssignment, CtBlock, CtBreak, CtCase, CtCatch, CtConstructorCall, CtContinue, CtDo, CtFor, CtForEach, CtIf, CtInvocation, CtLocalVariable, CtOperatorAssignment, CtReturn, CtSwitch, CtSynchronized, CtThrow, CtTry, CtTryWithResource, CtUnaryOperator, CtWhile, CtYieldStatement.

The class maintains stacks of strings and lists of sentences to obtain sequence of statements leading up to the assertion statement, and following the assertion statement in each method as

[2]Link to MyCustomScanner Class: `https://gist.github.com/BhavyaC16/fe6d3e626badbd4b1216f0b7ab9727a4`

described in chapter 4. The implementation works correctly for any depth of nested blocks.

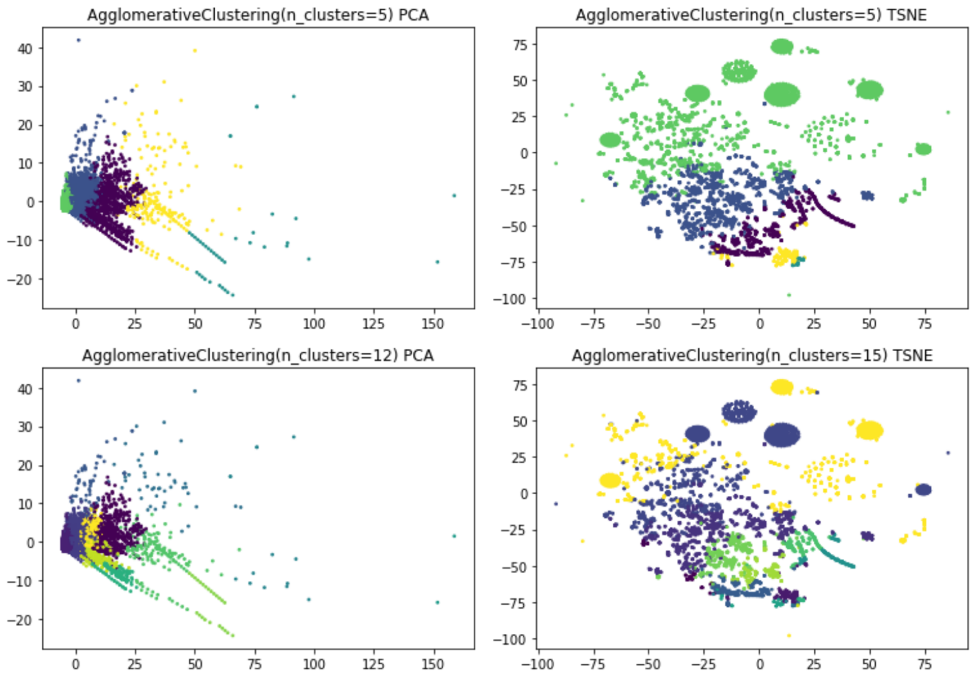


Figure 5.2: Agglomerative clustering for varying buckets

Considering the following motivating example:

```
void WeekendDays(){
    int num_weekdays = 5;
    int num_totaldays = 7;
    String[] weekends = {"Friday", "Saturday", "Sunday"};
    assert weekends.length == (num_totaldays - num_weekdays);
    return(weekends);
}
```

We will obtain the following sentences using the custom scanner class for this method:

- ‘Statements so far’ string: CtAssignment CtAssignment CtAssignment CtAssert
- ‘Statements beyond’ string: CtReturn

Using this pair of strings for each assertion that occurs in a project, we can construct n-grams (up to a limit) as follows:

- 1-gram: **CtAssert**
- 3-gram: CtAssignment **CtAssert** CtReturn
- 5-gram: CtAssignment CtAssignment **CtAssert** CtReturn None
- 7-gram: CtAssignment CtAssignment CtAssignment **CtAssert** CtReturn None None

These n-grams were then clustered using agglomerative clustering after being one-hot encoded. We had explored agglomerative clustering using the ‘Statements so far’ string as a feature. The results can be seen in figure 5.2. After performing clustering, it became challenging to develop a hypothesis for the clusters formed.

5.4.2 Qualitative Analysis

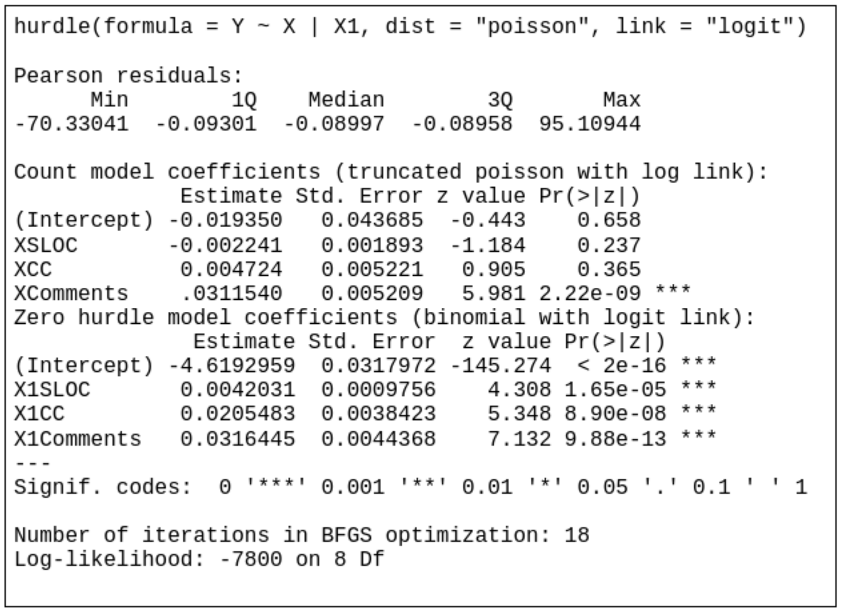
We then moved to performing qualitative analysis to identify the programming constructs before or after which assertions occur, and manually annotated the representative dataset, while forming intuition about the context in which the developer must have added the assertion, and reflect on the intent of the developer.

Chapter 6

Findings: The Use of Assertions in Java Projects

6.1 Correlation between SLOC, CC, Number of comments and use of Assertions

The below table shows coefficients computes for the hurdle model. We make the following interpretations from this model:



```
hurdle(formula = Y ~ X | X1, dist = "poisson", link = "logit")

Pearson residuals:
      Min        1Q    Median        3Q       Max
-70.33041  -0.09301  -0.08997  -0.08958  95.10944

Count model coefficients (truncated poisson with log link):
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.019350   0.043685  -0.443    0.658
XSLOC       -0.002241   0.001893  -1.184    0.237
XCC          0.004724   0.005221   0.905    0.365
XComments    .0311540   0.005209   5.981 2.22e-09 ***
Zero hurdle model coefficients (binomial with logit link):
              Estimate Std. Error  z value Pr(>|z|)
(Intercept) -4.6192959  0.0317972 -145.274  < 2e-16 ***
X1SLOC       0.0042031  0.0009756    4.308 1.65e-05 ***
X1CC         0.0205483  0.0038423    5.348 8.90e-08 ***
X1Comments   0.0316445  0.0044368    7.132 9.88e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Number of iterations in BFGS optimization: 18
Log-likelihood: -7800 on 8 Df
```

Figure 6.1

- The SLOC variable has **positive correlation for the hurdle component**. To estimate the impact of SLOC, we can take the exponential of the coefficient. Adding the first Line of Code to a method leads to an expected increase in the number of assertions by a factor of exp(0.0042031) = 1.0042098366 or 0.42%. The p-value is less than 0.001, showing that the result is statistically significant. This shows that adding the first line of code to a method has a significant impact on the assert occurrence of the method.

- The coefficient corresponding to the SLOC variable is **negative for the count component**. By taking the exponential of the SLOC coefficient, we can find that a unit decrease in the SLOC relates to a decrease in adding an assertion to a method by a factor of exp(-0.002241) = 0.997761 or 0.2241%.

- We can note that the coefficient corresponding to the CC variable is positive for the hurdle component. Thus having Cyclomatic Complexity from 0 to 1 is positively correlated to the number of asserts. To estimate the impact of cyclomatic complexity, we can take the exponential of the coefficient. Having non-zero CC (from 0 to 1) of a method significantly impacts having chance of adding assertion in a method by a factor of exp(0.0205483) = 1.020760 or 2.05%. The p-value is less than 0.001 showing that the result is statistically significant.

- The coefficient corresponding to the CC variable is also **positive for the count component**. By taking the exponential of the coefficient, we can find that a unit increase in the Cyclomatic complexity relates to a increase in adding assert to a method by a factor of exp(0.004724) = 1.004735 or 0.4724%.

- We can note that the coefficient corresponding to the Comments variable is **positive for the hurdle component**. Thus having comments in our method from 0 to 1 is positively correlated to the number of asserts. To estimate the impact of Comments, we can take the exponential of the coefficient. Adding the first Comment to a method leads to an expected increase in the number of assertions by a factor of exp(0.0316445) = 1.032 or 3.16%. The p-value is less than 0.001 showing that the result is statistically significant.

- The coefficient corresponding to the comment variable **for the count component is also positive**, and the p-value is also less than 0.001. By taking the exponential of the comment coefficient, we can find that a unit increase in the Comments relates to an increase in adding assert to a method by a factor of exp(0.0311540) = 1.0316443649 or 3.11%.

6.2 Varying Use of Assertions with change in Project Domains

We observe that the distribution of type of assertions varies across project domains, as seen in the figure.

For instance, projects in the application software domain make frequent use of media resources such as graphics, audio files, sounds, and videos, leading to an increased proportion of Null Condition Checks, as opposed to other assertion types.

Similarly, documentation projects involved programming tutorials, implementation of data structures, and solving of interview questions (such as LeetCode questions), leading to increased proportion of Collection Data and Length Checks, and Arithmetic or Logic Comparison Checks as compared to any other category.

Web and non-web libraries and frameworks make extensive use of Minimum and Maximum value constraint checks since they include graphics, visualization and app development frameworks.

We observe that Resource Lock checks and Initialization checks are predominantly only performed by system software projects.

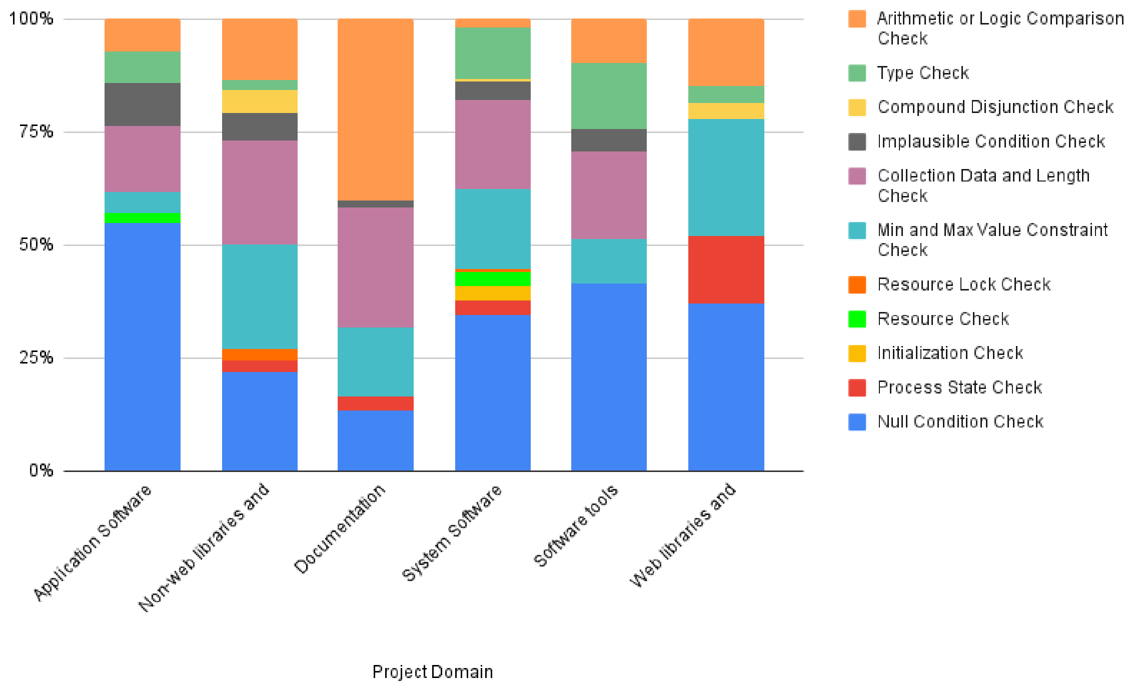


Figure 6.2

6.3 Reactive and Proactive use of Assertions

We observe 336 assertions (7.64%) to be reactive assertions, and remaining 4061 assertions (92.35%) to be proactive assertions. This shows that assertions are predominantly authored proactively. We identify the following reasons for assertions to be added proactively to the code base:

- While rolling out a new feature, developers often add assertions that validate assumptions about the newly authored code, or help with testing assumptions about previously authored code being used/referenced by the new feature.
- Developers often add assertions while refactoring code to ensure that the refactored code achieves the same functionality as the previous code.
- Developers add assertions to ensure backward compatibility of publicly exposed APIs.
- Developers very often author assertions while performing code optimizations, and these assertions are mostly Arithmetic or Logic Comparison Checks.
- We observe that assertions are seldom modified to add an assertion message, which may help with better documentation of the program.
- Lastly, developers seldom switch exceptions to assertions, taking into consideration usage scenarios and expected run-time exceptions.

6.4 Program-points where Assertions frequently occur

The following graph represents most frequent statements as n-grams immediately preceding (blue columns) and immediately succeeding (orange bars) the assert statements. Please visit `https://bit.ly/rq4-result` for qualitative insights.

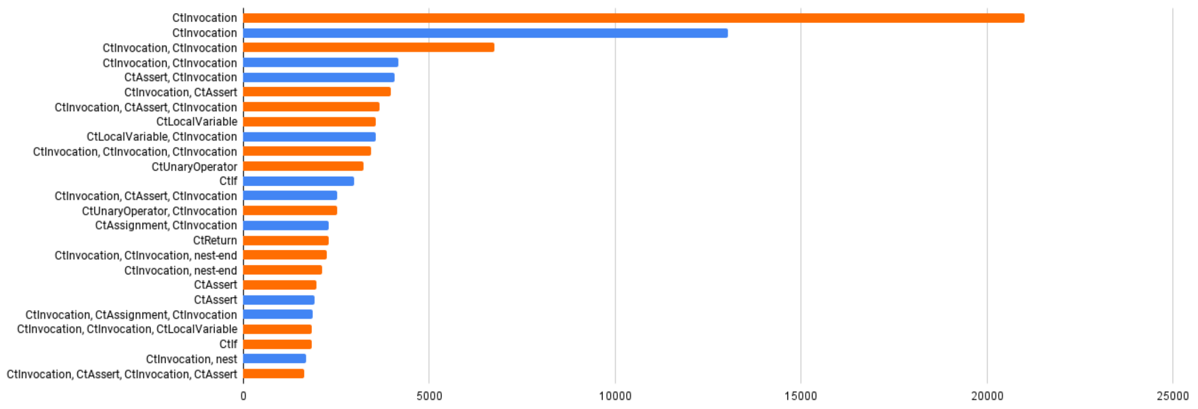


Figure 6.3

Chapter 7

Ongoing and Future Work

7.1 Ongoing work

I am currently working on implementing backward flow data dependence analysis, considering the assertion as source statement. This will enable us to analyse variables present in the assertion predicates (RQ 5) and propose heuristics to predict which variables must be evaluated in the assertion predicate. I am also working on articulating insights from RQ 4 findings about the program points where assertions frequently occur.

7.2 Future Work

Following the study of assertions, we aim to develop a tool for automated assertion generation for Java. The insights that will be obtained from the above research questions will help in development of the tool by extracting meaningful features. This will allow us in developing a heuristics and learning based approach for proposing domain-aware candidate assertions for development projects.

Proposing Candidate Assertions

The generation of a large number of candidate assertions can be problematic for the developer, and there needs to be filtering of assertions to allow assertions only for useful program properties. For this, we plan on analysing the function signatures (method name, input arguments and return value type), as well as property names and types. This could also involve building ASTs for assertion predicates and defining an extensive set of templates (For example: comparison of a constant and a symbol, comparison of multiple symbols, etc). Following this, we shall use program analysis techniques for code generation, to obtain syntactically correct assertion statements.

The final implementation step for the project would involve development of an IDE Eclipse plugin, which can possibly allow developers to tweek assertion recommendation settings (like variables, types of relations, frequency, etc).

Evaluation

Lastly, to evaluate our tool, we can recruit software developers and programmers as participants and conduct a user evaluation study. The participants can be assigned programming tasks with varying complexity and we can observe their interaction with the tool via Concurrent Think Aloud sessions. This will help us understand if the tool helps the developer in understanding their code or debugging it with ease.

Further, we can also run our assertion generation tool for existing Java open-source projects after removing assertion statements from them, and compare the original statements and their placement in the code, to the statements and program points generated by our tool.

Bibliography

[1] Baudry, B., Le Traon, Y., and Jezequel, J.-M. Robustness and diagnosability of oo systems designed by contracts. In *Proceedings Seventh International Software Metrics Symposium* (2001), pp. 272–284.

[2] Bi, T., Xia, X., Lo, D., and Aleti, A. A first look at accessibility issues in popular github projects. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2021), pp. 390–401.

[3] Borges, H., Hora, A., and Valente, M. T. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), pp. 334–344.

[4] Briand, L. C., Labiche, Y., and Sun, H. Investigating the use of analysis contracts to support fault isolation in object oriented code. *SIGSOFT Softw. Eng. Notes 27*, 4 (July 2002), 70–80.

[5] Casalnuovo, C., Devanbu, P., Oliveira, A., Filkov, V., and Ray, B. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), ICSE '15, IEEE Press, p. 755–766.

[6] Kochhar, P. S., and Lo, D. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering* (New York, NY, USA, 2017), EASE'17, Association for Computing Machinery, p. 298–307.

[7] Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience 46* (2015), 1155–1179.

[8] Pham, L. H., Thi, L. L. T., and Sun, J. Assertion generation through active learning. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (2017), pp. 155–157.

[9] Ray, B., Posnett, D., Devanbu, P., and Filkov, V. A large-scale study of programming languages and code quality in github. *Commun. ACM 60*, 10 (sep 2017), 91–100.

[10] Viera, A., and Garrett, J. Understanding interobserver agreement: The kappa statistic. *Family medicine 37* (06 2005), 360–3.

[11] Wang, C., He, F., Song, X., Jiang, Y., Gu, M., and Sun, J. Assertion recommendation for formal program verification. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)* (2017), vol. 1, pp. 154–159.

[12] Watson, C., Tufano, M., Moran, K., Bavota, G., and Poshyvanyk, D. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), ICSE '20, Association for Computing Machinery, p. 1398–1409.