AUTOMATED CODE QUALITY METRICS FOR CONCURRENT SOFTWARE

NADIA MIRSHAFIEE

This dissertation was submitted in part fulfilment of requirements for the degree of

MSc Software Development

DEPT. OF COMPUTER AND INFORMATION SCIENCES

UNIVERSITY OF STRATHCLYDE

AGUSTE 2019

Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(Please tick) Yes [ ] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is.

I confirm that I wish this to be assessed as a Type 1 2 3 4 5

Dissertation (please circle)

Signature:

Date:

Abstract

The aim of this project is to suggest code quality metric utilised for concurrent software. Since there cannot be a universal definition of quality, managing it can be a thorny problem that needs to be addressed. This problem is addressed in the scope of this study to the extent that the structural and underlying principles of synchronisation are considered.

Acknowledgments

I would like to express my sincere gratitude to my patient and supportive supervisor Dr Konstantinos Liaskos for the thoughtful comments and recommendations on this dissertation.

I also highly appreciate the University of Strathclyde, department of computer and information sciences and all their member's staff for all their support for the duration of my study.

Table of Contents

Table of Figures

Table of Listings

1 Chapter 1: Introduction

1.1 Background

"You cannot control what you cannot measure."

(De Marco, 1962)

This quote has become moto to evaluate quality in the software industry, which implies that measurement is a vital aspect of any software quality concept. In the programming level, it can be considered that developing and applying code quality metrics are derived from this quote. Employing code quality metrics in analysing source codes can be an optimum technique to manage and enhance the code quality. The main reason is that they provide valuable information based on code patterns that can be invisible during development. It can broaden the understanding of the current level of code quality that can lead to detect and reduce the number of possible errors and faults. Running automated static analyser that applies appropriate metrics over source codes early and often can facilitate reaching this purpose. However, to measure and evaluate quality, we must first define 'quality' to clarify the expectations.

In Juran quality handbook (2010) it is argued that defining the meaning of quality makes it manageable. Consequently, when quality is manageable, it can be delivered to expectations and satisfaction. There have been efforts to provide a universal definition of quality and clarify its matters. Multiple meanings of the word 'quality' were provided, which many of these definitions may fall short for different purposes. Juran (2010) suggests a definition to apply to any situation: "Quality means fitness for purpose." He was an evangelist for quality and quality management; thus, his handbook reflected different sectors, including the software industry.

Professional bodies and members of software industry such as The Institute of Electrical and Electronics Engineers (IEEE) and The Software Engineering Institute (SEI) provide different standards, frameworks, and guidelines for the educational and technical advancement of the software industry and allied disciplines. They provide the fundamental knowledge that forms the underlying fundamental for understanding the scope of software quality and taking corrective action on

managing it. Nevertheless, according to the general definition of quality by Juran, it is not possible to have any unique interpretation for this term and therefore any fixed measure for it.

This work covers only quality in programming level. ISO 24765 (ISO 17a) states that software is:

> "Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system."

This definition indicates that the programme is only one component of software deliverables. Programmes are source codes that have been designed, reviewed, and unit tested to ensure that they behave on expected instructions (April & Laporte, 2018). In addition to not have a particular definition of quality, numerous different programming languages have their syntax and standards of coding. This would seem to indicate that it is elaborate to have measures that can cover all aspects of code quality in depth while satisfying the needs of all software experts and customers.

The next section points out the areas that this work focuses on and the rationale for this study.

1.2 Work-study focus

This work is a study to help advancement in managing quality in the code concept. Referring to the previous section, measurement has a crucial role in quality management. Therefore, the main focus of this work is on suggesting appropriate *code quality metrics*. Since *Java* is one of the widely popular and highly in demand *programing language*, it is chosen as an area of study. It is clear that all aspect of the code quality of a language is unlike to cover in a dissertation. Therefore, the area is limited to part of the *concurrency concept*.

To fulfil the desired purpose, the process of applying a metric or set of metrics should follow correctly in addition to select useful metrics. Galin (2018) suggests that

this process is similar to the implementation of new methodologies or procedures and involves the following steps:

a. Assigning responsibility for given information by metrics.
b. Provide instruction regarding the new metrics for programmes.
c. Follow-up includes:
    c.1. Support for taken actions based on instruction and provision of further information if needed.
    c.2. Control of metrics reporting for completeness and accuracy.

This research only focuses on two matters: firstly, *suggesting code quality metrics* to highlight valuable information (not easy to notice by programmer or unit testing) for further analysis and evaluations. Secondly, designing a *static tool* which *automatically applies these metrics* in concurrent contexts. However, following all steps of applying quality metrics suggested by Galin (2018) is out of the scope of this study.

1.3 Overall aim and project objectives

The overall aim of this project is suggesting code quality metrics utilised for concurrent software. These metrics are expected to provide useful information about concurrent programmes based on their code patterns to managing the quality. They also are expected to derive other metrics. However, in order to suggest appropriate metrics, it is felt necessary to advance the understanding of the critical concepts pertinent to the subjected area. Within the context of the MSc dissertation, the objectives of this project are to:

1. Broaden the understanding of:
    1.1. Quality
    1.2. Code quality
    1.3. Code quality metric
    1.4. Non-concurrent software
    1.5. Concurrent software
2. Suggesting code quality metrics utilised for concurrent software.

3. Evaluating suggested metrics.

Objective one firstly focuses on quality and code quality metric in general. Subsequently, its focus is on the fundamental understanding of non-concurrent software and code quality metrics utilised for them. Finally, it explores a fundamental understanding of concurrent software limited to the subjected area of this work. This study will make critical contributions to advancement in measuring code quality through objectives two and three. The listed objectives should not be seen as unrelated activities since they are necessarily interlinked and decisive steps to achieve a satisfactory outcome.

1.4 Value of this research

Today there are many automated tools which statically examining source codes with applying code quality metrics. Although it is a method to measure the quality of the code, they also are used to analysing the source codes to take necessary steps such as refactoring to address software problems. This would seem to indicate that rely on individual programmers and different forms of testing cannot be extremely efficient methods to find errors and faults in source code and consequently to guarantee the quality of the code. Therefore, quality metrics can have a critical application in measuring and enhancing quality.

The attempt in this work was on contributing to the development of efficient code quality metrics utilised for concurrent software in a number of influential ways. To begin with, the understanding of expected quality in the subjected area of this study are deepened. Secondly, appropriate metrics are suggested based on obtaining knowledge. Apart from these, a tool is designed to statically evaluate the suggestions of this study.

The next chapter examines pieces of literature pertinent to the objectives of this study, beginning with producing a summary of what is quality and code quality metric.

2 Chapter 2: Literature Review

2.1 Introduction to literature review

This Literature Review is conducted to attain objective one of this project. Firstly, quality and code quality metric will be examined in general. Subsequently, a summary of non-concurrent software knowledge will be explored to recognise the crucial distinction between non-concurrent and concurrent software. Finally, this chapter broadens the understanding of concurrent software limited to the subjected area.

In this study, it is attempted to mainly use reference pieces of literature to avoid the tendency toward any specific theory or method. Importantly, developing an idea with relying on universal standards will improve the reliability of the idea. This literature review will provide a sufficient fundamental knowledge about concurrent concept and code quality in a structured way to facilitate the understanding of problematic areas of concurrent concept that code quality metrics can address.

At the end of this chapter, it is aimed at a clear understanding of code quality metrics' applications is exhibited, that will lead to suggesting appropriate metrics for concurrent context. Moreover, it is hoped that this study will be informative for those who are taking their first steps in empirical research in the field of concurrent programming.

2.2 Quality and Code quality metrics

As SWEBOK guide (2014) states, there are various attributes to measure the quality of software such as maintainability, portability, testability, usability, and correctness. Moreover, SWEBOK guide (2014) point outs that there is an interesting distinction between quality attributes discernible at runtime (e.g. performance) with those not discernible at runtime (e.g. testability), and those related to the architecture's intrinsic qualities (e.g. correctness). This implies that measuring quality is a vast concept since there are numerous aspects to observe measure, quantify, and qualitatively assess. Therefore, there is not any universal definition and a single framework to measure it. Importance of different attributes can be considered in pertinence with the subjected area to gain unsurpassed result specific to that context.

According to the Institute of Electrical and Electronics Engineers (IEEE 730) [IEE 14], software quality is:

> "The degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations [Institute of Electrical and Electronics Engineers."

This definition has two different aspects. The first aspect describes a quality software as the one that can satisfy all the specified requirements in the software requirements document, which is out of the scope of this work. However, April & Laporte (2018) point out that the second aspect of this definition specifies that it also must satisfy the requirements that are not necessarily described in the documentation. This definition comes from the quality perspective of Juran.

The definition of quality by Juran (2010) is "Quality means fitness for purpose." He points out that to be fit for purpose, every produced service or good must meet two specifications. To begin with, it must be valid and have the right features to satisfy customer requirements and needs. Secondly, it must have a few failures to be efficient for superior performance. It shows that he considers two aspects for quality, "features that meet customer needs" and "freedom from failures". The second aspect implies that any quality software is expected to possess implicit characteristics. In this concept, code quality has a crucial role as part of overall software quality.

As Juran (2010) suggests, higher quality leads to having fewer errors, faults, and failures. Therefore, a professionally developed piece of code is expected to follow adequate software engineering principles and standards to avoid any error or fault and consequently to guarantee the code and overall software quality. According to SWEBOK guide (2014), preventing errors is a preferable approach to software quality than to correct the faults and failures. Thus, it is arguable that to prevent errors; it is best to start analysing the code as soon as it is written. In this way, quality is given

priority from the very start of development. Hence, a programmer can improve the code during the creation phase.

There are other techniques to improve code quality. One of them can be adherence to standards is an efficient method to have a high-quality code. In addition to this, April & Laporte (2018) consider code review technique for detecting errors and faults, which leads to reduce the error and faults. Apart from these, SWEBOK guide (2014) states refactoring as reengineering technique that can be used to improve a program structure without changing its behaviour. Performing all aforementioned techniques can be done by the advantage of automated tools that their basic functionality is analysing the source code.

As mentioned in the previous chapter, having units of measurement is a decisive step to control the code quality. Therefore, code quality metrics become particularly a concern in the software industry. For instance, the aforementioned tools can function effectively by employing appropriate code quality metrics.

Galin (2018) proposed several numbers of objectives for software quality metrics that are presented in figure 1.

Figure 1. Objectives of software quality metrics (Galin, 2018)

Source: After ISO/IEC Std. 90003-2014 Sec. 8 (ISO/IEC 2014).

• To assist management to monitor and control development and maintenance of software systems and their process improvements by:
• Observing the conformance of software product to functionality and other requirement, regulations, and conventions.
• Serving as a data source for process improvement by:
  o Identifying cases of low performance requiring improvement.
  o Demonstrating the achievements of process improvement proposals (corrective actions)

To select applicable and successful quality metrics, Galin (2018) points out metrics characteristics and metrics implementation characteristics that figure 2 presents them.

Figure 2. Required characteristics for quality metrics (Galin, 2018)

Required characteristics	Explanation
Metrics characteristics	
✓ ***Relevant***	Related to an attribute of substantial importance
✓ ***Valid***	Measures the required attribute
✓ ***Reliable***	Produces similar results when applied under similar conditions
✓ ***Comprehensive***	Applicable to a large variety of implementations an situations
✓ ***Mutually exclusive***	Does not measure attributes measured by other metrics
Metrics implementation characteristics	
✓ ***Easy and simple***	The implementation of the metrics data collection is simple and performed with minimal resources
✓ ***Does not required independent data collection***	Metrics data collection can be integrated with other project data collection systems: that is, employee attendance
✓ ***Immune to biased interventions by interested parties***	The data collection and processing system is protected from unwanted changes; additions and deletions

Taking account of these characteristics and aforementioned objectives, useful quality metrics can be proposed to measure required attributes efficiently. According to Galin (2018), *software product metrics* are distinguishable from *software process metrics*. Quantitative representation of the product's attributes, as experienced by

user are *software product metrics* such as productivity and reliability that is out of the scope of this project. Quantitative representation of software processes is *Software process metrics* that are experienced by developers and maintainers. Code quality metrics belong to this category.

Significant efforts were made over the past years to define code quality metrics that can be employed to develop approaches and tools for analysing source codes. Microsoft (2018) states that *code metrics* are a set of software measures that discovered information by code analysis and give a clear insight into the current code quality. Gaining advantage of code quality metrics facilitates taking steps to enhance code quality and consequently, overall software quality.

SWEBOK guide (2014) categorises software quality control techniques to static and dynamic. Dynamic techniques involve executing the software, which is out of the scope of this work. Static techniques involve analysing software source code without execution, that is the subject of this work. To assist the progress of code management, static analyser tools are designed for employing code quality metrics to evaluate code quality, facilitate applying standards, following coding best practices, and conduct refactoring.

2.3 Non-concurrent software

According to Goetz et al. (2006), the elderly generation of computer did not have operating systems, and they executed a single program sequentially, which means from beginning to end. The sequential programming is natural, do one thing at a time, in sequence-mostly as it models the way humans work. These programs had direct access to all the resources of the machine (Goetz et al., 2006). The sequential programmes are known as non-concurrent programmes.

2.3.1 Code quality metrics utilised for non-concurrent software

As argued at the start of this chapter, analysing and auditing the code quality can be a robust process. Over the years, a wide range of simple to sophisticated

metrics have been suggested to facilitate quality management. The different metrics provide information at different levels of abstraction within the source code. This study provides a list of metrics that are useful to assess non-concurrent concepts. However, they cannot provide valuable information in the context of concurrent programming. These metrics are categorised as follow:

- ✓ **Size metrics**: Logical Source Statement (LSS) & Physical Source Statement (PSS), Source Lines Of Code (SLOC) or Lines Of Code (LOC), Method Lines Of Code (MLOC), Nested Block Depth (NBD), Complexity metric
- ✓ **CK metrics**: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Lack of Cohesion in Methods (LCOM)
- ✓ **Basic metrics**: Number Of Classes (NOC), Number Of Methods (NOM), Number Of Fields (NOF), Number Of overridden Methods (NORM), Number Of Parameters (PAR), Number Of Static Methods (NSM), Number Of Static Fields (NSF) (Bigonha et al., 2015)

2.3.1.1 Size metrics

> "Software size matters and you should measure it."
>
> — Gerush & West (2014)

In the concept of software attributes, the size can be the first and one of the key attributes that come to consideration. As Boehm et al. (2007) state size is a measure of effort and time required to develop a software. Moreover, it is a unit to derive other metrics for software analysis and its quality measurement. According to IEEE [IEEE 92] counting the source statements is a measure to define the software size and there are two frequently used size measures: the *logical source statement (LSS)* and the *physical source statement (PSS)*.

A brief study of the non-concurrent software was conducted to prepare the grand for developing a clear understanding of concurrent software.

2.4 Concurrent software

Most of the time, applications are expected to do more than one activity at a time. For instance, a bank account application is expected to make the direct debit at the same time the user is transferring money from their account, and it should receive any money that is transferred to this account from anywhere. Applications with this behaviour are known as concurrent applications.

In concurrent programming, Oracle (2017) declares threads as one of the basic units of execution. Threads provide independent paths of execution through source code for Java applications (Friesen, 2015). From the application programming point of view, every Java application comprises at least one thread, called the *main thread* that executes the program's `main()` method (Goetz et al. 2006; Friesen, 2015). Moreover, according to Oracle (2017), this *main thread* can create additional threads.

Oracle (2017) states that an essential characteristic of the Java platform is multithreaded execution. Goetz et al. (2006) argue that threads are executed concurrently and asynchronously concerning each other in the lack of precise coordination. By the same token, Friesen (2015) states that in multithreaded execution, threads do not interfere with each other, and each of them has their own path that differs from each other. It happens due to allocating separate stack to each thread via the Java virtual machine (JVM). Given these explanations implies a thread can have its own copy of local variables, method parameters, and return value by using the separate JVM stack. Moreover, different path means each thread has its own instruction to execute code sequences and can track it by its stack (Friesen, 2015).

Listing 1 demonstrates a withdrawal method of a `BankAccount` class as an example. The idea is to demonstrate what happens in concurrent applications if multithread execution of the same portion of code takes place simultaneously.

Listing 1. Demonstrating multithread execution in the absence of synchronisation

```
 5  Public class BankAccount {
 ⋮        ⋮
20       public boolean withdraw(double withdrawAmount) {
21           if (withdrawAmount > balance) {
22             System.out.println("Insufficient Funds!!!");
23             return false;
24           }
25       else {
26             balance -= withdrawAmount;
27             return true;
28           }
29         }
 ⋮        ⋮
60  }
```

Execution of listing 1's `withdraw` method is as follow:

a. Line 20 -> Get a withdrawal amount
b. Line 21 -> Check the balance of the bank account to know if there is a sufficient amount for making a withdrawal.
    b.1. If it is more than the amount of balance:
        b.1.1. Line 22 -> the method prints the appropriate message
        b.1.2. Line 23 -> Return false
    b.2. Otherwise, the method makes the withdrawal:
        b.2.1. Line 25 -> Enter the else block
        b.2.2. Line 26 -> Reduce the amount of balance by withdrawal amount
        b.2.3. Line 27 -> Return True
c. Line 29 -> Exit the method

Consider a bank account has £200 balance, and two threads (T1: direct debit and T2: money transfer) are calling this method at the same time for this bank account. Check out figure 3.

Figure 3. Two threads are calling same non-synchronized method simultaneously

Tread T1 (direct debit £150)	**Tread T2 (money transfer £100)**	**Balance**
The scheduler gives T1 time slice Check if balance is sufficient The answer is yes Enter the `else` block *The scheduler paus the T1*		← £200 £200 £200
	The scheduler gives T2 time slice Check if balance is sufficient The answer is yes Enter the `else` block Make withdrawal *The scheduler paus the T1*	← £200 £200 £200 → £100
The scheduler gives T1 time slice Make withdrawal !!		£100

Suppose that the thread scheduler chose T1 first to run its tasks (Eliminate running line 20). Thus, T1 is running, and T2 is waiting for the scheduler to give it a time slice to run its tasks. T1 runs line 21 of code and checks (read balance value) if there is sufficient balance for £150 withdrawal. The answer is yes, thus T1 enters the `else` block in line 25. At this time, the thread scheduler pauses the T1 and gives the time slice to T2 to run. T2 runs line 21 to check (read balance value) if there is sufficient balance for its withdrawal: £100. Since T1 is in line 25 and still has not made the withdrawal, the balance is £200 and the answer is yes. Thus T2 enters the `else` block, makes a withdrawal in line 26 (write back the balance value) and therefore the balance is reduced to £100 now. At some point, scheduler pauses T2 and give the time slice to the T1. Since T1 already checked the balance and is inside the else block in line 25, it does not do it again, and it makes a withdrawal in line 26, although the balance is not sufficient now. This situation is known as a *race condition*, which causes hard-to-find faults in applications.

Goetz et al. (2006) point out in the absence of explicit coordination threads interfere with one another, and the ordering of operations is unpredictable. Threads

can interact with modifying shared data, i.e. a thread may write the value of shared data that another thread is in the middle of using (Goetz et al., 2006; Friesen, 2015). Occurring interactions can pose various safety hazards, such as deadlock and race condition (Friesen, 2015). Consequently, safety hazards make an application thread-unsafe, which means incorrect in a multithreaded context.

The evidence shows shared data are major potential parts of making an error and having faults in a multithreaded context. Therefore explicit coordination is vital for thread access to a critical portion of code that modifies the shared data. As Goetz et al. (2006) states, a variety of synchronisation techniques is provided by Java for coordinating such access. When data remain consistent by using Java's synchronisation-oriented language features correctly, it is possible to avoid raising issues of accessing shared data through multithreaded execution (Friesen, 2015; Oracle, 2017).

Following sub-sections outline the Java's synchronisation-oriented language features in the scope of this research to understand their correct use.

2.4.1 Synchronisation

Recall that threads interact primarily by sharing access to shared data. Goetz et al. (2006) argue that in the absence of synchronisation, the timing and order of such access is given by the compiler, hardware, and runtime, and it is unpredictable. This implies that multithreaded execution in the absence of synchronisation undermine safety. Oracle (2017) suggest Java's synchronisation-oriented language features are used to make concurrent applications thread-safe in context of multithreaded execution. An application is thread-safe if only one thread can have access to a critical portion of code – which is responsible for reading or writing on shared data – at a time (Friesen, 2015; Oracle, 2017). By the same token, Friesen (2015) argues using the synchronisation causes the threads to execute in a serial manner, which means only one thread can have access to a synchronised code section at a time.

On the other hand, Oracle (2017) argues that synchronisation can introduce thread contention such as starvation and livelock. This status occurs when two or more threads simultaneously attempt to access the same resource. It leads the Java runtime to execute threads slower, or even to stop their execution. Thus it could be concluded that synchronisation should use in an appropriate way to avoid any possible fault and failure in application. Considering this fact, sufficient details of synchronisation will be provided in the following sub-sections to cover the subject area of this study.

2.4.2 Properties of synchronisation

In programming, Oracle (2017) declares a set of actions is **atomic** if they execute all at once entirely and effectively or does not happen at all. Moreover, until an atomic action is complete, no side effects of the action are visible. The synchronisation is a JVM feature that ensures that threads execute a critical portion of code atomic (Friesen, 2015; Oracle, 2017). This implies a specific section does not be interrupted between threads and a thread execute it all at once. At the same time, all other threads which call the synchronised portion of the code should wait for the one that is inside the code to exit it (Oracle, 2017). This property of synchronisation is known as *Atomicity* (Friesen, 2015; Oracle, 2017).

Friesen (2015) argues that "synchronisation also exhibits the property of *visibility*". In the absence of synchronisation in multithreaded execution, if one thread updates the value of a shared variable, it is possible that another thread does not see the update when it needs it and uses it (Goetz et al., 2006). The visibility guarantees access to the most recent changes to shared variables by the thread executing in a synchronised portion of code (Friesen, 2015).

The *happens-before relationship* is another property of synchronisation (Oracle, 2017). There is a happens-before relationship between statements due to the way they are written. This implies that when a thread calls a portion of code, e.g. method, it executes method's statements in order. However, as Oracle (2017) declares if more than one thread read or write on a same shared data in a critical portion of code at

the same time, there is no ordering between the actions of different threads. Oracle (2017) suggests using synchronisation to establish a happens-before relationship between threads' actions. It causes the results of writing by one thread to be always visible to another thread for reading and shard data to remain consistent in multithreaded execution.

This triple concept (*Atomicity, visibility, happens-before relationship)* are fundamental to correct concurrent programming (thread-safe). Synchronisation involves different techniques in Java that each of them is associated with some of these concepts. Following sections will describe some of these techniques.

2.4.3 Synchronisation object and monitor lock

The synchronisation is implemented for monitoring the access to a critical portion of the code, hence correlates with an entity known as *monitor lock* (Friesen, 2015; Oracle, 2017). Oracle (2017) declares that every Java object is associated with a monitor lock and can be used as a synchronisation object. Thread has to acquire a monitor lock before accessing the synchronised code and release the lock when it exits the code (Friesen, 2015). Oracle (2017) points out that monitor lock guarantees the exclusive access to the synchronised portion of code and establishes a happens-before relationship between subsequent acquisitions of the same lock that are essential for visibility.

As Oracle (2017) states, the time between a thread acquire – lock – a *monitor lock* and release it, the thread is said to own the lock. Threads cannot own a monitor lock simultaneously. In other words, only one thread can own the monitor lock at a time, and any other threads should wait as long as a thread hold the lock (until owner releases the lock). The ***scope** of the lock* defines as the amount of time that a thread owns the lock (Gagne et al., 2010; Oracle, 2017).

Locks are designed to be acquired more than once by the threads which already own them in order to prevent deadlock (Friesen, 2015). This property enables

reentrant synchronisation. Reentrant synchronisation is where a thread owing a lock, directly or indirectly, invokes several synchronised code sets which use the same lock.

2.4.4 Synchronisation techniques

As argued, Java language has a variety of techniques to synchronise a portion of code, which they guarantee some properties of synchronisation. To spotlight the scope of this research, the pertinent techniques will expound and exemplify in following sub-sections.

2.4.4.1 Synchronised method

In the context of concurrency, using the `synchronized` keyword is the original style of Java synchronisation programming whenever shared variables need to read or write. Oracle (2017) declares that to synchronise a method, the `synchronized` keyword is used in its header. For example, Listing 2 shows the increment method, which is synchronised:

Listing 2. Synchronised method

```
public synchronized void first(){
      counter++;
}
```

Friesen (2015) points out that the `synchronized` keyword is associated with both the property of *Atomicity* and property of *visibility*. By the same token, Oracle (2017) argues that making a method synchronised has two effects:

1. The thread access to a synchronised method will be serialized; hence, two or more invocations of them cannot be interleaved on the same object. This indicates the synchronised method exhibits the property of *Atomicity*.
2. *A happens-before relationship* is associated with a synchronised method that is, a happens-before relationship will be automatically established with any subsequent invocation of a synchronised method. This ensures modifications in the shared data are visible to all threads.

Given these statements, it could be concluded that the `synchronized` keyword guarantees the triple property of synchronisation.

Regarding the monitor lock, there is a difference between the associated synchronisation object of static and non-static methods. As Olsson (2018) states, a static method is associated with a Class itself (Class object), not with any instance of the class (this). Therefore, when a thread invokes a static synchronised method, it acquires the monitor lock of the Class object. A non-static method is associated with the instance of the class (this) (Olsson, 2018). Hence, when a thread invokes a non-static synchronised method, it acquires the monitor lock associated with 'this', which represents a current object (an instance of the class). Given this evidence, it is evident that the static and non-static synchronised method lock on different objects.

2.4.4.2 Synchronised block

In Java, `synchronized` keyword also use to synchronise a critical block of statements (synchronised block) for serialising the threads access to them (Gagne et al., 2010). Friesen (2015) declares that the header of a synchronised block is consist of `synchronized` keyword along with an object that is used as a synchronisation object. Listing 3 demonstrates the syntax:

Listing 3. Synchronised block

```
public void SyncBlock {
      synchronized (/*SynchronisationObject*/) {
            /* statements inside synchronised block*/
      }
      /* statements inside method*/
}
```

There are two reasons for using the synchronised block in preference to the synchronised method. Firstly, only a critical block of statements which is responsible for reading or writing on the shared data will be synchronised instead of the entire method (Gagne et al., 2010). This reduces the locking scoop and thus improves the

performance of the application. Secondly, the synchronisation object is exceptionally flexible, and the programmer has substantial liberty to choose it. Regarding the monitor lock, it is associated with a declared synchronisation object.

2.4.4.2.1 Nested synchronised block

Synchronised blocks can be nested; however, it can increase the likelihood of error in code. Deadlock describes a situation where thread A waits for a lock that thread B is holding and thread B is waiting for a lock that thread A is holding. When Deadlock runs, each thread is waiting for the other to exit, and neither thread can make progress (Oracle, 2017). There is a potential deadlock with nested synchronised blocks.

Consider Listing 4, which presents an example of deadlock situation arises in nested synchronised block:

Listing 4. Nested synchronised block causes deadlock

```
 9  public class NestedSynchBlock {
 ⋮        ⋮
14        public void first(){
15              synchronized(lock1){
16                    synchronized(lock2){
 ⋮                          ⋮
20                    }
21              }
22        }
23
24        public void second(){
25              synchronized(lock2) {
26                    synchronized(lock1) {
 ⋮                          ⋮
30                    }
31              }
32        }
33
34  }
```

Suppose two or more threads invoke `first` and `second` methods, then a deadlock situation will run. Friesen (2015) points out that the following execution sequence can happen if two threads (T1 and T2) call methods in such a situation:

a. T1 calls `first`, obtains the lock assigned to the `lock1` and enters the block (but has not yet acquired the lock assigned to the `lock2`).
b. T2 calls `second`, obtains the lock assigned to the `lock2` to proceed further (but has not yet acquired the lock assigned to the `lock1`).
c. T1 acquires the lock associated with `lock2`while T2 holds it. Therefore, T1 should wait outside of the inner block until T2 release the lock.
d. T2 acquires the lock associated with `lock1`to finish its tasks and exit the synchronised block. However, T1 holds the lock associated with `lock1`. Thus, T2 should wait outside of the inner block until T2 release the lock.
e. Since both threads hold the needed lock by each other, neither of them can proceed, and there is a deadlock situation.

In this example, since the locks are given in reverse order in nested synchronised block to the threads, threads waiting for each other to release the locks to progress which will never happen.

2.4.4.3 Variables

As Oracle (2017) states the Java programming language defines the four kinds of variables: Instance Variables (Non-Static Fields), Class Variables (Static Fields), Local Variables, and Parameters. A clear understanding of variables is essential to do programming correctly. The role of instance variables and class variables are highly significant in the context of concurrent programming. Moreover, Java provides a volatile variable in the context of synchronisation.

This subsection expounds on the variables from the concurrency point of view.

2.4.4.3.1 Static variable

The `static` modifier is used to create class variables. It means there is only one copy of this kind of variables, which belongs to the class itself (Olsson, 2018). Consider Listing 5:

Listing 5. Instance variable vs class variable

```
public class Variables {
      int a = 0;          // instance/non-static variable
      static int b = 0;   // class/static variable
      ⋮
}
```

Listing 5's field '`a`' is declared without the `static` modifier, that is, '`a`' is a non-static field or instance variable. Oracle (2017) points out that each instance of a class has its non-static fields (instance variables) to store its individual states. In other words, the value of non-static variables is unique to each object. In this example, the non-static variable '`a`' will be created as a new copy for each new instance of listing 5' class.

The `static` modifier is used to create Listing 5's fields '`b`' as a static field or class variable. There is only one copy of a static variable, which belongs to the class itself regardless of how many times the class has been instantiated (Olsson, 2018). This implies each static variable is created only once for each class and is shared with all the instances of a class (objects). Consequently, any change in the value of a static variable affects the other objects' operations. For Listing 5, only one copy of static variable '`b`' will be created which is belong to class `Variables` itself, and all instances of the class have access to this variable.

As argued, there is only one copy of static variables which can be accessed or altered by any objects of a class. It can lead a concurrent application to behave unexpectedly in a multithread environment. According to these facts, it could be concluded that the existence of static variables in the code can cause lots of issues

during concurrent execution. Therefore static variables are not suggested to use in concurrent applications unless they are made thread-safe in multithread context.

One method to avoid the creation of subtle bugs by static variables is making them into the constant. As Olsson (2018) considers, the `final` modifier is used to make a variable into a constant, which means this variable cannot be reinitialized once it's been set. Friesen (2015) suggests that final often use to ensure thread safety when synchronisation is not used in the context of an immutable (unchangeable) class. It means multiple threads can safely access to the final object. The `final` modifier only exhibits the property of *visibility*.

2.4.4.3.2 Volatile variable

Java provides the `volatile` keyword as a weaker method of synchronisation, which can only use in the context of field declarations (Friesen, 2015). Listing 6 presents the source code which declares the variable `start` as volatile.

Listing 6. Declaring volatile variable

```
public class VolatileVar {
      private volatile boolean start;
      ⋮
}
```

Read and write on volatile variables are atomic. Recall that atomic action cannot be interfered by other threads. However, Oracle (2017) argues that not all need to synchronise atomic actions obviates by volatile variables. They claim that it is still probable to have memory consistency errors and volatile variables only reduce the risk of them. On the other hand, to accessing shared variable Oracle (2017) suggests that declaring them as simple atomic is a more efficient technique compared with access to them through synchronised code. Although extra care is needed to prevent memory consistency errors.

Declaring a variable as volatile causes writing on it establishes a happens-before relationship with following reads of that variable (Oracle, 2017). This means that all threads always read the latest change to a volatile variable as well as the possible side effects of the codes that made the change. In other words, a volatile variable is always visible to other threads. Thus it could be concluded that the `volatile` keyword is associated with the property of *visibility* (Friesen, 2015) and also is associated with the property of *the happens-before relationship.*

Friesen (2015) suggests that using the `volatile` keyword can be the right choice when *visibility* is an issue in comparison with using the `synchronized` keyword for synchronisation. Since synchronised keyword solves the mutual exclusion problem in addition to the visibility problem, it increases the performance cost whit attempting to acquire the lock, which is not necessary when there is only a visibility problem. Therefore, choosing the `synchronized` keyword for visibility problem is wasteful. However, using `volatile` technique makes an issue for the programme when atomicity is required.

2.5 Literature review conclusion

Objective one of this project achieved in this chapter. The study of relevant literature revealed that quality is a complex concept. To begin with, there is no unique definition of quality. As argued, it is because the definition of quality will be profoundly affected by needs that should be satisfied (Juran, 2010). Apart from this, to guarantee a high level of quality, an intricate framework should provide to follow. Arriving at a deeper understanding of quality at the code level, it is arguable that having code quality metrics that can be used regardless of specific requirements of a project can be advantageous and convenient to manage code quality.

The subject area of this study is concurrent software, which its scope was defined. The review of literature stressed that to write a correct concurrent program (thread-safe), appropriate synchronisation technique should choose to avoid faults and failures that mostly fall into one of the triple properties of synchronisation

(*Atomicity, visibility, happens-before relationship)*. However, it is complicated to make sure that a concurrent application is made thread-safe correctly. It is because at compile-time, we are not given any errors and concurrent application runtime behaviour is very unpredictable in the context of multithreaded execution. Thus, the concern is that any problem of thread-unsafe (incorrect in a multithreaded context) application may occur randomly.

It could be concluded code quality metrics utilised for concurrent software can use to effectively measure and enhance the code quality by analysing the source code. These metrics provide information based on the code pattern, which cannot be given by doing the usual test, such as unit testing. In addition to measuring the current status of code quality, provided information can alert to the potential errors in concurrent source code, which lead to rectify them and accordingly improve the code quality. Code quality metrics also can derive further metrics concerning the more specific needs of each application. To that end, a list of essential metrics is presented by which useful information can be received in a concurrent programming context.

The next chapter of this study will detail the research strategy and techniques that are adopted to conduct this work.

3 Chapter 3: Research Method

3.1 Introduction to research method

Several objectives are set within the context of MSc dissertation for this work (stated in chapter one). This chapter is dedicated to producing a summary of the applied research method and techniques for pursuing the objectives of this project.

Considering Biggam (2015) guide, appropriate research strategy and proper techniques are attempted to employ for conducting a valid study, which following sections will present them. Furthermore, a detailed record of work steps is provided to improve the reliability of the study. Not only this, reference literature and general principals are mostly used to minimise bias and strengthen reliability.

3.2 Research strategy

The experimental research strategy is adopted to conduct this work. The purpose of the experimental strategy is to continuously develop the science by adding valid and reliable novel theories (Biggam, 2015). Strathclyde PGT dissertation handbook (18-19) suggests an experimental research structure to follow:

> "Designing an experiment and associated software to test the performance of a system or system component(s), analysing the data collected, and forming recommendations and conclusions based on this analysis."

According to this structure, an experimental strategy is therefore concerned with attempts to examine theory through an experiment with the help of designing appropriate software. Therefore, the experimental approach provides a framework that is required to attain these project objectives.

With a rough guide by Biggam (2015), the following steps are taken to adopt an experimental approach:

a. **Define the problem** – objective one deepens the understanding of the subjected area and defines the problem that needs to be addressed (Chapter 2).

b. **Formulate a theory** – objective two addresses the problem were defined by objective one by proposing novel ideas (Chapter 4).
c. **Implement an experiment to examine the formulated theory** – objective three will be met by going through the following procedure (Chapter 5 & 6):
    c.1. Design static code analysis tool
    c.2. Determine the sample source code to do the evaluation
    c.3. Perform the experiment
    c.4. Verify result
    c.5. Accept or reject the formulated theory

3.3 Source code sampling

Sampling is the process of selecting a set of elements from a subjected area with the intent of making consistent and unbiased statements about it. Do et al., (2005) points out representativeness as an essential characteristic of the sample. It is because representativeness makes profound impacts on the applicability of the conclusions to the rest of the subjected area.

To meet representativeness in this study, the sample source codes for evaluation are selected from the Software-artifact Infrastructure Repository (SIR). SIR is hosted at the University of Nebraska at Lincoln. This repository provides programmes containing intentional errors to use in controlled experimentations with testing and analysis techniques (Do et al., 2005). It can help to conduct a proper evaluation and consequently develop the reliable of it.

3.4 Framework for experimental evaluation

This study provides a framework to conduct the experimental evaluation for achieving reliable results. This framework has three steps, as follow:

a. **Description**: details the targeted source code to perform analysis
b. **Result**: represents the output of running analysis tool to analyse targeted Java source code (in class level)

c. **The verification of result and findings**: compares the result with proposed information for each metric (in chapter 4), and examines the result based on literature review and suggested metrics (chapter 2 & 4)

The rationale and operational details of the research strategy and techniques applied in this study were provided in this chapter. The next chapter will expound the suggested metrics that are derived by the literature review.

4 Chapter 4: Code Quality Metrics Utilised for Concurrent Software

4.1 Introduction to code quality metrics utilised for concurrent software

Analyse and audit of source codes to measure their quality can include a wide variety of concerns with respect to principles and standards such as adherence to coding standards, code structure for testability, and analysis of algorithms (SWEBOK, 2014). Objective two of this research aims to suggest code quality metrics utilised for concurrent software, which is derived as an outcome of the knowledge gained from the literature review (objective one), and this chapter will explicate it.

4.2 Metric 1 – Synchronised Methods in Class (SMIC)

It is evident that when analysis takes place, the indispensable step is having the fundamental units of measurement to take next steps. First basic unit of measurement in the scope of this study is the synchronised method, which is indispensable to enable performing some further analysis, e.g. to examine the synchronisation objects associated with synchronised methods. Therefore, Synchronised Methods in Class (SMIC) metric aims to distinguish synchronised methods from non-synchronised methods of a class.

To measure code quality concerning the synchronised method, it is decisive to understand the effect of the synchronised method on it. In 'synchronisation object and monitor lock' sub-section, the scope of the lock is defined as the amount of time that a thread owns a lock. This implies having many synchronised methods in a class may yield locking scope that is too large. There can be two main reasons for extending the scope of lock in this situation. To begin with, all statements inside a lock always are executed sequentially not concurrently, although atomic execution may not be necessary for all of them (presents the wrong choice of synchronisation). Secondly, having a lot of synchronised methods (static or non-static method) can mean the same synchronised object (Class itself or this) is used to protect different shared data and operations, which is not the optimum way of synchronisation. It causes threads unnecessarily wait for the same lock to execute different operations, although these operations may not need to be protected against each other. Consequently, extending the locking scope negatively affect the performance of the application, and

it can cause unpredictable behaviour. Apart from these, a high number of synchronised methods can increase the chance of having deadlock failure.

Although using the synchronised method is thread-safe, it cannot be always efficient due to the cost of synchronisation all the time which method is invoked. Thus, SMIC metric also aims to provide useful information to get an understanding of current code quality of a class in perspective. Given these explanations, this study suggests providing several items of information by the SMIC metric, which they will be exemplified with listing 7.

Listing 7. Sample class to exemplify the SMIC metric

```
10  public class SynchronizedMethod {
⋮         ⋮
20        public synchronized void first(){
⋮               /* statements */
32        }
33
34        public synchronized int second(){
⋮               /* statements */
42        }
43
44        public void third() {
⋮               /* statements */
53        }
54
55        public synchronized boolean fourth() {
⋮               /* statements */
61        }
62
63        public String fifth() {
⋮               /* statements */
70        }
71  }
```

Listing 7 shows a Java class consists of five methods. Referring to the method declaration, methods in lines 20, 34, and 55 are synchronised, and methods in lines 44 and 63 are non-synchronised. The SMIC metric is expected to provide the following information by analysing this class:

I.	**Name of synchronised methods**	**:** `first, second, fourth`
II.	**Number of synchronised methods**	**:** 3
III.	**Number of non-synchronised methods**	**:** 2
IV.	**The total number of methods**	**:** 5

4.3 Metric 2 – Synchronised Methods Line of Code (SMLOC)

As discussed in 'synchronised block' sub-section, a suggestion to improve the performance of the application and consequently enhancing the quality of the code is minimising the scope of locking. One of the best practice that results in a smaller locking scope can be only synchronising a block of critical statements (synchronised block) of a method instead of the entire it. It is clear that choosing synchronised block over synchronised method in methods consists of one statement makes no difference from the locking scope perspective. However, growing the number of statements inside the synchronised method can imply the higher likelihood of the need to change the synchronisation technique.

Apart from the locking scope perspective, increasing the number of statements can consequently increase the possibility of having programming errors in the synchronisation concept. For instance, in the high number of statements, it seems probable that there is different shared variables and functions protected with a same synchronised object (associated with the method), although they may need to be synchronised with variant techniques. This situation can lead to having concurrency faults in the application.

the Considering discussion above, Synchronised Methods Line of Code (SMLOC) metric aims to provide information about the number of statements inside the synchronised method to highlight methods that have the potential for changing their

technique of synchronisation. Previously mentioned matters would seems to indicate a high number of statements imply a law level of code quality. In order to increase a better understanding of the code quality status, several pieces of information are suggested to provide by SMLOC metric, which they will be exemplified with listing 8.

Listing 8. Sample class to exemplify the SMLOC metric

```
10  public class StmtCount {
⋮         ⋮
20        public synchronized void first(){
21              /* 1 statement */
22        }
23
24        public synchronized int second(){
⋮               /* 5 statements */
30        }
31
32        public void third() {
⋮               /* 6 statements */
39        }
40
41        public synchronized boolean fourth() {
⋮               /* 2 statements */
44        }
45
46        public String fifth() {
⋮               /* 8 statements */
55        }
56  }
```

Listing 8 shows a Java class consists of five methods with different numbers of statements. Referring to the method declaration, methods in lines 20, 24, and 41 are synchronised, and methods in lines 32 and 46 are non-synchronised. The SMLOC metric is expected to provide the following information by analysing this class:

I. **Name of synchronised methods along with number of their statements:**

`First [1 statement], second [5 statements], fourth [2 statements]`

II. **The total number of statements inside synchronised method** : 8

III. **Number of synchronized methods** : 3

IV. **Number of non-synchronized methods** : 2

V. **The total number of statements inside non-synchronized methods** : 14

VI. **The total number of methods** : 5

4.4 Metric 3 – Synchronised Blocks in Class (SBIC)

In the scope of this study, another basic unit of measurement is synchronised block, which is essential to enable performing some further analysis, e.g. to examine the synchronisation objects associated with synchronised blocks. Therefore, identifying synchronised blocks of a Java class are targeted at Synchronised Blocks in Class (SBIC) metric. This study proposes that SBIC metric provides several items of information, which they will be exemplified with listing 9.

Listing 9. Sample class to exemplify the SBIC metric

```
10  public class SynchronizedBlock {
 ⋮        ⋮
20        public void first(){
21              synchronized (/*monitorObject*/) {
 ⋮                    /* statements inside synchronised
                      block*/
32              }
 ⋮              /* statements inside method*/
38        }
39
40        public int second(){
 ⋮              /* statements */
46        }
47
48        public void third() {
49              synchronized (/*monitorObject*/) {
 ⋮                    /* statements inside synchronised
                      block*/
55              }
56              synchronized (/*monitorObject*/) {
 ⋮                    /* statements inside synchronised
                      block*/
63              }
64        }
65
66        public boolean fourth() {
 ⋮              /* statements */
70        }
71
72  }
```

Listing 9 shows a Java class consists of four methods. Referring to the method declaration, methods in lines 20 and 48 consists of synchronised blocks. The SBIC metric is expected to extract the following information by analysing this class:

I. **Name of methods consist of synchronised blocks along with number of their synchronised blocks:** `First [1 block], third [2 blocks]`

II. **The total number of methods** **:** `4`

III. **Number of methods consist of synchronised blocks** **:** `2`

IV. **The total number of synchronised blocks** **:** `3`

4.5 Metric 4 - Nested Synchronised Block in Class (NSBIC)

Nested Synchronised Blocks can be precarious if not used delicately. As mentioned in 'nested synchronised block' subsection of chapter four, they can be a potential source of deadlock. Moreover, considering reentrant concept discussed in 'monitor lock' subsection of chapter four, if nested synchronised block uses the same synchronised object in all its levels it means inner synchronised blocks also associate with the same lock. Thus, there is no protection and a thread reentrant all the levels that use the same lock. Although, it is not the aim of using nested synchronised blocks.

Given these explanations, it is arguable that synchronised block can be another unit of code quality measurement in the scope of concurrency, which leads to suggest Nested Synchronised Block in Class (NSBIC) metric. Noticing nested synchronised blocks can help to prevent deadlock with reconsidering the code pattern - either to detect and fix a probable deadlock or to avoid having nested synchronised block if possible. Furthermore, NSBIC metric can facilitate further analyses. For instance, it can derive a metric to detect reentrant mistakes in nested synchronised blocks. Provided information by SBIC metric will be exemplified with listing 10.

Listing 10. Sample class to exemplify the NSBIC metric

```
10  public class NestedSynchronizedBlock {
 ⋮        ⋮
20        public void first(){
21              synchronized (/*monitorObject*/) {
 ⋮                    /* statements inside synchronised
                      block*/
32                    synchronized (/*monitorObject*/) {
 ⋮                          /* statements inside inner
                            synchronised block*/
39                    }
40              }
 ⋮              /* statements inside method*/
46        }
47
48        public int second(){
 ⋮              /* statements */
55        }
56
57        public void third() {
58              synchronized (/*monitorObject*/) {
 ⋮                    /* statements inside synchronised
                      block*/
63                    synchronized (/*monitorObject*/) {
 ⋮                          /* statements inside inner
                            synchronised block*/
70                    }
71              }
72        }
73
74  }
```

Listing 10 shows a Java class consists of three methods. Methods in lines 20 and 57 consist of nested synchronised blocks. The NSBIC metric is suggested to extract the following information by analysing this class:

I. **Name of methods consist of nested synchronised blocks:** `First, third`

II. **The number of methods consist of nested synchronised blocks :** `2`

III. **The total number of methods :** `3`

4.6 Metric 5 - Synchronised Blocks Line of Code (SBLOC)

As mentioned in the 'synchronised block' subsection of literature review, the synchronised block is used over the synchronised method to reduce the lock scope. However, choosing a reasonably certain portion of code is decisive to achieve the desired aim. Therefore, increasing the number of statements inside a synchronised block can be an alert for the possibility of unessential synchronisation on some statements. This situation generates a more significant locking scope unnecessarily.

Synchronised Blocks Line of Code (SBLOC) metric aims to calculate the number of statement inside the synchronised blocks to give an insight into the probability of such a mistake. By the same token, SBLOC can derive other metrics that one of them will be suggested in the next section.

This study proposes providing several items of information by the SBLOC metric, which they will be exemplified with listing 11.

Listing 11. Sample class to exemplify the SBLOC metric

```
10  public class NumberOfStmtSyncBlock {
 ⋮        ⋮
20        public void first(){
21              synchronized (/*monitorObject*/) {
 ⋮                    /* 5 statements inside synchronised
                      block*/
32              }
 ⋮              /* statements inside method*/
38        }
39
40        public int second(){
 ⋮              /* statements */
46        }
47
48        public void third() {
49              synchronized (/*monitorObject*/) {
 ⋮                    /* 3 statements inside synchronised
                      block*/
55              }
56              synchronized (/*monitorObject*/) {
 ⋮                    /* 2 statements inside synchronised
                      block*/
63              }
64        }
65
66  }
```

Listing 11 shows a Java class consists of three methods. Methods in lines 20 and 48 consists of synchronised blocks. The SBLOC metric is suggested to provide the following information by analysing this class:

I. **Name of methods consist of synchronised blocks along with number of synchronised blocks' statements:** `First [one sync block with 5 statements], third [one sync block with 3 statements, one sync block with 2 statements]`

II. **The total number of method/s** **:** `3`

III. **Method/s consist of synchronized block/s** : 2

IV. **The total number of synchronized block/s** : 3

V. **The total number of statements inside synchronised block/s** : 10

4.7 Metric 6 - Compare Synchronised Line of Code in Class (CSLOCIC)

One method to measure the concurrent code quality can be to ensure the appropriateness of used synchronisation technique. To evaluate choosing the synchronised block over synchronised method, the first step can be comparing the number of their statements. Using the SBLOC metric (metric 5), Compare Synchronised Line of Code in Class (CSLOCIC) metric is suggested to provide useful information regarding this comparison.

Consider if a method contains only a synchronised block (zero statements except for this block), there may not be any legitimate reason to choose synchronise block over synchronise method. It is worth to stress that this insight is added to provided information by SBLOC metric that was mentioned in the previous section. Therefore, CSLOCIC is supposed to deepen the understanding of code quality in compare with SBLOC (metric 5). Proposed information by CSLOCIC metric will be exemplified with listing 12.

Listing 12. Sample class to exemplify the CSLOCIC metric

```
10  public class CompareSyncStmt {
 ⋮        ⋮
20       public void first(){
21            synchronized (/*monitorObject*/) {
 ⋮                 /* 5 statements inside synchronised
                   block*/
32            }
 ⋮            /* 1 statement inside method*/
38       }
39
40       public int second(){
 ⋮            /* statements */
46       }
```

Listing 12 shows a Java class consists of two methods. The method in lines 20 consists of a synchronised block. The CSLOCIC metric is expected to extract the following information by analysing this class:

I. **Name of methods consist of synchronised blocks along with number of synchronised blocks/number of statements inside synchronised blocks/number of statements belong to method except the synchronised blocks:** `First [one sync block with 5 statements / 1 statement outside the synchronised block]`

II. **The total number of methods** : `2`

III. **Synchronised methods** : `0`

IV. **Non-synchronized methods without synchronised block** : `1`

V. **Non-synchronized methods consist of synchronised blocks** : `1`

VI. **Methods consist of 0 statement except synchronised blocks** : `0`

4.8 Metric 7 - Static Variables in Class (SVIC)

According to the 'variables' subsection of the literature review chapter, variables play a vital role to correct programming. As discussed static variable is the most problematic one in multithreading execution among all other types. The evidence leads to suggest Static Variables in Class (SVIC) metric. SVIC aims to extract several items of information that can provide clues about possible errors in a concurrent context. This information will be exemplified with listing 13.

Listing 13. Sample class to exemplify the SVIC metric

```
10  public class Variables {
11
12        boolean a = true;
13        static int b = 0;
14        static String c = "Hello";
15        static final int d = 6;
 ⋮        ⋮
35  }
```

In listing 13 Java class, four variable is declared. Referring to the variable declaration, the variable in line 12 is non-static, variables in lines 13 and 14 are static non-final, and variable in line 15 is static final. The SVIC metric are expected to extract following information by analysing this class:

I. **Name of static variable with declaring its final status :**
`b [static non-final], c [static non-final], d [static final]`

II. **The total number of variables : 4**

III. **Non-static variable : 1**

IV. **The total number of static variables : 3**

V. **Static non-final variable : 2**

VI. **Static final variable : 1**

4.9 Metric 8 - Volatile Variables in Class (VVIC)

In the context of multithread programming, *atomicity* is desired to guarantee most of the time. However, as discussed in 'volatile variable' subsection of literature review chapter, volatile cannot address problems related to atomicity. Hence, the use of this variable is limited to very restricted cases. Considering this, Volatile Variables in Class (VVIC) metric aims to provide several items of information about the status of a volatile variable in a Java class.

VVIC metric mainly focuses on comparing the number of volatile variables of a class with non-volatile ones. A sizeable number of volatile can alert about the inappropriateness of used synchronisation technique. Consequently, it can mean the low code quality and the possibility of producing faults in the application. According to the discussion above about volatile variable, it should be noted that sizeable number can mean more than one or two in this concept.

Listing 14 will exemplify the proposed information that VVIC metric provides.

Listing 14. Sample class to exemplify the VVIC metric

```
10  public class VolatileVar {
11        private volatile boolean a;
12        private volatile boolean b;
13        private volatile boolean c;
14        private int d;
15        private static e;
 ⋮        ⋮
36  }
```

In this Java class, five variables are declared. Referring to variable declaration variables in lines 11. 12, and 13 are volatile, and variables in lines 14 and 15 are non-volatile. The VVIC metric is expected to extract the following information by analysing this class:

I.	**Name of volatile variable**	**:** `a, b, c`
II.	**The total number of variables**	**:** `5`
III.	**Non-volatile variables**	**:** `2`
IV.	**Volatile variables**	**:** `3`

4.10 Metric 9 - Synchronization Objects Associated with Synchronised Methods (SOAWSM)

In 'synchronisation object' subsection of the literature review chapter, it was considered that the monitor lock has a crucial role in synchronisation. According to 'concurrent software' section of literature review chapter, a thread that needs to access a synchronised portion of the code has to acquire the monitor lock of a synchronisation object that is associated with that synchronised code. The thread owns the lock exclusively during execution and releases the lock when execution is done. Exclusive access emphasises that a monitor lock is designed to belong to only one thread at a time. Accordingly, if other threads acquire a monitor lock which is owned by other thread, they should wait until it is released. It proves that lock on different objects means there is no synchronisation between them.

Referring to discussions in the aforementioned chapter, static and non-static synchronised methods associate with different synchronisation objects (Class object itself and 'this' respectively); therefore, they lock on different monitor locks. Since the synchronisation object is not declared directly in synchronised methods, there is a good chance of making a mistake in using this technique of synchronisation to protect shared variables. Listing 15 demonstrates an example of this situation:

Listing 15. Sample class to exemplify the SOAWSM metric

```
10  public class SyncObjects {
 ⋮        ⋮
20       Public static synchronized int add(int number) {
21            a += number;
22            return a;
23       }
24
25       public synchronized int subtract (int number) {
26            a -= number;
27            return a;
28       }
 ⋮        ⋮
50  }
```

In listing 15 example, some operations on 'a' are specified in two different methods. It is absolutely necessary to have atomic access to 'a' in order to guarantee consistent behaviour. Strictly speaking, only one thread can be in either method `add` or `subtract`. Since different synchronisation objects are associated with these methods, two different locks are involved. Consider, thread T1 can invokes `add` at the same time that thread T2 invokes `subtract`. Since each thread locks on different objects, there is no happens-before ordering between operations. Therefore, the behaviour of application will depend on thread scheduler and hence is unpredictable.

It could be concluded that an application is thread-safe if either static synchronised methods or non-static synchronised methods are used to protect the

same shared variable. In plain English, threads that access to the same shared variables must acquire the same lock, or there will be no synchronisation and consequently thread-safety.

To avoid such mistakes, this study proposes provides several items of information by Synchronization Objects Associated with Synchronised Methods (SOAWSM) metric, which they will be exemplified based on listing 15 as follow:

I. **Name of method and its associated synchronised object** :
   `add [Class object itself], subtract [this]`
II. **The total number of methods** : `2`
III. **Non-synchronized methods** : `0`
IV. **The total number of synchronized methods** : `2`
V. **Locks on Class object** : `1`
VI. **Locks on this** : `1`

4.11 Metric 10 - Synchronisation Objects Associated with Synchronised Blocks (SOAWSB)

As mentioned in the 'concurrent software' section of literature review, having synchronised block allows to lock on variant synchronisation objects as a monitor, and the monitor lock has a central role in synchronisation. Having plenty of choices can increase the probability of making a mistake in concurrent programming and leads to thread-unsafety. Reviewing the monitor lock associated with synchronised blocks can be constructive to prevent potential errors. The first and crucial step for this analysis is to categorise the type of variant objects that can be used as a monitor. Following types can be considered for this purpose:

1. Class Object
2. 'this' which represents a current object
3. Lock Object
4. Any Object

Considering these types, this study proposes that Synchronisation Objects Associated with Synchronised Blocks (SOAWSB) metric provides several items of information, which they will be exemplified with listing 16.

Listing 16. Sample class to exemplify the SOAWSB metric

```
10  public class SyncBlockObject {
11        private final Object object = new Object();
12        private final Lock lock = new ReentrantLock();
 ⋮        ⋮
20        public void first(){
21              synchronized (SyncBlockObject.class) {
 ⋮                    /* statements */
27              }
28              synchronized (this) {
 ⋮                    /* statements */
33        }
34
35        public void second() {
36              synchronized (object) {
 ⋮                    /* statements */
42              }
43              synchronized (lock) {
 ⋮                    /* statements */
50              }
51              synchronized (this) {
 ⋮                    /* statements */
59              }
60        }
61
62        public boolean third() {
 ⋮              /* statements */
70        }
71
72  }
```

Listing 16 shows a Java class consists of three methods. Referring to the method declaration, methods in lines 20 and 35 consists of synchronised blocks. The SOAWSB metric is expected to extract the following information by analysing this class:

I. **Name of method consist of synchronised blocks and their associated synchronised object:** `first [lock on class object, lock on this], second [lock on class field, lock on lock object, lock on this]`
II. **The total number of methods** : 3
III. **Methods without synchronised blocks** : 1
IV. **Methods consist of synchronised blocks** : 2
V. **The total number of synchronised blocks** : 5
VI. **Locks on (Class object)** : 1
VII. **Locks on [this]** : 2
VIII. **Locks on {Lock object}** : 1
IX. **Locks on ''field''** : 1

4.12 Metric 11 - Synchronised Blocks inside Synchronized Methods (SBISM)

Referring to the 'synchronised method' subsection of the literature review chapter, statements inside synchronised methods are already thread-safe. Using synchronised block inside such methods can represent an error and negatively affect the performance of the application. This study proposes Synchronised Blocks inside Synchronized Methods (SBISM) metric to provide several items of information in this regard, which they will be exemplified with listing 17.

Listing 17. Sample class to exemplify the SBISM metric

```
10  public class SynchronizedBlock {
 ⋮        ⋮
20        public void first(){
21              synchronized (/*monitorObject*/) {
 ⋮              ⋮
32        }
33
34        public int second(){
 ⋮              ⋮
46        }
47
48        public synchronized void third() {
49              synchronized (/*monitorObject*/) {
 ⋮              ⋮
55        }
56  }
```

Listing 17 shows a Java class consists of three methods. Referring to the method declaration, methods in lines 48 is synchronised and also consist of synchronised block. The SBISM metric is expected to extract the following information by analysing this class:

I. **Name of synchronised method consist of synchronised :** `third [1 synchronised block]`

II. **The total number of synchronised methods : 1**

III. **The total number of synchronised blocks inside sync methods : 1**

5 Chapter 5: Static Analysis Tool

5.1 Introduction to the static analysis tool

Attaining the objective three of this project that was declared in chapter one required the development of an analysis tool to apply the suggested metrics. This analysis tool is termed as static since does not involve executing the source code. This chapter will provide a summary of the vital aspects that made this tool operational and illustrates the output frameworks of running it against Java source code.

5.2 Tool specifications

The Java programing language and Eclipse IDE are used to develop this analysis tool. Moreover, Maven is adopted to manage the required dependencies. This tool depends on only one external library, which is called JavaParser. Following sub-section will be dedicated to expounding on this library.

5.2.1 JavaParser – external used library

The JavaParser library is a Java language parser that facilitates the interaction with Java source codes (Bruggen et al., 2019). Employing this library is exceptionally efficient to identify interesting patterns in source code compare to writing laborious Abstract Syntax Tree (AST) traversal to parse the source codes. Most importantly, it makes the developed tool generic for Java language.

There are accurate sources to guide through employing this library such as a complete Javadoc, a dedicated book, a dedicated website, a repository hosted on GitHub (JavaParser, 2019). The figure 4 represents the maven configuration to setup this library.

Figure 4. The maven configuration to setup JavaParser

```
<dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-symbol-solver-core</artifactId>
    <version>3.14.11</version>
</dependency>
```

5.3 Tool instruction

This tool needs to be imported as a Maven project to function correctly. Each class of this tool implements one metric that suggested in the previous chapter and extracts the proposed information by analysing the Java source code in class level. To apply a method to any Java classes, the tool needs to be provided by the intended Java class path. The following section will illustrate the general template to present the tool output.

5.4 The implementation of metrics and their output

As mentioned in the previous section, each class of this tool implements one metric. Analysing a Java class with no method are considered as an exception in all classes of the tool (except classes associated with SVIC and VVIC metric), and an appropriate message is given. In this tool, the attempt is made to always present results in a consistent format. Therefore, a consistent template is designed to display extracted information. The figure 5 represents the general template for the output of such an exceptional analysis.

Figure 5. The general template for tool output with a message for the exception of no method

```
This class has 'no' method.


------------------------------------------------------------------------
/*The same output template of each metric are shown in this part,
which will be illustrated in following subsections.*/
```

It is worth noting that the question mark (**?**) will be used in illustrating the output frameworks where the tool will show the **different numbers** based on each analysis.

5.4.1 Metric 1 – Synchronised Methods in Class (SMIC)

The `MethodSync` class of the analysis tool applies *SMIC* metric to analyse the Java classes. The source code of this class can be found in the Appendix A.1. Running this class against source code cause to extracts information that is proposed in SMIC section of chapter 4. The figure 6 represents a designed template to reveal them.

Figure 6. SMIC metric template for tool output

```
-> '/*name of synchronized method*/' is synchronized method

/*Tool lists all the synchronized methods with the same format as
above*/
⋮

------------------------------------------------------------------------
The total number of method/s   :  ?
Non-synchronized method/s      :  ?
->> Synchronized method/s      : '?'
```

Since synchronised methods are targeted at SMIC, a situation that a class does not consist of any synchronised method are considered as an exception, and an appropriate message will be given as figure 7.

Figure 7. SMIC metric template for tool output with a message for the exception of no sync method

```
This class has 'no' synchronized method.


------------------------------------------------------------------------
The total number of method/s   :  ?
Non-synchronized method/s      :  ?
->> Synchronized method/s      : '0'
```

5.4.2 Metric 2 – Synchronised Methods Line of Code (SMLOC)

The `StmtMethodSync` class applies SMLOC metric to analyse Java classes (see Appendix A.2 for source code). The figure 8 represents the template to show the suggested information in SMLOC section of chapter four by this tool.

Figure 8. SMLOC metric template for tool output

```
-> '/*name of synchronized method*/' is synchronized method with
[/*number of statements inside this method: ? */] statements.

/*Tool lists all the synchronized methods with the same format as
above*/
⋮


------------------------------------------------------------------------
The total number of method/s                                    :  ?
Non-synchronized method/s                                       :  ?
The total number of statements inside non-synchronized method/s    :  ?
->> Synchronized method/s                                       : '?'
->> The total number of statements inside synchronized method/s    : [?]
```

Synchronised methods are targeted at SMLOC. Hence, a situation that a class does not consist of any synchronised method is considered as an exception, and an appropriate message will be given as figure 9.

Figure 9. SMLOC metric template for tool output with a message for the exception of no sync method

```
This class has 'no' synchronized method.

------------------------------------------------------------------------
The total number of method/s                                        :  ?
Non-synchronized method/s                                           :  ?
The total number of statements inside non-synchronized method/s     :  ?
->> Synchronized method/s                                          : '0'
->> The total number of statements inside synchronized method/s     : [0]
```

5.4.3 Metric 3 – Synchronised Blocks in Class (SBIC)

The `BlockSync` class of the analysis tool applies the *SBIC* metric to analyse the Java classes (see Appendix A.3 for the source code). Running this class against Java source code causes to extracts information that is proposed in the SBIC section of chapter four. The figure 10 represents a designed template to show them.

Figure 10. SBIC metric template for tool output

```
-> '/*name of method consist of synchronized block*/' method has
[/*number of synchronized block/s within this method: ? */] synchronized
blocks

/*Tool lists all the methods consist of synchronized block/s with the
same format as above*/
⋮

------------------------------------------------------------------
The total number of method/s                    :   ?
->> Method/s consist of synchronized block/s    :  '?'
->> The total number of synchronized block/s    :  [?]
```

Since synchronised blocks are targeted at SBIC, a situation that a class does not consist of any synchronised block are considered as an exception, and an appropriate message will be given as figure 11.

Figure 11. SBIC metric template for tool output with a message for the exception of no sync block

```
This class has 'no' synchronized block.


----------------------------------------------------------------------
The total number of method/s                      :   ?
->> Method/s consist of synchronized block/s     :  '0'
->> The total number of synchronized block/s     :  [0]
```

5.4.4 Metric 4 – Nested Synchronised Block in Class (NSBIC)

The `NestedSync` class of the analysis tool applies *NSBIC* metric for analysing Java classes to extract suggested information in *NSBIC section* of literature review. The source code of this class can be found in the Appendix A.4. The figure 12 represents a designed template to show the output of this tool.

Figure 12. NSBIC metric template for tool output

```
-> '/*name of method consist of nested synchronized block*/' has nested
synchronized block

/*Tool lists all the methods consist of synchronized block/s with the
same format as above*/
⋮


----------------------------------------------------------------------
The total number of method/s                                    :   ?
The number of method/s consist of nested synchronized block/s :  '?'
```

Since nested synchronised methods are targeted at SMIC, a situation that a class does not consist of any of them is considered as an exception, and an appropriate message will be given as figure 13.

Figure 13. NSBIC metric template for tool output with a message for the exception of no nested sync block

```
This class has 'no' nested synchronized block.


------------------------------------------------------------------------
The total number of method/s                                    :   ?
The number of method/s consist of nested synchronized block/s :  '0'
```

5.4.5 Metric 5 – Synchronised Blocks Line of Code (SBLOC)

The `StmtBlockSync` class of the analysis tool applies *SBLOC* metric to analyse the Java classes. The source code of this class can be found in the Appendix A.5. Running this class against source code cause to extracts information that is proposed in *SBLOC* section of chapter four. The figure 14 represents a designed template to reveal them.

Figure 14. SBLOC metric template for tool output

```
-> '/*name of method consist of synchronized block*/' method has
(/*number of synchronized block/s within this method: ? */)
synchronized blocks with [/*number of statements inside synchronized
block/s respectively: ?, ? */] statement/s respectively.

/*Tool lists all the methods consist of synchronized block/s with the
same format as above*/
⋮


------------------------------------------------------------------
The total number of method/s                                :  ?
->> Method/s consist of synchronized block/s                : '?'
->> The total number of synchronized block/s                : (?)
-->>The total number of statements inside synchronized block/s: [?]
```

Since statements inside synchronised blocks are targeted at *SBLOC*, a situation that a class does not consist of any synchronised block are considered as an exception, and an appropriate message will be given as figure 15.

Figure 15. SBLOC metric template for tool output with a message for the exception of no sync block

```
This class has 'no' synchronized block.


-----------------------------------------------------------------------
The total number of method/s                                    :  ?
->> Method/s consist of synchronized block/s                    : '0'
->> The total number of synchronized block/s                    : (0)
-->>The total number of statements inside synchronized block/s: [0]
```

5.4.6 Metric 6 – Compare Synchronised Line of Code in Class (CSLOCIC)

The `CompareStmt` class of the analysis tool applies *CSLOCIC* metric to analyse the Java classes (see Appendix A.6 for the source code). Running this class against source code cause to extracts proposed information that in *CSLOCIC* section of chapter four were exemplified. The figure 16 represents a designed template to show them.

Figure 16. CSLOCIC metric template for tool output

```
-> '/*name of method consist of synchronized block*/' is non-
synchronized method with (/*number of synchronized block/s within
this method: ? */) synchronized block with [/*number of statements
inside synchronized block/s respectively: ?, ? */] statement/s
respectively.
--> It has [/*number of statements inside method except its
synchronized block/s: ? */] statement/s except its synchronized
block/s.

/*Tool lists all the methods consist of synchronized block/s with the
same format as above*/
⋮


-----------------------------------------------------------------------
The total number of method/s                                      :  ?
Synchronized method/s                                             :  ?
Non-synchronized method/s without synchronized block              :  ?
->> Non-synchronized method/s consist of synchronized block/s     : '?'
-->> Method/s consist of [0] statement except synchronized block/s : [?]
```

Since comparison is targeted at SMIC, having synchronised block is necessary to proceed further. A situation that methods of a class do not consist of any synchronised block is considered as an exception, and an appropriate message will be given as figure 17.

Figure 17. CSLOCIC metric template for tool output with a message for the exception

```
This class has 'no' synchronized block inside non-synchronized method/s.


-----------------------------------------------------------------------
The total number of method/s                                        :  ?
Synchronized method/s                                               :  ?
Non-synchronized method/s without synchronized block                :  ?
->> Non-synchronized method/s consist of synchronized block/s      : '0'
-->> Method/s consist of [0] statement except synchronized block/s : [0]
```

5.4.7 Metric 7 – Static Variables in Class (SVIC)

The `StaticVariable` class of the analysis tool applies the *SVIC* metric to analyse the Java classes (see Appendix A.7 for source code). Running this class against source code cause to extracts information that is proposed in the SVIC section of chapter four. The figure 18 represents a designed template to reveal them.

Figure 18. SVIC metric template for tool output

```
-> '/*name of variable*/'  variable is (/*static final or non-
fianl*/)

/*Tool lists all static final and non-final variables.*/
⋮


-------------------------------------------------------------------------
The total number of variables               :  ?
Non-static variable                         :  ?
->> The total number of static variable/s   : '?'
--->> Static non-final variable             : (?)
--->> Static final variable                 : [?]
```

Since variables are targeted at SMIC, a situation that a class does not have any variable are considered as an exception, and an appropriate message will be given as figure 19.

Figure 19. SVIC metric template for tool output with a message for the exception of no variable

```
This class has 'no' variable.


------------------------------------------------------------------------
The total number of variables                :  0
Non-static variable                          :  0
->> The total number of static variable/s    : '0'
--->> Static non-final variable              : (0)
--->> Static final variable                  : [0]
```

5.4.8 Metric 8 – Volatile Variables in Class (VVIC)

The `VolatileVariable` class of the analysis tool applies *VVIC* metric to analyse the Java classes. The source code of this class can be found in the Appendix A.8. Running this class against source code cause to extracts information that is proposed in VVIC section of chapter four. The figure 20 represents a designed template to reveal them.

Figure 20. VVIC metric template for tool output

```
-> '/*name of volatile variable*/' is volatile variable


/*Tool lists all the volatile variable.*/
⋮


---------------------------------------------------------------------
The total number of variable/s    :  ?
Non-volatile variable/s           :  ?
->> Volatile variable/s           : '?'
```

Since variables are targeted at VVIC metric, a situation that a class does not have any variable are considered as an exception, and an appropriate message will be given as figure 21.

Figure 21. VVIC metric template for tool output with a message for the exception of no variable

```
This class has 'no' variable.


------------------------------------------------------------------
The total number of variable/s    :  0
Non-volatile variable/s           :  0
->> Volatile variable/s           : '0'
```

5.4.9 Metric 9 – Synchronization Objects Associated with Synchronised Methods (SOAWSM)

The `MethodSyncObject` class of the analysis tool applies SOAWSM metric to analyse the Java classes (see Appendix A.9 for source code). Running this class against source code cause to extracts proposed information in SOAWSM section of chapter four. The figure 22 represents a designed template to reveal them.

Figure 22. SOAWSM metric template for tool output

```
-> '/*name of synchronised method*/' method lock on (class object)
itself
-> '/*name of synchronised method*/' method lock on [this]

/*Tool lists all the synchronised methods and detects their
associated synchronisation objects*/
⋮

------------------------------------------------------------------
The total number of method/s                  :  ?
Non-synchronized method/s                     :  ?
->> The total number of synchronized method/s  : '?'
-->> Lock/s on (Class object)                 : (?)
-->> Lock/s on [this]                         : [?]
```

Since synchronisation object are targeted at SMIC, a situation that a class does not consist of any synchronised method are considered as an exception, and an appropriate message will be given as figure 23.

Figure 23. SOAWSM metric template for tool output with a message for the exception of no syncing method

```
This class has 'no' synchronized method to lock on any object.


-----------------------------------------------------------------------
The total number of method/s                    :  ?
Non-synchronized method/s                       :  ?
->> The total number of synchronized method/s   : '0'
-->> Lock/s on (Class object)                   : (0)
-->> Lock/s on [this]                           : [0]
```

5.4.10 Metric 10 – Synchronisation Objects Associated with Synchronised Blocks (SOAWSB)

The `BlockSyncObject` class of the analysis tool applies SOAWSB metric to analyse the Java classes. The source code of this class can be found in the Appendix A.10. Running this class against source code cause to extracts information that is proposed in section SMIC of chapter four. The figure 24 represents a designed template to reveal them.

Figure 24. SOAWSB metric template for tool output

```
-> '/*name of method consists of synchronised block*/' method has
synchronized block that lock on ((Class object) / [this] / {Lock object}
/ ''field'')

/*Tool lists all the methods consist of synchronized block/s and defines
the associated synchronised object with each block separately with the
same format as above*/
⋮

------------------------------------------------------------------------
The total number of method/s                    :   ?
Method/s without synchronized block/s           :   ?
->> Method/s consist of synchronized block/s    :  '?'
->> The total number of synchronized block/s    :   ?
-->> Lock/s on (Class object)                   :  (?)
-->> Lock/s on [this]                           :  [?]
-->> Lock/s on {Lock object}                    :  {?}
-->> Lock/s on ''field''                        : ''?''
```

Since synchronisation object associated with a synchronised block is targeted at SMIC, a situation that a class does not have any synchronised block are considered as an exception, and an appropriate message will be given as figure 25.

Figure 25. SOAWSB metric template for tool output with a message for the exception of no sync block

```
This class has no synchronized block.


------------------------------------------------------------------------
The total number of method/s                    :   ?
Method/s without synchronized block/s           :   ?
->> Method/s consist of synchronized block/s    :  '0'
->> The total number of synchronized block/s    :   0
-->> Lock/s on (Class object)                   :  (0)
-->> Lock/s on [this]                           :  [0]
-->> Lock/s on {Lock object}                    :  {0}
-->> Lock/s on ''field''                        : ''0''
```

5.4.11 Metric 11 – Synchronised Blocks inside Synchronized Methods (SBISM)

The `SyncBlockInSyncMethod` class of the analysis tool applies *SBISM* metric to analyse the Java classes. The source code of this class can be found in the Appendix A.11. Running this class against source code cause to extracts information that is proposed in SBISM section of chapter four. The figure 26 represents a designed template to reveal them.

Figure 26. SBISM metric template for tool output

```
-> '/*name of method consists of synchronised block*/' is
synchronized method and has [/*number of synchronized block/s within
this method: ? */] synchronized block!

/*Tool lists all the synchronised methods consist of synchronized
block/s with the same format as above*/
⋮

------------------------------------------------------------------
The total number of sync method/s                       : '?'
The total number of sync block/s inside sync method/s   : [?]
```

Since synchronised block inside the synchronised method is targeted at SMIC, two exceptions are considered for this metric. Firstly, if a class does not have any synchronised method, appropriate message will be given as figure 27.

Figure 27. SBISM metric template for tool output with a message for the exception of no sync method

```
This class has 'no' synchronized method.


------------------------------------------------------------------
The total number of sync method/s                       : '0'
The total number of sync block/s inside sync method/s   : [0]
```

Secondly, if synchronised methods do not have any synchronised block appropriate message will be given as figure 28.

Figure 28. SBISM metric template for tool output with a message for the exception of no sync block in sync method

```
This class has 'no' synchronized block inside its synchronized method/s.


-----------------------------------------------------------------
The total number of sync method/s                        : '0'
The total number of sync block/s inside sync method/s    : [0]
```

5.5 The verification of the tool

This tool is required to implement all suggested metrics by this study properly and extract the proposed information accurately. To verify that this tool meets the requirements and fulfils its intended purpose, some actions take place.

Due to the nature of this tool, unit testing could not be used to verify it. Therefore, the verification was done under close observation. In the first step, different code patterns were written for each method to examine the tool could function correctly. Subsequently, the tool performance was observed for analysing the noticeable number of source code that randomly was selected from SIR. The results verify that this tool can meet the requirements and fulfils its intended purpose.

Next chapter will demonstrate conducted experimental evaluations by the advantage of this tool.

6 Chapter 6: Experimental Evaluation

6.1 Introduction to experimental evaluation

The aim of implementing the experiments is to evaluate the suggested metrics in chapter 4. The procedure to conduct this evaluation was discussed in chapter 3. This is the final step to attain objective three of this project (Evaluating suggested metrics). One Java source code sample is chosen for each metric to perform the experiment that the following sections will detail them. Since the source codes are extremely lengthy they can be accessed along with digital source codes of this project.

6.2 Metric 1 – Synchronised Methods in Class (SMIC)

6.2.1 Description

Listing 18 shows a class of account programme. SIR states that this programme demonstrates parallel concurrency faults (deadlock and race). Listing 18's class is responsible for doing some account operations. It has four methods that are declared in lines 23, 27, 31 and 44 that all of them are synchronised. Although its functions are synchronised, SIR declares there is an interleaving, which causes mentioned faults on the program.

Listing 18. Provided class by SIR to evaluate the SMIC metric

```
 3  public class Account {
 ⋮        ⋮
13        synchronized void depsite(…){
14              ⋮
15        }
16
17        synchronized void withdraw(…){
18              ⋮
19        }
20
21        synchronized void transfer(…){
 ⋮              ⋮
32        }
33
34        synchronized void print(){
35        }
36
37  }
```

6.2.2 Result

The figure 29 represents the output of running a static analysis tool to analyse the listing 18's class.

Figure 29. Result of the SMIC metric evaluation

```
-> 'depsite' is synchronized method
-> 'withdraw' is synchronized method
-> 'transfer' is synchronized method
-> 'print' is synchronized method

---------------------------------------------------------------------
The total number of method/s   :  4
Non-synchronized method/s      :  0
->> Synchronized method/s      : '4'
```

6.2.3 The verification of result and findings

The result is as expected:

I. **Name of synchronised methods:** All four methods of the class are identified as synchronised, and their name is shown correctly (see lines 23, 27, 31 and 44 of listing 18).

II. **Number of synchronised methods:** Is equal to the number of listed methods in (I.).

III. **Number of non-synchronised methods:** Does not exist.

IV. **The total number of methods:** Is equal to synchronised ones in this case.

The analysis shows that all four methods of listing 18's class are synchronised. As discussed in the SMIC section, it can negatively affect the performance of the application by extending locking scope and increase the potential for having a deadlock. Furthermore, SIR discloses that this code has deadlock and race concurrency faults. Given this evidence, it is arguable that this class has a low code quality, and choosing the synchronised method was not efficient technique to synchronise all critical portion of codes. Therefore, revising the design of the class is worth to rectify the errors, which leads to enhance its quality.

6.3 Metric 2 – Synchronised Methods Line of Code (SMLOC)

6.3.1 Description

Displayed class in listing 19 is used to manage the allocation and freeing of blocks. It has three methods that are declared in lines 35, 93, and 106 that all of them are synchronised. Although all of its functions are synchronised, SIR states that there is a synchronisation gap between methods `getFreeBlockIndex` and `markAsAllocatedBlock` in which anything can be done.

Listing 19. Provided class by SIR to evaluate the SMLOC metric

```
  8  public class AllocationVector {
  ⋮        ⋮
 35        synchronized public int getFreeBlockIndex(){
  ⋮              /* 12 statements */
 86        }
  ⋮
 93        synchronized public void  markAsAllocatedBlock(…){
  ⋮              /* 1 statement */
 99        }
  ⋮
106        synchronized public void  markAsFreeBlock(…){
  ⋮              /* 1 statement */
112        }
113
114  }
```

6.3.2 Result

The figure 30 represents the output of running a static analysis tool to analyse listing 19's class.

Figure 30. Result of the SMLOC metric evaluation

```
-> 'getFreeBlockIndex' is synchronized method with [12] statements.
-> 'markAsAllocatedBlock' is synchronized method with [1] statement.
-> 'markAsFreeBlock' is synchronized method with [1] statement.


------------------------------------------------------------------------
The total number of method/s                                    :  3
Non-synchronized method/s                                       :  0
The total number of statements inside non-synchronized method/s   :  0
->> Synchronized method/s                                       : '3'
->> The total number of statements inside synchronized method/s   : [14]
```

6.3.3 The verification of result and findings

The result is as expected:

I. **Name of synchronised methods along with number of their statements:** All three methods of the class are identified as synchronised, and their *name* and *number of their statements* are shown correctly (see lines 35, 93, and 106 of listing 19).

II. **The total number of statements inside synchronised methods:** Is equal to the sum of the shown numbers of statements in (I.).

III. **Number of synchronised methods:** Is equal to the number of listed methods in (I.).

IV. **Number of non-synchronized methods:** Does not exist.

V. **The total number of statements inside non-synchronized methods:** it is zero as there is no non-synchronized method.

VI. **The total number of methods:** Is equal to synchronised ones in this case.

The result shows that the two methods of this class have one statement. The early assumption is made about these methods is that their design can be correct. However, according to the aforementioned matters in section SMLOC of chapter four, the method `getFreeBlockIndex` with 12 statements seems to have a low level of code quality. Some clues can reinforce this perception. Firstly, as mentioned in description sub-section, SIR statements about the existence of synchronisation gap in `getFreeBlockIndex` method. Secondly, since all methods of this class are synchronised, it may be a warning that the programmer has only used one technique for synchronisation, regardless of considering variant techniques to choose the most appropriate one.

The evidence suggests that this class needs to be redesigned to bridge its synchronisation gap and therefore develops its quality.

6.4 Metric 3 – Synchronised Blocks in Class (SBIC)

6.4.1 Description

Listing 20 shows a class of sleepingBarber programme. Listing 20's class is responsible for doing some account operations. It has two methods that both of them

consist of synchronised blocks. The method declared in lines 22 has four synchronised blocks (see lines 23, 31, 36, and 40), and the method declared in line 49 has one synchronised block (see line 50). Although its functions are synchronised, SIR declares there is an interleaving, which causes mentioned faults on the program.

Listing 20. Provided class by SIR to evaluate the SBIC metric

```
77  class BarberShop {
 ⋮        ⋮
22        public void requestCustomer(){
23              synchronized (…) {
 ⋮                    ⋮
29
30              }
31
 ⋮              synchronized (…) {
34                   ⋮
35              }
36
37              synchronized (…) {
38                   ⋮
39              }
40
 ⋮              synchronized (…) {
46                   ⋮
47              }
48        }
49
50        public void getHairCut(){
 ⋮              synchronized (…) {
77                   ⋮
78              }
79        }
    }
```

6.4.2 Result

The figure 31 represents the output of running a static analysis tool to analyse listing 20's class.

Figure 31. Result of the SBIC metric evaluation

```
-> 'requestCustomer' method has [4] synchronized blocks
-> 'getHairCut' method has [1] synchronized block


---------------------------------------------------------------------
The total number of method/s                    :   2
->> Method/s consist of synchronized block/s     :  '2'
->> The total number of synchronized block/s     :  [5]
```

6.4.3 The verification of result and findings

The result is as expected:

I. **Name of methods consist of synchronised blocks along with number of their synchronised blocks:** all the proposed information are shown accurately (see lines 22, 23, 31, 36, 40, 49, and 50 of listing 20).

II. **The total number of methods:** Is equal to the number of listed methods in (I.) in this case.

III. **Number of methods consist of synchronised blocks:** Is equal to the number of listed methods in (I.).

IV. **The total number of synchronised blocks:** Is equal to the sum of the number of synchronised blocks listed in (I.).

The analysis shows that this class has two methods which they contain synchronised block. It also indicates the number of synchronised blocks in each method. This would seem to imply that the SBIC metric could accurately measure this Java class for a proposed unit of measurement and can make further analyses possible.

6.5 Metric 4 – Nested Synchronised Block in Class (NSBIC)

6.5.1 Description

Listing 21 demonstrates a Java class provided by Friesen (2015). This 21's class has two methods declared in lines 14 and 24 that both of them consist of synchronised blocks. Friesen (2015) states that this class is a typical example of deadlock.

Listing 21. Provided class by Friesen (2015) to evaluate the NSBIC metric

```
 9  public class DeadlockDemo {
 ⋮        ⋮
14        public void instanceMethod1(){
15              synchronized(lock1){
16                    synchronized(lock2){
 ⋮                          ⋮
20                    }
21              }
22        }
23
24        public void instanceMethod2(){
25              synchronized(lock2) {
26                    synchronized(lock1) {
 ⋮                          ⋮
30                    }
31              }
32        }
33
34  }
```

6.5.2 Result

The figure 32 represents the output of running a static analysis tool to analyse listing 21 class.

Figure 32. Result of the NSBIC metric evaluation

```
-> 'instanceMethod1' has nested synchronized block
-> 'instanceMethod2' has nested synchronized block


------------------------------------------------------------------------
The total number of method/s                                    :   2
The number of method/s consist of nested synchronized block/s :  '2'
```

6.5.3 The verification of result and findings

The result is as expected:

I. **Name of methods consist of nested synchronised blocks:** Methods consist of a nested synchronised block are identified, and their name is shown correctly (see lines 14 and 24 of listing 21).

II. **The number of methods consist of nested synchronised blocks:** Is equal to the number of listed methods in (I.).

III. **The total number of methods:** it is equal to (II.) in this case.

The analysis shows that this class has two methods that both of them contain nested synchronised block. As discussed in NSBIC section of chapter four, it is necessary to be careful with nested synchronised block, since they increase the likelihood of making errors. Friesen (2015) statement about the existence of deadlock in this code reinforces this view. Thus, it could be concluded that this class does not have adequate code quality, and its error should rectify.

6.6 Metric 5 – Synchronised Blocks Line of Code (SBLOC)

6.6.1 Description

Listing 22 illustrates a Java class that is part of the 'log' programme. SIR states that this programme demonstrates a null pointer exception fault due to unprotected field access. Listing 22's class has two methods that are declared in lines 86 and 118 that one of them consists of synchronised block.

Listing 22. Provided class by SIR to evaluate the SBLOC metric.

```
 40  class LoadXMLAction
 41      extends AbstractAction
 42  {
  ⋮        ⋮
 86       public void actionPerformed(…){
  ⋮             ⋮
108       }
  ⋮
118       private int loadFile(…)
119          …
120       {
121            synchronized (…) {
  ⋮                 /* 12 statements inside synchronised
                    block */
137            }
138       }
139  }
```

6.6.2 Result

The figure 33 represents the output of running a static analysis tool to analyse listing 22's class.

Figure 33. Result of the SBLOC metric evaluation

```
-> 'loadFile' method has (1) synchronized block with [12] statement/s.


---------------------------------------------------------------------
The total number of method/s                                  :  2
->> Method/s consist of synchronized block/s                 : '1'
->> The total number of synchronized block/s                 : (1)
-->>The total number of statements inside synchronized block/s: [12]
```

6.6.3 The verification of result and findings

The result is as expected:

I. **Name of methods consist of synchronised blocks along with number of synchronised blocks' statements:** Method consists of a synchronised block are identified, its name and the number of statements inside its synchronised block are shown correctly (see line 118).

II. **The total number of method/s:** Is equal to declared methods in the class (see line 86 and 118).

III. **Method/s consist of synchronised block/s:** Is equal to the number of listed methods presented in (I.).

IV. **The total number of synchronised block/s:** In this case, it is equal to presented information in (I.).

V. **The total number of statements inside synchronised block/s:** In this case, it is equal to presented information in (I.).

The result shows that one out of two methods of this class consists of a synchronised method, which its number of statements is significantly high. Referring to the discussion in SBLOC section of chapter four, it can be a clue about unnecessary synchronisation that can negatively affect the performance of the application hence decrease the code quality. SIR statements about presenting concurrency fault in this programme support the view that there can be programming errors in this source code. Therefore, reviewing the design of this class seems constructive.

6.7 Metric 6 – Compare Synchronised Line of Code in Class (CSLOCIC)

6.7.1 Description

For performing this experiment, AsyncAppender class of 'log' programme are chosen that is of a reasonable length. This class has 19 methods that 11 of them are consist of synchronised blocks. This noticeable number of methods can make the findings more sensible.

6.7.2 Result

The figure 34 represents the output of running a static analysis tool to analyse the class mentioned in previous subsection.

Figure 34. Result of the CSLOCIC metric evaluation

```
-> 'addAppender' is non-synchronized method with (1) synchronized block
with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.


-> 'append' is non-synchronized method with (1) synchronized block with
[1] statement/s.
--> It has [6] statement/s except its synchronized block/s.


-> 'close' is non-synchronized method with (2) synchronized blocks with
[2, 2] statement/s respectively.
--> It has [1] statement/s except its synchronized block/s.


-> 'getAllAppenders' is non-synchronized method with (1) synchronized
block with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.


-> 'getAppender' is non-synchronized method with (1) synchronized block
with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.


-> 'isAttached' is non-synchronized method with (1) synchronized block
with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.


-> 'removeAllAppenders' is non-synchronized method with (1) synchronized
block with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.


-> 'removeAppender' is non-synchronized method with (1) synchronized
block with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.
```

```
-> 'removeAppender' is non-synchronized method with (1) synchronized
block with [1] statement/s.
--> It has [0] statement/s except its synchronized block/s.


-> 'setBufferSize' is non-synchronized method with (1) synchronized
block with [4] statement/s.
--> It has [1] statement/s except its synchronized block/s.


-> 'setBlocking' is non-synchronized method with (1) synchronized block
with [2] statement/s.
--> It has [0] statement/s except its synchronized block/s.

------------------------------------------------------------------------
The total number of method/s                                    :  19
Synchronized method/s                                           :  0
Non-synchronized method/s without synchronized block            :  8
->> Non-synchronized method/s consist of synchronized block/s   : '11'
-->> Method/s consist of [0] statement except synchronized block/s: [8]
```

6.7.3 The verification of result and findings

The result is as expected:

I. **Name of methods consist of synchronised blocks along with number of synchronised blocks/number of statements inside synchronised blocks/number of statements belong to method except the synchronised blocks:** Methods consist of the synchronised block are identified, their names and the numbers of statements inside their synchronised blocks are shown correctly (see Appendix).

II. **The total number of methods:** Is equal to the declared method in this class

III. **Synchronised methods:** This class has not any synchronised method

IV. **Non-synchronized methods without synchronised block:** Is equal to the correct number

V. **Non-synchronized methods consist of synchronised blocks:** Is equal to the number of listed methods presented in (I.).

VI. **Methods consist of 0 statement except synchronised blocks:** Is equal to the number of listed methods identified with zero statements in (I.).

The result shows that most of the methods (11 out of 19) contain a synchronised block. On the other hand, there is no synchronised method. Considering the noticeable number of methods, it can imply that the synchronisation technique was employed inappropriately. Other aspects of the result reinforce this view. Most methods with the synchronised block (8 out of 11) have no other statements. As discussed in CSLOCIC section of chapter four, there can be no legitimate reason to choose synchronise block over the synchronise method. The view becomes even stronger for methods that have one synchronised block with one statement.

In addition to all of these, this is a faulty programme in concurrent context as SIR aforementioned statement (in the previous section). Thus, it could be concluded that the code quality cannot be outstanding, and the code design needs to be reviewed.

6.8 Metric 7 – Static Variables in Class (SVIC)

6.8.1 Description

A class of 'elevator' programme is shown in listing 23. SIR states that this programme demonstrates a concurrency fault that is array manipulation problems in a multithreaded context. This class has a noticeable number of variable that can broaden the understanding of the value of provided information.

Listing 23. Provided class by SIR to evaluate the SVIC metric

```
 4  public class Elevator implements Runnable{
 5
 6     public static final int MOVING_UP = 1;
 7     public static final int NO_DIRECTION = 0;
 8     public static final int MOVING_DOWN = -1;
10     public static final int MOVING = 1;
11     public static final int STOPPED = 0;
13     public static final int DOOR_OPEN = 1;
14     public static final int DOOR_CLOSED = 0;
15     private static final long FLOOR_WAIT_TIME = 1000;
16     public static final long FLOOR_TRAVEL_TIME = 1000;
17     private static final long INACTIVE_TIME = 1000 * 2;
18     private static final int MAX_OCCUPANCY = 20;
19     private int elevatorID;
20     private int doorState;
21     private int motionState;
22     private int motionDirection;
23     private volatile int currentFloorNumber;
24     private boolean requestDoorOpen;
25     private boolean[] destinationList = new
    boolean[Building.MAX_FLOORS]; // of type int
26     private static ElevatorController elevatorController;
27     private Vector riders = new Vector();
28     private Thread activeElevator;
29     private Logger log;
30     private volatile boolean keepRunning;
 ⋮     ⋮
299  }
```

6.8.2 Result

The figure 35 represents the output of running a static analysis tool to analyse listing 23 class.

Figure 35. Result of the SVIC metric evaluation

```
-> 'MOVING_UP'  variable is [static final]
-> 'NO_DIRECTION'  variable is [static final]
-> 'MOVING_DOWN'  variable is [static final]
-> 'MOVING'  variable is [static final]
-> 'STOPPED'  variable is [static final]
-> 'DOOR_OPEN'  variable is [static final]
-> 'DOOR_CLOSED'  variable is [static final]
-> 'FLOOR_WAIT_TIME'  variable is [static final]
-> 'FLOOR_TRAVEL_TIME'  variable is [static final]
-> 'INACTIVE_TIME'  variable is [static final]
-> 'MAX_OCCUPANCY'  variable is [static final]
-> 'elevatorController'  variable is (static non-final)


----------------------------------------------------------------------
The total number of variables               :  23
Non-static variable                         :  11
->> The total number of static variable/s    : '12'
--->> Static non-final variable              : (1)
--->> Static final variable                  : [11]
```

6.8.3 The verification of result and findings

The result is as expected:

I. **Name of static variable with declaring its final status:** Static final and non-final variable are identified, and their names are shown correctly (see lines 6 to 18, and 26 of listing 23).

II. **The total number of variables:** See lines 6 to 30 of listing 23

III. **Non-static variable:** see lines 19 to 25, and 27 to 30 of listing 23

VII. **The total number of static variables:** Is equal to the number of listed variables in (I.).

IV. **Static non-final variable:** See line 26 of listing 23

V. **Static final variable:** See lines 6 to 18 of listing 23

The result shows that almost half of the variables (12 of 23) are static. As discussed in the 'static variable' subsection of the literature review chapter, they can cause an issue during concurrent execution. However, the result also shows that 11 out of 12 of static variables were declared as final that is they are thread-safe. Since SIR states this programme has array manipulation problems that are not related to these static variables, this would seem to indicate declaring them as final could protect this code against possible concurrency issues (from the static variable perspective), although the number of them is noticeable.

6.9 Metric 8 – Volatile Variables in Class (VVIC)

6.9.1 Description

For conducting this experiment, another class of 'elevator' programme is chose that listing 24 shows it. This class has a noticeable number of variable that can deepen the understanding of the value of the provided information.

Listing 24. Provided class by SIR to evaluate the VVIC metric

```
 5  public class Person implements Runnable {
 7     public static final int WAITING = 1;
 8     public static final int TAKING_STAIRS = 2;
 9     public static final int WORKING = 3;
10     public static final int WALKING_OUTSIDE = 4;
11     public static final int RIDING = 5;
12     public static final int GOING_NOWHERE = -1;
13     public static final int OUTSIDE = -1;
14     public static final int IN_ELEVATOR = 0;
15     private static Building building;
16     private int personID;
17     private int destination;
18     private int location;
20     private int activity;
21     private Elevator elevator;
22     private Floor floor;
23     private Thread activePerson;
24     private Logger log;
25     private volatile boolean keepRunning;
```

6.9.2 Result

The figure 36 represents the output of running a static analysis tool to analyse listing 24's class.

Figure 36. Result of the VVIC metric evaluation

```
-> 'keepRunning' is volatile variable


---------------------------------------------------------------------
The total number of variable/s    :  18
Non-volatile variable/s           :  17
->> Volatile variable/s           : '1'
```

6.9.3 The verification of result and findings

The result is as expected:

I. **Name of volatile variable:** Volatile variable are identified, and its name is shown correctly (see line 25 of listing 24).
II. **The total number of variables:** See lines 7 to 25 of listing 24
III. **Non-volatile variables:** See lines 7 to 24 of listing 24
IV. **Volatile variables:** See line 25 of listing 24

The result shows that although this class has a noticeable number of variables (25), only one of them is volatile. According to 'volatile variable' subsection of literature review chapter and VVIC section of chapter four, it can imply the appropriateness of used synchronisation technique. The SIR statement (discussed in the previous section) reinforce the aforementioned view that this volatile variable does not cause any problem for this programme. It would seem to indicate that the code could have the acceptable quality from the volatile perspective.

6.10 Metric 9 – Synchronization Objects Associated with Synchronised Methods (SOAWSM)

6.10.1 Description

For performing this experiment, another class of 'log' programme are chosen that is of a reasonable length. This class has 15 methods that 11 of them are synchronised. This noticeable number of methods can make the findings more sensible.

6.10.2 Result

The figure 37 represents the output of running a static analysis tool to analyse mentioned class in previous subsection.

Figure 37. Result of the SOAWSM metric evaluation

```
-> 'getTimeZone' method lock on [this]
-> 'setTimeZone' method lock on [this]
-> 'getLocale' method lock on [this]
-> 'setLocale' method lock on [this]
-> 'getPattern' method lock on [this]
-> 'setPattern' method lock on [this]
-> 'getOutputFormat' method lock on [this]
-> 'setOutputFormat' method lock on [this]
-> 'getDateFormatInstance' method lock on [this]
-> 'setDateFormatInstance' method lock on [this]
-> 'configure' method lock on [this]

------------------------------------------------------------------------
The total number of method/s                    :  15
Non-synchronized method/s                       :  4
->> The total number of synchronized method/s   : '11'
-->> Lock/s on (Class object)                   : (0)
-->> Lock/s on [this]                           : [11]
```

6.10.3 The verification of result and findings

The result is as expected:

I. **Name of method and its associated synchronised object:** Synchronised methods, their names, and their associated synchronisation object are identified correctly.
II. **The total number of methods:** Is equal to declared methods in the class
III. **Non-synchronized methods:** see source code
IV. **The total number of synchronised methods:** Is equal to the number of listed methods presented in (I.).
V. **Locks on Class object:** Since this class has no static synchronised method, it is equal to zero in this case.
VI. **Locks on this:** Is equal to the number of listed methods associated with 'this' presented in (I.).

The result shows that most of the methods are synchronised (11 out of 15). Moreover, all of them lock on 'this' that is the same synchronisation object. Although it can imply the better code quality with a small number of synchronised methods as discussed in SOAWSM section of chapter four, it can have a completely different meaning with a noticeable number of synchronised methods. Referring to 'concurrent software' section of the literature review chapter, using the same synchronisation object not only can cause faults but also can increase the locking scope and negatively affect the performance of the application.

There is awareness of concurrency faults in this programme from SIR statement. Therefore, it could be concluded that synchronisation is done inappropriately and programme needs to be redesigned for enhancing the code quality.

6.11 Metric 10 – Synchronisation Objects Associated with Synchronised Blocks (SOAWSB)

6.11.1 Description

Listing 25 illustrates a Java class that is part of the 'WrongLock' programme. SIR states that this programme demonstrates the wrong lock fault that is threaded obtain different locks. Listing 25's class has two methods that are declared in lines 15 and 25

that both of them consist of synchronised block and lock on different synchronisation objects.

Listing 25. Provided class by SIR to evaluate the SOAWSB metric

```
 9  public class WrongLock {
10        Data data;
 ⋮        ⋮
15        public void A(){
16              synchronized (data) {
 ⋮              ⋮
23        }
24
25        public void B() {
26              synchronized (this) {
 ⋮              ⋮
29        }
30  }
```

6.11.2 Result

The figure 38 represents the output of running static analysis tool to analyse listing 25 class.

Figure 38. Result of the SOAWSB metric evaluation

```
-> 'A' method has synchronized block that lock on ''field''
-> 'B' method has synchronized block that lock on [this]


---------------------------------------------------------------------
The total number of method/s                    :   0
Method/s without synchronized block/s           :   0
->> Method/s consist of synchronized block/s    :  '2'
->> The total number of synchronized block/s    :   2
-->> Lock/s on (Class object)                   :  (0)
-->> Lock/s on [this]                           :  [1]
-->> Lock/s on {Lock object}                    :  {0}
-->> Lock/s on ''field''                        : ''1''
```

6.11.3 The verification of result and findings

The result is as expected:

I. **Name of method consist of synchronised blocks and their associated synchronised object:** Methods consist of a synchronised block, their names, and their associated synchronisation object are identified correctly (see lines 15, 16, 25, and 26 of listing 25).

II. **The total number of methods:** Is equal to the declared method (see line 15 and 25 of listing 25)

III. **Methods without synchronised blocks:** This class has no method with this specification.

VII. **Methods consist of synchronised blocks:** Is equal to the number of listed methods presented in (I.).

IV. **The total number of synchronised blocks:** Is equal to the number of presented lock listed in (I.).

V. **Locks on (Class object):** Zero in this class.

VI. **Locks on [this]:** see line 26 of listing 25

VII. **Locks on {Lock object}:** Zero in this class.

VIII. **Locks on ''field'':** see line 16 of listing 25

The result shows that this class only has two methods that both of them contain a synchronised block. Moreover, each of them is associated with different synchronisation object (this and field). As discussed in the previous experiment, with a small number of synchronisation units (block in this case) using the same lock can be appropriate most of the time. This idea concerning the subjected class is supported by SIR statement about the existence of the wrong fault in this programme.

The evidence suggests that this programme has a low level of code quality and need to reconsider developing its quality.

6.12 Metric 11 - Synchronised Blocks inside Synchronized Methods (SBISM)

6.12.1 Description

Listing 26 shows a class of 'accountsubtype' programme. SIR states that this programme demonstrates parallel concurrency faults (deadlock and race). This class has only one method that is declared in line 6 and consists of a synchronised block in line 8.

Listing 26. Provided class by SIR to evaluate the SBISM metric

```
 1  public class BusinessAccount extends Account {
⋮        ⋮
 6       public synchronized void transfer(…){
 7            …
 8            Synchronized (dest) {
⋮        ⋮
10  }
```

6.12.2 Result

The figure 39 represents the output of running a static analysis tool to analyse listing 26 class.

Figure 39. Provided class by SIR to evaluate the SBISM metric

```
-> 'transfer' is synchronized method and has [1] synchronized block!


----------------------------------------------------------------------
The total number of sync method/s                        : '1'
The total number of sync block/s inside sync method/s    : [1]
```

6.12.3 The verification of result and findings

The result is as expected:

I. **Name of synchronised method consist of synchronised:** Synchronised method consists of a synchronised block, its name, and the number of synchronised block in it are identified correctly (see lines 6 to 8 of listing 26).

II. **The total number of synchronised methods:** Is equal to the number of listed method presented in (I.).

III. **The total number of synchronised blocks inside sync methods:** Is equal to the number of presented synchronised block inside relevant method listed in (I.).

The result shows this class only have one synchronised method, and this method consists of synchronised block. As discussed in SBISM section of chapter four, this situation can represent the code has error. SIR statements about deadlock and race concurrency errors that this programme demonstrate reinforce this view. The evidence shows this code has low code quality and need to be redesigned for rectifying its errors and enhancing its quality.

7 Chapter 7: Recommendation and conclusion

7.1 Project objectives: Summary of findings and conclusions

This section will revisit the objectives of the project to draw a conclusion. Within the context of the MSc dissertation, three objectives were defined for this project. A valuable aspect of this study relates to objective two – suggesting code quality metrics – that can be practically utilised in concurrent concept. Attaining this objective were reflected in the Literature Review findings. Objective three took this research one step further through comparing theory with practice – that is, evaluating suggested code quality metrics by the static code analysis tool.

Drawn conclusions are divided into objectives stated at the start of this work to ensure if they have been met.

7.1.1 Project objective 1: Broaden the understanding of critical concepts of subjected area

The literature identified a professionally developed piece of code is expected to present the superior quality that is having few failures to be efficient for outstanding performance. However, evaluating code quality is an exceptionally complicated procedure. It is because any universal definition of quality cannot be provided, and it can be highly variable depending on the interesting context. It seems code quality metrics as units of quality measurement can facilitate this procedure by extracting valuable information based on analysing the source codes.

One of the main conclusion that can be drawn from this study can be an approach to manage quality consist of the following steps:

a. Choosing the area of interest (or project).
b. Defining critical importance of it (Defining quality in a particular concept (or project) with considering basic standards and particular needs of this concept (or project)).
c. Developing and applying quality metrics as measures to evaluate quality.
d. Taking steps to manage and enhance the quality based on the result of the evaluation.

In the context of concurrent programming, the conclusion appears to be extra effort is needed to have a correct concurrent code. Since any error are not given at compile-time, it is complicated to make sure that concurrent source code is made thread-safe and have superior quality. More critical, employing the optimum technique of synchronisation based on the problem that needs to address helps to reduce the performance cost and have a cleaner approach, which can lead to enhance the code quality. To achieve this aim, a deep understanding of the problem that needs to address and properties associated with each synchronisation techniques are decisive.

7.1.2 Project objective 2: Suggesting code quality metrics utilised for concurrent software

The optimum quality metrics are those that give useful information about source codes, and they can inspire codes in an interesting context. A conclusion that can be drawn from this study is that the primary step to evaluate source code is to define the basic units of measurement in a concurrent context such as the synchronised method and block. These units make conducting further analysis possible and can derive other metrics.

More important, according to synchronisation standards and principals, suggested code quality metrics are only appropriate to evaluate Java classes in isolation. It implies that they can have a completely different meaning if a class are considered as part of an entire programme.

7.1.3 Project objective 3: Evaluating suggested metrics

The conclusion that can be drawn on static analysis tool is that without a doubt developing an analysis tool as generic is extremely decisive to perform accurate and reliable analysis. Furthermore, a tool needs to implement appropriate metrics to function effectively and efficiently.

Regarding suggested metrics, the conclusion appears to be that they are accurate to the extent that the structural and underlying principles of synchronisation are considered in the interpretation of extracted information. Since there is no awareness about the logic of programme, proper evaluation of the chosen synchronisation technique cannot be possible. Moreover, appraising a class in isolation from the entire programme can be vague.

7.2 The challenges

This section raises the challenge of this work and illustrates the approaches were used to address them for providing a real insight into it.

The result of conducting these experiments cannot be generalised to the broader study area. It is because although the general standards and principals are considered through this study, the experiments could not perform on ample source codes. Since this study is an MSc project, and there is time limitation, a tried and tested research strategy is applied to conduct a valid in-depth study instead of generalised one.

Moreover, as discussed in source code sampling section of chapter three, it is attempted to choose representative source codes for performing an experimental evaluation to improve the reliability of the study, although it was tough to find classes that represent the intended quality to highlight the value of extracted information by each metric.

In term of depending on the analysis tool to evaluate suggested metrics, at first, there were considerable difficulties in developing a tool that can analyse all patterns of source codes. They finally were dealt with by using a generic library that was expounded in JavaParser subsection of chapter five.

Nonetheless, this study is expected to be valid and reliable by adopting the aforementioned procedures to minimise the effect of limitations and problems.

7.3 Recommendations

The review of the literature made it clear that evaluating code quality is a comprehensive concept. This study took a primary step to evaluate the code quality in the concurrent context. However, the lack of work in this subject area was recognised. In that respect, this study offers to:

1. classify the concurrent concept by synchronisation techniques as subject areas
2. identify code quality metrics for each subject area separately (by broadening the understanding about that subject in great depth)
3. combine different metrics for analysing source code to gain a comprehensive insight into its quality

8 References

April, A. & Laporte, C.Y. (2018) *Software Quality Assurance*. Hoboken, Wiley-IEEE Computer Society, Inc. Available from: https://ieeexplore.ieee.org/servlet/opac?bknumber=8268017 [Accessed 9th June 2019]

Bhatia,S. & Malhotra, J. (2014) A survey on impact of lines of code on software complexity. In: *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)*. Unnao, IEEE. Available from: https://doi.org/10.1109/ICAETR.2014.7012875

Biggam, J. (2015) *Succeeding with Your Master's Dissertation – A step-by-step handbook*. 3rd edition. Berkshire, Open University Press.

Bigonha, M.A., Ferreira, K.A. & Filó, T.G. (2015) A Catalogue of Thresholds for Object-Oriented Software Metrics. In: *The First International Conference on Advances and Trends in Software Engineering*. Barcelona, IARIA. pp. 48-55

Boehm, B., Deeds-Rubin, S., Nguyen, V. & Tan, T (2007) *A SLOC Counting Standard*. Available from: http://sunset.usc.edu/csse/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf [Accessed 28th May 2019]

Bourque, P. ed. & Fairley, R.E. ed. (2014) *The Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*. Version 3.0. IEEE Computer Society. Available from: https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3 [Accessed 25th January 2019]

Bruggen, D.V., Smith, N. & Tomassetti, F. (2019) *JavaParser: Visited - Analyse, transform and generate your Java code base*. Leanpub. Available from: https://leanpub.com/javaparservisited [Accessed 29th May 2019]

Chidamber, S.R. & Kemerer, C.F. (1991) Towards a metrics suite for object oriented design. In: Paepcke, A. (ed.): SPLASHSystems, Programming, and Applications: *OOPSLA '91 Conference proceedings on Object-oriented programming systems, languages, and applications*. New York, ACM. pp. 197-211

Chidamber, S.R. & Kemerer, C.F. (1994) A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 20 (6), 476 - 493. Available from: https://doi.org/10.1109/32.295895

DeMarco, T. (1982) *Controlling Software Projects: Management, Measurement, and Estimates*. US, Pearson Education.

Do, H., Elbau, S. & Rothermel, G. (2005) Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*. 10 (4), 405–435. Available from: https://link.springer.com/article/10.1007/s10664-005-3861-2 [Accessed 14th June 2019]

JavaParser (2019) *Java 1-12 Parser and Abstract Syntax Tree for Java*. Available from: https://github.com/javaparser/javaparser [Accessed 29th May 2019]

JavaParser (2019) *javaparser-core 3.14.11 API*. Available from: https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.14.11 [Accessed 29th May 2019]

JavaParser (2019) *JavaParser for Processing Java Code.* Available from: https://javaparser.org [Accessed 29th May 2019]

Jeffery, R. & Trendowicz, A. (2014) *Software Project Effort Estimation - Foundations and Best Practice Guidelines for Success*. Switzerland, Springer International. Available from: https://doi.org/10.1007/978-3-319-03629-8

Juran, J.M. (2010) *Juran's quality handbook: the complete guide to performance excellence*. 6th edition. New York, McGraw Hill. Available from: https://www.dawsonera.com/readonline/9780071629720 [Accessed 9th June 2019]

Friesen, J. (2015) *Java Threads and the Concurrency Utilities*. Berkeley, Apress. Available from: https://link-springer-com.proxy.lib.strath.ac.uk/book/10.1007%2F978-1-4842-1700-9#toc [Accessed 21st June 2019]

Gagne, G., Galvin, P.B. & Silberschatz, A. (2010) *Operating Systems Concepts with Java*. 8th edition. The United States, John Wiley & Sons, Inc.

Galin, D (2018) Software Process Quality Metrics. In: Galin, D (2018) *Software Quality: Concepts and Practice*. Hoboken, John Wiley & Sons, Inc. Available from: https://ieeexplore.ieee.org/document/8343650 [Accessed 27th May 2019]

Goetz, B., Bloch, J., Bowbeer, J., Holmes, D., Lea, D. & Peierls, T. (2006) *Java Concurrency in Practice*. Massachusetts, Pearson Education.

IEEE (2014) IEEE 730-2014. *IEEE Standard for Software Quality Assurance Processes*. New York, the Institute of Electrical and Electronics Engineers.

Institute of Electrical and Electronics Engineers (IEEE) (1993) IEEE Std. 1045-1992. *IEEE Standard for Software Productivity Metrics*. USA, IEEE.

ISO/IEC/IEEE (2017) BS ISO/IEC/IEEE 24765:2017. *Systems and software engineering -- Vocabulary*. London, BSI Standards Limited.

Microsoft (2018) *Code metrics values*. Available from: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019 [Accessed 27th May 2019]

Olsson, M. (2018) *Java Quick Syntax Reference*. Second edition. Berkeley, Apress. Available from: https://link-springer-com.proxy.lib.strath.ac.uk/book/10.1007%2F978-1-4842-3441-9 [Accessed 21st June 2019]

Oracle. (2017) *The Java™ Tutorials: Atomic Access*. Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html [Accessed 4th June 2019].

Oracle. (2017) *The Java™ Tutorials: Deadlock*. Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html [Accessed 7th June 2019].

Oracle. (2017) *The Java™ Tutorials: Intrinsic Locks and Synchronization*. Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html [Accessed 3th June 2019]

Oracle. (2017) *The Java™ Tutorials: Memory Consistency Errors.* Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html [Accessed 3th June 2019]

Oracle. (2017) *The Java™ Tutorials: Processes and Threads*. Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html [Accessed 7th June 2019].

Oracle. (2017) *The Java™ Tutorials: Synchronization.* Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html [Accessed 25th May 2019]

Oracle. (2017) *The Java™ Tutorials: Synchronized Methods.* Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html [Accessed 25th May 2019].

Oracle. (2017) *The Java™ Tutorials: Variables*. Available from: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html [Accessed 6th June 2019]

Robert, P. (1992) *Software Size Measurement: A Framework for Counting Source Statements*. Pittsburgh, Software Engineering Institute (SEI). Available from: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11689 [Accessed 3rd June 2019]

University of Strathclyde. (2018) *PGT Dissertation Handbook - Academic Year 2018/2019*. Glasgow, Department of Computer and Information Sciences. Available from: https://local.cis.strath.ac.uk/wp/wp-content/uploads/2018-19_Dissertation_Gudielines_V1.pdf [Accessed 8th June 2019]

9 Appendices

9.1 Appendix A: Analysis Tool

9.1.1 Appendix A.1: Metric 1 – Synchronised Methods in Class (SMIC)

```
public class MethodSync {

	public static void main(String[] args) throws FileNotFoundException {
		final File file = new
File("src/main/java/com/sampleClasses/Account.java");
		final CompilationUnit cu = StaticJavaParser.parse(file);

		final List<MethodDeclaration> methodList =
cu.findAll(MethodDeclaration.class);
		int syncMethod = 0;

		for(MethodDeclaration method : methodList) {
			if(method.isSynchronized()) {
				syncMethod ++;
				System.out.println("-> '" + method.getName() + "' is
synchronized method");
			}
		}
		System.out.println("\n");
		if (methodList.size() == 0) {
			System.out.println("This class has 'no' method.");
			System.out.println("\n");
		}
		else if (syncMethod == 0) {
				System.out.println("This class has 'no' synchronized
method.");
				System.out.println("\n");
		}
		System.out.println("----------------------------------------------------
---------------");
		System.out.println("The total number of method/s   :  "
						+ methodList.size());
		System.out.println("Non-synchronized method/s      :  "
						+ (methodList.size() - syncMethod));
		System.out.println("->> Synchronized method/s      : '"
						+ syncMethod + "'");
	}

}
```

9.1.2 Appendix A.2: Metric 2 – Synchronised Methods Line of Code (SMLOC)

```
public class StmtMethodSync {

	public static void main(String[] args) throws FileNotFoundException {
	 final File file = new
File("src/main/java/com/sampleClasses/AllocationVector.java");
	 final CompilationUnit cu = StaticJavaParser.parse(file);

	 final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
	 int syncMethod = 0;
	 int totalNonSyncMethodStmt = 0;
	 int totalSyncMethodStmt = 0;

	 for (MethodDeclaration method : methodList) {
			if(method.isSynchronized()) {
				syncMethod += 1;
				List<Statement> methodStmt = method.getBody()

	.map(blockStmt -> blockStmt

	.getStatements())

	.get();
				totalSyncMethodStmt += methodStmt.size();
				if (methodStmt.size() == 0) {
					System.out.println("-> '" + method.getName() +
								"' is
synchronized method with [no] statement.");
				}
				else if (methodStmt.size() == 1) {
					System.out.println("-> '" + method.getName() +
								"' is
synchronized method with [1] statement.");
				}
				else {
					System.out.println("-> '" + method.getName() + "' is
synchronized method with ["
								+
methodStmt.size() + "] statements.");
				}
			}
```

```
                        else {
                                List<Statement> nonSyncMethodStmt = method.getBody()

                        .map(blockStmt -> blockStmt

                        .getStatements())

                        .get();
                                totalNonSyncMethodStmt += nonSyncMethodStmt.size();
                        }
                }
        System.out.println("\n");
                if (methodList.size() == 0) {
                        System.out.println("This class has 'no' method.");
                        System.out.println("\n");
                }
                else if (syncMethod == 0) {
                System.out.println("This class has 'no' synchronized method.");
                System.out.println("\n");
                }
                System.out.println("----------------------------------------------------
----------------------");
                System.out.println("The total number of method/s
:  "
                                                + methodList.size());
                System.out.println("Non-synchronized method/s
:  "
                                                + (methodList.size() - syncMethod));
                System.out.println("The total number of statements inside non-synchronized
method/s     :  "
                                                + totalNonSyncMethodStmt);
                System.out.println("->> Synchronized method/s
: '"
                                                + syncMethod + "'");
                System.out.println("->> The total number of statements inside synchronized
method/s     : ["
                                                + totalSyncMethodStmt + "]");

        }

}
```

9.1.3 Appendix A.3: Metric 3 – Synchronised Blocks in Class (SBIC)

```
public class BlockSync {

        public static void main(String[] args) throws FileNotFoundException {
         final File file = new File("src/main/java/com/sampleClasses/BarberShop.java");
         final CompilationUnit cu = StaticJavaParser.parse(file);

         final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
         int methodHasSyncBlock = 0;
               int syncBlock = 0;

         for (MethodDeclaration method : methodList) {
             final long count = synchronizedBlockCount(method);
             if (count != 0) {
                methodHasSyncBlock ++ ;
                 syncBlock += count;
                 if (count == 1) {
                       System.out.println("-> '" + method.getName()
                                                          + "' method has [1]
synchronized block");
                 }
                 else {
                       System.out.println("-> '" + method.getName()
                                                          + "' method has [" + count +
"] synchronized blocks");
                 }
             }
         }
         System.out.println("\n");
         if (methodList.size() == 0) {
               System.out.println("This class has 'no' method.");
               System.out.println("\n");
         }
         else if (syncBlock == 0) {
               System.out.println("This class has 'no' synchronized block.");
               System.out.println("\n");
         }
         System.out.println("----------------------------------------------------------------
---------");
         System.out.println("The total number of method/s                    :   "
                                                 + methodList.size());
         System.out.println("->> Method/s consist of synchronized block/s     :  '"
```

```
                                                + methodHasSyncBlock + "'");
        System.out.println("->> The total number of synchronized block/s     :  ["
                                            + syncBlock + "]");
    }

    private static long synchronizedBlockCount(MethodDeclaration method) {
        final Optional<BlockStmt> methodBody = method.getBody();
        return methodBody.map(blockStmt -> blockStmt.getStatements()
                .stream()
                .filter(Objects::nonNull)
                .filter(Statement::isSynchronizedStmt)
                .count()).orElse(0L);
    }

}
```

9.1.4 Appendix A.4: Metric 4 – Nested Synchronised Block in Class (NSBIC)

```
public class NestedSync {

    public static void main(String[] args) throws FileNotFoundException {
        final File file = new File("src/main/java/com/sampleClasses/DeadlockDemo.java");
        final CompilationUnit cu = StaticJavaParser.parse(file);

        final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
        int methodHasNestedSyncBlock = 0;
        int syncNestedBlock = 0;

        for (MethodDeclaration method : methodList) {
            if (haveNestedSynchronizedBlock(method)) {
               methodHasNestedSyncBlock ++;
                 syncNestedBlock++;
                 System.out.println("-> '" + method.getName() + "' has nested
synchronized block");
            }
        }
        System.out.println("\n");
        if (methodList.size() == 0) {
               System.out.println("This class has 'no' method.");
               System.out.println("\n");
        }
        else if (syncNestedBlock == 0) {
               System.out.println("This class has 'no' nested synchronized block.");
```

```
                System.out.println("\n");
            }
            System.out.println("-----------------------------------------------------------
---------");
            System.out.println("The total number of method/s
:   "
                                               + methodList.size());
            System.out.println("The number of method/s consist of nested synchronized
block/s   :  '"
                                               + methodHasNestedSyncBlock + "'");
        }


        private static boolean haveNestedSynchronizedBlock(MethodDeclaration method) {
            final AtomicBoolean haveNestedSynchBlock = new AtomicBoolean(false);

            method.getBody().map(blockStmt -> {
                final NodeList<Statement> statements = blockStmt.getStatements();
                for (Statement statement : statements) {
                    statement.ifSynchronizedStmt(synchronizedStmt -> {
                        if (isSynchronized(synchronizedStmt.getBody())) {
                          haveNestedSynchBlock.set(true);
                        }
                    });
                }
                return haveNestedSynchBlock;
            });
            return haveNestedSynchBlock.get();
        }


        private static boolean isSynchronized(BlockStmt blockStmt) {
            final AtomicBoolean isSynch = new AtomicBoolean(false);
            final NodeList<Statement> statements = blockStmt.getStatements();

            for (Statement statement : statements) {
                statement.ifSynchronizedStmt(synchronizedStmt -> {
                  isSynch.set(true);
                });
            }
            return isSynch.get();
        }
}
```

9.1.5 Appendix A.5: Metric 5 – Synchronised Blocks Line of Code (SBLOC)

```
public class StmtBlockSync {

    public static void main(String[] args) throws FileNotFoundException {
        final File sFile = new
File("C:/Users/Nadia/Downloads/Compressed/log4j3/LoadXMLAction.java");
        final CompilationUnit cu = StaticJavaParser.parse(sFile);

        final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
        int methodHasSyncBlock = 0;
        int syncBlock = 0;
        int totalSyncStmt = 0;

        for (MethodDeclaration method : methodList) {
            final List<Integer> countArrays = countSynchronizedBlockLine(method);
            if (!countArrays.isEmpty()) {
              methodHasSyncBlock++;
              for (Integer syncStms : countArrays) {
                     totalSyncStmt += syncStms;
              }
              syncBlock += countArrays.size();
              if (countArrays.size() == 1) {
                     System.out.println("-> '" + method.getName()
                                                          + "' method has (1)
synchronized block with "
                                                          + countArrays.toString() + "
statement/s.");
              }
              else {
                     System.out.println("-> '" + method.getName()
                                                          + "' method has (" +
countArrays.size()
                                                          + ") synchronized blocks with
"
                                                          + countArrays.toString() + "
statement/s respectively.");
              }
            }
        }
        System.out.println("\n");
        if (methodList.size() == 0) {
```

```
                        System.out.println(" This class has 'no' method.");
                        System.out.println("\n");
                }
        else if (methodHasSyncBlock == 0) {
                System.out.println(" This class has 'no' synchronized block.");
                System.out.println("\n");
        }
        else {

        }
        System.out.println("------------------------------------------------------------
--------------------");
        System.out.println("The total number of method/s
:  "
                                            + methodList.size());
                System.out.println("->> Method/s consist of synchronized block/s
: '"
                                            + methodHasSyncBlock + "'");
                System.out.println("->> The total number of synchronized block/s
: ("
                                            + syncBlock + ")");
                System.out.println("-->> The total number of statements inside
synchronized block/s     : ["
                                            + totalSyncStmt + "]");
    }


    private static List<Integer> countSynchronizedBlockLine(MethodDeclaration method) {
        final List<Integer> results = new ArrayList<>();

        final BlockStmt blockStmt = method.getBody().get();
        final NodeList<Statement> statements = blockStmt.getStatements();
        for (Statement statement : statements) {
            if (!statement.isSynchronizedStmt()) {
                continue;
            }

results.add(statement.findFirst(BlockStmt.class).get().getChildNodes().size());
        }
        return results;
    }

}
```

9.1.6 Appendix A.6: Metric 6 – Compare Synchronised Line of Code in Class (CSLOCIC)

```
public class CompareStmt {

	public static void main(String[] args) throws FileNotFoundException {
	 final File file = new
File("C:/Users/Nadia/Downloads/Compressed/log4j3/AsyncAppender.java");
	 final CompilationUnit cu = StaticJavaParser.parse(file);

	 final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
	 int syncMethod = 0;
	 int nonSyncMethod = 0;
	 int methodHasSyncBlock = 0;
	 int wrongSync = 0;

	 for (MethodDeclaration method : methodList) {
		final List<Integer> countArrays = countSynchronizedBlockLine(method);
			if(!method.isSynchronized()) {
				nonSyncMethod ++;

				if (!countArrays.isEmpty()) {
			methodHasSyncBlock ++;
			List<Statement> methodStmt = method.getBody()

		.map(blockStmt -> blockStmt

		.getStatements())

		.get();
			if (countArrays.size() == 1) {
				int remStmt = methodStmt.size() - countArrays.size();
				System.out.println("-> '" + method.getName()
								+ "' is non-
synchronized method with (1) synchronized block with "
								+
countArrays.toString() + " statement/s.");
				System.out.println("--> It has [" + remStmt
								+ "] statement/s except
its synchronized block/s.");
				System.out.println("\n");
				if (remStmt == 0) {
					wrongSync ++;
```

```
                        }
                    }
                    else {
                        int remStmt = methodStmt.size() - countArrays.size();
                        System.out.println("-> '" + method.getName()
                                                                + "' is non-
synchronized method with (" + countArrays.size()
                                                                + ") synchronized
blocks with "
                                                                +
countArrays.toString() + " statement/s respectively.");
                        System.out.println("--> It has [" + remStmt
                                                + "] statement/s except its
synchronized block/s.");
                        System.out.println("\n");
                        if (remStmt == 0) {
                            wrongSync ++;
                        }
                    }
                }
                    }
                    else {
                        syncMethod ++;
                    }
            }
        System.out.println("\n");
            if (methodList.size() == 0) {
                System.out.println("This class has 'no' method.");
                System.out.println("\n");
            }
            else if (methodHasSyncBlock == 0) {
            System.out.println(" This class has 'no' synchronized block inside non-
synchronized method/s.");
            System.out.println("\n");
        }
            System.out.println("-----------------------------------------------------
----------------------");
            System.out.println("The total number of method/s
:  "
                                                + methodList.size());
            System.out.println("Synchronized method/s
:  "
                                                + syncMethod);
```

```
                System.out.println("Non-synchronized method/s without synchronized block
:  "
                                                        + (nonSyncMethod -
methodHasSyncBlock));
                System.out.println("->> Non-synchronized method/s consist of synchronized
block/s       : '"
                                                        + methodHasSyncBlock + "'");
                System.out.println("-->> Method/s consist of [0] statement except
synchronized block/s  : ["
                                                        + wrongSync + "]");
        }


        private static List<Integer> countSynchronizedBlockLine(MethodDeclaration method)
{
         final List<Integer> results = new ArrayList<>();

         final BlockStmt blockStmt = method.getBody().get();
         final NodeList<Statement> statements = blockStmt.getStatements();
         for (Statement statement : statements) {
             if (!statement.isSynchronizedStmt()) {
                 continue;
             }

results.add(statement.findFirst(BlockStmt.class).get().getChildNodes().size());
         }
         return results;
     }

}
```

9.1.7 Appendix A.7: Metric 7 – Static Variables in Class (SVIC)

```
public class StaticVariable {

        public static void main(String[] args) throws FileNotFoundException {
                final File file = new
File("C:/Users/Nadia/Downloads/Compressed/elevator/Elevator.java");
                final CompilationUnit cu = StaticJavaParser.parse(file);
```

```
		final List<FieldDeclaration> fieldDeclarations =
cu.findAll(FieldDeclaration.class);
		int staticVar = 0;
		int StaticFinalVar = 0;

		for (FieldDeclaration field : fieldDeclarations) {
			if (field.isStatic()) {
				final String name =
field.getVariables().get(0).getName().asString();
				staticVar ++;
				if (field.isFinal()) {
					StaticFinalVar ++;
					System.out.println("-> '" + name + "'  variable is
[static final]");
				} else {
					System.out.println("-> '" + name + "'  variable is
(static non-final)");
				}
			}
		}
		System.out.println("\n");
	 if (fieldDeclarations.size() == 0) {
		System.out.println("This class has 'no' variable.");
		System.out.println("\n");
	 }
	 System.out.println("----------------------------------------------------------------
---------");
	System.out.println("The total number of variables		:  "
					+ fieldDeclarations.size());
	System.out.println("Non-static variable			:  "
					+ (fieldDeclarations.size() - staticVar));
	System.out.println("->> The total number of static variable/s	: '"
				+ staticVar + "'");
	System.out.println("--->> Static non-final variable		: ("
					+ (staticVar - StaticFinalVar) + ")");
	System.out.println("--->> Static final variable		: ["
					+ StaticFinalVar + "]");
	}

}
```

9.1.8 Appendix A.8: Metric 8 – Volatile Variables in Class (VVIC)

```
public class VolatileVariable {

    public static void main(String[] args) throws FileNotFoundException {
        final File file = new
File("C:/Users/Nadia/Downloads/Compressed/elevator/Person.java");
        final CompilationUnit cu = StaticJavaParser.parse(file);

        final List<FieldDeclaration> fieldDeclarations =
cu.findAll(FieldDeclaration.class);
        int volVar = 0;

        for (FieldDeclaration fieldDeclaration : fieldDeclarations) {
            if (fieldDeclaration.isVolatile()) {
                System.out.println("-> '" + fieldDeclaration.getVariables()
                                                    .get(0).getNameAsString() + "' is
volatile variable");
                volVar++;
            }
        }
        System.out.println("\n");
        if (fieldDeclarations.size() == 0) {
              System.out.println("This class has 'no' variable.");
              System.out.println("\n");
        }
        System.out.println("-----------------------------------------------------------
---------");
        System.out.println("The total number of variable/s    :  "
                                                    + fieldDeclarations.size());
              System.out.println("Non-volatile variable/s           :  "
                                                    + (fieldDeclarations.size() -
volVar));
              System.out.println("->> Volatile variable/s           : '"
                                                    + volVar + "'");
    }
}
```

9.1.9 Appendix A.9: Metric 9 – Synchronization Objects Associated with Synchronised Methods (SOAWSM)

```
public class MethodSyncObject {

	public static void main(String[] args) throws FileNotFoundException {
		final File file = new
File("C:/Users/Nadia/Downloads/Compressed/log4j3/DateFormatManager.java");
		final CompilationUnit cu = StaticJavaParser.parse(file);

		final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
		int syncMethod = 0;
		int classLock = 0;
		int thisLock = 0;

		for(MethodDeclaration method : methodList) {
			if(method.isStatic() && method.isSynchronized()) {
				classLock ++;
				System.out.println("-> '" + method.getName() + "' method
lock on (class object) itself");
			}
			else if (!method.isStatic() && method.isSynchronized()) {
				thisLock ++;
				System.out.println("-> '" + method.getName() + "' method
lock on [this]");
			}
		}
		syncMethod = classLock + thisLock;
		System.out.println("\n");
		if (methodList.size() == 0) {
			System.out.println("This class has 'no' method.");
			System.out.println("\n");
		}
		else if (syncMethod == 0) {
			System.out.println("This class has 'no' synchronized method to lock
on any object.");
			System.out.println("\n");
		}
		System.out.println("-----------------------------------------------------
---------------");
		System.out.println("The total number of method/s                    :  "
                                        + methodList.size());
		System.out.println("Non-synchronized method/s                       :  "
                                        + (methodList.size() - syncMethod));
		System.out.println("->> The total number of synchronized method/s   : '"
                                        + syncMethod + "'");
		System.out.println("-->> Lock/s on (Class object)                   : ("
```

```
                                                + classLock + ")");
                System.out.println("-->> Lock/s on [this]                              : ["
                                                + thisLock + "]");
        }
}
```

9.1.10 Appendix A.10: Metric 10 – Synchronisation Objects Associated with Synchronised Blocks (SOAWSB)

```
public class BlockSyncObject {

        public static void main(String[] args) throws Exception{
         final File file = new
File("C:/Users/Nadia/Downloads/Compressed/wrongLock/wrongLock.java");
         final CompilationUnit cu = StaticJavaParser.parse(file);

         final List<MethodDeclaration> methodList = cu.findAll(MethodDeclaration.class);
         ArrayList<String> methodHasSyncBlock = new ArrayList<String>();
         int syncBlock = 0;
         int classLock = 0;
         int thisLock = 0;
         int lockLock = 0;
         int fieldLock = 0;

         for (MethodDeclaration method : methodList) {
             if (haveClassSynchronizedBlock(method)) {
                classLock ++;
                methodHasSyncBlock.add(method.getName().asString());
                 System.out.println("-> '" + method.getName() +
                                                "' method has synchronized block that
lock on (Class object)");
             }

             if (haveThisSynchronizedBlock(method, cu)) {
                thisLock ++;
                methodHasSyncBlock.add(method.getName().asString());
                 System.out.println("-> '" + method.getName() +
                                                "' method has synchronized block that
lock on [this]");
             }
```

```
            if (haveLockFieldSynchronizedBlock(method, cu)) {
               lockLock ++;
               methodHasSyncBlock.add(method.getName().asString());
                System.out.println("-> '" + method.getName() +
                                                   "' method has synchronized block that
lock on {Lock object}");
            }

            if (haveFieldSynchronizedBlock(method, cu)) {
               fieldLock ++;
               methodHasSyncBlock.add(method.getName().asString());
                System.out.println("-> '" + method.getName() +
                                                   "' method has synchronized block that
lock on ''field''");
            }
        }

        ArrayList<String> cleanMethodHasSyncBlock = new ArrayList<String>();
        cleanMethodHasSyncBlock = removeDuplicates(methodHasSyncBlock);

        syncBlock = classLock + thisLock + lockLock + fieldLock;

        System.out.println("\n");
        if (methodList.size() == 0) {
              System.out.println("This class has no method to analyse its blocks.");
              System.out.println("\n");
        }
        else if (syncBlock == 0) {
              System.out.println("This class has no synchronized block.");
              System.out.println("\n");
        }
        System.out.println("-----------------------------------------------------------
---------");
        System.out.println("The total number of method/s                    :   "
                                                   + methodList.size());
              System.out.println("Method/s without synchronized block/s          :   "
                                                   + (methodList.size() -
cleanMethodHasSyncBlock.size()));
              System.out.println("->> Method/s consist of synchronized block/s   :  '"
                                                   + cleanMethodHasSyncBlock.size() +
"'");
              System.out.println("->> The total number of synchronized block/s   :   "
                                                   + syncBlock);
              System.out.println("-->> Lock/s on (Class object)                  :  ("
```

```
                                            + classLock + ")");
            System.out.println("-->> Lock/s on [this]                          :  ["
                                            + thisLock + "]");
            System.out.println("-->> Lock/s on {Lock object}                   :  {"
                                            + lockLock + "}");
            System.out.println("-->> Lock/s on ''field''                       : ''"
                                            + fieldLock + "''");
    }

       private static boolean haveClassSynchronizedBlock(MethodDeclaration method) {
        final AtomicBoolean haveClassSynchronized = new AtomicBoolean(false);

        method.getBody().map(blockStmt -> {
            final NodeList<Statement> statements = blockStmt.getStatements();

            for (Statement statement : statements) {
                statement.ifSynchronizedStmt(synchronizedStmt -> {
                    final ExpressionMetaModel metaModel =
synchronizedStmt.getExpression().getMetaModel();
                    if (metaModel.getTypeName().equals("ClassExpr")) {
                        haveClassSynchronized.set(true);
                    }
                });
            }
            return haveClassSynchronized;
        });
        return haveClassSynchronized.get();
    }


       private static boolean haveThisSynchronizedBlock(MethodDeclaration method,
CompilationUnit compilationUnit) {
        final AtomicBoolean haveSynchronized = new AtomicBoolean(false);

        method.getBody().map(blockStmt -> {
            final NodeList<Statement> statements = blockStmt.getStatements();

            for (Statement statement : statements) {
                statement.ifSynchronizedStmt(synchronizedStmt -> {
                    final String fieldName =
synchronizedStmt.getExpression().toString();
                    if (fieldName.equals("this")) {
                        haveSynchronized.set(true);
                    }
```

```
                });
            }
            return haveSynchronized;
        });
        return haveSynchronized.get();
    }



    private static boolean haveLockFieldSynchronizedBlock(MethodDeclaration method,
CompilationUnit compilationUnit) {
        final AtomicBoolean haveFieldSynchronized = new AtomicBoolean(false);

        method.getBody().map(blockStmt -> {
            final NodeList<Statement> statements = blockStmt.getStatements();

            for (Statement statement : statements) {
                statement.ifSynchronizedStmt(synchronizedStmt -> {
                    final String fieldName =
synchronizedStmt.getExpression().toString();
                    final FieldDeclaration field = findField(fieldName,
compilationUnit);
                    if (field == null) {
                        return;
                    }
                    final String fieldDeclaration = field.toString().replace("private",
"").replace("final", "").trim();
                    if (fieldDeclaration.startsWith("Lock")) {
                        haveFieldSynchronized.set(true);
                    }
                });
            }
            return haveFieldSynchronized;
        });
        return haveFieldSynchronized.get();
    }



    private static boolean haveFieldSynchronizedBlock(MethodDeclaration method,
CompilationUnit compilationUnit) {
        final AtomicBoolean haveFieldSynchronized = new AtomicBoolean(false);

        method.getBody().map(blockStmt -> {
            final NodeList<Statement> statements = blockStmt.getStatements();
```

```
            for (Statement statement : statements) {
                statement.ifSynchronizedStmt(synchronizedStmt -> {
                    final String fieldName =
synchronizedStmt.getExpression().toString();
                    final FieldDeclaration field = findField(fieldName,
compilationUnit);
                    if (field == null) {
                        return;
                    }
                    final String fieldDeclaration = field.toString().replace("private",
"").replace("final", "").trim();
                    if (!fieldDeclaration.startsWith("Lock")) {
                        haveFieldSynchronized.set(true);
                    }
                });
            }
            return haveFieldSynchronized;
        });
        return haveFieldSynchronized.get();
    }


    private static FieldDeclaration findField(String name, CompilationUnit
compilationUnit) {
        return compilationUnit.findFirst(FieldDeclaration.class,
                fieldDeclaration -> fieldDeclaration.getVariables()
                        .stream()
                        .anyMatch(f -> f.getNameAsString().equals(name))
        ).orElse(null);
    }


/**
 *  Function to remove duplicates from an ArrayList
 * @param list
 * @return
 */

    private static ArrayList<String> removeDuplicates (ArrayList<String> list) {
        ArrayList<String> cleanList = new ArrayList<String>();
        for (String element : list) {
            if (!cleanList.contains(element)) {
               cleanList.add(element);
```

```
            }
        }
        return cleanList;
    }

}
```

9.1.11 Appendix A.11: Metric 11 - Synchronised Blocks inside Synchronized Methods (SBISM)

```
public class SyncBlockInSyncMethod {

        public static void main(String[] args) throws FileNotFoundException {
                final File file = new
File("C:/Users/Nadia/Downloads/Compressed/accountsubtype/BusinessAccount.java");
                final CompilationUnit cu = StaticJavaParser.parse(file);

                final List<MethodDeclaration> methodList =
cu.findAll(MethodDeclaration.class);
                int syncMethod = 0;
                int syncBlock = 0;

                for(MethodDeclaration method : methodList) {
                        final long count = synchronizedBlockCount(method);
                        if(method.isSynchronized()) {
                                syncMethod ++;
                                if (count != 0) {
                                        syncBlock += count;
                                        if (count == 1) {
                                                System.out.println("-> '" + method.getName()
                                                                                + "' is
synchronized method and has [1] synchronized block!");
                                        }
                                        else {
                                                System.out.println("-> '" + method.getName()
                                                                                + "' is
synchronized method and has [" + count + "] synchronized blocks!");
                                        }
                                }
                        }
                }
                System.out.println("\n");
```

```
		if (methodList.size() == 0) {
			System.out.println("This class has 'no' method.");
		}
		else if (syncMethod == 0) {
			System.out.println("This class has 'no' synchronized method.");
			System.out.println("\n");
		}
		else if (syncBlock == 0) {
			System.out.println("This class has 'no' synchronized block inside
its synchronized method/s.");
			System.out.println("\n");
		}
		System.out.println("----------------------------------------------------
-------------------");
		System.out.println("The total number of sync method/s
: '"
							+ syncMethod + "'");
		System.out.println("The total number of sync block/s inside sync method/s
: ["
							+ syncBlock + "]");
	}


	private static long synchronizedBlockCount(MethodDeclaration method) {
	 final Optional<BlockStmt> methodBody = method.getBody();
	 return methodBody.map(blockStmt -> blockStmt.getStatements()
			 .stream()
			 .filter(Objects::nonNull)
			 .filter(Statement::isSynchronizedStmt)
			 .count()).orElse(0L);
    }

}
```