



Framework for Secure Coding :

An algorithmic approach for real-time detection of secure coding guideline violations

Group members

S.L. Dasanayake : 14000156

A. Mudalige : 14000954

M.L.T. Perera : 14001144

Supervisor: Dr. Prasad Wimalaratne

Co-Supervisor: Mr. Chaman Wijesiriwardana



**University of Colombo School of Computing
Sri Lanka**

Submitted in partial fulfillment of the requirements of the
B.Sc(Hons) in Software Engineering 4th Year Project (SCS4123)

January 14, 2019

Declaration

We certify that this dissertation does not incorporate, without acknowledgment, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. We also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name:

.....
Signature of Candidate

Date:

Candidate Name:

.....
Signature of Candidate

Date:

Candidate Name:

.....
Signature of Candidate

Date:

This is to certify that this dissertation is based on the work of Mr. S.L.Dasanayake, Mr. A.Mudalige, and Mr. M.L.T.Perera under my supervision. The dissertation has been prepared according to the format stipulated and is of the acceptable standard.

Supervisor Name: Dr. Prasad Wimalaratne

.....
Signature of Supervisor

Date:

This is to certify that this dissertation is based on the work of Mr. S.L.Dasanayake, Mr. A.Mudalige, and Mr. M.L.T.Perera under my supervision. The dissertation has been prepared according to the format stipulated and is of the acceptable standard.

Co-Supervisor Name: Mr. Chaman Wijesiriwardana

.....
Signature of Co-Supervisor

Date:

Abstract

Secure Software Development refers to the process of developing software applications with minimised security vulnerabilities. In the release or maintenance phase of the Software Development Life Cycle(SDLC), fixing specific bugs is very expensive than correcting such issues during the coding or development phase. Therefore it is essential to minimise these software defects within the coding phase itself by adhering to a set of coding best practices that are referred as secure coding guidelines. Following of these guidelines has been a challenging and time-consuming task due to the lack of knowledge among developers regarding such guidelines and the fact that currently there exists only a manual mechanism of checking these guidelines using a checklist. This dissertation proposes a plugin-based framework for IntelliJ IDEA Integrated Development Environment that focuses on developing a mechanism to automate the process of detecting secure coding guideline violations found in the source code of a software application. The framework is based on the secure coding guidelines introduced by Software Engineering Institute Computer Emergency Response Team (SEI CERT) known as the SEI CERT secure coding rules. These secure coding rules include guidelines for avoiding coding and implementation errors, as well as low-level design errors.

In order to implement the secure coding rules, the rules were classified into three granularity levels namely Method, Class and Package level. A total of 15 secure coding rules, five from each granularity level have been implemented in this framework in the form of violation detection algorithms. The source code fragments associated with each violation detection algorithm are obtained via the Abstract Syntax Tree generated by the parser and are stored in data structures such as ArrayLists and HashMaps. Violation detection algorithms use these stored source code fragments to detect secure coding rule violations. A significant feature of this framework is the extensibility mechanism in which violation detection algorithms could be added with minimal effort during future development of the framework. Performance optimisation has also been achieved to minimise resource consumption and reduce latency, by improved system design with the support of software design patterns.

Apart from detecting secure coding rule violations in the source code, the framework will also provide the necessary countermeasures to overcome those violations. In addition, the framework could be used as a teaching tool for users who are unaware of the secure coding rules due to its features such as tooltips, tools windows, syntax highlighting. Using this framework, a software developer would be able to adhere to secure coding rules and ensure the security aspect of a software application. The secure coding plugin-based framework has been deployed to the JetBrains plugin repository enabling to be downloaded by the required users.

Acknowledgements

This product based Software Engineering project is the final result of continuous commitment and dedication among the members of our group with great support from various personnel that assisted us in numerous ways.

A very special gratitude goes to Dr. Prasad Wimalaratne, our main supervisor, for giving us the seed idea of the project and providing guidance and necessary support to complete this project in a successful manner. He had been a great motivator and an advisor for us in order to overcome major obstacles faced during this project.

We also appreciate the great support provided by Mr. Chaman Wijesiriwardana, our co-supervisor, lecturer at the University of Moratuwa for the assistance given to us related to depth theoretical and subject wise matter of this project.

Our special thanks are extended to the evaluating panel at the preliminary and interim defenses that provided us important feedback and showed the places of improvement of the project.

Finally, we would like to extend our deepest gratitude to our beloved parents in supporting in achieving goals in academic career and overcome unexpected barriers in our lives.

Table of Contents

Declaration	i
Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Figures	vii
List of Tables	viii
List of Acronyms	ix
Chapter 1 : Introduction	1
1.1 Motivation	1
1.2 Aims and objectives	2
1.3 Scope.....	2
1.4 Structure of the dissertation.....	3
Chapter 2 : Background Study	4
2.1 Introduction	4
2.2 Review of similar systems.....	6
2.3 Related work and limitations.....	8
2.3.1 Related work	8
2.3.2 Limitations of current approaches.....	9
2.4 Secure coding guidelines.....	10
2.4.1 Secure coding.....	10
2.4.2 Secure coding guidelines	11
2.4.3 SEI CERT Secure coding rules.....	13
2.5 Summary	15
Chapter 3 : Analysis and Design	16
3.1 Introduction	16
3.2 Problem analysis	16
3.3 Design assumptions and dependencies	17
3.4 Secure coding rules classification criteria	18
3.5 Product perspective.....	19
3.5.1 Dependencies	19

3.5.2 Design and Implementation constraints	20
3.6 System architecture design	21
3.7 Parser selection.....	23
3.7.1 Compilers	23
3.7.2 Parsers	24
3.7.3 Rationale for the need of a parser	24
3.7.4 Parser selection	24
3.7.5 JavaParser.....	26
3.8 System modeling	28
3.8.1 Class diagram.....	28
3.8.2 Design patterns utilized	29
3.9 Justification as a Framework.....	30
3.10 Summary	31
Chapter 4 : Implementation.....	32
4.1 Introduction	32
4.2 IntelliJ IDEA plugin development.....	32
4.3 Method level implementation.....	34
4.3.1 Introduction	34
4.3.2 Implementation procedure	35
4.4 Class level implementation	37
4.4.1 Introduction	37
4.4.2 Implementation procedure	38
4.5 Package level implementation	41
4.5.1 Introduction	41
4.5.2 Implementation procedure	42
4.6 Alignment of implementation with system design	44
4.7 Countermeasures for violated guidelines	47
4.8 Test procedure	48
4.9 Deployment	49
4.10 Summary	51
Chapter 5 : Evaluation and Results	52
5.1 Introduction	52
5.2 Project based evaluation.....	53

5.2.1 Introduction	53
5.2.2 Results of Project based evaluation.....	54
5.2.3 Conclusion.....	60
5.3 Extensibility based evaluation	62
5.3.1 Introduction	62
5.3.2 Addition of a new secure coding violation detection algorithm	62
5.3.3 Addition of a new source code granularity level	65
5.3.4 Modification of existing secure coding violation detection algorithms	66
5.3.5 Modification of existing data structures	67
5.4 Performance based evaluation	68
5.4.1 Introduction	68
5.4.2 Results of performance based evaluation	69
5.4.3 Benchmark tool comparison	74
5.5 User based evaluation	75
5.5.1 Introduction	75
5.5.2 Analysis of results.....	75
5.5.3 Conclusion.....	78
5.6 Summary	79
Chapter 6 : Conclusion.....	80
6.1 Future work	81
References.....	82
Appendix.....	85
Appendix A : Terminology	85
Appendix B : Classification of secure coding rules.....	86
Appendix C : Secure coding rules with algorithms and respective source code fragments	99
Appendix D : Other design artifacts	105
Appendix E : Violation detection	109
Appendix F : Evaluation results	111
Appendix G : Deployment results	117
Appendix H : Individual contribution.....	118
USER'S MANUAL.....	124

List of Figures

Figure 2.1: Costs of fixing bugs based on the phase of SDLC	4
Figure 2.2: Software security best practices applied to various software artifacts.....	5
Figure 2.3: Detection of bugs using SpotBugs plugin	6
Figure 2.4: Detection of a code quality issue using sonarLint plugin	7
Figure 2.5: Levels and Priority ranges.....	14
Figure 3.1: Product interaction with internal and external environment	20
Figure 3.2: High-level (tier) architecture	23
Figure 3.3: Class diagram	28
Figure 4.1: Detection of the method level secure coding rule violations	36
Figure 4.2: Countermeasures for the detected method level violations.....	36
Figure 4.3: Detection of the class level secure coding rule violations.....	40
Figure 4.4: Countermeasures for the detected class level violations	40
Figure 4.5: Detection of the package level secure coding rule violations	43
Figure 4.6: Countermeasures for the detected package level violations	43
Figure 4.7: Example countermeasures for the detected violations	47
Figure 4.8: Framework for secure coding plugin deployed in the JetBrains Plugin Repository	50
Figure 4.9: Plugin deployed in the JetBrains Plugin Repository.....	51
Figure 5.1: Project based evaluation procedure diagram.....	54
Figure 5.2: ERR08J violation as detected by the secure coding plugin.....	58
Figure 5.3: Multiple rule violations and same rule violation at multiple places as detected by secure coding plugin	61
Figure 5.4: Response time versus lines of code	73
Figure 5.5: Percentage wise usage of Programming languages	76
Figure 5.6: Percentage wise awareness of secure coding	76
Figure 5.7: Percentage wise use of secure coding standards while coding.....	77
Figure 5.8: Gradings received for usability of plugin.....	78

List of Tables

Table 2.1: Comparison of secure coding guidelines	11
Table 2.2: Main categories in SEI CERT secure coding rules	12
Table 3.1: Classification criteria	18
Table 3.2: Comparison between JavaParser and ANTLR	27
Table 4.1: Comparison of IntelliJ IDEA plugin development approaches	33
Table 4.2: Justification of method level secure coding rules	34
Table 4.3: Justification of class level secure coding rules	37
Table 4.4: Justification of Package level secure coding rules	41
Table 5.1: Results of Project based evaluation	54
Table 5.2: Detection of secure coding rules by static code analysis tools	58
Table 5.3: Summary of the detection of secure coding rules by static code analysis tools	59
Table 5.4: Memory and CPU consumption with design patterns and without design patterns for individual rule	69
Table 5.5: Memory and CPU consumption with design patterns and without design patterns for above six rules	70
Table 5.6: Average response time with design patterns and without design patterns for individual rule ...	70
Table 5.7: Average total response time with design patterns and without design patterns for six rules ...	72
Table 5.8: Average response time for different lines of code	73
Table 5.9: Benchmark tool comparison using Secure Coding plugin and SonarLint.....	74

List of Acronyms

AST	Abstract Syntax Tree
CERT	Computer Emergency Response Team
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
SDLC	Software Development Life Cycle
SEI	Software Engineering Institute
OWASP	Open Web Application Security Project

Chapter 1 : Introduction

1.1 Motivation

The software security field based on the concept of Secure Software Development originated in the early 2000s [1]. As a result a set of software security best practices which involve in identifying and understanding common software threats, designing software focussing on security and subjecting all software artifacts through a thorough security analysis [2] have been introduced.

In the early 2000s, there were several critical cyber attacks such as Nimda, Code Red, etc that caused uncertainty among users with regards to the technology they were using every day [3]. This resulted in a need to improve the security, privacy, and reliability of the technology. As a result in the year of 2002, Microsoft's "Trustworthy Computing" initiative [1] was launched by Bill Gates with the primary intention of ensuring the technology used by people is secure and reliable. This initiative mainly focused on people, process, and technology to tackle the software security problem and overcome issues related to them.

Carnegie Mellon University has established a laboratory named CyLab comprising a set of researchers related to the field of cybersecurity [4]. A research on Cyber Security Engineering was done by CyLab has identified that organizations which focus on security in the early stages of the SDLC have seen major reductions in operational vulnerabilities, resulting in reductions in software patching. CyLab is currently researching on developing an advanced tool that could assist in identifying parts of web pages that are vulnerable to DOM Cross Site Scripting (XSS) attacks [4] which could be considered as a major step towards ensuring web security.

In the current context, smartphones have become a vital member in the daily lives of people and they contain information such as user's location, contacts and other sensitive information that has resulted in huge security breaches by various unauthorized parties such as hackers, attackers, etc. The software security related researchers at CyLab are currently engaged in research to improve the smartphone privacy and controlling access to third-party libraries that may contain malicious elements [4] with the intention of minimizing the security breaches into sensitive information. From these facts, it could be concluded that many parties are currently committed to ensuring software security by engaging in various ways.

1.2 Aims and objectives

The goal of this project is to build a framework to detect secure coding guideline violations in real-time in order to assist and encourage software developers to adhere to these guidelines. The main objectives of this project are,

1. Identification of existing approaches and their limitations.
2. Conducting a literature review on secure coding.
3. Study SEI CERT Oracle Secure Coding rules [5] and identify the most suitable set of rules that are to be implemented in the framework.
4. Designing and implementing a methodology to integrate these secure coding rules into the proposed framework.
5. Notify developers about potential violation of secure coding rules while they are coding in a real-time manner.
6. Evaluate the capability of the plugin-based framework to detect secure coding rule violations.

1.3 Scope

The proposed framework in this dissertation is only focused on analyzing projects based on Java programming language. It is focused on the coding or implementation phase of the SDLC. The proposed plugin is covering a selected set of secure coding guidelines and only focus on source codes written using IntelliJ IDEA Integrated Development Environment (IDE). The plugin supports on the fly catching of secure coding rule violations.

1.4 Structure of the dissertation

The remaining sections of this thesis are as follows. Chapter 2 is associated with the literature review and background study related to the project. Chapter 3 provides a detailed explanation of the architecture of the project and Chapter 4 explains the implementation of the project. Chapter 5 illustrates the evaluation methodologies along with their results and Chapter 6 concludes the dissertation along with a discussion regarding future work.

Chapter 2 : Background Study

2.1 Introduction

According to a research conducted by Tricentis which is an Austrian software testing firm, the total economic loss to the world in the year of 2016 as a result of software bugs, software failures and other vulnerabilities was approximately \$1.1 trillion including 606 software failures in 314 companies which affected approximately 3.6 billion people mainly in the areas of consumer and retail technology [6]. These facts convey an important message regarding the significance of following secure coding practices while writing the source code of a program since it will reduce the time and cost of developing less vulnerable software applications.

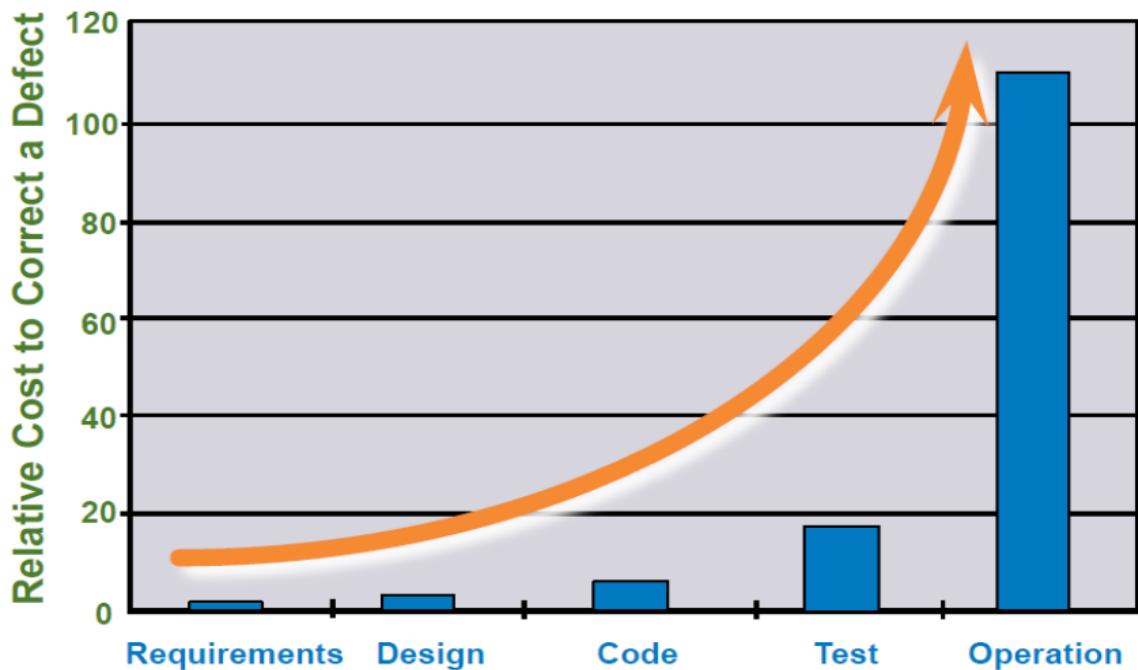


Figure 2.1: Costs of fixing bugs based on the phase of SDLC [7]

Figure 2.1 states that code fixes performed after the release or maintenance period known as the reactive approach are more expensive than correcting such issues during the coding phase (proactive approach) [8]. It is also visible that the increase in the cost is approximately 6 times [8] than fixing it in the coding phase which can be considered as a significant amount. Not only the cost but the time needed to correct the issues in the post-release period is also high since the source code is complex after the software application has been completely developed. Software

developers might also forget the contents of the source code since coding was done sometimes back. Therefore it will be difficult for them to rectify a large number of defects at once.

A major contribution to the secure software development was made by Gary McGraw by writing several important books namely “How to break code”, “Building security in [2]”, “Java security”, etc. Out of them “Building security in” is the most prominent book related to the development of secure software which mainly consists of facts [7] that indicates the importance of following secure coding concept while developing software. According to Gary McGraw, software security must be incorporated into all levels of the SDLC [2]. In this book, Gary also mentioned a set of best practices [9] known as the seven touchpoints as shown in *figure 2.2*, which may need to be followed in order to develop secure software.

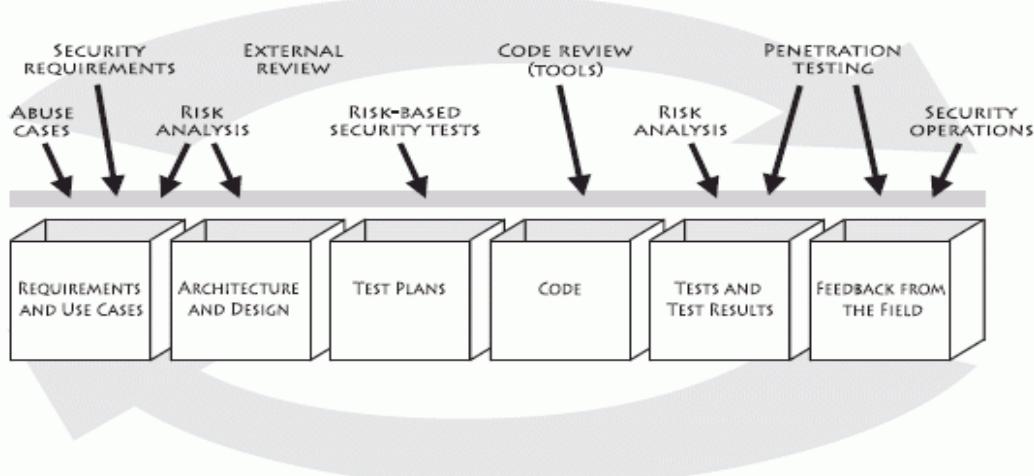


Figure 2.2: Software security best practices applied to various software artifacts [9]

The descending order of effectiveness of the seven touchpoints [10] has been identified as follows :

Code review → Risk analysis → Penetration testing → Risk analysis → Abuse cases → Security requirements → Security operations

It could be seen from the above order of effectiveness, the importance of *code reviews* and that they mainly involve in examining the source code, identifying issues and correcting them in order to improve source code quality. Also, it could be concluded that source code plays a major role in building secure software since code reviews are associated with the source code.

2.2 Review of similar systems

There are few similar static code analysis tools that mainly focus on source code quality issues and bug detection but these tools do not give the exact solution for the real world problem which is to detect the secure coding rule violations and to encourage developers to write secure code. Following are two major Software Engineering solutions currently available.

1. SpotBugs [11]

SpotBugs is a tool which uses the concept of bug patterns to detect bugs in Java bytecode. It can be considered as the successor of FindBugs tool and it is available as a free software which is distributed under the terms of GNU Lesser General Public License. Currently, SpotBugs supports more than 400 bug patterns with reference to Open Web Application Security Project (OWASP) Top 10 and Common Weakness Enumeration (CWE). Bugs have been classified into four main categories namely *Scariest*, *Scary*, *Troubling* and *of concern*.

It is used majorly as a plugin and supports major IDEs such as Eclipse, IntelliJ IDEA, NetBeans, etc and when a bug is detected in the source code, a small bug icon is displayed at the beginning of the source code line in the IDE. Also, command line integrations may be done with build tools such as Gradle, ANT, and Maven. SpotBugs is also extensible where new detectors may be added through plugins such as fb-contrib and find-sec-bugs. SpotBugs tool is capable of detecting relevant bugs in source code but not secure coding rule violations and can not be considered as a solution for identifying such violations.

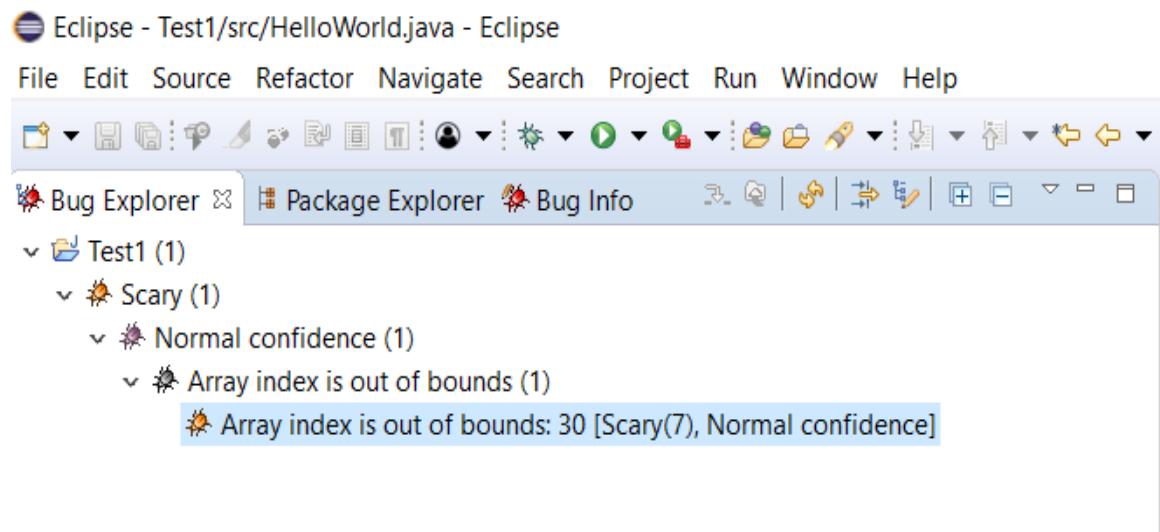


Figure 2.3: Detection of bugs using SpotBugs plugin

2. SonarLint [12]

SonarLint is a tool recently introduced to the Sonar family. After its introduction, Sonar family comprises 3 main tools namely SonarLint, SonarCloud, and SonarQube. SonarLint exists as a plugin and provides on the fly detection of source code quality issues and these issues have been classified into 3 main categories namely *Vulnerabilities*, *Bugs* and *Code smells*. It supports the currently existing major IDEs namely Eclipse, IntelliJ IDEA, Microsoft Visual Studio, VS code and Atom.

SonarLint comprises of several major features such as bug detection which involves in detecting common mistakes, tricky bugs and known vulnerabilities, provides on the fly instant feedback when the bugs are detected, provides guidance regarding the countermeasures for such bugs, uncovering old issues, provides descriptions about the errors that have arose in the source code, etc. But the on the fly feedback is provided only when the Java class is *saved* not while the user types the source code in the IDE. Similar to SpotBugs, SonarLint is incapable of detecting secure coding rule violations that occur in the source code and thus can not be considered as a solution for identifying such violations that occur in source code.

The screenshot shows a Java code editor with a file named `HelloWorld.java`. The code contains a simple `System.out.println` statement that is highlighted in blue, indicating it is being analyzed by the SonarLint plugin. A tooltip window appears over the line of code, containing the following text:

- Replace this use of `System.out` or `System.err` by a logger.
- 1 quick fix available:
- [Open description of rule squid:S106](#)

At the bottom right of the tooltip, there is a note: "Press 'F2' for focus".

Figure 2.4: Detection of a code quality issue using the sonarLint plugin

2.3 Related work and limitations

2.3.1 Related work

The research paper published by Steve Lipner discusses the *Trustworthy Computing Security Development Lifecycle* [1] which is adopted by Microsoft for the development of software that needs to withstand malicious attacks. This methodology introduces a series of security-focused activities and deliverables to each of the phases of the development process. During implementation or coding phase of the SDLC the development team codes the software. In this phase, secure coding standards can be applied in order to prevent forms of security vulnerabilities. In addition, static code analysis tools and code reviews can also be conducted.

Malik Imran published a paper on secure software development model [13] which mainly discuss based on Extreme Programming (XP) technique, a new model which has been designed to focus on the concept of iterative development of secure software. At each phase of the SDLC, security requirements are considered and iteratively updated. During security design and implementation, threat modeling can be used in order to identify threats, vulnerabilities, and their countermeasures. During implementation, the known security vulnerabilities and their countermeasures can be taken into consideration while developing software.

Noopur Davis from Carnegie Mellon University [14] published a paper on a survey conducted to discuss existing SDLC processes. This article provides overview information about existing processes, standards, life-cycle models, frameworks, and methodologies that support or could support secure software development. The target audience of this paper includes program managers, project managers, developers, software engineering process group (SEPG) members who want to integrate security into their standard software development processes. This document also provides overviews of process models, processes, and methods that support one or more phases of the SDLC focussing on areas of secure software development.

V.S. Mdunyelwa, J.F. van Niekerk and L.A. Futcher in this [15] research paper, discussed whether software developers and students know about secure coding practices. Security breaches in web applications are mostly caused by programmers failure to adhere to secure coding practices, such as those recommended by the OWASP. Programmers cannot adhere if they are not educated regarding the secure coding best practices in the first place. It discusses secure coding practices according to the OWASP guidelines. It further discusses what the Association for Computing Machinery (ACM) curricular guidelines for Information Technology states in terms of secure coding practices.

Currently, many static code analysis tools are used by software companies to detect security vulnerabilities and bugs. But these tools are mainly used when its close to a major software release which is commonly known as batch style analysis [16]. At this point, the software developers that

programmed the software application might have forgotten the coding context and also correcting such issues is extremely complex due to the low effectiveness of static code analysis tools since they exhibit a large number of false positives [17]. As a result building and patching which is the concept used to overcome this issue where patching is done to the relevant issues, but the real errors exist in the source code level.

The concept of Just In Time(JIT) static code analysis introduced by Justin Smith along with five other scholars [16] mainly focuses on code development along with bug fixing. Unlike traditional batch-style analysis tools, a JIT analysis tool presents warnings to software developers over time, providing the most relevant results quickly, and computing less relevant results incrementally later [16]. In this paper the JIT static code analysis concept which has been proposed mainly involves the integration of static code analysis into the development workflow, allowing developers to immediately see the impact of their changes in the code without preventing them from performing other coding tasks.

2.3.2 Limitations of current approaches

It could be concluded from [1], [13], [16], [14], [15] and [17] that the development or the coding phase of the Secure SDLC is extremely important. But in the current context, there exists no automated mechanism to support software developers adhere to secure coding guidelines in order to minimize the introduction of security vulnerabilities during the coding phase. In the current context, there exist solutions such as *SpotBugs* and *SonarLint* that mainly focuses on detecting security vulnerabilities and source code quality issues but do not detect secure coding guideline violations. Therefore currently there does not exist any automated mechanism to support software developers adhere to secure coding guidelines while they are writing source code.

2.4 Secure coding guidelines

2.4.1 Secure coding

As mentioned previously in the introduction section of the report it could be seen that insecure coding practices committed by software developers during the coding phase of the SDLC incur heavy costs in developing a software application due to wastage of time and other resources. In order to overcome this issue, the concept of secure coding has been introduced.

The concept of secure coding primarily states that Computer Software should be developed in a manner such that accidental introduction of bugs and security vulnerabilities during the development or coding phase needs to be minimized or prevented by the use of appropriate guards, typically following a set of best practices known as the secure coding guidelines.

It has been discovered that there are 3 main reasons for developers to follow these set of secure coding guidelines when they are writing the source code of a computer software program. They are [18],

1. A set of best practices developed with expertise support -

The secure coding guidelines have been developed by analyzing the feedback from various software developers based on the vulnerabilities that they encounter in various software projects over a period of time. Also, many experienced people in the security field with the necessary expertise have involved in designing these guidelines. Thus making these guidelines a set of best practices for software developers.

2. Creating efficiencies through standard coding practices -

Following secure coding guidelines when writing source code will allow the creation of consistency in the source code and also create a common communicating mechanism among various developers that follow different coding techniques (Similar to software design patterns that create a method of communication among developers). This would definitely save a massive amount of time since any new developer that has joined a project could get a quick understanding of the source code since it is in a standard form.

3. Reduction of costs significantly -

Since the software bugs and vulnerabilities are prevented entering into the source code at an early stage(coding stage) of the SDLC it would significantly reduce developer time incurred on correcting the bugs and vulnerabilities that are identified during the testing stage. This will allow the developers to invest their time in another project thus reducing

the average cost of a project (Because minimizing the unnecessary activities in a software project reduces overall project cost). It has been found that the cost of building secure software following secure coding practices is much lower than correcting security-related issues after the software application has been developed [19].

2.4.2 Secure coding guidelines

During the literature review, secure coding guidelines provided by 3 parties namely *OWASP, Oracle, and SEI CERT* were identified. A comparison analysis based on several selected parameters was carried out in order to select the most feasible set of secure coding guidelines to be implemented in the project. In *Table 2.1* Significant refers to equal or greater than 50%, Less refers to less than 50% of the resources.

Table 2.1: Comparison of secure coding guidelines

Parameters	SEI CERT [5]	Oracle [20]	OWASP [19]
Number of Resources / References available	Significant	Less	Less
Code examples provided	Significant	Significant	Less
Nature (language specific/not)	Language specific	Language specific	Not language specific
Security domain Coverage (Security areas covered)	Significant coverage	Low coverage	Significant coverage
Prioritization of guidelines	Yes	No	No

Based on the analysis shown in *Table 2.1* it could be concluded that the secure coding rules provided by *SEI CERT* are the most suitable set of secure coding standards to be implemented in the solution of the project. As previously mentioned there are 19 categories of secure coding rules and a brief explanation of them is shown in *Table 2.2*.

Table 2.2: Main categories in SEI CERT secure coding rules

Category	Explanation
Input Validation and Data Sanitization (IDS)	<p>Input validation mainly involves testing input data provided by a user into an application and prevents improperly formed data from entering into that application.</p> <p>Data sanitization involves the process of deliberately, permanently, and irreversibly removing or destroying the data stored on a memory device.</p>
Declarations and Initializations (DCL)	<p>Mainly associated with declaring and initializing of variables.</p>
Expressions (EXP)	<p>Expressions are usually used to produce a new value or assign a value to a variable. Expressions are built using values, variables, operators and method calls.</p>
Numeric types and Operations (NUM)	<p>Numeric types are used to handle various numbers using operations such as assignment, arithmetic, unary, etc.</p>
Characters and Strings (STR)	<p>Related to the use of strings and character data types in Java Programming language.</p>
Object Orientation (OBJ)	<p>A programming language model organized around objects along with the concepts of inheritance, encapsulation, polymorphism and abstraction.</p>
Methods (MET)	<p>Represents the behaviors of class instances(objects).</p>
Exceptional behavior (ERR)	<p>Mainly associated with handling exceptions during the execution of a program.</p>
Visibility and Atomicity (VNA)	<p>Focuses on accessibility and independence of variables and objects.</p>
Locking (LCK)	<p>Is a synchronization technique that allows at most one thread to own and make changes to a resource.</p>

Thread Application Programming Interface (THI)	Associated with thread functions and their states (New, Runnable, Blocked, Waiting).
Thread Pools (TPS)	Thread pools typically consist of idle threads which are to be used in future processes.
Thread Safety Miscellaneous (TSM)	Mainly comes into effect in a multi-threaded code where multiple threads share common resources such that consistency is maintained.
Input Output (FIO)	Involves in performing reading(input) and writing(output) operations using stream data.
Serialization (SER)	This is the process of converting an object into a stream of bytes in order to store the object and reconstruct it later.
Platform security (SEC)	Associated with the security concerns of the Java platform
Runtime environment (ENV)	Concerned with privileges of the Java Runtime Environment.
Java Native Interface (JNI)	It is a framework that enables Java code running in a Java Virtual Machine to call and be called by <i>native</i> applications
Miscellaneous (MSC)	Concerned of guidelines that do not fall on any of the above mentioned categories.
Android (DRD)	Associated with mobile application development based on Java.

2.4.3 SEI CERT Secure coding rules

The SEI of Carnegie Mellon University along with their CERT group have introduced a set of secure coding standards known as SEI CERT coding standards for a set of *specific* programming languages including C, C++, Java, Perl and Android [21]. The professors and lecturers at Carnegie Mellon University has actively contributed to improve these standards and promote them by writing books such as “The CERT Oracle Secure coding standard for Java” [5], “Java Coding guidelines for reliable and secure programs” [5], “Secure coding in C and C++” [21], “The CERT

C++ secure coding standard” [21], etc. This implies that SEI CERT has provided a huge set of resources to promote the concept of secure coding among the software developer community.

SEI CERT has provided a set of secure coding standards for Java Programming language known as Rules (19 rules) and Recommendations (12 recommendations) collectively known as *guidelines* [5]. Violation of a rule may result in a defect that may adversely affect the reliability, safety, and security of software whereas a recommendation typically suggests improving code quality [22]. Hence the impact of secure coding rules is much higher than that of the recommendations. SEI CERT has provided comprehensive documentation for these guidelines along with compliant and non-compliant code examples making it simpler for the developers to understand the guidelines. Also, these guidelines have been prioritized based on 3 major parameters [23] as shown in *Figure 2.5*.

- 1) **Severity** - The serious nature of the consequences or outcomes of the rule is ignored.
- 2) **Likelihood** - The probability of a flaw introduced by violating the rule could lead to a security vulnerability.
- 3) **Remediation cost** - Cost involved correcting the existing unsecure code to comply with the rule.

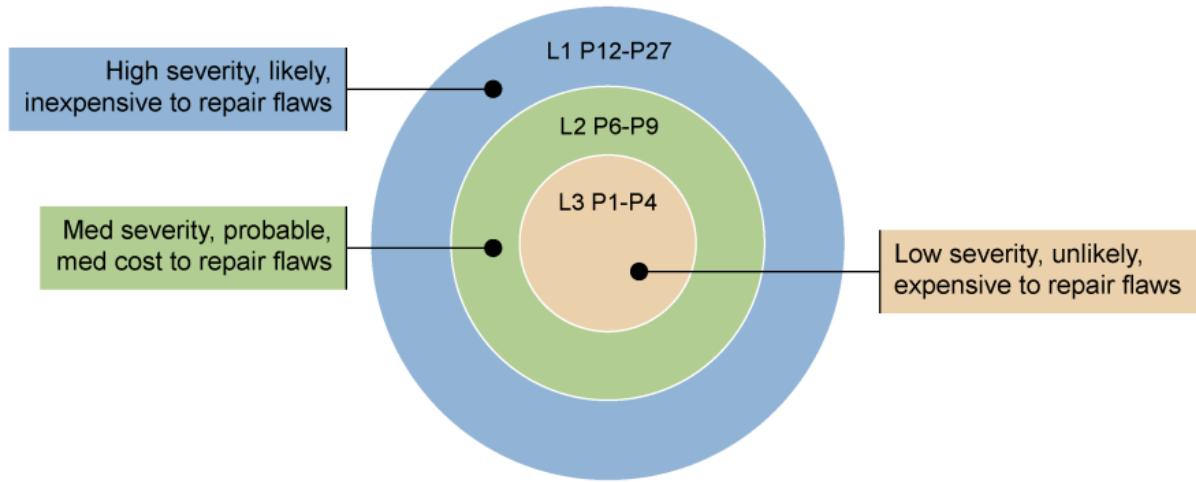


Figure 2.5: Levels and Priority ranges

2.5 Summary

There is a huge increase in remediation costs when software defects are detected and corrected at post-implementation phase of the SDLC when compared with the development or coding phase of the SDLC. Following of coding best practices by software developers known as *secure coding guidelines* while developing of software applications, can be considered as the well known and accepted method to overcome this issue since these best practices primarily focus on vulnerabilities that may arise in the source code level.

The review of existing approaches gives an idea that there exists no automated mechanism to identify the violation of secure coding guidelines in the source code in order to support software developers to adhere to them in order to minimize the accidental introduction of security vulnerabilities while coding. Developing a software product that is able to detect these violations in the form of a plugin-based framework could be considered as the best solution for the aforementioned problem.

It could be concluded that three parties namely OWASP, SEI of Carnegie Mellon University along with their CERT and Oracle have majorly contributed to the promotion of the secure coding concept among software developers. Out of these three parties, the coding best practices introduced by SEI CERT known as the Secure Coding Rules could be considered as the most feasible set of best practices to be implemented in the plugin-based framework.

Chapter 3 : Analysis and Design

3.1 Introduction

The design and analysis chapter mainly describes the proposed design of the framework that was implemented in order to provide a solution for the aforementioned problem. Based on the literature review that was carried out in the background study several design methodologies were identified. The system architecture and the system model was developed based on those design methodologies. The system architecture consists of three main layers namely the *Presentation layer*, *Application layer* and the *Data layer* with relevant components for each layer. The workflow process was also identified during the design phase in order to get a thorough view of the proposed solution. The *system modelling* stage mainly focuses on the *class diagram* which provides a static view of the plugin-based framework along with the relevant design patterns. Use of design patterns is a main focus in Product based Software Engineering projects. Several parser selection criteria were also analyzed in this phase.

3.2 Problem analysis

The goal of the project is building a framework to detect secure coding guideline violations in real-time to assist developers to adhere to secure coding best practices. Most of the Software developers are unaware of this concept and the ones that follow secure coding guidelines while writing source code encounter many difficulties in adhering them due to the manual cheat sheet approach which is lengthy and consumes a huge amount of time. In order to achieve the goal, an extensive background study was conducted by referring relevant artifacts such as white papers, dissertations, existing tools, etc.

The acquired knowledge from the background study was used to identify relevant requirements, design system architecture and system model, identify related components of the framework, etc. Remedies for the limitations identified in existing Software Engineering solutions, related concepts in white papers and dissertations, workflow and interaction between the components were incorporated into the system design of the framework. Since the solution involves the development of a plugin-based framework for IntelliJ IDEA IDE, the relevant approaches for plugin development were observed and parser libraries were analyzed in order to identify the most feasible parser to be used in the solution.

3.3 Design assumptions and dependencies

1. This plugin-based framework requires the user to be a person with basic Java programming knowledge. The user of this plugin would be a software developer who should be able to fix security vulnerabilities shown by the secure coding framework, after referring to countermeasures given.
2. The user is responsible for following general coding standards such as proper indentation, variable declaration, proper usage of brackets etc. This is important because the source code written by developers needs to be parsed in order to analyze it by the secure coding plugin. Therefore, the source needs to be syntactically correct. JavaParser library, which is used in this project allows parsing of slightly syntactically incorrect source codes but when designing this framework the assumption is made that user will write syntactically correct source codes. This can be easily achieved with the help of IntelliSense feature of IntelliJ IDEA. This framework will aid developers to write more secure code but not with general coding standards such as proper indentation, variable declaration, proper usage of brackets and etc.
3. This framework requires the user to have a compatible version of IntelliJ IDEA IDE or any other supported JetBrains IDE up and running (community edition or proprietary commercial edition). The user can access JetBrains plugin repository through their IDE and this plugin will only be shown to the user if it is compatible with the IDE version of the user.
4. This framework assumes it will detect the violations of secure coding rules based on the source code granularity levels namely Method level, Class level and Package level. At present the framework supports 15 secure coding rules given by SEI CERT Secure Coding Standard for Java. This number can increase as this plugin is designed and built in a way that supports the addition of new secure coding rules. (Extensibility is a key feature of the secure coding plugin)
5. The framework will not depend on any other software or service but it is using an existing parser library to transform the source code into an Abstract Syntax Tree (AST). The parser used in this projects is JavaParser library. This JavaParser library is integrated into the system. Therefore this plugin works as a standalone system and each user who downloads it will have their own copy.
6. The framework makes the assumption that the user has an active internet connection. Internet connection would be required when downloading the plugin from the plugin repository and also when accessing the links provided by the plugin to find more details

about the detected secure coding rule violations. These details can be used when applying fixes to detected secure coding rule violations.

3.4 Secure coding rules classification criteria

In order to implement the secure coding rules in the plugin-based framework, the rules were initially classified into three main granularity levels namely *Method level*, *Class level* and *Package level* based on the source code fragments that triggered the violations. The main focus of the Method level classification was to identify the source code fragments that lie inside a specific method. The source code fragments that are interrelated between two or more methods of the same class are classified under Class level granularity. Package level granularity primarily focussed on the source code fragments that lie between two or more classes inside the same package. The classification criteria for source code fragments that fall into each granularity level are as shown in *Table 3.1*.

Table 3.1: Classification criteria

Method Level	Class Level	Package Level
<p>Focuses on the source code fragments that belongs to the java.lang package (default package) and exist inside a method of a class.</p> <ul style="list-style-type: none"> 1. Method parameters in method signature 2. Local variables 3. Loop controls (for, for each, while, do while) with no method calls 4. Exceptions belonging to the java.lang package (eg- 	<p>Focuses on the source code fragments of the java.lang package(default package) that is inside a class but lies outside a method.</p> <ul style="list-style-type: none"> 1. Names of class variables 2. Data types of class variables 3. Access modifiers of class variables 4. Method names in a method signature 5. Return types of methods in a method signature 	<p>Focuses on the source code fragments that belongs to classes outside the existing class.</p> <ul style="list-style-type: none"> 1. Methods belong to packages outside java.lang package 2. Extended classes outside java.lang package 3. Library imports 4. Implemented interfaces which are outside java.lang package

<pre> NullPointerException) 5. Threads(That fall into java.lang package) 6. Try, catch, Finally blocks </pre>	<p>6. Access modifiers of methods in a method signature</p>	<p>5. Instances of classes outside java.lang package</p>
--	---	--

A set of classified secure coding rules (100 rules) based on the above-mentioned classification criteria can be found in *Appendix B*. A set of 15 secure coding rules 5 belonging to each granularity level out of the classified set of rules has been implemented in the framework.

3.5 Product perspective

The software product developed in this project is an open source product. It is a plugin-based framework for IntelliJ IDEA, which is an IDE. The plugin can be used as an aid for software developers to correct secure coding rule violations they have done when coding. This is an entirely new software product and not an extension of an already existing product.

3.5.1 Dependencies

This software product does not depend on any other software or service but it is using an existing parser(JavaParser) which is integrated into the system. *Figure 3.1* shows how the product is interacting with the internal and external environment.

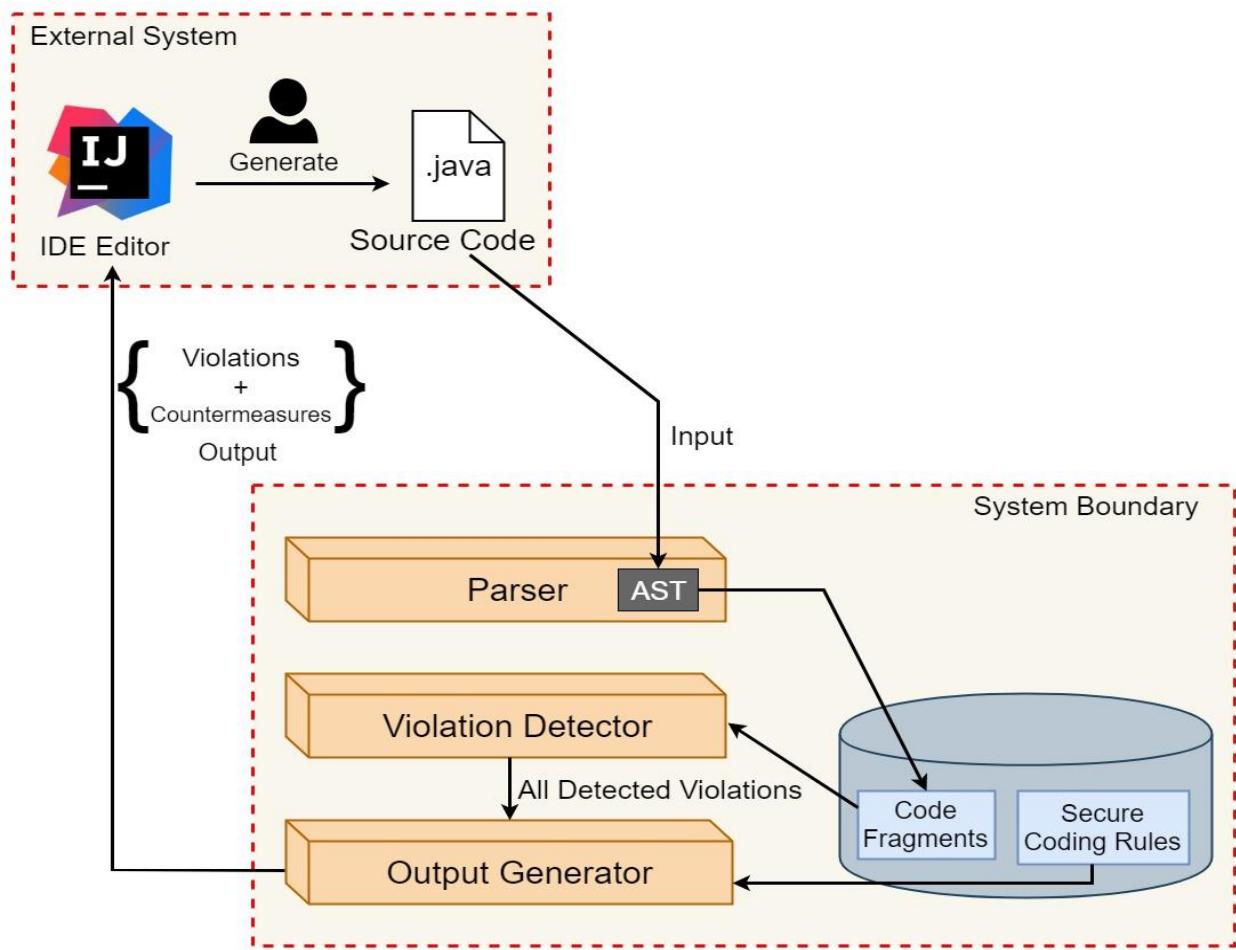


Figure 3.1: Product interaction with internal and external environment

3.5.2 Design and Implementation constraints

The software product developed is intended to provide real-time feedback to its users. Therefore, secure coding rules and code fragments are stored in a common data structure and that improves the performance of the plugin. Software design is made transparent and simple in order to provide the ability to extend the framework when a new secure coding rule or a new granularity level needs to be added to the framework.

The software product developed consists of three main components. They are *parser*, *violation detector* and *output generator*. Each component would be using the results given by its preceding component as its input. Following are the description of these components and component architecture diagram for the proposed system.

1. **Parser** - This component is taking a source code as its input and it transforms it into a structure called an AST. The parser is also providing methods to get code fragments by traversing the AST it created. A code fragment represents a set of items like a list of local variables, a list of class variables, a list of method names etc. The output of the parser component is the code fragments.
2. **Violation Detector** - This component has three subcomponents namely *Method level violation detector*, *Class level violation detector* and *Package level violation detector*. These subcomponents represent three main granularity levels in the source code. Under each subcomponent, there are algorithms for detecting secure coding rule violations relevant to granularity level it specifies. These algorithms will take code fragments given by parser component as its input. Each algorithm is giving an output stating whether the particular secure coding rule it represents, has been violated or not. Finally, the violation detector component outputs all of these secure coding rule violations it has identified to the output generator component.
3. **Output Generator** - The main intention of this component is to receive the results given by violation detector component as its input and rank them according to priority. Priority level [23] specified under each secure coding rule of "SEI CERT Oracle Coding Standard for Java" is used for this purpose. After ranking, secure coding rule violations with the highest priority is given as the output of this component. This output along with the full specification of relevant secure coding rules are used to notify the developers, as real-time feedback.

3.6 System architecture design

The proposed system is a standalone system which runs on an IDE. It could be seen from *Figure 3.2* that the plugin-based framework could be organized into three *horizontal layers*, with each layer performing a specific role within the system. These three layers are the *presentation layer*, *Application layer* and *Data layer*. The main idea behind this architecture is the separation of concerns. This architecture is also used for communication due to its simplicity. Following are the descriptions of the roles and responsibilities of each layer.

1. **Presentation Layer** - Users of the proposed system would be using an IDE to interact with the system. Therefore, IDE could be considered as the presentation layer, which is responsible for handling all user interface logic. Presentation layer only needs to know how

to format user data for displaying and doesn't even need to think about from where the data is coming. The presentation layer will pass the source code of the currently opened source file in the editor of the IDE to its underneath layer. The presentation layer is also responsible for showing the results that it will receive from the application layer.

2. **Application layer** - This layer consists of 3 main components. They are parser, violation detector and output generator. Application layer handles the business logic of the system. The parser is responsible for parsing the source code it received from the presentation layer and creating an AST. Violation detector will utilize the output of parser to implement violation detection algorithms. Output generator is responsible for generating the output and passing it back to the presentation layer. Output generator would also be making use of the data layer to perform its tasks. All these three subcomponents in the application layer represent three main business logic in the system. The application layer is also responsible for moving and processing data between its two surrounding layers. Following are the three subcomponents mentioned in the application layer.
 1. **Parser** - Represents an existing parser called JavaParser which will be used to create an AST from the source code. This subcomponent in application layer will directly interact with the presentation layer.
 2. **Violation Detector** - This subcomponent will interact with the parser to get the generated AST and use it to detect secure coding rule violations.
 3. **Output Generator** - This subcomponent will interact with violation detector component of the application layer and the data layer to detect the set of secure coding rule violations to be shown to the user. This subcomponent would also be interacting with the presentation layer. The user will see the results when it is passed from the output generator to the presentation layer.
3. **Data layer** - The proposed system will use a selected data structure such as ArrayLists, HashMaps etc to store its data. Data of the system would be a secure coding rule set and code fragments extracted by traversing an AST. Output generator component of the application layer will interact with the data layer to carry out its tasks.

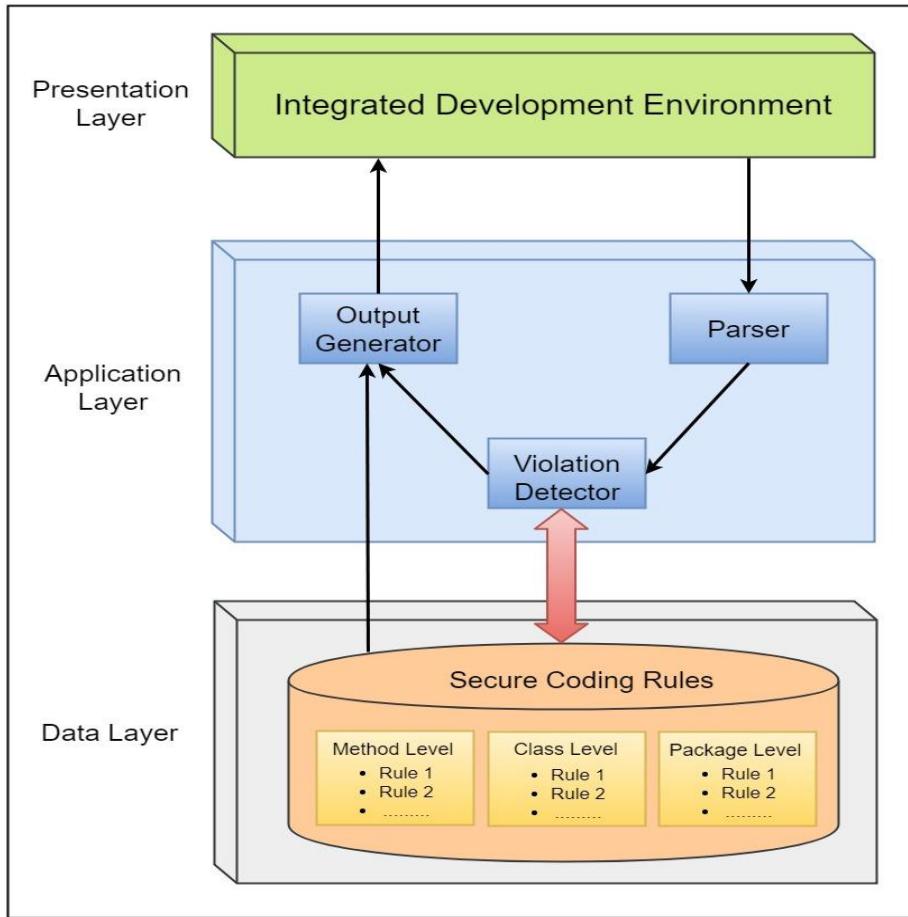


Figure 3.2: High-level (tier) architecture

3.7 Parser selection

3.7.1 Compilers

Source codes written by developers in a particular programming language needs to be understandable by machines in order to execute it. The process of converting a source code written by humans to a format which is understandable by machines(machine-code) can be done using a compiler. A compiler is a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer. A *parser* is a part of the compiler which is used during syntax checking or parsing phase in the compilation process.

Compilation process consists of several steps [24]. These steps are lexical analysis, syntax analysis/parsing, type checking, intermediate code generation, register allocation, machine code

generation and finally the assembly and linking. During the lexical analysis phase, the source code is read and divided into tokens, each of these tokens corresponds to a symbol in the programming language. For example, a symbol can be a variable name, a keyword or a number. During syntax analysis/parsing phase, the list of tokens produced by the lexical analysis phase will be arranged in a tree structure. This tree reflects the structure of the program. The output of the parsing phase is a syntax tree or an AST. AST focuses more on the abstract relationships between the components of source code. This syntax tree or AST will be used during the next phases of compiling to get the machine-code of the relevant source code.

3.7.2 Parsers

Parsers are used during the syntax analysis/parsing phase of a compiler. The input to a parser is a list of tokens and the output is a syntax tree. A parser can be written by hand or automatically generated by parser generators like ANTLR, Bison, JavaCC etc. A parser usually consists of two parts called a lexer/scanner/tokenizer and proper parser. Lexer scans the input and produces the matching tokens, and the parser scans these tokens and produces a syntax tree or an AST.

3.7.3 Rationale for the need of a parser

We follow two main steps to identify secure coding rule violations. They are AST creation and AST analysis. An AST omits unnecessary syntactic details of the source code and describes the source code in a convenient format for analyzing.

This project involves identifying secure coding rule violations using well-defined algorithms. These algorithms need code fragments to operate. For example, a code fragment can be a list of methods declared with the private access modifier, a list of synchronized methods etc. In order to get code fragments specific to the source code, the AST relevant to this code needs to be walked/traversed and analyzed. Hence, this project requires a parser to extract an AST from the source code.

3.7.4 Parser selection

In order to analyze a source code, it needs to be represented in a format that is good for analyzing. Best format suited for analyzing a source code (which is in concrete syntax) is an AST. An AST is a tree which contains the nodes that are necessary for representing the source code. Source code needs to be parsed to get an AST. Three main approaches can be followed to parse the source code. They are,

1. By developing a custom parser.

2. By using a tool or library to generate a parser: for example ANTLR, that can be used to build parsers for any language.
3. By using an existing library supporting that specific language: for example JavaParser library to parse source code written in Java.

The first approach is writing a parser by hand. It is a time-consuming task. In this case, the source codes written in Java programming language need to be parsed. Since Java is a very popular programming language, there are many open source libraries available to use which means that building a parser by hand is a rework.

The second approach to parse a language is by generating a parser using a parser generator. ANTLR which stands for “ANother Tool for Language Recognition” is a very popular parser generator. It can be used to generate parsers for reading, processing, executing, or translating structured text or binary files [25]. To follow this approach (using ANTLR), grammar needs to be defined. Then ANTLR will generate a parser using the grammar that is defined. The parser given by ANTLR can be used to build and walk parse trees or ASTs. ANTLR is also capable of generating parsers for the same grammar in many languages (Java, C#, Python etc). Building a parser using ANTLR may have been a better approach if the source codes of many programming languages need to be analyzed.

The third approach is to use an existing parser or a library. JavaParser is one such library which can be used for processing source code written in Java. Using JavaParser library, Java code can be parsed to get an AST and also this AST can be processed or analyzed by using the methods JavaParser have defined. In order to process an AST, its nodes need to be traversed. There are two main approaches to traverse an AST. They are by using a visitor or by using a recursive iterator [26]. A visitor is usually used when specific types of nodes in the AST needs to be operated. An iterator is used when all sort of nodes in an AST needs to be processed. JavaParser provides built-in methods and functions to follow both of these approaches.

JavaParser also provides a number of methods and function that can be used to analyze an AST generated by it. All these methods are well-documented in a Javadoc [27]. Other than parsing and analyzing, JavaParser is also capable of transforming and generating a source code given an AST. JavaParser also has an excellent online community to discuss and get help when needed. JavaParser contributors also provide a book called "JavaParser visited" which can be used by its users as a reference on how to use JavaParser in their projects. Due to all these benefits, JavaParser is used in this project.

3.7.5 JavaParser

JavaParser is a library to *parse, analyze, transform and generate Java code from an AST*. These are the four main functions of JavaParser. As the name suggests, it works on Java programming language. In this project, JavaParser is used to get an AST from Java code and analyze it using the built-in methods they have provided. Generating an AST using JavaParser can be done with just two lines of code. Following are the main features of JavaParser library [28].

1. JavaParser supports all versions of Java from 1 to 9.
2. JavaParser supports lexical preservation and pretty printing which means that it can parse Java code, modify it and print it back either with the original formatting or pretty printed.
3. JavaParser can be used along with JavaSymbolSolver which is used for symbol resolution. For example, using JavaSymbolSolver, an AST created by JavaParser can be analyzed and find the declarations connected to each of its element.
4. JavaParser supports almost all the new features of Java (lambdas, generics, type inference etc).

JavaParser will not only create an AST but also its built-in methods can be used to analyze an AST to get code fragments required by the secure coding rule violation detection algorithms. Following are the steps that were followed when identifying code fragments using JavaParser.

1. Parse the source code with JavaParser.
2. Traverse/walk the AST using the visitor design pattern or iterator design pattern.
3. While traversing/walking the AST, use built-in methods of JavaParser to get code fragments and store them in a data structure.

Due to the above-mentioned benefits, it could be concluded that *JavaParser* is the most suitable parser to be used in the proposed plugin-based framework. *Table 3.2* depicts a comparison between JavaParser and ANTLR.

Table 3.2: Comparison between JavaParser and ANTLR

JavaParser	ANTLR
A parser - Grammar already defined (User friendly)	A parser generator - Grammar needs to be defined (Less user friendly)
An AST can be generated - More focussed	A parse tree can be generated - Less focussed
Supports only Java programming language	Supports several programming languages
The reference guide is currently being written	The reference guide already exists

3.8 System modeling

3.8.1 Class diagram

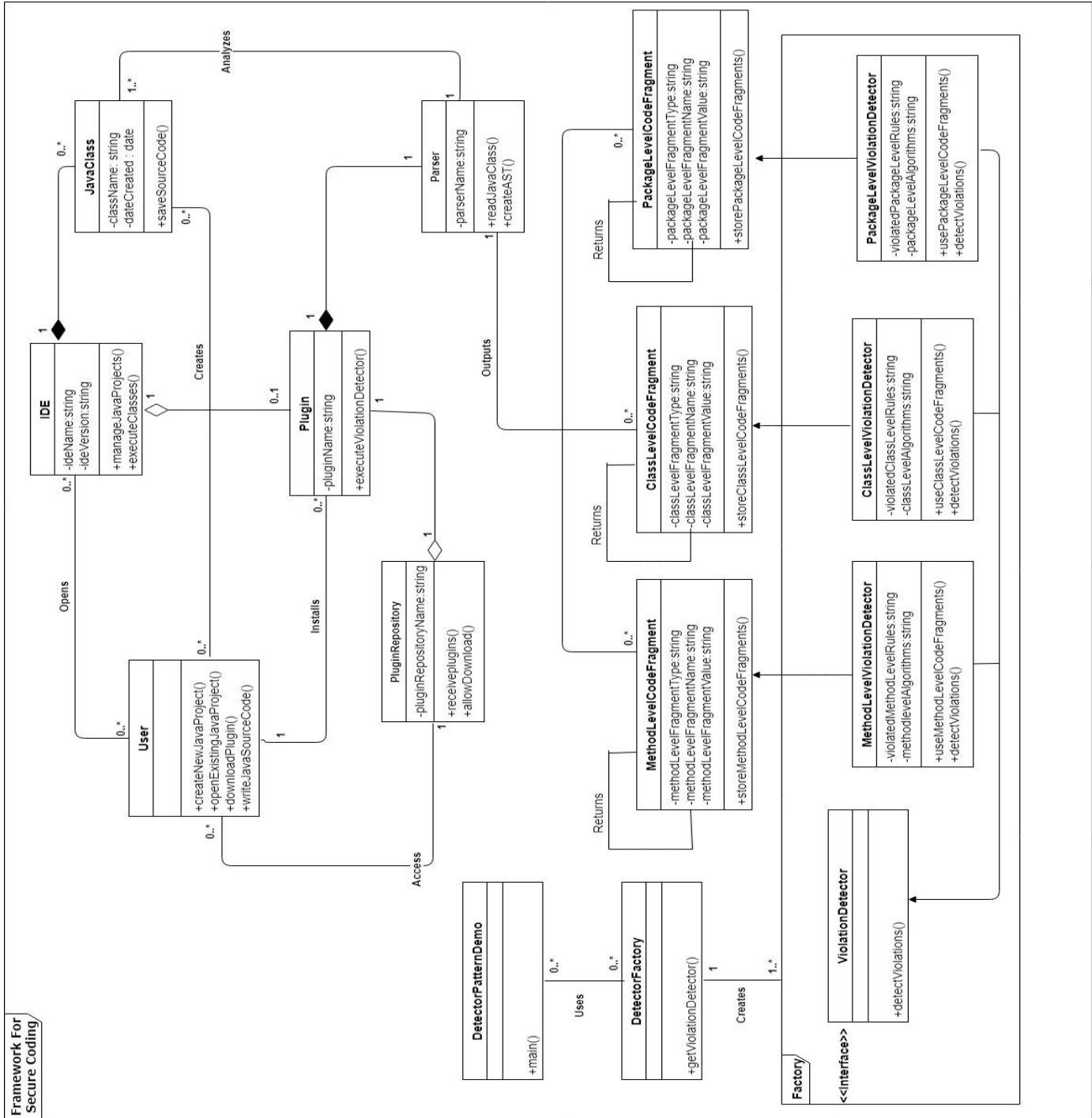


Figure 3.3: Class diagram

3.8.2 Design patterns utilized

In this project, two main design patterns are utilized during the implementation phase. They are given below.

1) Singleton pattern [29]

The *Singleton* design pattern is a creational design pattern which ensures that only a single object or instantiation of a class is created, through which other class objects access the singleton class members(methods and variables). According to the class diagram in the previous page, the three code fragment classes namely MethodLevelCodeFragment, ClassLevelCodeFragment and PackageLevelCodeFragment consist of data structures such as HashMaps and ArrayLists which are used store the code fragments belonging to each granularity level namely method level, class level and package level.

Any violation detector class corresponding to each source code fragment class, i.e. MethodLevelViolationDetector class corresponds to MethodLevelCodeFragment class)will only need a single instantiation of the code fragment class to access the source code fragments stored in the data structures. I.e. a single instance of MethodLevelCodeFragment class, is sufficient for one or more MethodLevelViolationDetector class to access the method level source code fragments stored in the MethodLevelCodeFragment class.

2) Factory pattern [29]

The *Factory* design pattern belongs to the category of creational design patterns and is mainly used to create an object of a class without exposing its internal class member details to the outside classes by accessing the created class object using a common interface.

As shown in the class diagram ViolationDetector is the common interface and the 3 violation detector classes namely MethodLevelViolationDetector, ClassLevelViolationDetector and PackageLevelViolationDetector are the concrete classes implementing the ViolationDetector interface. The DetectorFactory class would be used to create objects of the three concrete classes by passing the class type. I.e. if MethodLevelViolationDetector is passed as the type then an object of this class would be generated by the ViolationDetector factory class.

3.9 Justification as a Framework

Following are four major features[30] that a framework consists of and how they are achieved by the project solution.

1. Reusability

The concept of reusability is one of the significant features of a framework since reusable components ensure less effort required by developers to add new functionalities to the existing framework. This feature has been achieved in the project solution by the system modelling (class diagram). The source code fragments relative to each granularity level are caught into data structures such as ArrayLists and HashMaps.

When a new secure coding rule violation detector algorithm is to be added the source code fragments stored in existing data structures could be utilized ensuring the reusability feature. I.e. if a new method level secure coding rule is to be implemented the method level secure coding rule violation detector algorithm may use the source code fragments that are stored in existing data structures found in MethodLevelCodeFragment.java class.

2. Extensibility

The concept of extensibility is a mandatory requirement to ensure that a framework could be easily extended in the future to add new functionalities with minimum changes to the existing source code base. This concept has been implemented in the project solution by the use of three common data structures(of type `HashMap<String, Object>`) in each of the three source code fragment classes which is descriptively explained in the Extensibility based evaluation in section 5.3 of the thesis. In order to extend the framework, a developer may import this common data structure and add new secure coding rule violation detection algorithm without any changes to the initial source code base thus ensuring the extensibility mechanism has been achieved in the project solution.

3. Low coupling

The framework implemented is an Object Oriented framework since it typically involves java classes and objects. An important feature of such a framework is loose coupling among the framework components. This means that the software components should be less dependant on other software components. In this project loose coupling refers to the respective Java classes being less interdependent between each other.

The project solution has achieved this feature through its system model and system design. When the system model or class diagram is considered each source code fragment class is extended by its violation detector class. I.e MethodLevelCodeFragment.java class is extended by MethodLevelViolationDetector.java class. Thus it could be concluded that there is less interdependence between the java classes in the secure coding framework.

4. High cohesiveness

Being highly cohesive is another vital feature of an object-oriented framework which mainly refers to each software component performing a specific task. In the framework, it typically means relative focusness of each java class when performing a particular task. Each Java class of the secure coding plugin-based framework performs a specific task.

The three code fragment java classes mainly focus on storing source code fragments corresponding to each granularity level. The three violation detector classes focus on detecting algorithms, the LiveParser class mainly focuses on the on the fly mechanism and generation of the AST. Thus it could be seen that each class in the framework performs a specific task hence ensuring a high level of cohesiveness.

The successful fulfilment of the above four features in a comprehensive manner by the project solution could be concluded as a *strong justification* for it to be defined as a framework.

3.10 Summary

In this chapter, the design of the framework for secure coding plugin is presented using the system architecture diagram, product interaction with internal and external environment diagram and class diagram(other design artifacts found in *Appendix D*). Prior to designing the architecture, the selected secure coding rules are classified under three granularity levels as method level granularity, class level granularity and package level granularity in order to apply design patterns and improve the performance of the plugin as well as to add extensibility feature. The rationale for the need of a parser and the approach followed when selecting JavaParser is also presented under this chapter. The features of a framework and how those features are included into the plugin is presented under the justification of a framework section.

Chapter 4 : Implementation

4.1 Introduction

This chapter provides information as to how the solution is provided, development approaches followed, how each component was developed, tool and technologies used in implementing the plugin-based framework. The source code snippets are also illustrated at places where they are applicable. The implementation of the solution of this project which is a Framework for Secure Coding was primarily achieved by focussing on the source code abstraction levels and categorising them into 3 granularity levels namely Method level, Class Level and Package level. Each of the 3 group members took the responsibility of implementing the secure coding violation detection algorithms belonging to a particular granularity level as their individual component. The approach to integrating the 3 individual components using appropriate design patterns described in the system model(Class diagram) is mentioned in this chapter.

4.2 IntelliJ IDEA plugin development

A plugin is a software component that adds a specific feature to an existing computer program. Plugin for an IDE has the same purpose of adding specific features to it. IntelliJ IDEA is currently the most popular IDE among Java developers according to statistics. This may be due to its user-friendliness and enhanced features. This IDE already has a lot of built-in plugins which adds different features, and they also provide plugin development capabilities to its users to let them customise and extend IDE's functionality based on their specific needs. *IntelliJ IDEA Community Edition* which is the open source version of IntelliJ IDEA includes the complete set of plugin development tools where its users can use to develop custom plugins.

The *IntelliJ platform* provides a foundation for building JetBrains IDEs and all of the infrastructure that these IDEs need. The most popular IDEs are IntelliJ IDEA, WebStorm, RubyMine, DataGrip etc. These IDEs are the products of the IntelliJ platform. The IntelliJ platform fully supports plugin development, and JetBrains (the software development company which created IntelliJ IDEA) hosts a plugin repository that can be used to distribute plugins that support one or more of these products. Plugins that support different products can also be created. Since IntelliJ platform is a JVM application written mostly in Java and Kotlin, the developers who intend to create plugins for their products should also be familiar with these languages, and associated tooling. *IntelliJ Platform SDK* [31] is the primary source of documentation for plugin developers who intend to write plugins that will extend the IntelliJ platform. It provides necessary guidance on how to package, deploy and test plugins.

The most common types of plugins are [31],

1. *Custom language support plugins* that provide basic functionality for working with a particular programming language.
2. *Framework integration plugins* which allow integrating a framework to the IDE.
3. *Tool integration plugins* which allow manipulating third-party tools and components directly from the IDE.
4. *User interface add-ons* which allow adding various changes to the standard user interface of the IDE.

Out of these types, the secure coding plugin falls under the category of custom language support plugin because it provides *syntax highlighting*, *inspections* and *countermeasures* for source codes written in Java programming language.

IntelliJ Platform SDK provides two possible workflows for building IntelliJ IDEA plugins. They are by using Gradle or by using Plugin DevKit. The recommended workflow is to use *Gradle*. *Table 4.1* provides a comparison between the two approaches.

Table 4.1: Comparison of IntelliJ IDEA plugin development approaches

Using Plugin Devkit [31]	Using Gradle [31]
<ul style="list-style-type: none">● Provides support for developing IntelliJ plugins using IntelliJ IDEA's own build system.● Provides its own SDK type and a set of actions for building plugins within the IDE.● Possible to run and debug the plugin directly from the IntelliJ IDEA.● Provides a simple way to package plugins and generate a JAR file inside the project directory, which can be installed and distributed.● Can easily upload JAR file to the IntelliJ Plugin Repository.	<ul style="list-style-type: none">● Gradle is an open-source build automation system.● The plugin build using Gradle will take care of the dependencies of the plugin project.● Could easily build the plugin against many versions of the base IDE and make sure that the plugin is not affected by API changes which may happen between major releases of the platform.● Provides tasks to run the IDE with the plugin and to publish it to the JetBrains plugins repository.● This is the recommended solution for building new IntelliJ plugins.

Secure coding plugin-based framework was built using the *Gradle approach* because of it being the currently recommended approach by JetBrains community for new plugin development projects.

4.3 Method level implementation

4.3.1 Introduction

The method level implementation section mainly consists of details regarding the implementation of the secure coding rules that fall under the method level, based on the classification criteria in *Table 3.1*. The *Table 4.2* illustrates the secure coding rules implemented under method level based on the source code fragment which leads to the violation and the appropriate justification for each rule to be classified as a method level rule.

Table 4.2: Justification of method level secure coding rules

Main category	Secure coding rule	Justification
Numeric types and Operations (NUM)	NUM09-J	Violation of the rule occurs due to using float as the data type of the loop counter used in for loops found inside a method.
Exceptional Behavior (ERR)	ERR04-J	Violation of the rule occurs as a result of the contents inside finally block of a method containing return, break, continue, or throw statements which belong to java.lang package.
Exceptional Behavior (ERR)	ERR07-J	Violation of the rule occurs as a result of throwing RuntimeException, Exception or Throwable which are exceptions belong to java.lang package, inside a method.
Exceptional Behavior (ERR)	ERR08-J	Violation of the rule occurs as a result of catching NullPointerException, Exception or Throwable exceptions that belong to java.lang.package, inside the methods of a class.
Exceptional Behavior (ERR)	EXP02-J	The use of equals() method inside a declared method of a class to compare 2 arrays leads to the violation. The equals() method belongs to the Object class of the java.lang package.

4.3.2 Implementation procedure

The implementation process was carried out after the relevant algorithms were designed for each secure coding rule. In order to achieve the method level implementation two main java classes, namely *MethodLevelCodeFragment* and *MethodLevelViolationDetector* classes were used. The *MethodLevelCodeFragment* classes mainly consist of data structures such as ArrayLists and HashMaps which are used to store the relevant method level source code fragments corresponding to each method level secure coding rule. Each rule may have a corresponding single data structure or many data structures based on the nature of the algorithm.

The relevant source code fragments required by each secure coding rule are stored in the data structures of the *MethodLevelCodeFragment.java* class with the support of the relevant JavaParser methods and private static inner classes. The *LiveParser.java* class of the framework is used to capture source code fragments the user types in IntelliJ IDEA IDE in an on the fly(real-time) manner. Each time the user types a source code an AST is generated by the JavaParser library and the relevant JavaParser methods are used to traverse the AST with the support of the *Visitor* design pattern found in the JavaParser library.

After the relevant method level source code fragments are stored in data structures in the *MethodLevelCodeFragment.java* class, the method level secure coding rules are implemented in the form of algorithms in the *MethodLevelViolationDetection.java* class. The method level secure coding rule violation detection algorithms access the data structures in the *MethodLevelCodeFragment.java* class in order to obtain the relevant source code fragments needed for the algorithm to implement the violation detection algorithm successfully.

For instance, if Num09-J secure coding rule which is the first method level rule mentioned in Table 4.2 is considered, it mainly states users not to use float as the data type of the loop counters inside for loops. This is mainly because it may adversely affect the precision of the results after the execution of the for loop. To implement this secure coding rule successfully, the source code fragment which is needed is the list of data types of the loop counters of the "for loops" present inside methods.

The data type of the loop counters is stored in a data structure (A HashMap in this instance) in the *MethodLevelCodeFragment.java* class and the violation detection algorithm in the *MethodLevelViolationDetector.java* class accesses this data structure and if a float data type is found a violation is detected. The selected method level secure coding rules along with the algorithms designed for them and the relevant JavaParser methods used to obtain relevant code fragments are *Appendix C*.

```

public class MethodLevelViolation {

    void calculate1() {
        try {
            for (float x = 1; x <= 1.0f; x += 0.1f) {
                System.out.println(x);
            }
        } catch (Throwable e) {
            System.out.print("Exception caught" + e);
        }
    }

    boolean isCapitalized1(String s) {
        if (s == null) {
            throw new RuntimeException("Null String");
        }
        if (s.equals("")) {
            return true;
        }
        return false;
    }
    private static boolean doLogic() {
        try {
            throw new IllegalStateException();
        } finally {
            System.out.println("logic done");
            return true;
        }
    }
    public void compare() {
        int[] arr1 = new int[20]; // Initialized to 0
        int[] arr2 = new int[20]; // Initialized to 0
        System.out.println(arr1.equals(arr2)); // Prints false
    }
}

```

Figure 4.1: Detection of the mentioned Method Level secure coding rules(in orange colour)

Violated Rules	Rule Description										
NUM09J violated at line [5] ERROR08J violated at line [8, 26] ERROR07J violated at [15] ERROR04J violated at [29] EXPO2J violated at [35]	<p style="color: blue;">NUM09J : Do not use floating-point variables as loop counters</p> <table border="1" data-bbox="910 1537 1351 1615"> <thead> <tr> <th data-bbox="910 1537 975 1558">Severity</th><th data-bbox="975 1537 1073 1558">Likelihood</th><th data-bbox="1073 1537 1237 1558">Remediation Cost</th><th data-bbox="1237 1537 1318 1558">Priority</th><th data-bbox="1318 1537 1351 1558">Level</th></tr> </thead> <tbody> <tr> <td data-bbox="910 1558 975 1615">Low</td><td data-bbox="975 1558 1073 1615">Probable</td><td data-bbox="1073 1558 1237 1615">Low</td><td data-bbox="1237 1558 1318 1615">P6</td><td data-bbox="1318 1558 1351 1615">L2</td></tr> </tbody> </table> <p data-bbox="820 1628 997 1649">Click here for more details</p>	Severity	Likelihood	Remediation Cost	Priority	Level	Low	Probable	Low	P6	L2
Severity	Likelihood	Remediation Cost	Priority	Level							
Low	Probable	Low	P6	L2							

Figure 4.2: Countermeasures(in Right side column) for the detected violations

4.4 Class level implementation

4.4.1 Introduction

Class granularity is an important granularity level in object-oriented programming since object-oriented programs are based on the hierarchy of classes, and well-defined and cooperating objects. Specifically, in Java programs, a class is a Java platform API library that defines a set of objects that share a common structure and behaviour. In Java programming language, all the code written by developers should be included inside a particular class which means that in order to have any piece of source code, a class is needed. Hence class level granularity is a must for any Java program. This granularity level covers the code written inside a class but lies outside a method and also belongs to the `java.lang` package that doesn't need to be imported explicitly.

Table 4.3 mentions the five secure coding rules supported by the framework that fall into the class granularity level. A justification is also provided to explain as to why they belong to class level granularity.

Table 4.3: Justification of class level secure coding rules

Main category	Secure coding rule	Justification
Methods (MET)	MET09J	Both equals and hashCode method belongs to <code>java.lang.Object</code> package which is auto-imported. In Java, <code>java.lang.Object</code> class requires that any two objects that are compared using the <code>equals()</code> method must produce the same integer result when the <code>hashCode()</code> method is invoked. The violation occurs when the <code>hashCode()</code> is not defined.
Object Orientation (OBJ)	OBJ05J	This secure coding rule states that accessor methods (getter methods) should not return mutable class private members(eg- private global variables) without making them defensive. The violation occurs when referencing to private mutable class members are returned.
Object Orientation (OBJ)	OBJ01J	This secure coding rule states to limit the access of fields such as global variables by using access modifiers such as private. The violation occurs if the accessibility of class variables are not limited.
Object Orientation (OBJ)	OBJ10J	This secure coding rule states to use the final keyword for public static non-final fields(eg- Global variables).

		I.e to make those fields into constants so that their values cannot be changed by an attacker. The violation occurs if the public static class variables are not declared as final.
Declarations and Initialization (DCL)	DCL00J	This secure coding rule violation arises due to the incorrect order of initialization of static field triggers inside a class but outside a method. This secure coding rule involves static class variables.

4.4.2 Implementation procedure

Class level implementation was carried out to add the ability to detect class level secure coding rule violations to the framework. A systematic approach was followed during the class level implementation. Secure coding rules that belong to class level were selected as mentioned in *Table 4.3*. This classification was done according to the class level classification criteria that was defined in *Table 3.1*. Single secure coding rule may require several source code fragments of different granularity levels. In that case, the granularity of the secure coding rule was selected according to its required code fragment with the highest level of granularity. If the highest level of granularity of code fragments required by a secure coding rule is class level, it was classified under class level granularity. Secure coding rules that belong to class level granularity will only use class level code fragments or method level code fragments.

Subsequently, the code fragments required by secure coding rules that belong to class level granularity were extracted by traversing the AST of the source file that is needed to be analysed. JavaParser library was used for this purpose of creating an AST and traversing through it. The code written for extracting code fragments were included in ClassLevelCodeFragment class or MethodLevelCodeFragment class according to its granularity level because the code fragments needed by class level secure coding rules can also be method level code fragments.

After extracting relevant code fragments, the secure coding rules that belong to class level granularity were represented by algorithms. These algorithms were then implemented using Java programming language and included into a single class called ClassLevelViolationDetector. Finally, the ClassLevelViolationDetector class was integrated into the framework using the factory design pattern. Countermeasures relevant to the secure coding rules that belong to class level granularity were also added by a separate class.

For example, let's consider how the above approach was applied to implement the secure coding rule that states to prevent class initialisation cycles (DCL00-J). This secure coding rule requires two code fragments. The first code fragment required is the list of class level variable declarations and the second code fragment is the list of assign statements inside the constructor. These required code fragments were extracted using JavaParser library. Then the secure coding rule was represented using an algorithm and then implemented using Java. Following the algorithm used to implement this secure coding rule.

1. Check whether a constructor is defined.
2. Get the full declaration of class variables along with their line number. (Line number is required because in this rule the order of class variables are important as well)
3. If a class variable has created an object of the same class, get class variables defined after that object creation.
4. Check whether these variables are used in expressions inside the constructor and if so DCL00-J is violated.

Finally, the code fragments, secure coding rule implementation and countermeasures were added to the framework respecting the design. Following figures illustrate as to how the class level secure coding rule violations were detected by the framework.

```

public class Cycle {
    private final int balance;
    private static final Cycle c = new Cycle();
    private static final int deposit = (int) (Math.random() * 100); // Random deposit
    public int total; // Number of elements
    private Date d;
    public static FuncLoader m_functions;
    public Cycle() { balance = deposit - 10; // Subtract processing fee }
    void add() {
        if (total < Integer.MAX_VALUE) {
            total++;
        }
    }
    public MutableClass() {
        d = new Date();
    }
    public Date getDate() {
        return d;
    }
    public static void main(String[] args) { System.out.println("The account balance is: " + c.balance); }
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
    }
}

```

Figure 4.3: Detection of the mentioned class level secure coding rules(in orange colour)

Violated Rules	Rule Description			
MET09-J violated at line [25]	MET09J : Classes that define an equals() method must also define a hashCode() method			
Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	L3
Click here for more details				

Figure 4.4: Countermeasures (in Right side column) for the detected violations

4.5 Package level implementation

4.5.1 Introduction

The proposed solution consists of three main granularity levels namely method level, class level and package level. As mentioned previously, SEI CERT Coding Rules have been classified into above three granularity levels based on classification criteria. This classification helps to implement relevant secure coding rules correctly.

Package level mainly focuses on secure coding rules which are neither belongs to method level nor class level. That is source code fragments that are imported from external classes and which lie outside the java.lang package falls into package level. *Table 4.4* shows secure coding rules implemented under package level and their relevance to package level based on code fragments with justifications.

Table 4.4: Justification of Package level secure coding rules

Main category	Secure coding rule	Justification
Thread Application Programming Interface (THI)	THI00-J	A violation occurs due to invoking of run() method directly Inside a class that implements the Runnable interface.
Serialization (SER)	SER01-J	A violation of the rule occurs due to incorrect method signatures of writeObject(), readObject(), readResolve() and writeReplace() methods. Serializable interface belongs to java.io Package.
Numeric types and Operations (NUM)	NUM10-J	A violation occurs due to passing double values instead of string values to the BigDecimal constructor which belongs to the BigDecimal class of java.math package.
Platform Security (SEC)	SEC07-J	A violation occurs due to overriding getPermissions() method without invoking super.getPermissions() method. getPermissions method belongs to URLClassLoader in java.net package.
Input Output (FIO)	FIO02-J	This rule violates due to delete() method is used to delete a specified file, but it does not indicate its success. It only throws SecurityException. No other exceptions are thrown, so the deletion can silently fail.

		This method includes in file class which is inherited from java.io package.
--	--	---

4.5.2 Implementation procedure

Initially, a set of secure coding rules that fall under package level were identified. Subsequent to the successful completion of designing of violation detection algorithms for selected secure coding rules, package level implementation was carried out. A mechanism was required to extract source code fragments which are used by violation detection algorithms. To achieve this task, a JavaParser library is used to generate an AST from the source code. The source code fragments were obtained by traversing along the generated AST. HashMaps and ArrayList are used to store relevant source code fragments for each secure coding rule.

A single data structure can contain common code fragments which are used by several secure coding rules. Single secure coding rule may use several code fragments which belongs to different granularity levels. Package level consist of PackageLevelCodeFragment and PackageLevelViolationDetector class. Package Level Code fragment class consist of data structures for store relevant code fragments and method which can be used to extract relevant source code fragments from the source code.

Then secure coding rules were represented as violation detection algorithms in order to detect violations in the source code. *PackageLevelViolationDetector* class contains package level secure coding rules implemented as violation detection algorithms. These violation detection algorithms used source code fragments of different granularity levels to detect violations.

Finally completed secure coding rules of package level need to be integrated into the framework. To achieve this factory design pattern was used to incorporate three granularity levels and build single unit (DetectorFactory class) which can be used to detect secure coding rule violations which fall into any of the above three levels. Countermeasures for the package level secure coding rules were also added to the framework to guide the user to resolve the detected violation.

Consider the third secure coding rule in Table 4.4 (NUM10-J). It states not to construct BigDecimal objects from floating-point literals. This is primarily because literal decimal floating-point numbers cannot always be precisely represented as an IEEE 754 floating-point value. Consequently, the BigDecimal(double value) constructor must not be passed a floating-point literal as an argument when doing so results in an unacceptable loss of precision. To implement this rule first need to identify relevant code fragments (Object creational expressions and arguments passed to the constructor). Then the list of object creational expression along with their

arguments is obtained and passed to the constructors. If a BigDecimal object exists, which is created from a double value passed as a parameter, then the rule NUM10-J is violated.

```

public static void main(String[] args) {
    THI00J foo = new THI00J();
    new Thread(foo).run();

    System.out.println(new BigDecimal( val: 0.1));

    File file = new File(args[0]);
    file.delete();
}

protected PermissionCollection getPermissions(CodeSource cs) {
    PermissionCollection pc = new Permissions();
    // Allow exit from the VM anytime
    pc.add(new RuntimePermission( name: "exitVM"));
    return pc;
}

public class SER01J_2 implements Serializable {
    private final long serialVersionUID = 123456789;

    private SER01J_2() {
        // Initialize..
    }

    private Object readResolve() {
        // ..
    }
}

```

Figure 4.5: Detection of the mentioned package level secure coding rules(in orange colour)

Secure Coding Plugin															
Violated Rules		Rule Description													
THI00J violated at line [14] SER01J violated at line [35] NUM10J violated at [16] SEC07J violated at [22] FIO02J violated at [19]					THI00J : Do not invoke Thread.run() <table border="1"> <thead> <tr> <th>Severity</th><th>Likelihood</th><th>Remediation Cost</th><th>Priority</th><th>Level</th></tr> </thead> <tbody> <tr> <td>Low</td><td>Probable</td><td>Medium</td><td>P4</td><td>L3</td></tr> </tbody> </table> Click here for more details	Severity	Likelihood	Remediation Cost	Priority	Level	Low	Probable	Medium	P4	L3
Severity	Likelihood	Remediation Cost	Priority	Level											
Low	Probable	Medium	P4	L3											

Figure 4.6: Countermeasures(in Right side column) for the detected violations

4.6 Alignment of implementation with system design

In the design chapter System modelling section which mainly includes the class diagram, it was mentioned that two main design patterns namely *Singleton* and *Factory* design patterns are to be used during the implementation of the solution. The main reasons for the use of design patterns are to improve the system design and this product based project being an industry level Software Engineering project.

1. Singleton design pattern

According to the class diagram, singleton pattern is to be used in the three source code fragment classes namely MethodLevelCodeFragment.java, ClassLevelCodeFragment.java and PackageLevelCodeFragment.java classes and the LiveParser.java class which involves reading the source code typed in the IDE in a real-time manner. The main reason for this is that to access the relevant data structures a single instance of each class is sufficient and also, creating too many instances of the same class would increase the overhead and degrade the performance of the framework by consuming an excess amount of main memory and increased Central Processing Unit (CPU) usage.

The following Listings provide evidence for the use of Singleton design patterns in the three source code fragment classes based on the *Lazy initialisation* of the Singleton design pattern.

Listing 4.1: Singleton initialization in MethodLevelCodeFragment.java class

```
Private static MethodLevelCodeFragment instance;  
Private MethodLevelCodeFragment();  
public static MethodLevelCodeFragment getInstance(){  
    if(instance == null){  
        instance = new MethodLevelCodeFragment();  
    }  
    return instance;  
}
```

Similarly, singleton initializations have been made in the other 3 classes namely ClassLevelCodeFragment.java, PackageLevelCodeFragment.java and LiveParser.java.

2. Factory design pattern

The primary purpose of using the Factory design pattern in the system design is to *integrate* the implementations of the three granularity levels and to ensure that they function in an effective, consistent manner after the integration. To incorporate the factory design pattern into the implementation process of the framework successfully, the DetectorFactory.java class and ViolationDetector.java interface was specifically created.

The DetectorFactory.java class primary focuses on *defining* the types of violation detector classes available in the framework namely MethodLevelViolationDetector class, ClassLevelViolationDetector class, and PackageLevelViolationDetector class. The main focus of this is to enable the creation of the objects of these violation detector classes appropriately in the LiveParser.java class and also support the future extensibility of the framework by allowing a developer to add a new violation detector granularity level easily.

Listing 4.2: Implementation of the Factory design pattern in DetectorFactory.java class

```
public class DetectorFactory {  
  
    public ViolationDetector getViolatorType(String ViolatorType){  
        if(ViolatorType == null){  
            return null;  
        }  
        if(ViolatorType.equalsIgnoreCase("MethodLevelViolationDetector")){  
            return new MethodLevelViolationDetector();  
        } else if(ViolatorType.equalsIgnoreCase("ClassLevelViolationDetector")){  
            return new ClassLevelViolationDetector();  
        } else if(ViolatorType.equalsIgnoreCase("PackageLevelViolationDetector")){  
            return new PackageLevelViolationDetector();  
        }  
        return null;  
    }  
}
```

The ViolationDetector.java interface supports the use of factory design pattern by defining the abstract methods that are overridden in the three violation detector classes namely MethodLevelViolationDetector.java, ClassLevelViolationDetector.java and PackageLevelViolationDetector.java classes to implement the violation detection algorithms

corresponding to the relevant secure coding rules belonging to the three granularity levels. It also supports extensibility by enabling the easy addition of a new violation detection algorithm to the framework.

Listing 4.3: Implementation of Factory design pattern in ViolationDetector.java interface

```
public interface ViolationDetector {  
  
    String rule1Detection();  
    String rule2Detection();  
    String rule3Detection();  
    String rule4Detection();  
    String rule5Detection();  
  
}
```

Listing 4.4: Overriding of the relevant abstract methods in MethodLevelViolationDetector.java class to implement violation detector algorithm of secure coding rule NUM-9J

```
public class MethodLevelViolationDetector extends AnAction implements  
ViolationDetector{  
  
    public String rule1Detection(){  
        try {  
            rule1Detection=detectViolationNUM09J();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return rule1Detection;  
  
    }  
}
```

4.7 Countermeasures for violated guidelines

Countermeasures are included in the plugin-based framework as a separate module since it is convenient to maintain a separate module for them and these are required by secure coding rules of all three granularity levels. To store countermeasures for selected secure coding rules, a HashMap data structure is used. *Listing 4.5* shows the structure of the HashMap created to store countermeasure data.

Listing 4.5: Implementation of CountermeasureData HashMap in order to store countermeasure data of implemented secure coding rules.

```
public HashMap<String, String> CountermeasureData = new HashMap<String, String>();
```

As shown in *Listing 4.5* it is clear that two string <key,value> pairs are maintained in the CountermeasureData HashMap. Each rule has a unique name in its description [5]. For example consider the THI00J rule which is the package level first rule as mentioned in the *Package level implementation*. (This rule represents the first rule under the Thread APIs (THI) main category). This name is used as the key of the HashMap. A rule description is included as the value of the HashMap. Rule description consists of three main parts including *rule name*, *risk assessment* relevant to the rule and *a link* to the rule in the SEI CERT coding standard web page.

Secure Coding Plugin					
Violated Rules		Rule Description			
		THI00J : Do not invoke Thread.run()			
THI00-J violated at line [14]					
SER01-J violated at line [35]		Severity	Likelihood	Remediation Cost	Priority
NUM10J violated at [16]		Low	Probable	Medium	P4
SEC07J violated at [22]					L3
FI002J violated at [19]		Click here for more details			

Figure 4.7: Example countermeasures for the detected violations

Figure 4.7 illustrates how the risk assessment of each secure coding rule is represented in the ToolWindow using an HTML table. The detected violations are listed down under the violated rules heading in LHS column. Countermeasures for the detected violations are shown in the RHS column of the ToolWindow, and they are linked to the relevant secure coding rules. If a user clicks on a violated rule shown in the LHS column, then appropriate countermeasures will be displayed in the RHS column. If a user needs to know more details relevant to a particular secure coding rule, he/she can click on the link at the bottom. Then the user will be directed through the web browser to the relevant rule of the SEI CERT secure coding web page.

4.8 Test procedure

The primary purpose of conducting software testing for this project was to find defects in the solution and to ensure that the plugin-based framework operates as mentioned in the specification. I.e. to verify whether actual results are aligned with the expected results. In the current context there exist two main categories of testing methodologies namely, functional testing and non-functional testing. Functional testing primarily focuses on unit testing, integration testing, and system testing to verify that the implemented plugin-based framework provides accurate results. Automated unit testing focussing on the three granularity levels namely Method, Class and Package levels was performed using the TestNG framework.

Non-functional testing was primarily performed on *performance* and *reliability* aspects of the plugin-based framework. The performance testing was carried out using software profiling tools such as VisualVM, JProfiler to verify that the framework functioned in an optimised manner with low memory consumption, Central Processor unit consumption and low latency(response times). The reliability testing was manually conducted by all three members focussing on the three levels of granularity levels namely method, class and package levels. This reliability testing methodology was carried out using the code snippets provided by SEI CERT [5] to verify whether the implemented violation detection algorithms detected the respective secure coding rule violations. Thus ensuring the reliability of the framework providing accurate results.

4.9 Deployment

The solution of this product based project is a plugin-based framework for IntelliJ IDEA IDE and specific to Java programming language. In the plugin development process, an IDE instance along with the developing plugin is used to make sure that it works as intended. After successful completion of the plugin development process the framework needs to be deployed. This is mainly because prior to the usage of custom plugin it needs to be deployed: built, installed, and then enabled using the Plugin Manager. If the plugin module *does not* depend on any libraries, a .jar archive will be created. Else, a .zip archive will be created including all the plugin libraries specified in the project settings.

In order to deploy the plugin-based framework, it needs to be verified that it works correctly and provides the expected results. The proper working of the framework was achieved by installing a built of the plugin from disk on a fresh instance of IntelliJ IDEA community edition. Subsequently, manual testing was conducted against some compliant and non-compliant codes to verify its correct functionality. After confirming that the framework functions as intended, the plugin version needs to be updated, as the JetBrains plugin repository will not accept multiple artefacts with the same version. After the next step was to upload the plugin to the plugin repository.

After submitting the secure coding plugin-based framework, it was successfully uploaded to the JetBrains plugins repository. After the uploading process, there was a pending JetBrains' review conducted by their plugin evaluating panel that assesses the standards of the plugin. This process took two business days, and after the evaluation conducted by IntelliJ IDEA authorities, approval was granted, and the framework was made publicly available. In the meantime, authors of the plugin can always access the plugin via direct link provided by them. However, it will not be publicly available (e.g. in search results and the product) until approved by JetBrains. All new plugins are typically reviewed within two business days. Authors will receive a notification as soon as the status of this review changes. Once the Framework for Secure Coding plugin has been approved by JetBrains plugin administration team, it was publicly available to download in the JetBrains Plugin Repository. *Figure 4.8* illustrates Framework for secure coding plugin deployed in the JetBrains Plugin Repository.

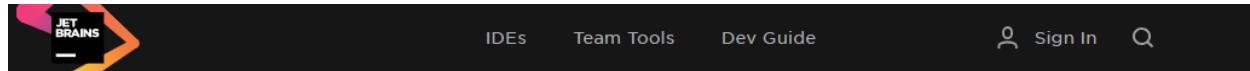


Figure 4.8: Framework for secure coding plugin deployed in the JetBrains Plugin Repository

As mentioned in the use case diagrams included in Appendix D, the plugin can be installed to the IntelliJ IDEA IDE in two ways. The first method is installing the plugin from disk. In order to do this first need to make sure that the plugin is compatible with the IDE version and if not then the plugin will not be able to install and run in the IDE. If it is compatible with the IDE version, then it can be downloaded from the JetBrains plugin repository. After successful installation, the plugin needs to be enabled using the Plugin Manager. The second method is to browse repositories for the plugin. This method is quite easy as a user can directly download and install the plugin at once. The plugin will appear in the search result only if it is compatible with the IDE version. If the plugin is not compatible with the IDE version, then it will not appear in the search results. As previously mentioned after a successful installation, the plugin needs to be enabled using the Plugin Manager.

Framework for Secure Coding downloads All updates

Unique downloads only

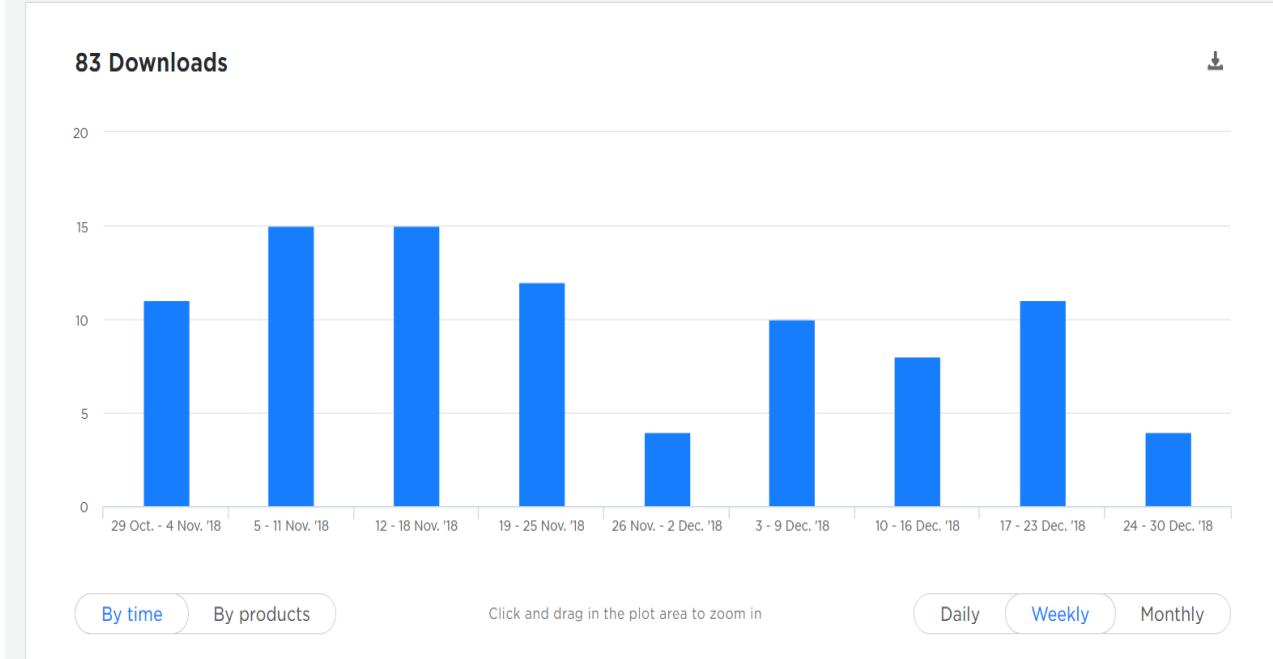


Figure 4.9: Downloads of deployed plugin on weekly basis

4.10 Summary

This chapter presented a detailed description of the solution provided, development approaches followed, how each component was developed, tools and technologies used in implementing the plugin-based framework. This chapter also discusses the overall process of IntelliJ IDEA plugin development. A detailed description of the implementation of 3 granularity levels was also presented in this chapter. Further, this chapter discussed decisions taken during the implementation process and justification for alignment of implementation with system design. In addition to that, this chapter also explained how the countermeasures for violated guidelines were implemented and integrated into the plugin-based framework.

Chapter 5 : Evaluation and Results

5.1 Introduction

The evaluation process was carried out to assess the solution of the product based project in order to verify whether the intended requirements have been met and are up to relevant standards. The main focus is to ensure that the plugin-based framework functions in a way that the expected objectives are achieved. In the evaluation process, four methods were selected in order to assess various aspects of the framework. They are Project-based evaluation, Extensibility based evaluation, Performance-based evaluation and User-based evaluation.

The Extensibility based evaluation methodology mainly assess the extent to which the existing framework could be extended with minimal changes to the existing source code base, by users in the future. Extensibility mechanism has been achieved by the use of a common data structure and with the support of design patterns used in the system design. 4 main aspects of extensibility were considered in this method of evaluation and they are addition of a new secure coding violation detection algorithm, addition of a new source code granularity level, modification of existing secure coding algorithms, and modification of existing data structures.

Project-based evaluation methodology primarily focuses on evaluating the extent to which the framework is capable of detecting secure coding rule violations found in existing open source projects. Other aims of this method of evaluation are to identify the most commonly violated secure coding rules by software developers while writing the source code and assessing the accuracy of the detected violations via comparison with other existing static analysis tools.

IntelliJ IDEA IDE consumes a considerable amount of main memory when it is being executed. Optimising the memory usage during the execution is vital to maintain the performance of the framework when it is being used. The Performance-based evaluation methodology mainly evaluates the system design of the framework by assessing the performance-related factors such as memory usage, Central Processing Unit(CPU) usage, response time(latency), etc to determine the extent to which the system design improved them.

The user-based evaluation was carried out in order to evaluate the usability aspects of the plugin-based framework based on the responses provided by the users that downloaded the plugin after its deployment to the JetBrains plugin repository. The main intention of this evaluation methodology is to assess the usability aspects of the plugin-based framework along with focussing on the improvements to be made in the future versions to be released.

5.2 Project based evaluation

5.2.1 Introduction

The project-based evaluation was carried out to check whether the secure coding plugin can detect secure coding rules violated by open source Java projects on Github code repository and to verify whether the detected violations are correct(True positive) or incorrect and use those results to measure the accuracy of algorithms used to detect secure coding rule violations.

This evaluation methodology assisted in *identifying false positives* and *false negatives* of the secure coding plugin-based framework if there is any. False positive means when the secure coding plugin detects and shows a secure coding rule violation that is not there. When the secure coding plugin does not detect a secure coding rule violation that is there, it is a false negative. Opposites of these two terms are true positives and true negatives. True positives are correctly detected violations, and true negatives mean not showing any violations even if there does not exist any.

Following is the procedure that was followed when conducting the project-based evaluation.

1. A set of open source projects in Java programming language found on Github code repository were selected [32].
2. Source files of these selected projects were scanned using the secure coding plugin-based framework to get the set of secure coding rule violations that it detects.
3. Source files were manually reviewed to identify any false positives, false negatives and then the correctness of the detected violations. During this step of manually evaluating the source files, they were manually tested against all the 15 secure coding rules supported by secure coding plugin ignoring the results given by the plugin at the first place. This approach allowed this evaluation technique to find false negatives if there exists any.

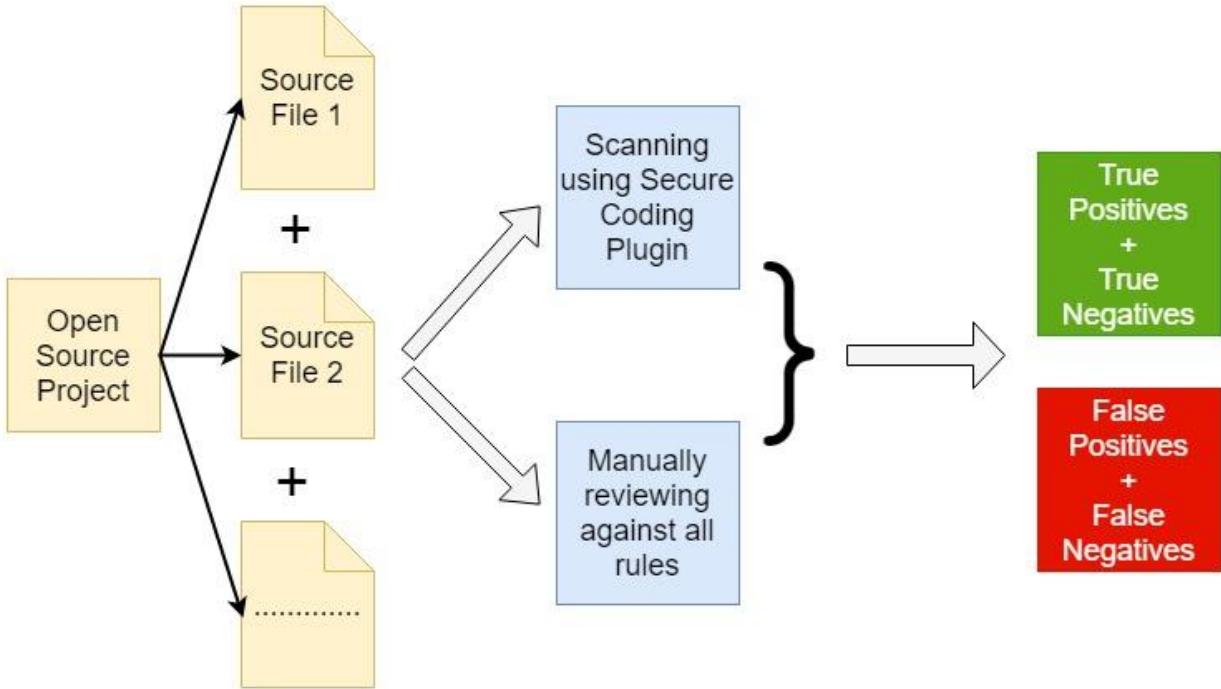


Figure 5.1: Project based evaluation procedure diagram

5.2.2 Results of Project based evaluation

Shown below are the results of the project based evaluation conducted. *Table 5.1* shows the projects considered, source files scanned, the results obtained when scanned and the justification after manually reviewing the detected violations. Appendix E contains screenshots of how secure coding plugin detected violations in open source projects mentioned *Table 5.1*.

Table 5.1: Open source projects considered for project based evaluation

Rule	Project	File name	Justification
ERR08	Arthas	TelnetConsole.java	In this code "Exception" is caught at seven places which violate the rule stating that programs must not catch NullPointerException or any of its ancestors namely RuntimeException, Exception, or Throwable. The secure coding plugin detected this vulnerability at all of these seven places.

		Arthas.java (Figure 5.2)	In this code, "Throwable" is caught and it violates the rule stating that programs must not catch NullPointerException or any of its ancestors namely RuntimeException, Exception, or Throwable. The secure coding plugin detected this vulnerability.
NUM09J	griDraw	PlotThread.java	In this code, two <i>floating-point variables</i> called "gsx" and "gsy" are used as loop counters. It violates the secure coding practice that states not to use floating-point variables as loop counters. The secure coding plugin detected this vulnerability in both of its occurrences.
ERR07J	Processing Spoon	UTCompiler.java	In this code, "RunTimeException" is thrown at two places. It violates the secure coding practice that states not to throw RuntimeException, Exception, or Throwable. The secure coding plugin detected this vulnerability in both of its occurrences.
		JDTBasedSpoon Compiler.java	In this code, "RunTimeException" is thrown at six places. It violates the secure coding practice that states not to throw RuntimeException, Exception, or Throwable. The secure coding plugin detected this vulnerability at all of its occurrences.
ERR04J	sonar-java	ReturnInFinally Check.java	In this code, return, break, continue and throw statements are used inside the finally block in multiple places. It violates the secure coding practice that states not to complete abruptly from a finally block. The secure coding plugin detected this vulnerability at all of its occurrences.
EXP02J	eureka	EurekaJacksonC odecIntegrationT est.java	In this code Object.equals() method is used to compare two character arrays. It violates the secure coding practice that states to use Arrays.equals() method when comparing two arrays. The secure coding plugin detected this vulnerability.
MET09J	12B	IntegerStack.java	In this code equals() has been given a new implementation but hashCode() is not defined. It violates the secure coding practice that states an equals() method is defined, hashCode() method should also be defined. The secure coding plugin detected this vulnerability.
OBJ05J	demo-	SerializableOpti	In this code, there is a private mutable class

	serialize-optional	onal.java	member called optional. This variable is also used inside the constructor which means it can be assigned with a value when creating an object. There is a public method called asOptional() that returns this variable. It violates the secure coding practice that states not to return references to private mutable class members. The secure coding plugin detected this vulnerability.
OBJ01J	eclipse.jdt.core	JavaModelManager.java	This code has declared several public class variables. For example, consider public integer variable called rawTimeStamp. It is used for different arithmetic operations like increment but as it is a public field, it can be altered by a client code. It violates the rule that states to limit accessibility of fields. The secure coding plugin detected this vulnerability.
OBJ10J	Arthas	GlobalOptions.java	This code has declared 8 non-final public static variables. It violates the rule that states not to use public static non-final fields. The secure coding plugin detected all these vulnerabilities.
		PlainTextHandler.java	This code has declared a non-final public static variable called NAME. It violates the rule that states not to use public static non-final fields. The secure coding plugin detected this vulnerability.
DCL00J	java-puzzlers	Puzzle49LargerThanLife.java	The Puzzle49LargerThanLife class declares a public static final class variable, which is initialized to a new instance of the Puzzle49LargerThanLife class. Another class variable called beltSize is initialized after that which means the initialization of the Puzzle49LargerThanLife instance class variable happens before the runtime initialization of the beltSize field because it appears lexically before the initialization of the beltSize field. The value of beltSize seen by the constructor, when invoked during the static initialization of Puzzle49LargerThanLife instance, is the initial value of integer variable beltSize (0) rather than the value assigned inside the constructor. It violates the rule that states to prevent class initialization cycles. The secure coding plugin detected this vulnerability.

THI00J	Thread_De mo	Foo.java	This code has explicitly invoked run() in the context of the current thread. It violates the rule that states not to invoke Thread.run(). The secure coding plugin detected this vulnerability.
SER01J	demo-serialize-optional	SerializableOpti onal.java	This code has declared a readResolve() method as private after implementing the Serializable interface. It violates the rule that states not to deviate from the proper signatures of serialization methods. The secure coding plugin detected this vulnerability.
NUM10J	cbioportal	SignificantlyMut atedGenesContro llerTest.java	In this code, BigDecimal() is passed with 0.1 or 0.2 as an argument. They are floating-point literals. This can make an unacceptable loss of precision. It violates the rule that states not to construct BigDecimal objects from floating-point literals. The secure coding plugin detected this vulnerability at all of its occurrences.
SEC07J	j2objc	SecureClassLoad er.java	This code which is intended to create a custom class loader overrides the getPermissions() method but does not call its superclass's more restrictive getPermissions() method. It violates the rule that states to call the superclass's getPermissions() method when writing a custom class loader. The secure coding plugin detected this vulnerability.
FIO02J	Spoon	JDTBasedSpoon Compiler.java	This code attempts to delete a file but gives no indication of its success. It violates the rule that states to detect and handle file-related errors. The secure coding plugin detected this vulnerability.

Figure 5.2 is a screenshot captured when Arthas.java file of Arthas project (refers to the Table 5.1) was scanned using secure coding plugin. It shows how ERRO8 violation was detected by the secure coding plugin and displayed to the user.

The screenshot shows a Java code editor with the file `Arthas.java` open. The code contains a `main` method with a try-catch block. The catch block is highlighted with a yellow background, and a tooltip message appears: "AnsiLog.error('Do not catch NullPointerException or any of its ancestors')". Below the editor, a "Secure Coding Plugin" interface is displayed, showing the violated rule `ERR08-J` at line [97]. The rule description is "ERR08J : Do not catch NullPointerException or any of its ancestors". A table provides severity information: Severity (Medium), Likelihood (Likely), Remediation Cost (Medium), Priority (P12), and Level (L1). A link "Click here for more details" is also present.

Rule Description				
Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Medium	P12	L1

Figure 5.2: *ERR08* violation as detected by the secure coding plugin

The open source Java project files that were used in the above task were further used to see whether the secure coding rules violated by them are detected by already existing popular static code analysis tools. The most popular static code analysis tools for Java are *SonarQube*, *SonarLint*, *FindBugs* and *CheckStyle*. *Table 5.2* shows the results when these open source project files were scanned using these tools and secure coding plugin-based framework.

Table 5.2: Detection of secure coding rules by static code analysis tools

Rule	Secure Coding Plugin	SonarQube	SonarLint	FindBugs	CheckStyle
ERR08	Detected	Not Detected	Not Detected	Not Detected	Not Detected
NUM09J	Detected	Not Detected	Not Detected	Not Detected	Not Detected
ERR07J	Detected	Detected	Detected	Not Detected	Not Detected
ERR04J	Detected	Detected	Detected	Not Detected	Not Detected

EXP02J	Detected	Detected	Detected	Not Detected	Not Detected
MET09J	Detected	Detected	Detected	Not Detected	Not Detected
OBJ05J	Detected	Not Detected	Not Detected	Not Detected	Not Detected
OBJ01J	Detected	Detected	Detected	Not Detected	Not Detected
OBJ10J	Detected	Detected	Detected	Not Detected	Not Detected
DCL00J	Detected	Not Detected	Not Detected	Not Detected	Not Detected
THI00J	Detected	Detected	Detected	Not Detected	Not Detected
SER01J	Detected	Not Detected	Not Detected	Not Detected	Not Detected
NUM10J	Detected	Detected	Detected	Not Detected	Not Detected
SEC07J	Detected	Not Detected	Not Detected	Not Detected	Not Detected
FIO02J	Detected	Detected	Detected	Not Detected	Not Detected

Table 5.3 shows the summary of the above analysis done using already existing static code analysis tools.

Table 5.3: Summary of the detection of secure coding rules by static code analysis tools

Tool name	Number of rules supported out of 15 rules implemented in the secure coding plugin	On the fly analysis?
Secure Coding Plugin	15	Yes
SonarQube	9	No
SonarLint	9	Yes
FindBugs	0	No
CheckStyle	0	No

5.2.3 Conclusion

The primary objective of the project based evaluation was to check the *accuracy* of the algorithms and the results given by the secure coding plugin. Results can be divided into four categories as *true positives*, *true negatives*, *false positives* and *false negatives*. Following is a summary of how the secure coding plugin was evaluated using open source trending Java projects on Github code repository.

Figure 5.3 shows that in most cases multiple violations were detected in the same source code and in some cases the similar violations were detected at numerous places of the same source code. Table 5.1 illustrates the results from the secure coding plugin which were manually reviewed and justified to be true in all the cases, and this further proves that the logic behind the algorithms used to detect secure coding rule violations by secure coding plugin-based framework is correct. Therefore it could be concluded that all the detected violations by secure coding plugin are true positives and the plugin does not give false positives.

The source files were manually reviewed against all *15 rules* supported by secure coding plugin to see whether the plugin did not detect any violations that are there. For example, to see whether the rule that states to prevent class initialisation cycles has been violated, the class variables defined, and constructor body of the source code was manually examined, and in this way, it could be identified whether the secure coding plugin gives any false negatives. After manually reviewing all the 15 rules, there were no false negatives found for the scanned source files, and therefore it can be concluded that secure coding plugin does not give any false negatives and hence it gives only true negatives.

Apart from identifying true positives, true negatives, false positives and false negatives by project based evaluation it was also found that the *most commonly violated secure coding rules* among the 15 rules supported by secure coding plugin were the rules that states, programs must not catch NullPointerException or any of its ancestors namely RuntimeException, Exception, or Throwable(ERR08) and not to use public static non-final fields (OBJ10).

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

F: Projects JDTBasedSpoonCompiler.java

JDTBasedSpoonCompiler.java

```

232
233     private Collection<? extends String> toStringList(List<SpoonFile> files) {
234         List<String> res = new ArrayList<String>();
235         for (SpoonFile f : files) {
236             if (f.isActualFile()) {
237                 res.add(f.toString());
238             } else {
239                 try {
240                     File file = File.createTempFile(f.getName(), ".java");
241                     file.deleteOnExit();
242                     IOUtils.copy(f.getContent(), new FileOutputStream(file));
243
244                     res.add(file.toString());
245                 } catch (IOException e) {
246                     throw new RuntimeException(e.getMessage(), e);
247                 }
248             }
249         }
250         return res;

```

Secure Coding Plugin

Violated Rules	Rule Description										
ERR08-J violated at line [125, 289, 441, 461, 739, 905] OBJ05-J violated at line [832] ERR07J violated at [126, 246, 682, 686, 690, 721] FIO02J violated at [359]	ERR07J : Do not throw RuntimeException, Exception, or Throwable										
	<table border="1"> <thead> <tr> <th>Severity</th> <th>Likelihood</th> <th>Remediation Cost</th> <th>Priority</th> <th>Level</th> </tr> </thead> <tbody> <tr> <td>Low</td> <td>Likely</td> <td>Medium</td> <td>P6</td> <td>L2</td> </tr> </tbody> </table> <p>Click here for more details</p>	Severity	Likelihood	Remediation Cost	Priority	Level	Low	Likely	Medium	P6	L2
Severity	Likelihood	Remediation Cost	Priority	Level							
Low	Likely	Medium	P6	L2							

Figure 5.3: Multiple rule violations and same rule violation at multiple places as detected by secure coding plugin

5.3 Extensibility based evaluation

5.3.1 Introduction

The Solution of this project mainly involves developing a framework. Extensibility is a mandatory key feature or a requirement that a framework should be comprised of. This type of evaluation involves the extent to which the Class diagram of the plugin-based framework supports its extensibility in order to accommodate future requirements such that the framework could be expanded easily without changes to the extending source codebase. When considering the Extensibility 4 major extensibility areas were considered. These aspects are as follows

5.3.2 Addition of a new secure coding violation detection algorithm

The violation detector algorithms corresponding to the implemented secure coding rules are stored in 3 Java classes based on their granularity levels. There are three code fragment classes corresponding to each violation detector classes that stores the source code fragments in various data structures such as ArrayLists and HashMaps. One or more data structures may be used for a particular secure coding violation detection algorithm.

In order to accommodate the addition of a new secure coding algorithm the implemented project solution uses a concept of a *common data structure* which has been introduced for each source code fragment class. It is a HashMap with String as the key(Store names of existing data structures) and Objects as the value(To store various data structures such as ArrayLists, HashMaps, etc.). I.e. a HashMap with the form of *HashMap<String, Object>* is being used as the common data structure. Three such common data structures are used in each source code fragment classes corresponding to each source code granularity level.

The object-oriented concept of Encapsulation has been used for implementing the common data structure approach. Here the common data structure has been declared with the private access modifier and relevant getter methods for each of the three common data structures have been declared in the three source code fragment classes. This supports the easy addition of new secure coding violation detection algorithms into the framework.

Listing 5.1: The common data structure and methods associated with it in MethodLevelCodeFragment.java class

```
private static Map<String, Object> methodLevelCommonHashMap = new  
HashMap<String, Object>();  
  
public static Map<String, Object> getMethodLevelCommonHashMap(){  
    return methodLevelCommonHashMap ;  
}
```

All the existing data structures used to store source code fragments in each granularity level are added to the common data structure corresponding to that granularity level. A new secure coding violation detection algorithm can be easily added to the framework by writing two java classes. I.e. one java class to store the code fragments related to the extended secure coding rule and the other java class to write the new secure coding violation detection algorithm. In the case where the necessary code fragments are already stored in the common data structure, then a separate class may need not to be written for the new secure coding violation detection algorithm since the code fragments could be directly obtained from the data structure.

Listing 5.2: Adding an existing data structure to the common data structure

```
public Map<Integer, ArrayList<Integer>> catchClause = new HashMap<Integer,  
ArrayList<Integer>>();  
  
methodLevelCommonHashMap.put("catchClause",new HashMap<Integer, Integer>());
```

After the new secure coding violation detection algorithm is written an object of the violation detection algorithm class needs to be instantiated in the LiveParser.java class and the relevant violation detection method needs to be called. The Num09J secure coding rule which was implemented in the initial code base was reimplemented using the extensibility mechanism. Successful results were achieved confirming that the *common data structure* based approach being a successful mechanism to extend the plugin-based framework to accommodate the addition of new secure coding violation detection algorithms.

Listing 5.3: Adding a new data structure of the ExtendeCodeFragments.java class to the common data structure

```
MethodLevelCodeFragment methodLevelInstance = new  
MethodLevelCodeFragment();  
  
public Map<String, Object> newCommonHashMap =  
methodLevelInstance.getcommonHashMap();  
  
public Map<Integer, ArrayList<Integer>> forCounter = new HashMap<Integer,  
ArrayList<Integer>>();  
  
public void addDataStructures(String name, Object O){  
  
    newCommonHashMap.put(name,O);  
  
}  
  
addDataStructures("forCounter",forCounter);  
  
BlockVisitor.visit(cu, (Map<Integer, Integer>)  
newCommonHashMap.get("forCounter"));
```

Listing 5.4: Writing a new secure coding violation detection algorithm in the ExtendedViolationDetector.java class.

```
public class ExtendedViolationDetector {  
  
    public void extenedViolation() throws IOException {  
        ExtendedCodeFragments cc = new ExtendedCodeFragments();  
  
        if(!((Map<Integer, Integer>)  
cc.newCommonHashMap.get("forCounter")).isEmpty()){  
  
            for(int i=1;i<((Map<Integer, ArrayList<Integer>>)  
cc.newCommonHashMap.get("forCounter")).size()+1;i++) {  
                System.out.println("The violations are at lines "+((Map<Integer,  
ArrayList<Integer>>) cc.newCommonHashMap.get("forCounter")).get(i));  
            }  
  
            System.out.println("Guideline Num09J is violated");  
            return;  
        }  
        System.out.println("Guideline not violated");  
    }  
}
```

5.3.3 Addition of a new source code granularity level

In the current context, the secure coding plugin-based framework caters three source code granularity levels namely Method, Class and Package levels. When the extensibility of the framework is considered, the addition of a new granularity level other than the previously mentioned three levels may also need to be focused. The approach for this has been already achieved by the System design represented by the class diagram of the plugin-based framework.

The use of the Factory design pattern has allowed the easy addition of a new source code granularity level into the framework. This could be simply achieved by adding the name of the new granularity level violation detector class into the DetectorFactory.java class which contains the names of the other three violation detector classes. Then the respective violation detector class could be created similar to the other existing violation detector classes in the LiveParser.java class.

Listing 5.5: Defining the new source code granularity level in DetectorFactory.java class

```
public ViolationDetector getViolatorType(String ViolatorType){  
    if(ViolatorType == null){  
        return null;  
    }  
    if(ViolatorType.equalsIgnoreCase("NewGranularityLevelViolationDetector")) {  
        return new NewGranularityLevelViolationDetector();  
    }  
}
```

5.3.4 Modification of existing secure coding violation detection algorithms

The secure coding violation detection algorithms corresponding to each granularity level are implemented in three separate violation detector classes namely MethodLevelViolationDetector.java, ClassLevelViolationDetector.java and PackageLevelViolationDetector.java. I.e. Secure coding violation detection algorithms falling under method level are written in the MethodLevelViolationDetector.java class of the framework. Along with time, the particular secure coding rules may be updated by SEI CERT, and thus the existing secure coding violation detection algorithms in the plugin-based framework may need to be changed accordingly, by committing minimal changes to the existing source code base.

This aspect of extensibility has also been achieved by the system design of the framework which involves following a *loose coupling* and *high cohesiveness* based approach. Loose coupling is achieved as a result of the secure coding violation detection algorithm class, corresponding to each granularity level being dependent only on the on the respective secure coding code fragment class and independent of all other classes. I.e. The MethodLevelViolationDetector.java class depends only on the source code fragments stored in the common data structure of the MethodLevelCodeFragment.java class. High cohesiveness or improved focuses is achieved by splitting the source code into three granularity levels namely method level, class level and package level. This has supported in achieving independence among secure coding violation detection algorithms belonging to granularity level.

Thus the system design has allowed any individual intending to change the existing secure coding violation detection algorithms in the future. Since the algorithms are independent among each other, they could be changed easily with a *minimum* negative impact on the other algorithms in the same granularity level or another granularity level.

5.3.5 Modification of existing data structures

Developers may intend to change the existing data structures used to store source code fragments based on future requirements. Modification of the data structures may primarily refers to either changing its *name* or the *datatype*. I.e. converting an ArrayList to a HashMap or vice versa. This aspect of extensibility which involves changing the existing data structures is achieved from the concept of the *common data structure*. The three common data structures of type HashMap<String, Object> that are instantiated in each of the three source code fragment classes namely MethodLevelCodeFragment.java, ClassLevelCodeFragment.java and PackageLevelCodeFragment.java can be used to modify the existing data structures used to store source code fragments corresponding to the relevant violation detection algorithms.

This extensibility aspect could be easily achieved as the common data structure used in this project is of type *HashMap<String, Object>*. The *String* data type is used for the key of the HashMap since the names of the data structures are to be stored. An *Object* is used as the value in order to store various data structure types such as ArrayLists and HashMaps. If a modification such as changing the datatype of an existing data structure is to be done, it could be easily achieved since HashMaps do not allow duplicate keys to be stored in it. A developer can easily change the data structure using the default *put()* function of the HashMap with the name of the existing data structure with a different data type as the existing data structure stored in the common data structure will be overridden by the new data structure since the name will be used. If the name of the data structure is to be changed the default *get()* method and *put()* method may be used. Thus it could be concluded that the use of a common data structure has ensured the modification of existing data structures by name and the data type.

Listing 5.6: Modification of the data type of existing data structures

```
public Map<Integer, ArrayList<Integer>> catchClause = new HashMap<Integer,  
ArrayList<Integer>>();  
  
commonHashMap.put("catchClause",new HashMap<String,Integer>());  
  
commonHashMap.put("catchClause",new ArrayList<String>());
```

5.4 Performance based evaluation

5.4.1 Introduction

The solution of this project primarily focuses on the fly detection of secure coding rule violations in the source code written in Java programming language. It may be necessary to evaluate the performance of the plugin with respect to the *size of the source code* and its *complexity*. Consuming a *tremendous amount* of time to detect violations when large source code is present in the IDE may reduce the effectiveness and usefulness of the plugin.

Usually IntelliJ IDEA IDE consumes a considerable amount of memory and Central Processing Unit(CPU). As a result, the performance may also need to be evaluated with respect to the *main memory* and *CPU usage* compared with the source code size with the use of suitable *Software profiling* tools. *VisualVM* and *JProfiler* were used as software profiling tools in this evaluation methodology.

In order to achieve this task performance has been evaluated in terms of the *time required* for the analysis to complete(time is taken to detect violations in the source code) commonly referred to as *latency*, *main memory usage*, and CPU usage. These criteria assisted in verifying whether the plugin-based framework efficiently used its resources and how it performed on different source codebases.

In addition to the above-mentioned performance evaluation criteria, main memory and CPU consumption of the plugin-based framework was also evaluated using design patterns used version of the framework and design patterns not used version. The main focus of this performance evaluation criteria is to assess the contribution of the system design in improving the performance of the framework. The violation detection of *six* secure coding rules in which *two* rules from each of the three granularity levels were considered in the performance evaluation methodology.

5.4.2 Results of performance based evaluation

Table 5.4: Memory and CPU consumption with design patterns and without design patterns for individual rule

Secure coding rule	Design Patterns Used Version		Without Design Patterns Used Version	
	Average Memory consumption	Average CPU consumption	Average Memory consumption	Average CPU consumption
ERR08J	 175.13 MB	 6.97 %	 218.00 MB	 50.75%
NUM09J	 165.33 MB	 10.53 %	 173.43 MB	 30.34 %
OBJ05J	 190.85 MB	 10.34 %	 183.23 MB	 29.56%
MET09J	 186.75 MB	 10.34 %	 195.68 MB	 35.83%
THI00J	 172.77 MB	 7.02 %	 179.12 MB	 20.40 %
SER01J	 175.65 MB	 9.66 %	 184.43 MB	 20.98 %

Table 5.4 illustrates average memory and CPU consumption for each rule which has been implemented in the plugin. According to *Table 5.4* all the rules in the design pattern used version consumes relatively less amount of main memory than in version without design patterns. Likewise, the design pattern used version consumes *significantly less* amount of CPU compared to the version without design patterns.

In the design pattern used version each violation detector class only need a single instance of relevant code fragment class. As a result, rules which fall into the same granularity level can reuse this code fragments for violation detection without regenerating them. Moreover, these code fragment objects have declared as global which can be accessible by all the methods in that class instead of declaring them as local objects. Thus it could be concluded that this will eventually result in less amount of memory consumption.

Table 5.5: Memory and CPU consumption with design patterns and without design patterns for above six rules.

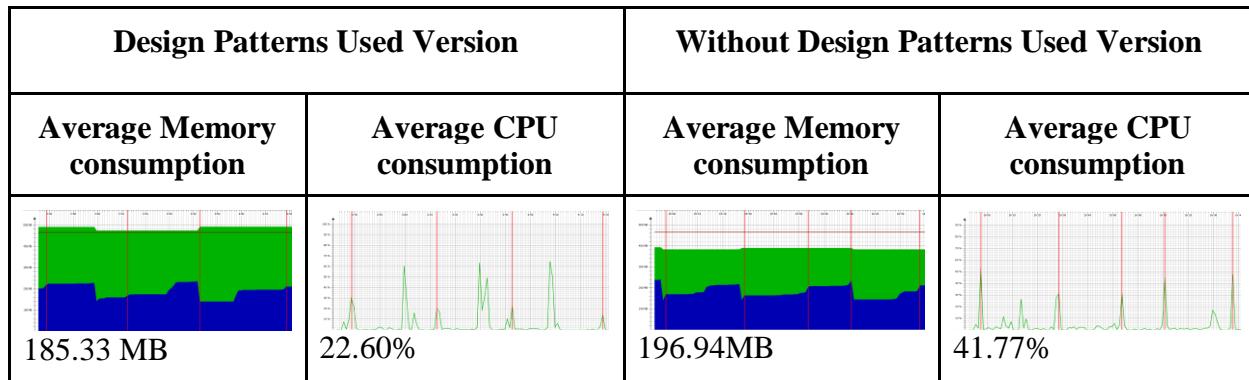


Table 5.5 shows that the design pattern used version consumes less amount of main memory and CPU compared to the version without design patterns. As mentioned earlier, since less number of objects are created of a violation detector class are created with the support of design patterns, CPU load is also small in design pattern used version. The above results also justify that adoption of design patterns have resulted in efficient consumption of CPU and main memory.

Table 5.6: Average response time with and without design patterns for individual rule.

Secure coding rule	Average response time with design patterns (Nanoseconds)	Average response time without design patterns (Nanoseconds)
ERR08J	<pre>Start time: 776383862925066 End time: 776383863311962 Time taken in nano seconds: 386896 . . . 407311.25</pre>	22973794.00

NUM09J	<pre>:runId=8 Start time: 776083388020418 Guideline ERRO8J has been violated! End time: 776083388710315 Time taken in nano seconds: 689897 Start time: 776093119953786 Guideline ERRO8J has been violated! End time: 776093120061778 Time taken in nano seconds: 107992 Start time: 776098123531346 Guideline ERRO8J has been violated! End time: 776098123644693 Time taken in nano seconds: 113347 Start time: 776106841153583 Guideline ERRO8J has been violated! End time: 776106841233015 Time taken in nano seconds: 79432</pre> <p>363133.50</p>	5437295.50
OBJ05J	<pre>:runId=8 Start time: 777191076637067 End time: 777191078976292 Time taken in nano seconds: 2339225 Start time: 777207014237946 End time: 777207014414660 Time taken in nano seconds: 176714 Start time: 777219410447884 End time: 777219410625044 Time taken in nano seconds: 177160 Start time: 777228190100642 End time: 777228190250134 Time taken in nano seconds: 149492 Start time: 777239877330355 End time: 777239877465568 Time taken in nano seconds: 135213</pre> <p>911124.50</p>	7775405.75
MET09J	<pre>:runId=8 Start time: 777429795198653 End time: 777429795775204 Time taken in nano seconds: 576551 Start time: 777443775739174 End time: 777443775842703 Time taken in nano seconds: 103529 Start time: 777450849937503 End time: 777450850097259 Time taken in nano seconds: 159756 Start time: 777457645739928 End time: 777457645902362 Time taken in nano seconds: 162434 Start time: 777466706698118 End time: 777466706785582 Time taken in nano seconds: 87464</pre> <p>716337.75</p>	17224237.25
THI00J	<pre>:runId=8 Start time: 777598489014076 End time: 777598489719591 Time taken in nano seconds: 705515 Start time: 777609012937137 End time: 777609013075920 Time taken in nano seconds: 138783 Start time: 777623672983734 End time: 777623673086817 Time taken in nano seconds: 103083 Start time: 777633850634986 End time: 777633850723789 Time taken in nano seconds: 88803 Start time: 777648200626708 End time: 777648200711495 Time taken in nano seconds: 84787</pre> <p>259046.00</p>	5745540.25

SER01J	<pre>:runId Start time: 777970732608985 End time: 777970733218558 Time taken in nano seconds: 609573 Start time: 777982837143610 End time: 777982837285517 Time taken in nano seconds: 141907 Start time: 777990487884625 End time: 777990487996187 Time taken in nano seconds: 111562 Start time: 777996119686484 End time: 777996119788229 Time taken in nano seconds: 101745 Start time: 778004568245621 End time: 778004568346919 Time taken in nano seconds: 101298</pre> <p>249563.50</p>	4561981.75
--------	---	------------

It could be seen from *Table 5.6* that design patterns used version has *less response time (low latency)* compared to the version without design patterns. According to this table, it is clear that design patterns used version function more efficiently. From the above six rules, MET09J has the *highest* response time since it consists of nested for loops. This result also justifies that adoption of design patterns in the system design has a positive effect on the response time of the plugin.

Table 5.7: Average total response time with design patterns and without design patterns for six rules

Total Response time with design patterns (Nanoseconds)	Total Response time without design patterns (Nanoseconds)
4468158.50	55829724.75

Average total response time with design patterns used version and without design patterns is shown in *Table 5.7*. According to the table, it is clear that the response time of version with design patterns is relatively less than the version without design patterns. Hence this result further concludes that the adoption of design pattern has reduced the overall response time of the plugin.

Table 5.8: Average response time for different lines of code

Lines of code	Average response time (Nanoseconds)
100	1315088.75
200	1482097.25
300	1598991.20
500	1641296.00

The average response time for different lines of code is illustrated in *Table 5.8: Average response time for different lines of code*. This table shows when lines of code increase response time also gradually increase. This is mainly because for large codebases, parser requires a significant amount of time to generate an AST.

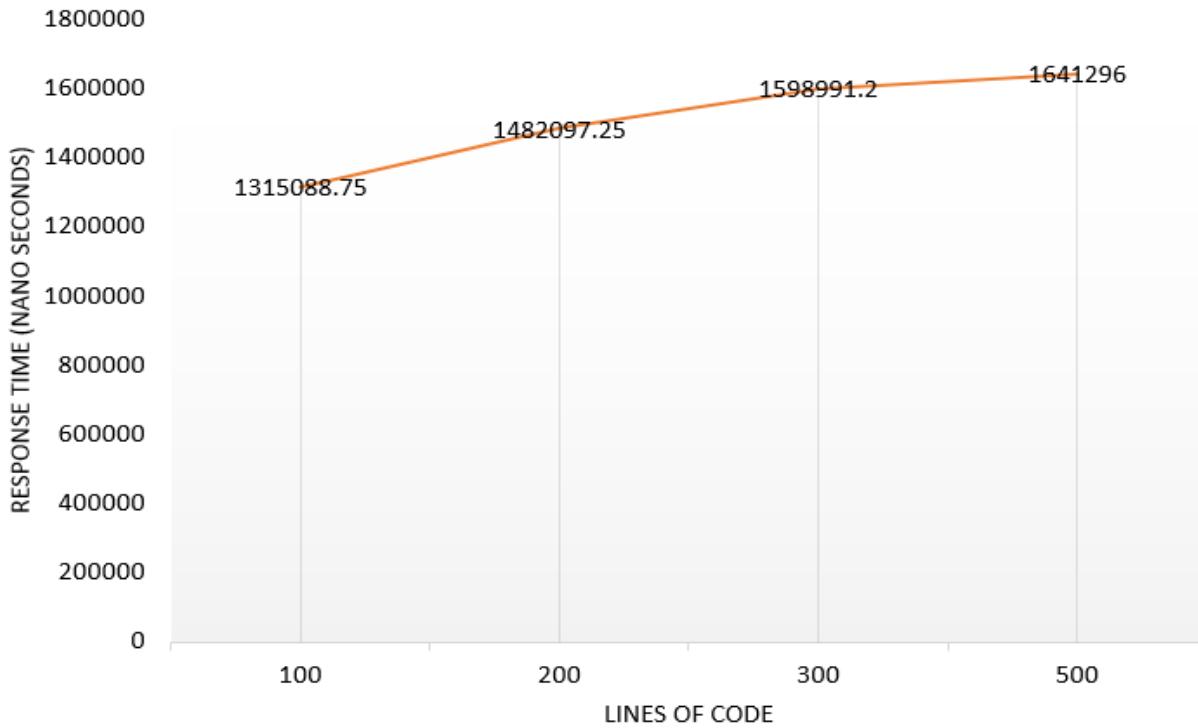


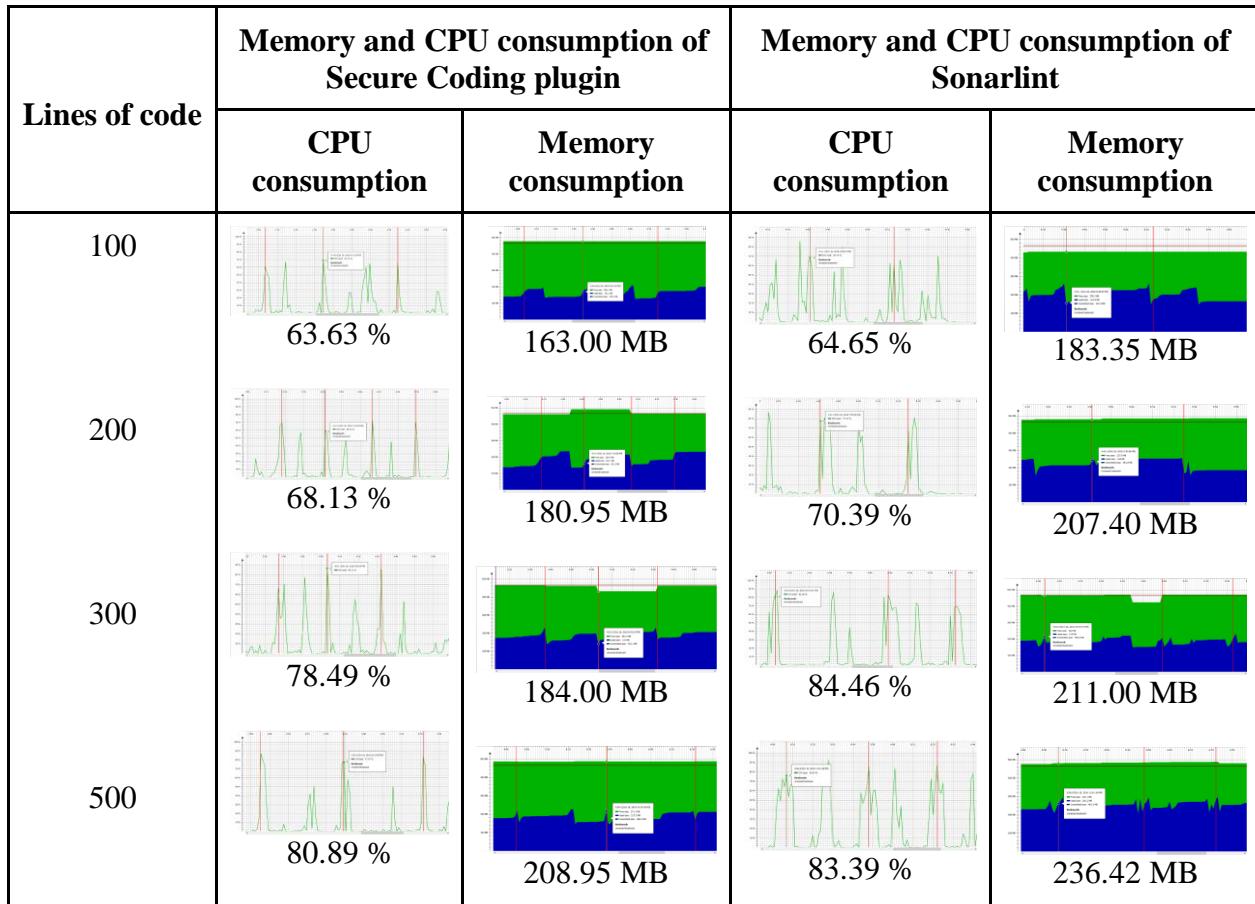
Figure 5.4: Response time versus lines of code

As mentioned earlier the response time should increase when the lines of code in the code base increase. As expected *Figure 5.4* illustrates this behaviour.

5.4.3 Benchmark tool comparison

This benchmark comparison was carried out for the Framework for Secure Coding plugin which has a total of 15 secure coding rules. Sonarlint IntelliJ IDEA plugin was selected as a benchmark tool which can be freely downloaded from the IntelliJ plugin repository.

Table 5.9: Benchmark tool comparison between Secure coding plugin-based framework and Sonarlint



The benchmark tool comparison using Secure Coding plugin and Sonarlint is illustrated in *Table 5.9*. It is clear that the secure coding plugin-based framework consumes significantly less amount of main memory compared to Sonarlint plugin. CPU consumption is also less with compared to Sonarlint plugin. For both tools, memory and CPU consumption gradually *increase* when the number of lines of code increase in the codebase. This evaluation concludes that the plugin was reasonable in its resource consumption with respect to benchmark tool.

5.5 User based evaluation

5.5.1 Introduction

The user-based evaluation was carried out in order to *assess the usability aspects* of the plugin-based framework along with its *impact on guiding* software developers to adhere to secure coding rules while writing source code. The evaluation was performed by allowing a set of users to provide feedback after downloading the framework directly from IntelliJ IDEA IDE or JetBrains plugin repository and using it.

5.5.2 Analysis of results

Feedback was obtained by enabling users that downloaded and used the the plugin-based framework, to fill a questionnaire. These questions were associated with the usability properties of the framework and other fundamental questions such as frequently used programming language, whether they are aware of secure coding, whether they follow secure coding guidelines when coding, if so what are the secure coding guidelines followed by them, etc. Users were also allowed to provide comments and recommendations for the improvement of the framework in order to improve the usability.

Based on the feedback from seven users it was evident that a *majority* of them mostly used Java programming language (71.4%). The secure coding plugin-based framework primarily supports the Java programming language, and it was also noticed that a *majority* of the feedback providers (57.1%) had heard about the concept of secure coding. The main reason for this is that the concept of secure coding was introduced in the early 2000s which is sometime back, along with the secure SDLC introduced by Microsoft [8].

What is the programming language you mostly use?

7 responses

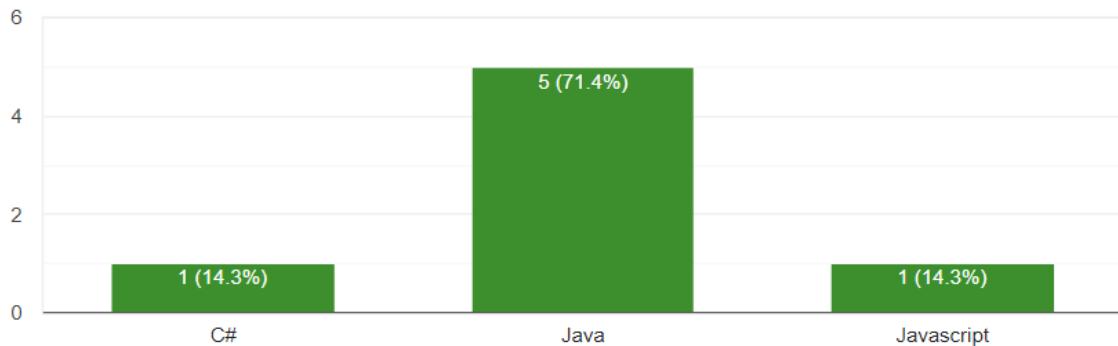


Figure 5.5: Percentage wise usage of Programming languages by the respondents

Are you aware of Secure Coding concept?

7 responses

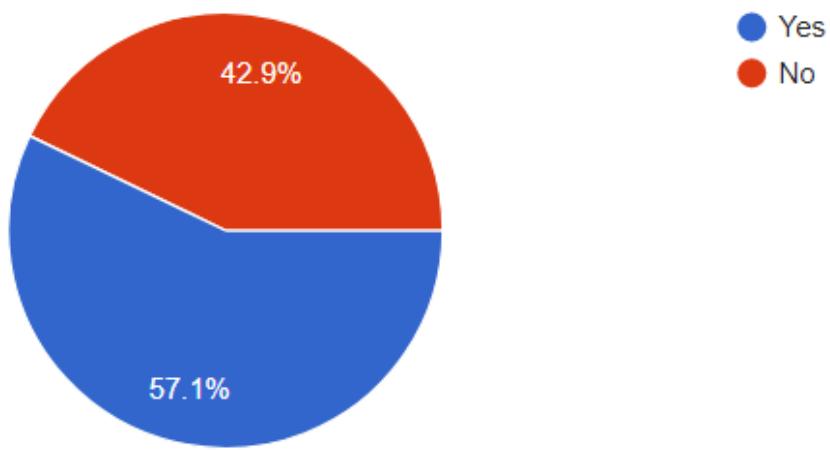


Figure 5.6: Percentage wise awareness of respondents regarding secure coding

The feedback as to whether the users who knew about secure coding concept followed secure coding guidelines while coding, provided negative results since a majority(75%) of them stated that they did not follow such guidelines while coding. The main reason for this could be identified as the lack of any automated tool focussed on identifying such violations in which the manual procedure of following a secure coding cheat sheet is highly time-consuming.

If yes, Do you follow any Secure Coding standards while coding?

4 responses

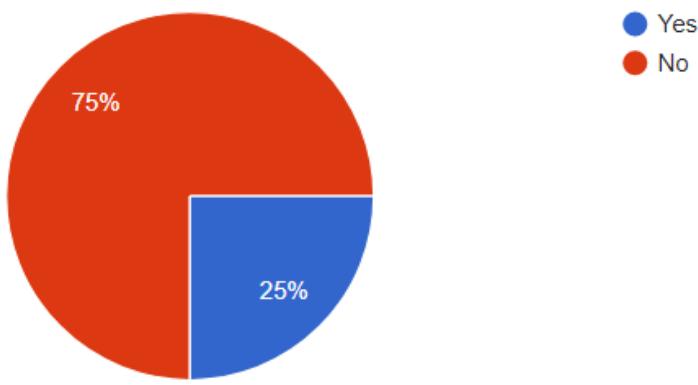


Figure 5.7: Percentage wise use of secure coding standards by respondents

The usability aspect of the plugin was assessed based on 5 major parameters. These were *ease* of installing the plugin into the IntelliJ IDEA plugin, *Performance* aspects such as latency, memory and processor usage, accuracy whether the mentioned secure coding rules are detected, Support as guide for the developers to learn secure coding best practices and User Friendliness such as tooltips, countermeasures, etc. The gradings for parameters in descending order were Excellent, Good, Fair and Poor and following are the respective responses received related to the *usability* aspect of the plugin.

Usability of the plugin



Figure 5.8: Grading received for usability of plugin

5.5.3 Conclusion

It could be concluded based on the usability related responses that the plugin-based framework had an average grading of *Good* since it is the highest grading received for three out of the five usability parameters apart from ease of installation and User friendliness parameters. Other than these the responses, the feedback respondents were requested to provide comments or suggest any other improvements in order to improve the versions of the plugin-based framework that would be released in the future. The feedback provided by the respondents could be found in *Appendix F*.

5.6 Summary

The project based evaluation methodology was mainly carried out in order to verify whether the secure coding rule violations detected in open source projects are accurate. A set of four existing static code analysis tool namely SonarQube, SonarLint, FindBugs, and Checker were used and from the results of this evaluation methodology, it was verified that the secure coding rule violation detections by the secure coding plugin-based framework were accurate. The aforementioned four static code analysis tools all together detect nine secure coding rule violations which have also been implemented in the secure coding framework. The comparison of the results from this evaluation methodology verified that the detection of secure coding rule violations by the framework developed in this product based project was accurate.

It could be concluded from the extensibility based evaluation methodology that the secure coding plugin-based framework can be modified in the future by developers, with minimal changes to the code base. The use of `HashMap<String, Object>` type of common data structure and Factory design pattern has enabled the extension of the plugin through various aspects such as addition of a new Secure coding violation detection algorithm, addition of a new source code granularity level, modification of existing secure coding algorithms and modification of existing data structures.

The performance based evaluation was performed using software profiling tools such as visualVM and JProfiler in order to assess the performance of the plugin-based framework based on its system design. The main intention was to identify whether the design provided a competitive advantage to the secure coding plugin-based framework on performance factors such as memory and Processor usage, latency which is the response time in detecting secure coding rule violations, etc.

The user based evaluation was carried out in order to assess the usability aspects of the plugin-based framework such as ease of installing, accuracy, performance, etc in which positive responses were received from the users that downloaded the plugin via the JetBrains plugin repository. Feedback was also received in the form of comments in order to make improvements in the future upgrades to be released.

Chapter 6 : Conclusion

The solution focuses on developing a mechanism to automate the process of detecting SEI CERT secure coding rule violations found in the source code of software applications. In the current context, most of the software developers follow a manual approach in which a checklist with secure coding guidelines is followed each time a part of source code is typed. A plugin-based framework was built as a proof concept for the automation of this manual process which currently consumes a massive amount of developer time unnecessarily.

The well-focussed background study identified that a significant amount of costs could be saved by preventing the introduction of security vulnerabilities during the coding or development phase of the SDLC, if software developers adhere to secure coding guidelines. Furthermore it was identified that there exist three sets of secure coding guidelines introduced by parties namely Oracle, SEI CERT and OWASP. The set of secure coding guidelines introduced by SEI CERT commonly referred to as *SEI CERT secure coding rules* were selected as the most suitable and feasible set of guidelines to be implemented in the secure coding plugin-based framework.

Subsequently, the SEI CERT secure coding rules were classified into *three granularity levels* namely Method, Class and Package respectively. Based on these three levels of classification a set of *100 secure coding rules* were classified and is found in *Appendix B*. The classification criteria immensely supported improving cohesiveness in analysing the secure coding rules and implementing them in the form of algorithms in the framework. The system design with the use of design patterns facilitated extensibility of the secure coding framework for future development purposes.

During the implementation process, a selected set of 15 secure coding rules were converted into algorithms and implemented in the framework in the form of an IntelliJ IDEA IDE plugin. An on the fly methodology was developed and integrated with the JavaParser, in order to build the *live parser* of the framework. This assisted in creating an AST in a real-time manner through which the necessary source code fragments for the violation detection algorithms are obtained and stored in data structures. Apart from the detection of secure coding rule violations in the source code, the *countermeasures* to overcome such violations were also provided to the user via a tool window. A *mechanism to extend* the framework with the support of Factory design pattern and a common data structure was introduced to support future developments.

Subsequent to the implementation process an executable version of the plugin-based framework was *deployed* to the JetBrains plugin repository. This enabled users to download and use the plugin-based framework. An evaluation process was carried based on four main aspects namely *Project-based*, *Extensibility-based*, *Performance-based* and *User-based* evaluations. The violations detected by the plugin-based framework can be concluded as accurate based on the results of the Project based evaluation. Extensibility-based evaluation assessed various aspects through which the framework could be extended for future work. The results from the

performance-based evaluation indicated that the system design of the plugin-based framework improved its performance with the support of appropriate design patterns. Based on user responses of the user based evaluation, it could be concluded that users have received positive experience when using the framework.

6.1 Future work

A significant amount of secure coding rules classified in *Appendix B* are user dependent and is therefore complex to be implemented in the form of algorithms. This could be considered as a limitation of this project and identification of a formal mechanism to convert these set of user-dependent rules into algorithms or any other suitable form to detect secure coding rule violations, could be considered as a vital task in the future development process of the plugin-based framework.

Secure coding plugin-based framework requires users to have a syntactically correct source code in the IDE editor. The reason for this is, JavaParser library can only parse syntactically correct source codes. Due to IntelliJ IDEA'S IntelliSense feature, users usually write syntactically correct source codes even though the code is incomplete or semantically incorrect. When a user makes a typo mistake, the whole code will become syntactically incorrect, and then the framework will not be able to work as intended. This is a limitation of this framework and hence finding an approach to analyse syntactically incorrect, or unparsable source codes can be considered as future work.

The framework developed in this product based project detects 15 SEI CERT secure coding rule violations. Integration of further secure coding rules into the framework could be considered as a vital task in the future work of this project. The ability to detect more and more secure coding rule violations in source codes could be considered as an aspect which improves the usability and importance of the secure coding plugin-based framework.

In the current context, the plugin-based framework supports Java programming language and focuses only on IntelliJ IDEA IDE. Existing static code analysis plugins such as SonarLint, SpotBugs support several programming languages and different IDEs such as NetBeans and Eclipse. SEI CERT has introduced secure coding rules for several languages such as C, C++, Perl, and Android. Updating the secure coding plugin-based framework to accommodate the secure coding rules in these different programming languages and modifying the framework to support several IDEs could be considered as future work associated with this project.

References

- [1] S. Lipner, "The trustworthy computing security development lifecycle," in Computer Security Applications Conference, 20th Annual, 2004.
- [2] G. McGraw, "Software Security: Building Security In", 2006.
- [3] SANS Institute, "Software Engineering - Security as a Process in the SDLC", Nithin Haridas, 2007.
- [4] "Software security", Cylab, 2018. [Online]. Available: <https://www.cylab.cmu.edu/research/software-security.html>. [Accessed: 08- Aug- 2018].
- [5] Software Engineering Institute, "SEI CERT Oracle Coding Standard for Java". [Online]. Available: <https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java>
- [6] Tricentis, "The Software Fail Watch: 2016 in Review", 2016.
- [7] R. Stites, "The Power of Peer Review Session", 2015.
- [8] "Benefits of the SDL", Microsoft. [Online]. Available: <https://www.microsoft.com/en-us/SDL/about/benefits.aspx>. [Accessed: 12- June- 2018].
- [9] G. McGraw, "Seven Touchpoints for Software Security", Swsec.com, 2006. [Online]. Available: <http://www.swsec.com/resources/touchpoints/>. [Accessed: 09- Jun- 2018].
- [10] G. McGraw. "Risk analysis in software design", 2004. [Online]. Available: <https://www.synopsys.com/blogs/software-security/software-risk-analysis>. [Accessed: 05- June- 2018].
- [11] SpotBugs. Retrieved from <https://spotbugs.github.io>
- [12] Sonarlint. Retrieved from <https://www.sonarlint.org>

- [13] M. I. Daud, “Secure Software Development Model: A Guide for Secure Software Life Cycle“. 2009.
- [14] N. Davis, “Secure Software Development Life Cycle Process“, 2013.
- [15] V. S. Mdunyelwa, J. F. van Niekerk and L. A. Futcher, “Secure Coding Practices in the Software Development Capstone Project“, 2017.
- [16] L. N. Quang Do, K. Ali, B. Livshits, E. Bodden, J. Smith and E. Murphy-Hill, “Cheetah: Just-in-Time Taint Analysis for Android Apps“, 2017.
- [17] F. Wedyan, D. Alrmuny and J. M. Bieman, “The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction“, 2009.
- [18] Synopsys, “Secure Coding Guidelines“, 2017. [Online].
Available: <https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/secure-coding-guidelines-datasheet.pdf>
- [19] OWASP, “OWASP Secure Coding Practices Quick Reference Guide“. [Online].
Available:
https://www.owasp.org/images/0/08/OWASP SCP_Quick_Reference_Guide_v2.pdf
- [20] Oracle, “Secure Coding Guidelines for Java SE“. [Online].
Available: <https://www.oracle.com/technetwork/java/seccodeguide-139067.html>
- [21] Software Engineering Institute, “SEI CERT Coding Standards“. [Online].
Available: <https://wiki.sei.cmu.edu/confluence/display/seccode>
- [22] Software Engineering Institute, “Rules vs. Recommendations“. [Online].
Available:
<https://wiki.sei.cmu.edu/confluence/display/seccode/Rules+vs.+Recommendations>
- [23] Software Engineering Institute, “Rule: Priority and Levels“. [Online].
Available:
<https://wiki.sei.cmu.edu/confluence/display/java/Rule%3A+Priority+and+Levels>
- [24] T. Mogensen, Basics of Compiler Design: Anniversary edition, (p. 2), 2011.
- [25] ANTLR. Retrieved from <http://www.antlr.org/about.html>

- [26] F Tomassetti, "Getting started with JavaParser: analyzing Java Code programmatically", 2016. [Online].
Available: <http://tomassetti.me/getting-started-with-javaparser-analyzing-java-code-programmatically>
- [27] Javadoc of JavaParser.
Available at: <https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.6.15>
- [28] G. Tomassetti, "Java Libraries That Parse Java: JavaParser", 2017. [Online].
Available: <https://tomassetti.me/parsing-in-java/#javaLibraries>
- [29] "Design Patterns in Java Tutorial", www.tutorialspoint.com. [Online].
Available: https://www.tutorialspoint.com/design_pattern/index.htm. [Accessed: 05- Aug- 2018].
- [30] N. M. Edwin, "Software Frameworks, Architectural and Design Patterns", 2014.
- [31] "IntelliJ Platform SDK", JetBrains IntelliJ Platform SDK, 2015. [Online].
Available: <http://www.jetbrains.org/intellij/sdk/docs/welcome.html>.
- [32] "Trending Java Repositories", GitHub.com. [Online].
Available: <https://github.com/trending/java>. [Accessed: 15- Nov- 2018].

Appendix

Appendix A : Terminology

Abstract Syntax Tree	- A tree representation of a source code.
Batch style analysis	- An analysis done as a whole at the end of a process rather than analysing over time.
Build Security In	- A set of principles, practices, and tools to design, develop, and evolve information systems and software that enhance resistance to vulnerabilities, flaws, and attacks.
Code fragments	- A list of related data found in a program.
IntelliSense	- A code completion aid provided by an IDE for developers.
On the fly(Just in time) analysis	- Analysis which is carried out dynamically rather than after a result of something.
Secure coding	- The practice of writing software that is protected from vulnerabilities.
Secure coding guideline	- A set of secure coding best practices.
Software patch	- A software update which is installed into an existing software program.
Static code analysis	- Analysis of computer software that is performed without actually executing programs.
Touchpoint	- A characteristic or specific weakness that renders an organization or asset open to exploitation by a given threat or susceptible to a given hazard.

Appendix B : Classification of secure coding rules

Based on the classification criteria in Table 3.1, the secure coding rules of each main category have been classified as follows.

Main Category	Sub category	Level	Justification
Input validation and Data Sanitization (IDS)	IDS01-J	Package	Normalize() method used is inherited from the Normalizer class of the java.text package.
	IDS03-J	Package	The main class used is Logger class which is inherited from the java.util package.
	IDS04-J	Package	The FileInputStream, ZipInputStream, BufferedInputStream objects are inherited from classes in the java.io package
	IDS06-J	Method	The violation occurs inside a method and the system.out.format() method mainly involved here is inherited from the System class of the lang package.
	IDS07-J	Method	The violation occurs inside a method and it is invoked in the exec() method which belongs to the Runtime class of the lang package.
	IDS08-J	Method	Data sanitization is to not done at the beginning inside a method which leads to an error. The Sanitization is to be done initiating a StringBuilder object which is inherited from the java.lang package.
	IDS11-J	Method	The violation occurs in a method when the unrepresentable characters in a string have not been removed before violation and it occurs in a string variable which belongs to the String class of the java.lang package.
Declarations and Initialization (DCL)	DCL00-J	Class	Arise due to incorrect order of initialization of static field triggers inside a class but outside a method. The datatypes of the static variables are either its own class or a member of the java.lang package.

Expression (EXP)	DCL01-J	Package	Class name is a public identifier of the java standard library. I.e the class with the relevant name already exists in another package and if that package is imported in the future it could lead to various conflicts in the program.
	DCL02-J	Package	Mainly involves the modification of the elements of a collection in a foreach loop. Collections belong to the java.util package.
	EXP00-J	Method	Ignoring the values returned by a particular method leads to this secure coding rule violation.
	EXP02-J	Method	The use of equals() method inside a declared method of a class to compare 2 integer arrays leads to the violation and equals() method belongs to the Object class of the java.lang package.
	EXP04-J	Package	The rule is violated by passing arguments to java collections framework methods that are different to the collection parameter type. The java collections framework belongs to the java.util package.
	EXP05-J	Class	Involves calling a method to assign a value to a variable inside the main method where program grants access to the unauthorized user because evaluation of the side effect infested subexpressions follows the left to right ordering rule.
Numeric Types and Operations (NUM)	NUM01-J	Method	The violation occurs when both arithmetic(+,-, /, *) and bitwise(<<, >>) operations are performed on the same integer variables inside a method.
	NUM02-J	Method	Rule violation arises due to the the division (/) pr modulus (%) operators resulting a value 0 when applied on 2 integer variables inside a method.
	NUM03-J	Class	Involves using int as the return type of class methods which is incapable of fully representing all numerical unsigned values (i.e have to use long as the return type in order to fully represent all numerical unsigned values).
	NUM04-J	Method	This rule is violated if float data type is used in a method where precise computations needs to be performed since int data types needs to be used for

Characters and Strings (STR)	NUM07-J	Method	such computations.
	NUM09-J	Method	Involves using NaN(constant holding Not-a-Number which belongs to inside a method of a class for comparisons which results always false).
	NUM10-J	Package	Violation occurs due to using float as the data type of the counter used in for loop inside methods of a class.
	NUM12-J	Method	Violation occurs due to passing a double values instead of string values to the BigDecimal constructor which belongs to the BigDecimal class of java.math package.
	NUM14-J	Method	The rule is violated if converting a numeric type into a lower type(eg- Int to byte) is performed without range checking since it may lead to data misinterpretation or loss.
	STR00-J	Method	Involves incorrectly using of arithmetic and logical shift operators inside a method of a class.
	STR01-J	Method	Rule violation occurs when String variable(belongs String class of java.lang package) inside a method has partial value of data since the variable has been created before data is made available.
	STR02-J	Method	Involves incorrectly using char and integer forms of the isLetter() method of the Character class which is in the java.lang package.
	STR03-J	Method/ Package	Violation occurs due to not defaultly using locale when comparing local dependant data inside a method of a class.
	STR04-J	Method	The violation occurs when non character data such as data stored in Integer variables(belonging to java.lang package) or BigInteger(belonging to java.Math package) are encoded as strings. So this rule can fall into both method or package level based on the non character data type used.

Object Orientation (OBJ)	OBJ01-J	Class	String object which belongs to java.lang package is being created. This occurs inside a method of a class.
	OBJ02-J	Package	Involves limiting the access of fields such as global variables by using access modifiers such as private.
	OBJ04-J	Class	Violation of this rule occurs when the contents of a superclass is changed which may adversely affect the program logic of the subclasses that it depends on.
	OBJ05-J	Class	Involves creating a constructor to copy instances of a mutable class when an argument of same type is passed into it.
	OBJ07-J	Package	States that accessor methods (getter methods) should not return mutable class private members(eg- private global variables) without making them defensive.
	OBJ08-J	Package	Involves preventing the creation of malicious classes by declaring a sensitive super class final.
	OBJ09-J	Class	States that Private access modifier needs to be used to hide the inner class and its methods.
	OBJ10-J	Class	Involves comparing classes(using getClass() method) not class names(using getClass.getName()) in order to verify whether the 2 classes are equal.
Methods (MET)	MET00-J	Method	If a method argument type does not adequately constrains the state of the argument (Ex- void setState(Object state) {...}) then need to check whether conditional statement like "if else" are used to validate these arguments. Method arguments should be validated to prevent incorrect calculations, runtime exceptions, violation of class invariants, and inconsistent object state.
	MET01-J	Method	Assertions should not be used for method argument

	MET02-J	Package	validations inside a method. Therefore method level.
	MET03-J	Class	Don't use deprecated fields, methods, or classes in code. Most of these packages need to be imported. Ex- java.util.
	MET04-J	Package	Methods that perform security checks should not allow malicious subclasses to override the methods and omits the checks. In java, auto imported java.lang class SecurityManager is used for security checking.
	MET05-J	Package	Overridden or hidden methods should be less accessible. Otherwise they permits a malicious subclass with access to the restricted methods. This vulnerability involves inheritance relationship, therefore package level.
	MET06-J	Package	Constructor of superclass should not call overridable methods. It may result in the use of uninitialized data, leading to runtime exceptions or to unanticipated outcomes. This vulnerability involves inheritance relationship, therefore package level.
	MET07-J	Package	Vulnerability occurs when the superclass has a clone method that invokes an overridable method of the same superclass. Only need to check method signatures and method content inside the superclass.
	MET08-J	Class	We should not declare a method that hides a method declared in a superclass or superinterface. This vulnerability involves inheritance relationship, therefore package level.
	MET09-J	Class	Equals method overriding can cause equality contract violations. Equality contract is preserved if equals method consistently return the same integer, provided no information used in equals comparisons on the object is modified.

Exceptional Behavior (ERR)	MET10-J	Package	integer result when the hashCode() method is invoked.
	MET12-J	Package	Classes that have implemented Comparable interfaces can violate general contract when implementing compareTo() method. If so a vulnerability occurs.
	ERR00-J	Method Package	Classes should not use finalizers. This may involve Superclass's finalizer as well.
	ERR01-J	Package	Inside catch blocks of source code, don't just print the exceptions which means suppressing or ignoring them. This will leak information about the structure and state of the process and also the behavior of the application will be unaffected by the exception being thrown.
	ERR02-J	Method	A vulnerability occurs when reading files, working with databases expose sensitive information. For reading files and databases, relevant packages like java.io, java.sql need to be imported.
	ERR03-J	Method	Vulnerability occurs when SecurityException is used. SecurityException belongs to java.lang package which is auto imported by java. Inside catch block of SecurityException, System.err.println(se), Console.printf(), System.out.print*(), or Throwable.printStackTrace() should not be used because this can cause security vulnerabilities.
	ERR04-J	Method	Vulnerability occurs when finally clause or catch block fails to restore prior object state which was there before method failure. This can cause incorrect results in calculations as well.
	ERR05-J	Method	Vulnerability occurs when statement inside finally block completes abruptly along with any exceptions thrown from the try or catch blocks. Therefore inside finally block the return, break, continue, or throw statements should not be used.
			Exceptions can be thrown from method calls inside finally blocks. These exceptions needs to be handled inside finally block itself to prevent vulnerabilities.

	ERR07-J	Method	Vulnerability occurs inside methods. A vulnerability may occur when catching RuntimeException. Therefore methods must not throw RuntimeException, Exception, or Throwable.
	ERR09-J	Method	When untrusted code is allowed to terminate the JVM, vulnerabilities can occur. For example invocation of System.exit() to terminate JVM can cause DoS attacks. These vulnerabilities occurs inside methods.
Visibility and Atomicity (VNA)	VNA00-J	Class	Multiple threads can share variables. To ensure the visibility of the most recent update on these variables, it must be declared volatile or the reads and writes must be synchronized. This shared variable is a class level variable.
	VNA02-J	Class	Compound operations (ex- *=, /=) on shared variables must be performed atomically to prevent data races and race conditions. To make these operations atomic, the methods these operations belongs to need to be declared as synchronized. Shared variable is a class variable therefore class level.
	VNA05-J	Class	We may get incorrect results from calculations, if atomicity is not ensured when reading and writing shared variables with 64-bit values (long) using multiple threads. Shared variable is a class variable. Therefore class level.
Locking (LCK)	LCK00-J	Class	Deadlocks may cause DoS, if the objects that require synchronization use their own intrinsic lock instead of private lock object idiom. This idiom requires the use of synchronized blocks within the class's methods rather than the use of synchronized methods. Lock is a class variable. Therefore class level.
	LCK01-J	Class	Deadlock may occur if synchronization is done on objects that may be reused. Examples of these objects are Boolean locks, boxed Integer objects, String literals, interned String objects etc. These objects are defined as class variables. Therefore class level.

Thread APIs (THI)	LCK02-J	Package	Vulnerability occurs between superclass and subclasses, when synchronization done on the class object returned by getClass() because the class object of the subclass is entirely distinct from the class object of the parent class. Therefore package level.
	LCK03-J	Package	Intrinsic locks should not be synchronized (Intrinsic means that we don't have to synchronize) of high-level concurrency objects. High-level concurrency objects belongs to java.util.concurrent.locks package. Therefore package level.
	LCK04-J	Class	Vulnerability occurs when synchronization is done on a collection view which has a backing collection which is accessible by other threads. These collections and backing collections are defined as class variables. Therefore class level.
	LCK05-J	Class	Vulnerability occurs when a code fails to synchronize access to a static field which is a class variable and also when it can be modified by an untrusted code. Therefore class level.
	LCK08-J	Class	An unreleased lock in any thread will prevent other threads from acquiring the same lock. Therefore need to make sure that actively held locks are released on exceptional conditions. Lock is defined as a class variable. Therefore class level.
	THI00-J	Package	Inside a class that implements Runnable interface, Thread object's run() method should not be invoked directly, because then the statements in the run() method are executed by the current thread rather than by the newly created thread.
	THI03-J	Class or Package	The wait() method must be invoked from a loop that checks whether a condition predicate holds and also the await() method of the condition interface also must be invoked inside a loop or else the program can lead to indefinite blocking and denial of service (DoS). .wait() belongs to java.lang.Object and .await() belongs to java.util.concurrent.locks.Condition package.

Thread Pools (TPS)	THI05-J	Package	Calling Thread.stop() from a class which implements Runnable interface will result in the release of all locks a thread has acquired. This may expose the objects protected by those locks when those objects are in an inconsistent state.
	TPS00-J	Class	When Thread-Per-Message design pattern is used to process incoming messages or requests attackers can carry out Dos attacks by sending multiple requests. To implement this Thread-Per-Message design, a class variable of ServerSocket (belongs to java.lang.Object package) type will be used. Therefore class level.
	TPS01-J	Package	A program that execute tasks that depend on the completion of other tasks in a bounded thread pool may lead to thread-starvation deadlock. This involves implementing an interface which belongs to java.util.concurrent package. Therefore package level.
Thread-Safety Miscellaneous (TSM)	TSM01-J	Method, class	The keyword this may be used only in the following contexts: <ul style="list-style-type: none"> • in the body of an instance method or default method • in the body of a constructor of a class • in an instance initializer of a class • in the initializer of an instance variable of a class • to denote a receiver parameter
	TSM02-J	Method	Threads are created using thread class inside the java.lang package.
	TSM03-J	Method	Violation occur since the partially initialized object can be made visible to other threads. During initialization of a shared object, the object must be accessible only to the thread constructing it.
	FIO00-J	Method	Before accessing a file need to check whether it is in a secure directory. Violation occurs since directly accessing to directories/file without checking its safety.
	FIO01-J	Package	The constructors for FileWriter do not allow the programmer to explicitly specify file access

	FIO02-J	Package	<p>permissions. The class FileWriter is inherited from the java.io package.</p> <p>The delete() method is used to delete a specified file but it gives no indication of its success. It only throws SecurityException. No other exceptions are thrown, so the deletion can silently fail. This method includes in file class which is inherited from java.io package.</p>
	FIO03-J	Package	<p>Violation occurs due to unexpected termination of JVM before invoke of deleteOnExit() method. Consequently, the temporary file is not deleted. deleteOnExit() method belong to the java.io package.</p>
	FIO04-J	Package	<p>Resources need to be released when they are not needed. When FileInputStream is used, the file needs to be explicitly closed. FileInputStream class belongs to java.io package. Database connection also need to close explicitly when an error occurs during execution of the SQL statement or during processing of the results. Connection interface belongs to java.sql package.</p>
	FIO05-J	Package	<p>CharBuffer created using wrap() or duplicate() methods must not return. CharBuffer defined in the java.nio package. Instead a read-only view of the char array can be returned in the form of a read-only CharBuffer.</p>
	FIO06-J	Package	<p>Violation occurs when multiple BufferedInputStream wrappers are used. BufferedInputStream class belongs to java.io package.</p>
	FIO07-J	Method	<p>The exec() method of the java.lang.Runtime class and the related ProcessBuilder.start() method can be used to invoke external programs. While running, these programs are represented by a java.lang.Process object. This process contains an input stream, output stream, and error stream. Incorrect handling of such external programs can cause unexpected exceptions, denial of service (DoS), and other security problems.</p>

Serialization (SER)	FIO08-J	Package	Violation occur when return value of the byte input method (read()) in FileInputStream class OR the return value of the character input method (read()) in FileReader class compare with -1(end of the stream). Both FileInputStream and FileReader classes are belongs to java.io package.
	FIO09-J	Package	The write() method, defined in the class java.io.OutputStream, takes an argument of type int the value of which must be in the range 0 to 255. Because a value of type int could be outside this range, failure to range check can result in the truncation of the higher-order bits of the argument.
	FIO10-J	Package	Incorrect use of the read() method can result in the wrong number of bytes being read or character sequences being interpreted incorrectly. This read() method belongs to FileInputStream and Reader class under java.io package.
	SER00-J	Package	If a class that implement Serializable without overriding its functionality changes, byte streams produced by users of old versions of the class become incompatible with the new implementation. Interface Serializable belongs to java.io package.
	SER01-J	Package	Violation of rule occurs due to incorrect method signature of writeObject(), readObject() and readObjectNoData() methods. These methods must be declared private for any serializable class. Serializable interface belongs to java.io Package.
	SER02-J	Package	Failure to sign and then seal objects during transit can lead to loss of object integrity or confidentiality. SealedObject constructor belongs to javax.crypto.SealedObject class. It uses the java.security.SignedObject class to sign an object when the integrity of the object must be ensured.
	SER03-J	Package	Violation occurs due to serialized form of sensitive data. Sensitive data that should never be serialized include cryptographic keys, digital certificates, and classes that may hold references to sensitive data at the time of serialization. Classes can be serialized by simply implementing the java.io.Serializable

Platform Security (SEC)	SER04-J	Method	<p>interface.</p> <p>security manager checks are omitted from the methods that are used in the serialization-deserialization process. security manager checks are included in the SecurityManager class which belongs to java.lang package.</p>
	SER05-J	Package	<p>Programs must not serialize inner classes.</p> <p>Serialization of inner classes can introduce platform dependencies and can cause serialization of instances of the outer class. Serializable interface belongs to java.io package.</p>
	SER09-J	Package	<p>Overridable methods should not invoke from readObject() method. This will provide the overriding method with access to the object's state before it is fully initialized. readObject() method belongs to ObjectInputStream class in java.io package.</p>
	SEC02-J	Package	<p>This violation refers to Java 1.5 java.io package.</p> <p>java.io.File is non final. As a result of that the getPath() method can be overridden so that the security check passes the first time it is called but the value changes the second time to refer to a sensitive file such as /etc/passwd. Class File belongs to java.io package.</p>
	SEC04-J	Method	<p>Violation occurs due to The check*() methods lack support for fine-grained access control. checkRead() method belongs to SecurityManager class which inherit from the java.lang package.</p>
	SEC06-J	Package	<p>By default, URLClassLoader verifies the signature using the public key contained within the JAR file.</p> <p>The default automatic signature verification process may still be used but is not sufficient.</p> <p>URLClassLoader class belongs to java.net package.</p>
	SEC07-J	Package	<p>Violation occurs due to overriding getPermissions() method without invoking super.getPermissions() method.</p> <p>getPermissions method belongs to URLClassLoader in java.net package.</p>
			<p>Untrusted environment variables can provide data</p>

Runtime Environment (ENV)	ENV02-J	Method	for injection and other attacks if not properly sanitized.
	ENV03-J	Method	Granting All Permission to untrusted code allows it to perform privileged operations. The permission java.lang.RuntimePermission applied to target createClassLoader grants code the permission to create a ClassLoader object. This permission is extremely dangerous because malicious code can create its own custom class loader and load classes by assigning them arbitrary permissions.

Appendix C : Secure coding rules with algorithms and respective source code fragments

Rule	Algorithm	Source code fragment and relevant JavaParser methods required for the algorithm
NUM09-J	<ol style="list-style-type: none"> 1. Check the data type of the loop counters of for loops inside methods. 2. If the data type is <i>float</i> violation else no violation 	<ol style="list-style-type: none"> 1. Checking the data type of the for loop counter <code>getInitialization().get(0).getChildNodes().get(0).getChildNodes().get(0).equals("float")</code>
ERR04-J	<ol style="list-style-type: none"> 1. Check the contents of the finally blocks in try catch statements 2. If it contains return, break, continue or throw statements violation occurs else no violation. 	<ol style="list-style-type: none"> 1. Checking whether finally block contains a return statement. <code>getFinallyBlock().flatMap(fb -> fb.findFirst(ReturnStmt.class)).isPresent()</code> 2. Checking whether finally block contains a break statement. <code>getFinallyBlock().flatMap(fb -> fb.findFirst(BreakStmt.class)).isPresent()</code> 3. Checking whether finally block contains a continue statement. <code>getFinallyBlock().flatMap(fb -> fb.findFirst(ContinueStmt.class)).isPresent()</code> 4. Checking whether finally block contains a throw statement. <code>getFinallyBlock().flatMap(fb -> fb.findFirst(ThrowStmt.class)).isPresent()</code>
ERR07-J	<ol style="list-style-type: none"> 1. Check the thrown exceptions inside methods. 2. If RuntimeException, Exception or Throwable has been thrown then violation else no violation 	<ol style="list-style-type: none"> 1. Checking whether thrown exception is a RuntimeException <code>getExpression().getChildNodes().get(0).equals("RuntimeException")</code> 2. Checking whether thrown exception is an Exception

		<p><code>getExpression().getChildNodes().get(0)).equals("Exception")</code></p> <p>3. Checking whether thrown exception is a Throwable <code>getExpression().getChildNodes().get(0)).equals("Throwable")</code></p>
ERR08-J	<ol style="list-style-type: none"> 1. Check the contents of the catch clause in try catch statements in a method. 2. If NullPointerException, Exception or Throwable are found in the catch block then violation else no violation 	<ol style="list-style-type: none"> 1. Checking whether NullPointerException is caught. <code>getParameter().getType().equals("NullPointerException")</code> 2. Checking whether Exception is caught. <code>getParameter().getType().equals("NullPointerException")</code> 3. Checking whether Throwable is caught. <code>(n.getParameter().getType().equals("Throwable"))</code>
EXP02-J	<ol style="list-style-type: none"> 1. Check whether equals() method has been called inside methods of a class. 2. Get the set of arrays declared inside methods 3. Check the parameters of the called equals() methods. 4. If the number of parameters is 1 and that parameter has a data type of an array then violation else no violation 	<ol style="list-style-type: none"> 1. Getting equals method <code>getName().asString().equals("equals")</code> 2. Getting number of arguments <code>getArguments().size() == 1</code> 3. Getting the list of arrays in methods <code>n.getName()</code>
MET09-J	<ol style="list-style-type: none"> 1. Get all the method names defined in a class. 2. Check whether the method name “equals” is there. 	<ol style="list-style-type: none"> 1. Getting the list of method names defined in a class <code>n.getNameAsString()</code>

	<ol style="list-style-type: none"> 3. If so check whether hashCode method is also defined. 4. If hashCode() is not defined then there's a vulnerability. 	
OBJ05-J	<ol style="list-style-type: none"> 1. Get class variables of object types, declared as private. 2. Check whether any method returns a value of these object types identified above. 3. If so then there's a vulnerability. 	<ol style="list-style-type: none"> 1. Getting the list of non primitive class variable declarations (!ff.getVariable(0).getType().isPrimitiveType() && ff.getModifiers().contains(Modifier.PRIVATE)) 2. Getting the list of return types of methods defined public void visit(ReturnStmt n, List<ReturnStmt> collector) { super.visit(n, collector); if(!n.toString().equals("return;")){ collector.add(n); } }
OBJ01-J	<ol style="list-style-type: none"> 1. Get the list of class variables declared as public. 2. Check whether this variable is used inside more than one method. 3. If so then there's a vulnerability. 	<ol style="list-style-type: none"> 1. Getting the list of class variables for (FieldDeclaration ff : n.getFields()) { collector.add(ff); }
OBJ10-J	<ol style="list-style-type: none"> 1. Get the list of public static class variables. 2. Check whether they are declared final as well. 3. If they are not declared as 	<ol style="list-style-type: none"> 1. Getting the list of public static non-final class variables (member.isPublic() && member.isStatic() && !member.isFinal())

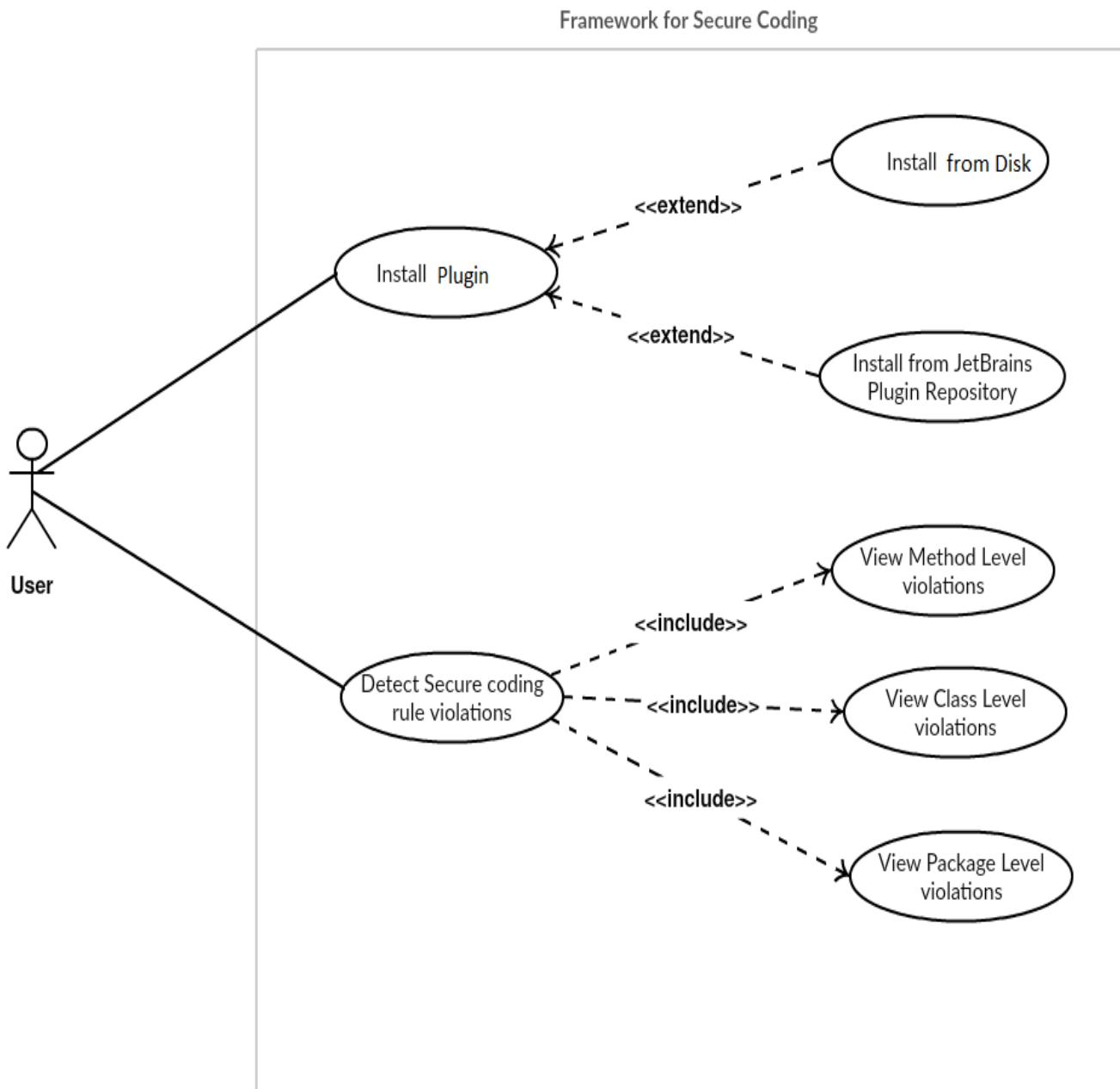
	final, then there's a vulnerability.	
DCL00-J	<ol style="list-style-type: none"> 1. Check whether a constructor is defined. 2. Get the full declaration of class variables along with their line number. 3. If a class variable has created an object of same class, get class variables defined after that object creation. 4. Check whether these variables are used in expressions inside constructor. 5. If so then there's a vulnerability. 	<ol style="list-style-type: none"> 1. Getting the full declarations of class variables n.getFields() 2. Getting the list of statements inside a constructor (n.getBody().getStatements()) 3. Checking whether the type of fields are of the same type as the object of the same class. for (FieldDeclaration member : ccf.clssvardeclarations) { if(member.getVariable(0).getTypeAsString() g().equals(ccf.className)){...} }
THI00-J	<ol style="list-style-type: none"> 1. Get implemented interface of class. 2. Check whether class implement runnable. 3. If Thread.run() is used then it is violation of rule. 	<ol style="list-style-type: none"> 1. Getting implemented interface. getNameAsString() 2. Checking whether it is runnable equals("Runnable") 3. Getting all expressions with run method. getNameAsString().equals("run")
SER01-J	<ol style="list-style-type: none"> 1. Get implemented interfaces and method signatures. 2. If class implement serializable then readObject() and writeObject() methods must be “private void” methods. 3. readResolve() and 	<ol style="list-style-type: none"> 1. Getting implemented interface. getNameAsString() 2. Checking whether it is serializable. ImplementedInterfaces.get(j).equals("Serializable") 3. Getting method declaration of readObject() and writeObject() methods and check for proper method signatures. (n.getNameAsString().equals("readObject"))&&(!(n.isPrivate())&&!n.isStatic()))))

	<p>writeReplace() methods should be private and static.</p> <p>4. If not it is a vulnerable code.</p>	<pre>(n.getNameAsString().equals("writeObject"))&&(!n.isPrivate()&&!n.isStatic()))</pre> <p>4. Getting method declaration of readResolve() and writeReplace() methods and check for proper method signatures.</p> <pre>n.getNameAsString().equals("readResolve")&& (n.isStatic() n.isPrivate())</pre> <pre>n.getNameAsString().equals("writeReplace")&& (n.isStatic() n.isPrivate())</pre>
NUM10-J	<p>1. Get object creation expressions.</p> <p>2. Check whether BigDecimal constructor is used.</p> <p>3. If its argument is a float or double then violation of rule.</p>	<p>1. Getting object creation expressions. ObjectCreationExpr</p> <p>2. Checking whether source code contains Big Decimal constructor.</p> <pre>getTypeAsString().equals("BigDecimal")</pre> <p>3. Checking whether constructor argument is double literal.</p> <pre>getArguments().get(0).isDoubleLiteralExpr()</pre>
SEC07-J	<p>1. Get method names in the class</p> <p>2. Check whether it override getPermissions() method.</p> <p>3. If it contains getPermissions() method then get method body and check whether is it calls super.getPermissions()</p> <p>4. If not then violation of the rule.</p>	<p>1. Getting method names invoke inside the class. MethodDeclaration</p> <p>2. Checking whether it contains getPermissions() method.</p> <pre>getNameAsString().equals("getPermissions")</pre> <p>3. Getting method body and then checking whether is it calls Permissions().</p> <pre>getBody().get().getRange().get().begin.line</pre> <pre>getBody().get().getRange().get().end.line</pre> <pre>getTypeAsString().equals("Permissions")</pre>
FIO02-J	<p>1. Get file objects created in the class.</p>	<p>1. Get list of method names invoked in the class. MethodCallExpr</p>

	<p>2. Check whether delete() method directly called without checking return inside if condition. Then violation of the rule.</p>	<p>2. Checking whether delete() method is called. <code>getNameAsString().equals("delete")</code></p> <p>3. Get file objects created in the class. <code>ObjectCreationExpr</code> <code>getTypeAsString().equals("File")</code> <code>getChildNodes().get(0).toString().equals(ObjectCReationExpress.get(j).getParentNode().get().getChildNodes().get(1).toString())</code></p> <p>4. Get conditional expression of if condition <code>getCondition().toString().equals(fileDeleteInstance)</code></p>
--	--	--

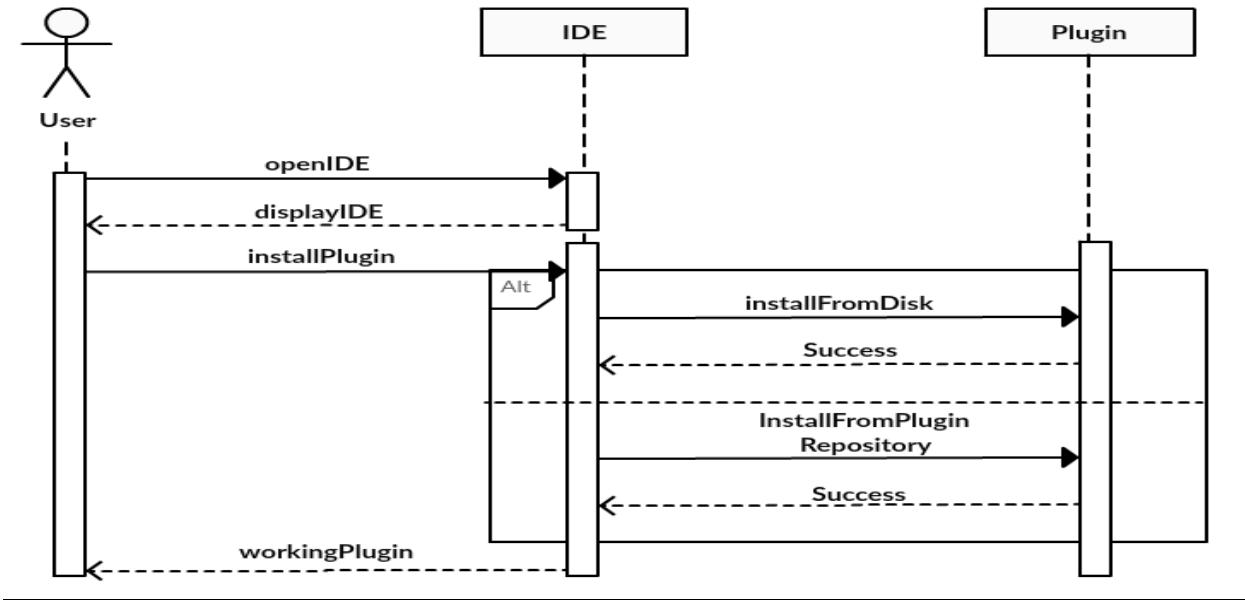
Appendix D : Other design artifacts

1. Use case diagram

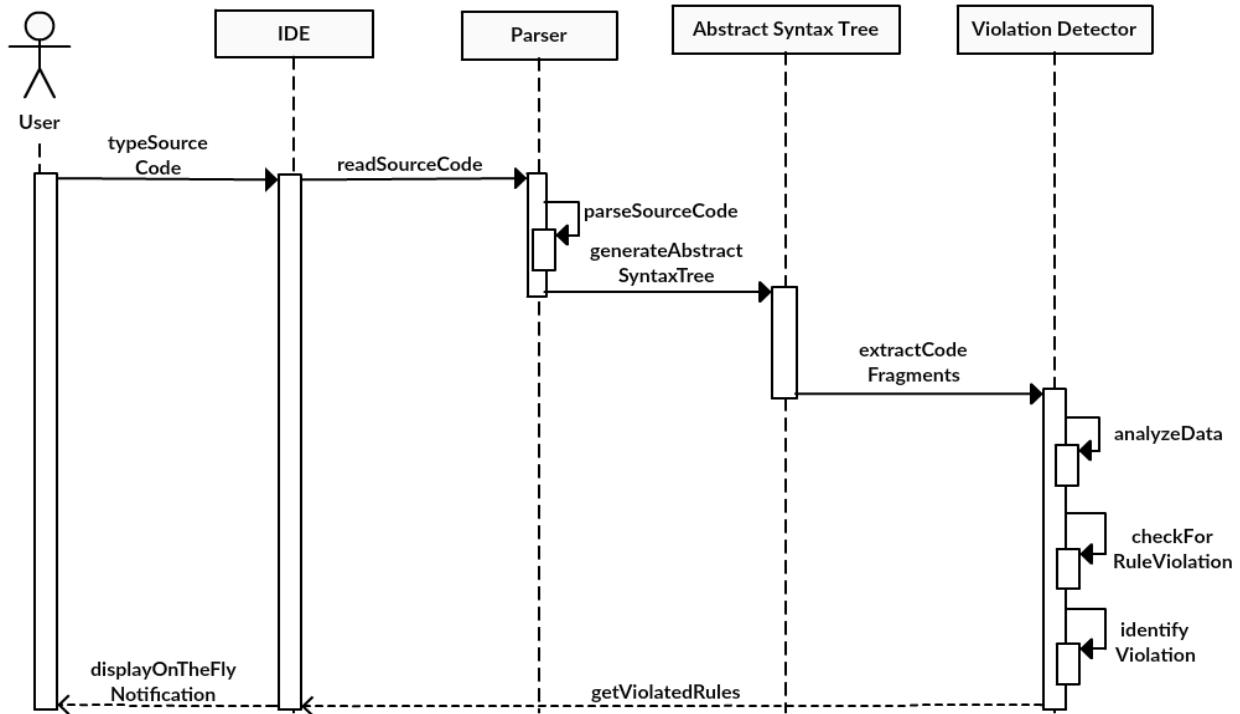


2. Sequence diagrams

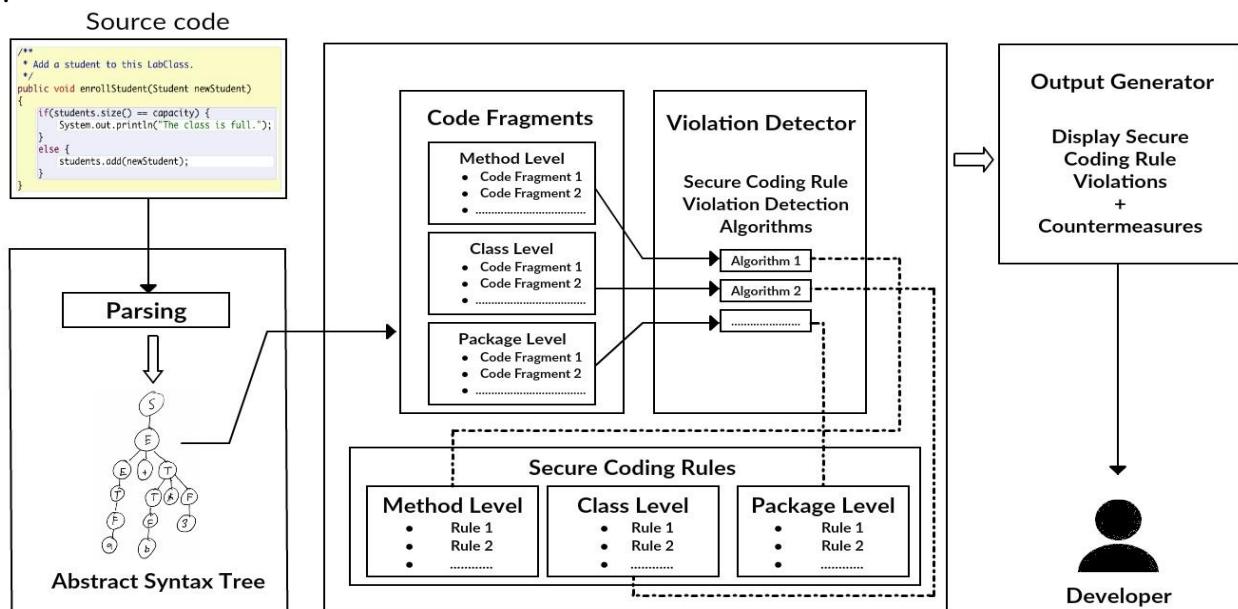
I) Sequence diagram for installing the plugin



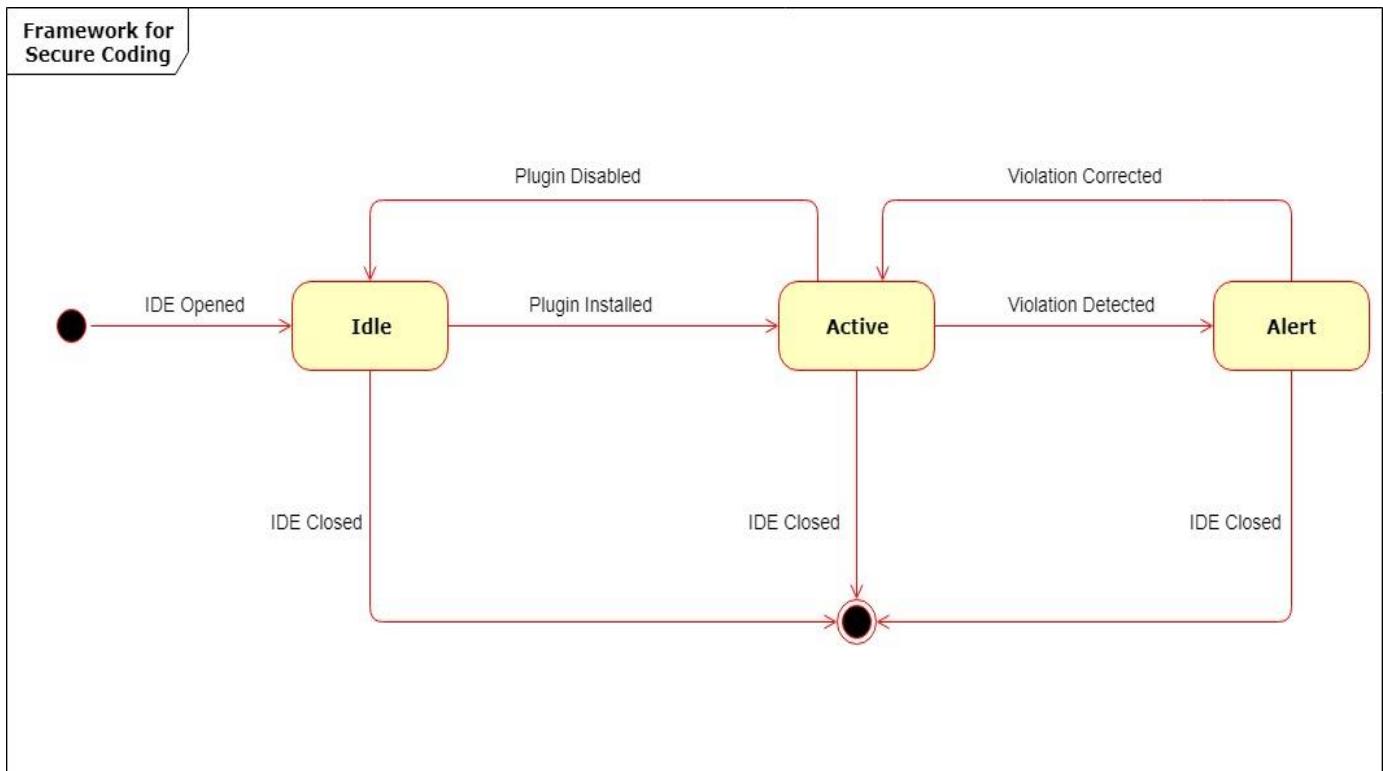
II) Sequence diagram for secure coding rule violation detection



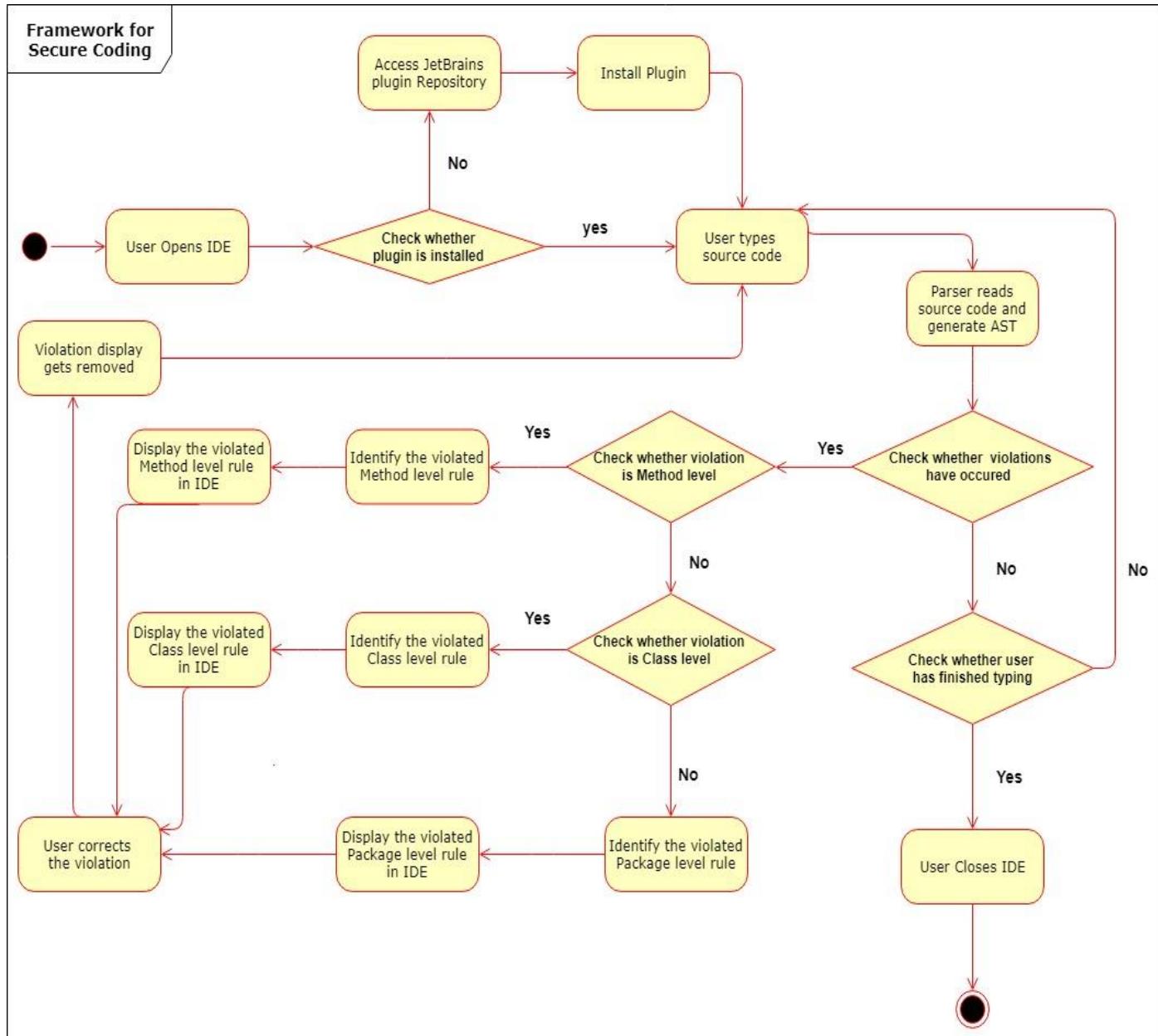
3. WorkFlow diagram



4. State transition diagram



5. Activity diagram for violation detection



Appendix E : Violation detection

PlainTextHandler

Secure Coding Plugin

Violated Rules	Rule Description										
OBJ10-J violated at line [12]	OBJ10 : Do not use public static nonfinal fields <table border="1"> <thead> <tr> <th>Severity</th> <th>Likelihood</th> <th>Remediation Cost</th> <th>Priority</th> <th>Level</th> </tr> </thead> <tbody> <tr> <td>Medium</td> <td>Probable</td> <td>Medium</td> <td>P8</td> <td>L2</td> </tr> </tbody> </table> Click here for more details	Severity	Likelihood	Remediation Cost	Priority	Level	Medium	Probable	Medium	P8	L2
Severity	Likelihood	Remediation Cost	Priority	Level							
Medium	Probable	Medium	P8	L2							

PlainTextHandler.java source file of "Arthas" project

TelnetConsole.java

Secure Coding Plugin

Violated Rules	Rule Description										
ERR08-J violated at line [46, 59, 79, 93, 101, 109, 193]	ERR08 : Do not catch NullPointerException or any of its ancestors <table border="1"> <thead> <tr> <th>Severity</th> <th>Likelihood</th> <th>Remediation Cost</th> <th>Priority</th> <th>Level</th> </tr> </thead> <tbody> <tr> <td>Medium</td> <td>Likely</td> <td>Medium</td> <td>P12</td> <td>L1</td> </tr> </tbody> </table> Click here for more details	Severity	Likelihood	Remediation Cost	Priority	Level	Medium	Likely	Medium	P12	L1
Severity	Likelihood	Remediation Cost	Priority	Level							
Medium	Likely	Medium	P12	L1							

TelnetConsole.java source file of "Arthas" project

PlainTextHandler.java

```
1 package com.taobao.artdas.core.shell.command.internal;
2
3 import ...
4
5 /**
6 * @author beivei30 on 20/12/2016.
7 */
8
9 public class PlainTextHandler extends StdoutHandler {
10     public static String NAME = "plaintext";
11     Do not use public static nonfinal fields
12     public static StdoutHandler inject(List<CliToken> tokens) { return new PlainTextHandler(); }
13
14     @Override
15     public String apply(String s) { return RenderUtil.ansiToPlainText(s); }
16
17 }
18
19
20
21
22
23
```

PlainTextHandler

Secure Coding Plugin

Violated Rules	Rule Description			
OBJ10-J violated at line [12]	OBJ10J : Do not use public static nonfinal fields			
Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

[Click here for more details](#)

PlotThread.java source file of "griDraw" project

```
233
234     private Collection<? extends String> toStringList(List<SpoonFile> files) {
235         List<String> res = new ArrayList<String>();
236         for (SpoonFile f : files) {
237             if (f.isActualFile()) {
238                 res.add(f.toString());
239             } else {
240                 try {
241                     File file = File.createTempFile(f.getName(), ".java");
242                     file.deleteOnExit();
243                     IOUtils.copy(f.getContent(), new FileOutputStream(file));
244
245                     res.add(file.toString());
246                 } catch (IOException e) {
247                     throw new RuntimeException(e.getMessage(), e);
248                 }
249             }
250         }
251         return res;
252     }

```

Secure Coding Plugin

Violated Rules	Rule Description										
ERR08-J violated at line [125, 289, 441, 461, 739, 905] OBJ05-J violated at line [832] ERR07J violated at [126, 246, 682, 686, 690, 721] FIO02J violated at [359]	<p>ERR07J : Do not throw RuntimeException, Exception, or Throwable</p> <table border="1"><thead><tr><th>Severity</th><th>Likelihood</th><th>Remediation Cost</th><th>Priority</th><th>Level</th></tr></thead><tbody><tr><td>Low</td><td>Likely</td><td>Medium</td><td>P6</td><td>L2</td></tr></tbody></table> <p>Click here for more details</p>	Severity	Likelihood	Remediation Cost	Priority	Level	Low	Likely	Medium	P6	L2
Severity	Likelihood	Remediation Cost	Priority	Level							
Low	Likely	Medium	P6	L2							

JDTBasedSpoonCompiler.java source file of "Spoon" project

Appendix F : Evaluation results

1. Project Based Evaluation results screenshots

core/.../java/com/taobao/arthas/core/GlobalOptions.java	13 hours ago	L22	%	▼
Make this "public static isUnsafe" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort				(cert, cwe)
Make this "public static isDump" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L35	%	▼
Make this "public static isBatchReTransform" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L46	%	▼
Make this "public static isUsingJson" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L57	%	▼
Make this "public static isDisableSubClass" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L68	%	▼
Make this "public static isDebugEnabledForAsm" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L78	%	▼
Make this "public static isSaveResult" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L89	%	▼
Make this "public static jobTimeout" field final ... 🔒 Vulnerability 🟢 Minor ⚡ Open Not assigned 20min effort	13 hours ago	L100	%	▼

OBJ10J rule violation of GlobalOptions.java file of "Arthas" project as detected by Sonarqube

GlobalOptions.java

```

1 package com.taobao.artdas.core;
2 public class GlobalOptions {
3     public static volatile boolean isUnsafe = false;
4
5     public static volatile boolean isDump = false;
6
7     public static volatile boolean isBatchReTransform = true;
8
9     public static volatile boolean isUsingJson = false;
10
11    public static volatile boolean isDisableSubClass = false;
12
13    public static volatile boolean isDebugForAsm = false;
14
15    public static volatile boolean isSaveResult = false;
16
17    public static volatile String jobTimeout = "1d";
18
19 }

```

Do not use public static nonfinal fields

GlobalOptions

Secure Coding Plugin

Violated Rules	Rule Description										
OBJ10-J violated at line [3, 4, 5, 6, 7, 8, 9, 10]	OBJ10J : Do not use public static nonfinal fields <table border="1" style="margin-top: 10px;"> <tr> <th>Severity</th> <th>Likelihood</th> <th>Remediation Cost</th> <th>Priority</th> <th>Level</th> </tr> <tr> <td>Medium</td> <td>Probable</td> <td>Medium</td> <td>P8</td> <td>L2</td> </tr> </table> Click here for more details	Severity	Likelihood	Remediation Cost	Priority	Level	Medium	Probable	Medium	P8	L2
Severity	Likelihood	Remediation Cost	Priority	Level							
Medium	Probable	Medium	P8	L2							

OBJ10J rule violation of GlobalOptions.java file of "Arthas" project as detected by Secure Coding Plugin

SonarLint: Current file Report Log

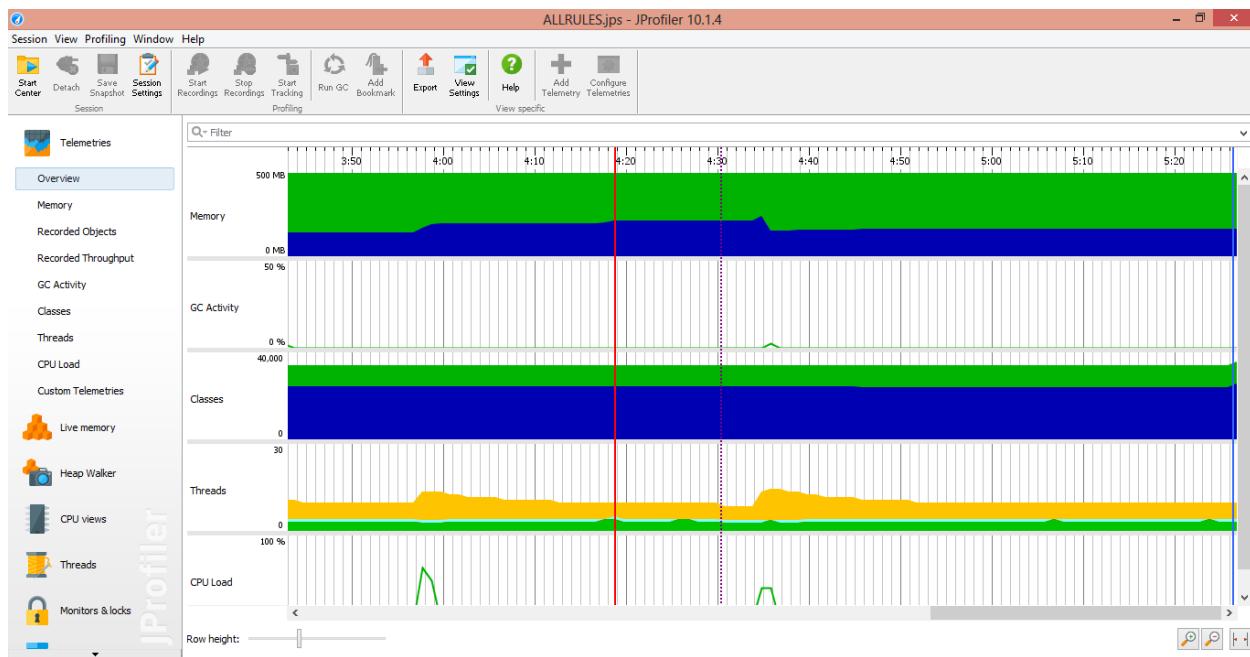
- Found 18 issues in 1 file
- GlobalOptions.java (18 issues)
 - (1, 13) Add a private constructor to hide the implicit public one.
 - (0, 0) Move this file to a named package.
 - (3, 35) Make this "public static isUnsafe" field final
 - (3, 35) Make isUnsafe a static final constant or non-public and provide accessors if needed.
 - (5, 35) Make this "public static isDump" field final
 - (5, 35) Make isDump a static final constant or non-public and provide accessors if needed.
 - (7, 35) Make this "public static isBatchReTransform" field final
 - (7, 35) Make isBatchReTransform a static final constant or non-public and provide accessors if needed.
 - (9, 35) Make this "public static isUsingJson" field final
 - (9, 35) Make isUsingJson a static final constant or non-public and provide accessors if needed.
- This file is not automatically analyzed

ANTLR Preview Tool Output CheckStyle FindBugs-IDEA Terminal SonarLint TODO

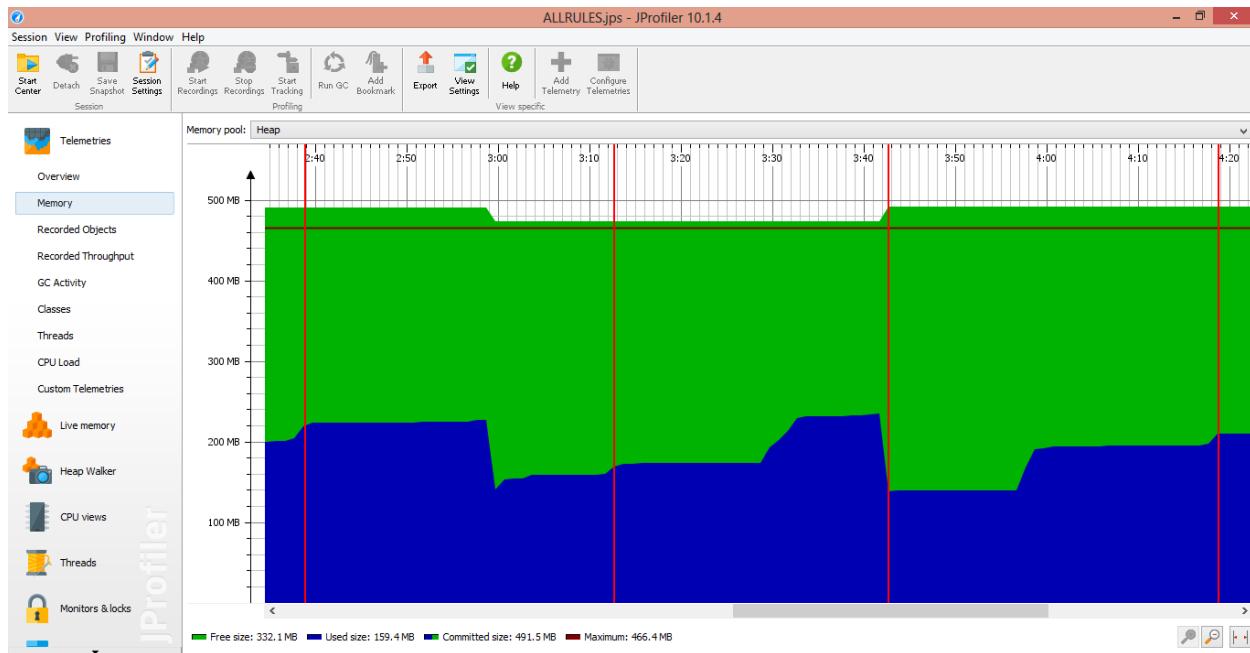
OBJ10J rule violation of GlobalOptions.java file of "Arthas" project as detected by SonarLint

2. Performance based evaluation

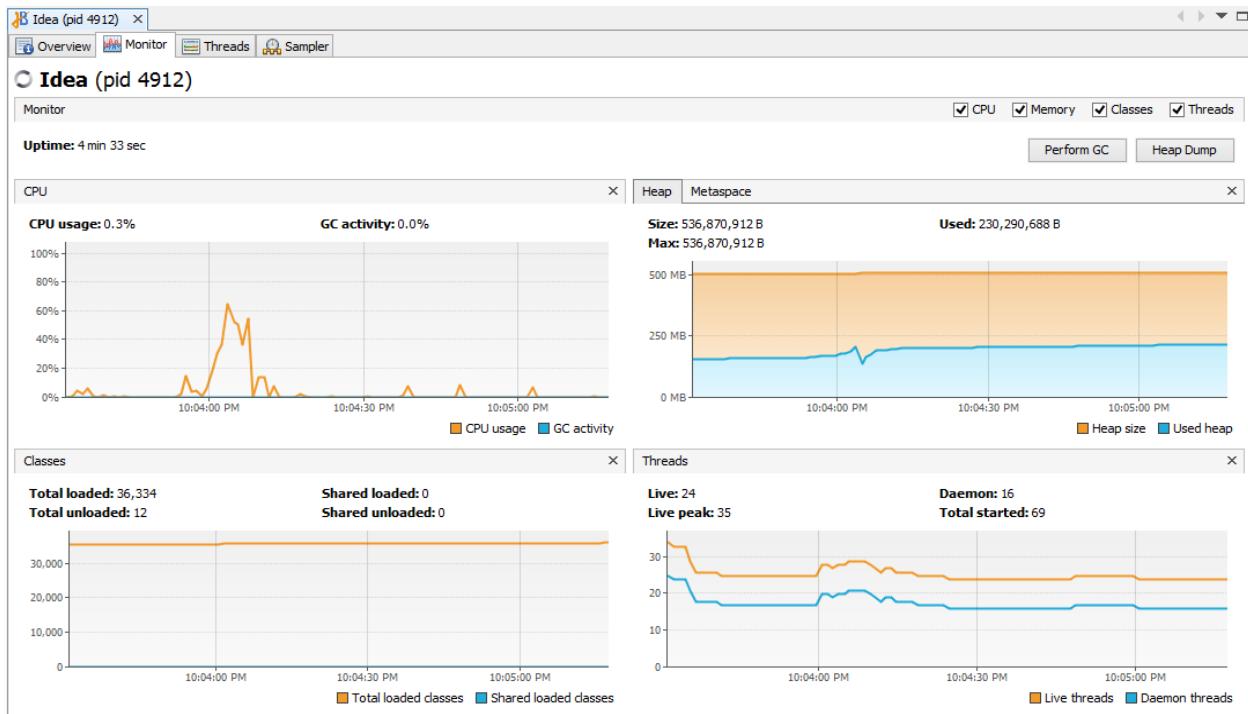
1. JProfiler Overview



2. JProfiler memory usage monitor



3. VisualVM monitor overview



4. User based evaluation form

Plugin Feedback Information

Basically to evaluate the IntelliJ Idea plugin "Framework for Secure coding"

What is the programming language you mostly use? *

Short answer text

Are you aware of Secure Coding concept? *

Yes

No

If yes, Do you follow any Secure Coding standards while coding?

- Yes
- No

If you follow any Secure Coding standard what is it?

- SEI CERT
- OWASP
- Oracle
- Other...

Usability of the plugin

*

Poor Fair Good Excellent

Usability of the plugin *				
	Poor	Fair	Good	Excellent
Ease of installing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Performance(Speed o...	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Accuracy (Whether m...	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Support as a teaching...	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
User friendliness (Too...	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Comments/ Any other improvements

Long answer text

4. Feedback and comments provided by respondents

Comments/ Any other improvements

6 responses

Actually this is a really good plugin. But there are some improvements which you can think about in future updates. You can improve number of secure coding guidelines covered by the plugin. You can look for other coding guidelines since this is mainly focused on SCI cert.

Works fine with IntelliJ IDEA 2018.2.5 (Community Edition). Didn't came across any issues but the computer does need to be connected to internet to find more details about detected violations. Overall positive experience.

Good plugin. This will help to promote the concept of secure coding among software developers.

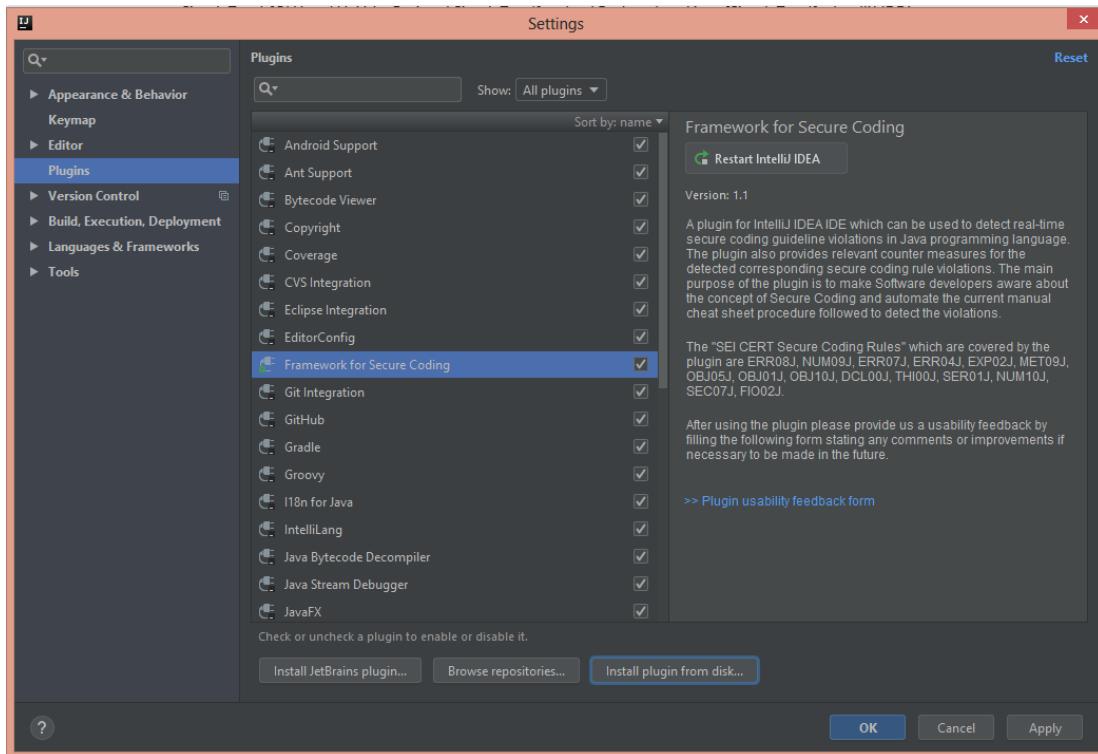
A well needed tool. I had not heard about secure coding best practises previously

Was not aware of secure coding previously. Got a good understanding regarding the concept from this plugin. Keep up the good work

A useful plugin for a beginner.

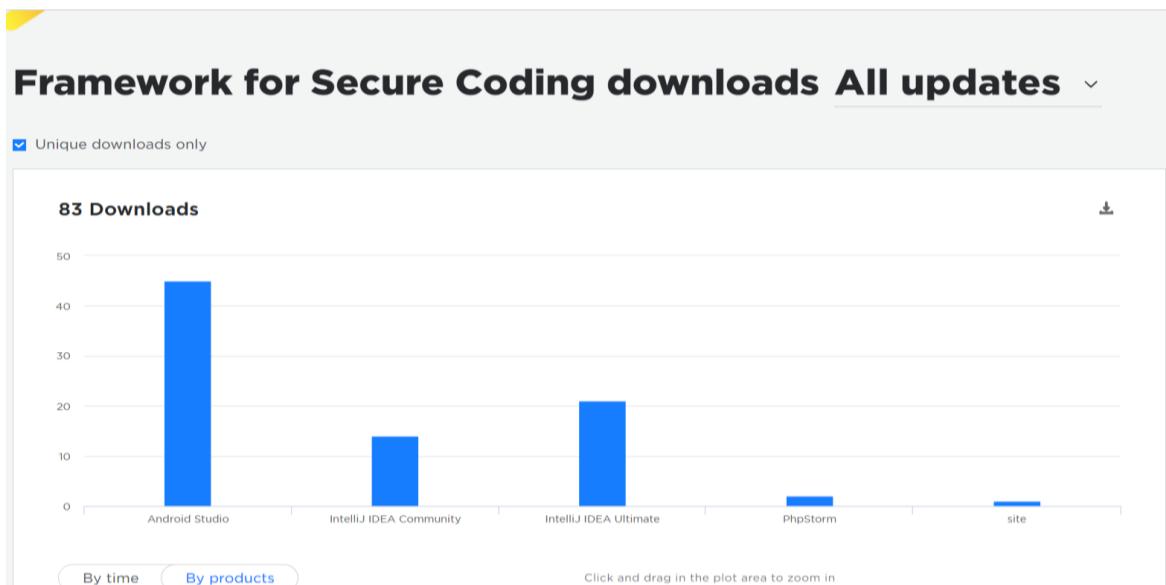
Appendix G : Deployment results

1. Downloading the deployed framework from IntelliJ IDEA IDE



Online Available at : <https://plugins.jetbrains.com/plugin/11265-framework-for-secure-coding>

2. Unique downloads by products



Appendix H : Individual contribution

Contribution of S.L.Dasanayake (Index number- 1400156)

Initially the currently existing problem needed to be identified. The component of the literature review which involves identifying the problem and the necessity of the framework. Similar systems such as Spotbugs and sonalint were reviewed practically and the limitations of them were identified. Studied the existing secure coding guidelines provided by Oracle,SEI cert and OWASP to identify the most suitable and feasible set of secure coding guidelines. Based on the study I identified SEI CERT secure coding rules to be the most suitable set of guidelines to be implemented in the framework , by detailed comparison study with other 2 secure coding guidelines.

In the design phase, my major contribution was to the system modelling aspect. The system model primarily consists of the class diagram drawn with a thorough study accommodating the necessary design patterns. The singleton design pattern was mainly used to improve the performance of the framework, and factory design pattern was added to the system model to support future extensibility of the plugin. Also, other design artefacts namely use case, activity diagram and state transition diagrams were contributions in the designing phase of the project.

The subcategories of secure coding rules of, six main categories of SEI CERT secure coding rules namely Input validation and Data Sanitization, Declarations and Initialization, Expression, Numeric Types and Operations, Characters and Strings, Object Orientation were classified into the three granularities based on their nature of the violation. I also contributed in designing the classification criteria for the method level secure coding rules which was my individual component. The violation detector algorithms for five selected method level secure coding rules were designed by me. The necessary source code fragments for these secure coding rules were obtained using static inner classes of the javaParser and stored in data structures such as HashMaps and arrayLists.

During the implementation phase, the necessary source code fragments for the method level secure coding rules were obtained using static inner classes of the javaParser and stored in data structures such as HashMaps and arrayLists. I implemented the selected five method level secure coding rules in the form of violation detection algorithms as my individual component implementation. Explanation of the implementation of method level algorithms has been explained in detail in this thesis. After the implementation of the violation detection algorithms, they were tested. This was done in order to verify whether these algorithms detected the respective method level secure coding rule violations in source code and expected results are obtained.

Apart from individual component implementation, I also developed the on the fly mechanism in of the framework which was then combined with the JavaParser to generate the live parser. The live parser is primarily used to generate AST through which the necessary source code fragments are obtained in a real-time manner. The design patterns namely Singleton and Factory mentioned in the system design were practically implemented to align the implementation and the system design. I did the integration of the implementations of the three individual components with the use of the factory design pattern. I used the singleton design pattern to avoid any unnecessary instantiation of classes in order to optimise the performance of the plugin based framework. This optimisation was verified from the results of the performance of based evaluation(not conducted by me) in which the framework with design patterns consumed less memory and showed low latency when compared with the framework implementation without design patterns.

I developed the extensibility mechanism which is a significant contribution to this project in order to accommodate future requirements such that the framework could be expanded easily with minimal changes to the extending source codebase. In Order to verify the success of the extensibility mechanism an evaluation was carried out Four main aspects were considered in extensibility based evaluation methodology which mainly involves the addition of new violation detection algorithms, new source code granularity level, modifying existing algorithms and modifying existing data structures were considered here. As a proof of concept of extensibility mechanism a violation detection algorithm which was implemented in the framework was reimplemented using the extensibility in order to verify the proper functionality of the mechanism.

Contribution of A.Mudalige (Index number- 14000954)

The main research area of the project was to find a technique for analysing source codes and detecting secure coding rule violations. After referencing to different techniques, we chose the algorithmic approach which involved using a parser to transform source code into an AST. A suitable parser was needed for this purpose and I reviewed and studied currently available parser libraries, parser generators and also how to write our own parser. I referred to “Basics of Compiler Design: Anniversary edition” by T. Mogensen to understand the compilation process. I referred parser generators like ANTLR and JavaCC. These parser generators required us to define our own grammar. Then I referred to parser libraries like JavaParser and IntelliJ IDEA’s built it Program Structure Interface (PSI) which can be used to create a tree structure from source codes. Due to many reasons like online community, easiness and nature of this project, I concluded that JavaParser is the most suitable parser to be used in this project. After choosing JavaParser I studied how to use its built in libraries and methods to extract basic code fragments that we would need in this project and for this purpose, I had to get the help of JavaParser online community since JavaParser user guides are still being written. I also referred to “JavaParser: Visited” book.

I referred to IntelliJ IDEA’s plugin development approaches and there are two possible workflows for building IntelliJ IDEA plugins. They are using Gradle and using plugin Devkit. After comparing and contrasting I chose that Gradle would be the better approach.

I further improved the design of the framework by designing the System architecture diagram, Component architecture diagram, Workflow diagram and Product interaction with internal and external environment diagram. These diagrams helped us to get a clear picture of our system and also for group communication. I referred to online tutorials when designing these diagrams that gives a both high level and an internal view of the system.

After designing the system I was involved in the implementation phase. We have three main implementations called method level, class level and package level in this project. I conducted class level classification and class level implementation. This is an important granularity level since every Java code requires a class. I classified six main categories of SEI CERT (Methods, Exceptional Behavior, Visibility and Atomicity, Locking, Thread APIs and Thread Pools) which involved around thirty secure coding sub rules. Out of 100 secure coding rules classified in this project I chose five class level secure coding rules and represented them using algorithms before implementing them in Java. During this process, the relevant code fragments required by each class level secure coding rule was identified as well. The class level secure coding rules in SEI CERT, I implemented are MET09-J, OBJ05-J, OBJ01-J, OBJ10-J and DCL00-J.

I implemented several features of the plugin during the implementation phase. They are syntax highlighting, tool window creation and adding annotations to highlighted codes by syntax

highlighter. I used IntelliJ IDEA's RangeHighlighter to highlight areas where a violation occurs. For this purpose, I identified the range of the areas that needs to be highlighted using JavaParser built-in methods and then passed them into the RangeHighlighter. I created the basic toolwindow to print the results of detected violations and this was further improved by other members of this project. Adding annotations and tooltips to highlighted areas in the source codes was a main challenge because there were no compatible built-in methods for this purpose. I had to create my own annotation handler by creating an editor mouse motion listener. For this I had to create a new class and implement EditorMouseMotionListener interface of IntelliJ Open API and define my own functions and logic in it. I integrated this mouse listener with syntax highlighter to make it work as intended. Same as the other members I carried out automated unit testing using TestNG for the code I wrote.

I conducted project based evaluation and for this selected a set of open source trending Java projects on Github repository. I conducted this evaluation in an unbiased manner and the main purpose of this evaluation was to see whether secure coding plugin detects vulnerabilities in these projects and if so are they accurate. In this evaluation, I also found the most common secure coding rules violated by developers. After project based evaluation, it was concluded that secure coding plugin works as intended.

Under documentation, I created the user manual which can be used by a developer to write secure code or to extend the framework by adding new secure coding rules or a new granularity level.

Contribution of M.L.T. Perera (Index number - 14001144)

Initially, a study was carried out in order to identify an approach to achieve the proposed goal of this project. Then the background of secure software development and secure coding guidelines were studied to acquire a thorough knowledge. During this process, several research papers and white papers were read in order to identify research areas in the literature. This study showed that related work is limited with relevant to secure coding practices since Secure Software Development originated in the early 2000s.

During the planning phase, I involved in determining the scope of this project along with other items which should be considered to achieve the goal. In the requirement gathering phase, I mainly involved in identifying the requirements of the plugin-based framework along with the major limitations and features of current software engineering solutions. During this process, I also involved in identifying the functional requirements of the solution.

I also designed system analysis and design artifacts such as sequence diagrams to understand the functionality of the system. This plugin-based framework is based on the SEI CERT secure coding rules for Java. The subcategories of last six main categories of SEI CERT secure coding rules including Thread Safety Miscellaneous, Input-Output, Serialization, Platform security, Runtime environment and Java Native Interface were classified into three granularity levels namely Method level, Class level and Package level based on classification criteria as mentioned previously. I designed the package level classification criteria. Package level primarily focused on the source code fragments that belong to classes outside the existing class. I.e. code fragments such as methods that belong to outside default package, extended classes and implemented interfaces outside default package, library imports, etc.

In order to implement these secure coding rules, five package level secure coding rules have been selected based on commonly violated guidelines, the likelihood of vulnerabilities and severity. Subsequently, algorithms for selected package level rules were implemented by me in the form of violation detection algorithms. The source code was parsed by a Java parser and the relevant source code fragments were identified. To store these extracted source code fragments, HashMaps and ArrayLists were used as data structures. Source code fragments extracted from an AST were passed to the violation detection algorithms in order to detect violations. The required source code fragments were extracted using relevant java parser methods, classes and interfaces. It was a challenging task. Finally necessary countermeasures for package level secure coding rules were designed.

I implemented the countermeasure component of the plugin-based framework to make it more user-friendly. Countermeasures were included in the plugin-based framework as a separate module since it was convenient to maintain a separate module for them and these were required by secure

coding rules of all three granularity levels. A separate HashMap data structure was maintained in the framework in order to facilitate this requirement. After successful completion of this component, it was integrated into the output generator in order to provide countermeasures for violated secure coding rules in the source code. Once a user clicks on the detected secure coding rule violation in the toolwindow, the relevant countermeasure will be displayed. During the implementation process, several difficulties were faced. By posting questions on JetBrains IDE Support, I was able to resolve them.

Following successful completion of the plugin development process, the framework needs to be deployed. The deployment procedure of this plugin-based framework was carried out by me. This was a challenging task because it is necessary to make sure that it works as intended. The proper working of the framework was achieved by installing a built on a fresh instance of IntelliJ IDEA IDE. Subsequently, manual testing was conducted. A zip archive was created including all the plugin libraries specified in the project settings. After submitting the secure coding plugin-based framework, it was successfully uploaded to the JetBrains plugins repository. Once the Framework for Secure Coding plugin has been approved by JetBrains plugin administration team, it was publicly available to download in the JetBrains Plugin Repository. Since we made some changes in the plugin based framework new updates of the plugin were also uploaded to the repository.

Finally, performance based evaluation of the plugin-based framework was carried out by me in order to ensure that its resource consumption was reasonable. Usually IntelliJ IDEA IDE consumes a considerable amount of memory and Central Processing Unit. As a result, the performance was evaluated concerning the main memory and CPU usage compared with the source code size with the help of suitable software profiling tools. VisualVM and JProfiler were used as software profiling tools in this evaluation methodology. Subsequently, results were extracted from above performance based evaluation and they were organized in a manner that was suitable for analysis. The main focus of this performance evaluation criteria was to assess the contribution of the system design in improving the performance of the framework. Benchmark tool comparison was also carried out using Sonarlint IntelliJ plugin. From the result obtained, it is clear that the plugin was reasonable in its resource consumption with respect to benchmark tool.

As a team, we were able to achieve this goal because of hard working and commitment of every member of the team who worked at same capacity and sharing of knowledge with the team.

USER'S MANUAL



Framework for Secure Coding

January, 2019

USER'S MANUAL

TABLE OF CONTENTS

1.0	GENERAL INFORMATION.....
1.1	System Overview
1.2	Points of Contact
1.3	Organization of the Manual
2.0	SYSTEM SUMMARY.....
2.1	System Configuration
2.2	Contingencies
3.0	GETTING STARTED.....
3.1	Installation
3.2	Running the plugin
3.3	Exit System
4.0	USING THE SYSTEM.....
4.1	Navigating the Results
4.2	Applying countermeasures
5.0	EXTENDING THE SYSTEM.....
5.1	Adding a new secure coding violation detection algorithm
5.2	Adding a new source code granularity level
6.0	UNINSTALLING.....

1.0 GENERAL INFORMATION

1.1 System Overview

This manual presents the Framework for Secure Coding plugin for IntelliJ IDEA. This plugin is a static code analyzer that automatically performs analysis of Java code written by software developers. This IntelliJ plugin aids software developers to write more secure codes by allowing them to inspect the results given by the plugin and applying countermeasures to detected secure coding rule violations.

1.2 Points of Contact

Following persons can be contracted for informational and troubleshooting purposes.

1. Sachintha Lasith Dasanayake (Developer) lasithd2@gmail.com
2. Lahiru Tharanga Perera (Developer) mlt.perera93@gmail.com
3. Arosha Mudalige (Developer) aroshamudalige1@gmail.com

1.3 Organization of the Manual

Section 1.0 of this manual presents the general information about the plugin and contact details of the relevant personals if any assistance is needed. Section 2.0 presents a summary of the system. Section 3.0 presents the instructions on how to install and run the plugin. Section 4.0 presents how to use the plugin and section 5.0 presents instructions on how to extend the plugin. Section 6.0 gives instructions on how to uninstall the plugin.

2.0 SYSTEM SUMMARY

2.1 System Configuration

Framework for secure coding plugin supports a variety of IDEs provided by JetBrains. They are *IntelliJ IDEA*, *PhpStorm*, *WebStorm*, *PyCharm*, *RubyMine*, *AppCode*, *CLion*, *GoLand*, *DataGrip*, *Rider*, *MPS* and *Android Studio*. After installation on the IDE, this plugin can be used immediately without any further configuration.

Following is the list of all the versions of the plugin and compatibility build numbers of each version.

VERSION	COMPATIBILITY	UPDATE DATE
▼ 1.2	182.3684–182.*	Dec 22, 2018
▼ 1.1	182.3684–182.*	Nov 06, 2018
▼ 1.0-SNAPSHOT	181.5087–181.*	Oct 28, 2018

2.2 Contingencies

Users cannot access previously analysed results since the plugin will save no user data or source codes. In case there is no Internet connection available users won't be able to access links provided by the plugin to access more details about the secure coding rules.

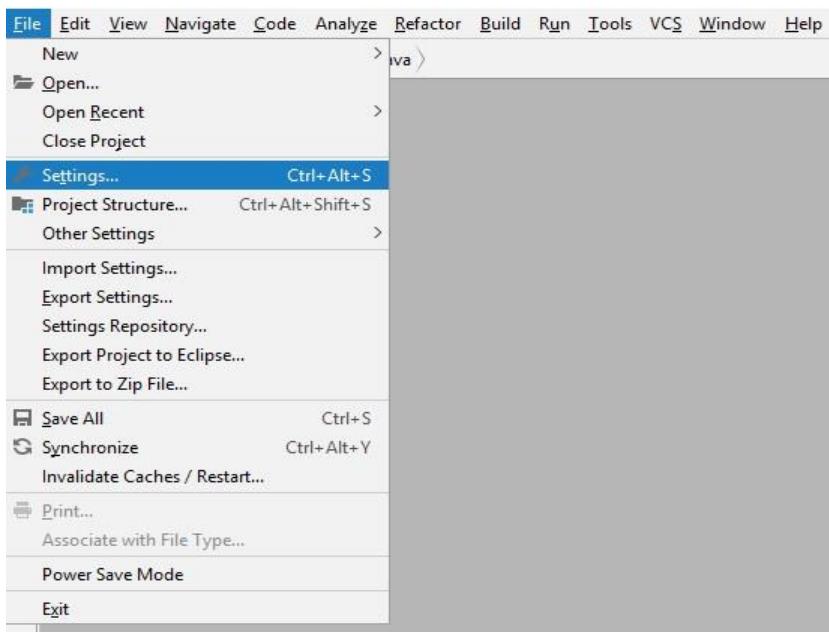
3.0 GETTING STARTED

3.1 Installation

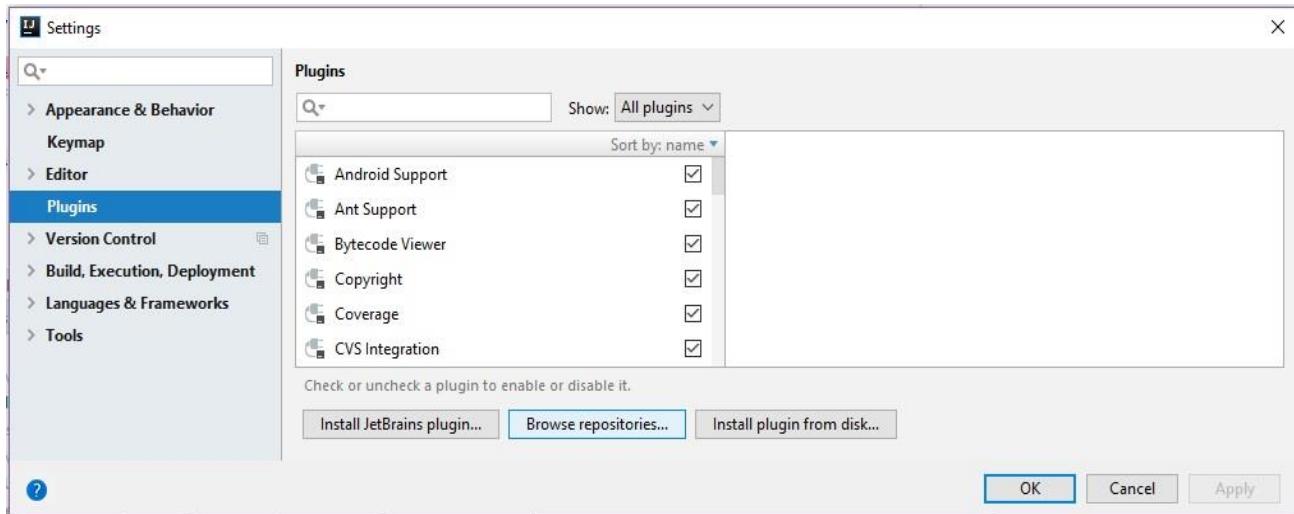
Framework for Secure Coding plugin is compatible with the following IDEs. If you have any of these IDEs installed, you can install the plugin using the instructions given below.

IntelliJ IDEA, PhpStorm, WebStorm, PyCharm, RubyMine, AppCode, CLion, GoLand, DataGrip, Rider, MPS and Android Studio

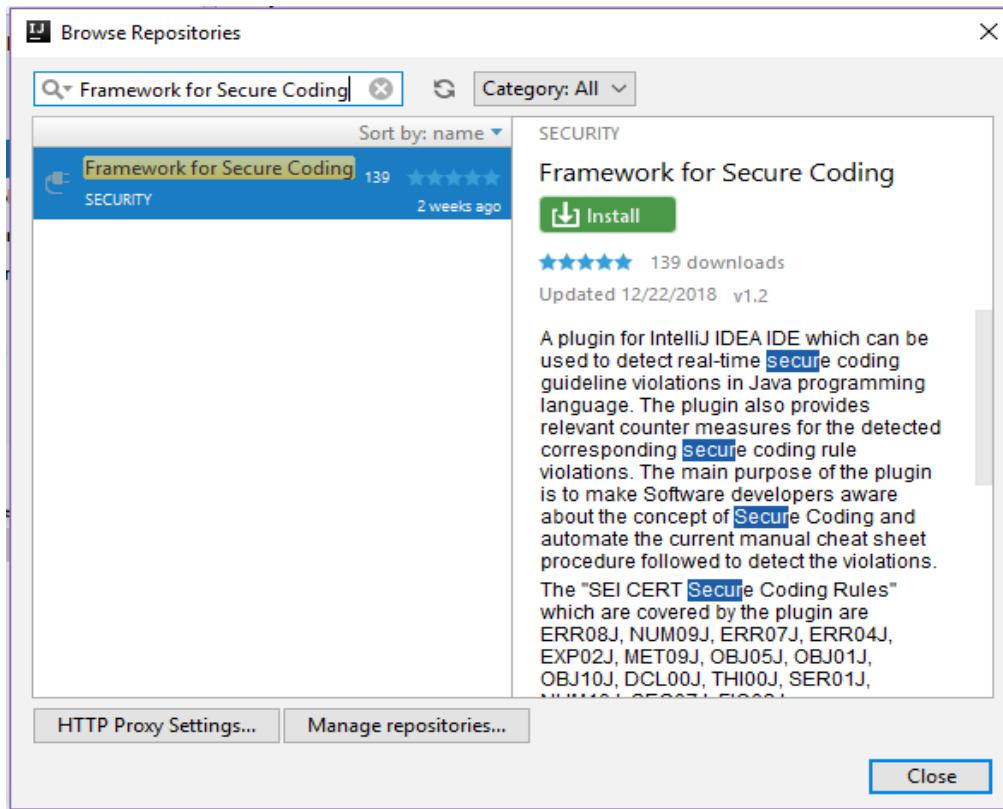
First of all, you need to navigate to settings panel by clicking *File->Settings*. The “Settings” panel will appear.



From there, choose the “Plugins” from the list on the left and the “Browse repositories...” button.



Now type “Framework for Secure Coding” in the search bar on the top. This step requires you to have an active internet connection. Under the search results select the plugin as shown in the below screenshot.



Finally, you have to click on the “Install” button to start the download and installation of the plugin. At the end of the installation, you will need to restart the IDE.

3.2 Running the plugin

Framework for Secure Coding plugin will automatically carry out analysis while developers are coding. This plugin will highlight the places in the code which violates a specific secure coding rule. Once the user hovers over to these highlighted places, a tooltip will be shown, briefly stating the violated secure coding rule. Then the user can access more information about these violations and how to correct them by navigating the results in the tool window.

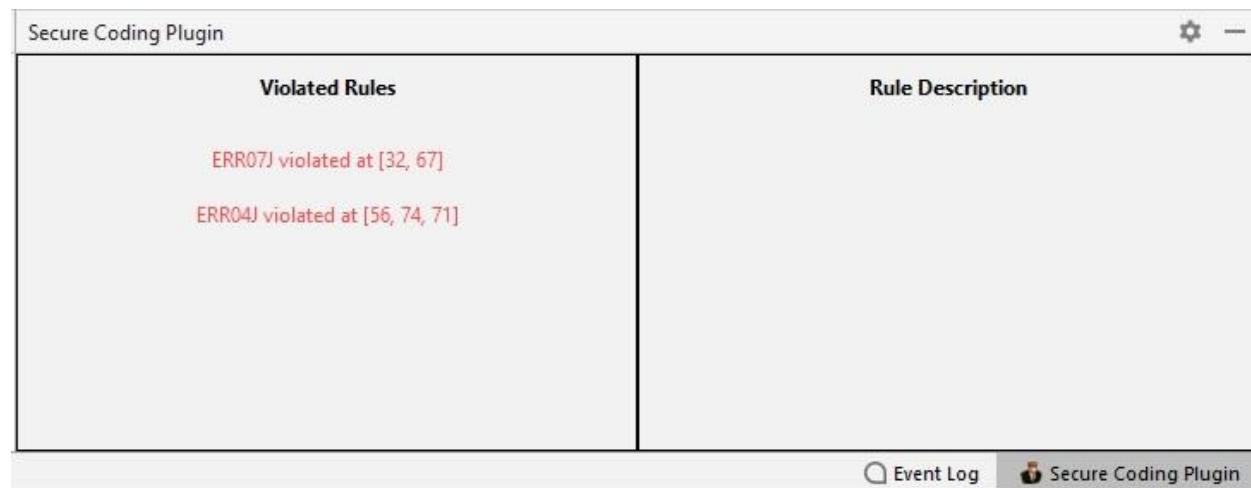
3.3 Exit Plugin

The plugin will exit once the IDE is closed and no user data or source codes will be saved.

4.0 USING THE SYSTEM

4.1 Navigating the Results

First of all, open the “Secure Coding Plugin“ tool window by clicking *View>Tool Windows>Secure Coding Plugin*. In the tool window, you can see two columns as “Violated Rule“ and “Rule Description“. “Violated Rules“ column will show every rule that has been violated by the currently opened source code in the IDE. The following screenshot depicts that each of the results in this column will show a rule id and the line number/numbers in the code that has violated that particular rule.



Once you click on these results the “Rule Description“ column will show you more information about these violated rules as shown in the below screenshot. This information includes the severity of the rule, the likelihood of the rule, the remediation cost of the rule, the priority of the rule etc.

The screenshot shows a window titled "Secure Coding Plugin". On the left, under "Violated Rules", there are two entries: "ERR07J violated at [32, 67]" and "ERR04J violated at [56, 74, 71]". On the right, under "Rule Description", it shows "ERR04J : Do not complete abruptly from a finally block". Below this, there is a table with columns: Severity, Likelihood, Remediation Cost, Priority, and Level. The values are: Low, Probable, Medium, P4, and L3 respectively. At the bottom, there is a link "Click here for more details".

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	P4	L3

4.2 Applying Countermeasures

On the bottom left of the “Rule Description“ column you will see a link to a web page where you can access the full specification of that secure coding rule as documented by SEI CERT. After opening this link in a browser you can get more detailed information about the secure coding rule. You can go through rule descriptions, code examples given by them and come up with a code that will eliminate the security vulnerability you previously had. Once you have eliminated these vulnerabilities you will no longer see them under tool window results.

5.0 EXTENDING THE SYSTEM

5.1 Adding a new secure coding violation detection algorithm

Follow the below steps when adding a new secure coding violation detection algorithm

1. Download the source code of the plugin from the bitbucket repository and set up a new project using it in IntelliJ IDEA. (Link to the source code is given in the plugin’s official page in JetBrains plugin repository.)
2. Navigate to the CodeFragments folder (/src/main/java/CodeFragments/). There you will find 3 code fragment classes corresponding to each violation detector classes namely method level, class level and package level that stores the source code fragments in various data structures such as ArrayLists and HashMaps.

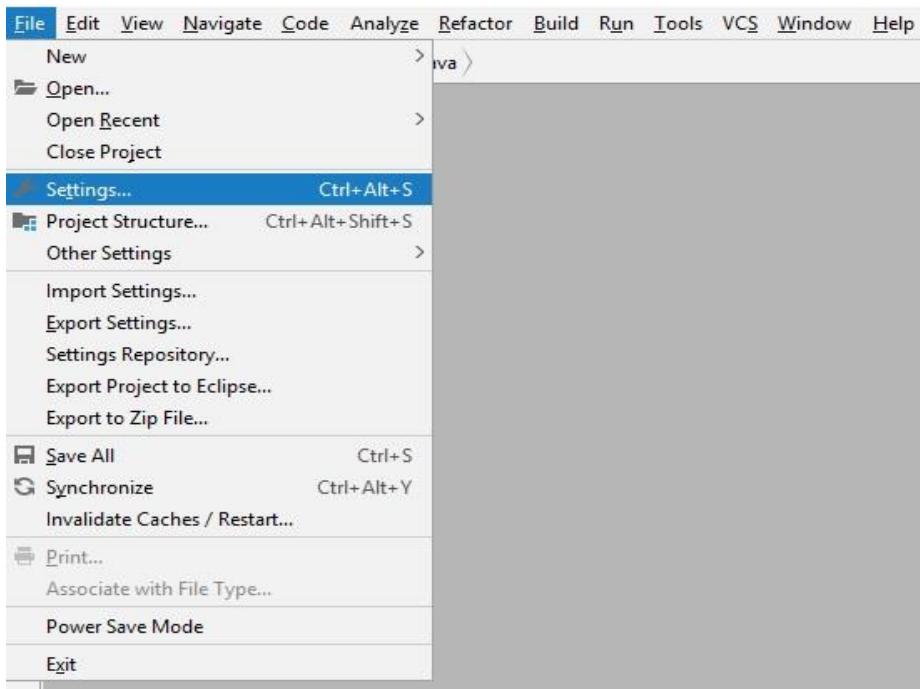
3. Access these three codeFragment classes and if the code fragments that the new secure coding algorithm need is found then use it. Otherwise, create a new class called ExtendeCodeFragments.java and define the code fragmnet extracting logic in it. Then add the extracted code fragment to the common data structure which is HashMap<String, Object>.
4. Navigate to the ViolationDetectors folder (/src/main/java/ViolationDetectors/). In this folder create a new class called ExtendedViolationDetector and implement the algorithm of the new secure coding rule in it.
5. Finally, the object of the violation detection algorithm class needs to be instantiated in the /src/main/java/Tools/LiveParser.java class and the relevant violation detection method needs to be called.

5.2 Adding a new source code granularity level

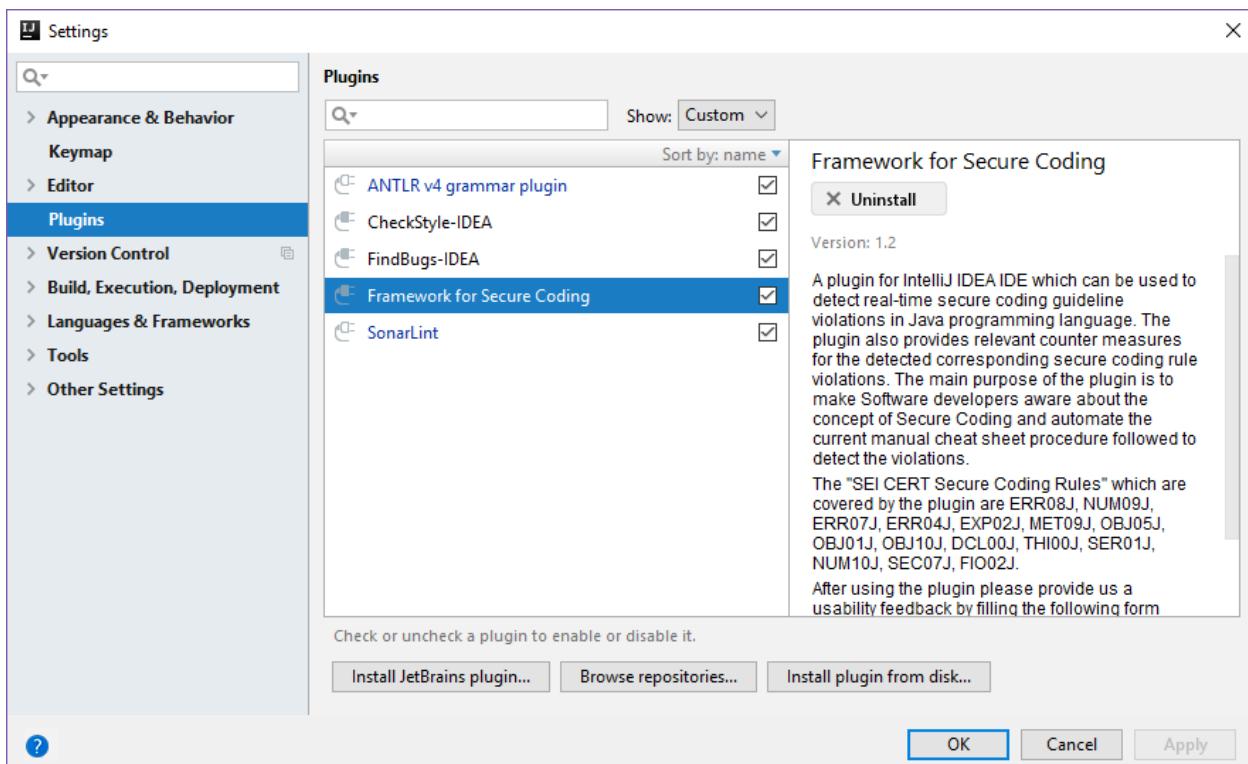
1. Download the source code of the plugin from the bitbucket repository and set up a new project using it in IntelliJ IDEA. (Link to the source code is given in the plugin's official page in JetBrains plugin repository.)
2. Add the name of the new granularity level violation detector class into the /src/main/java/ViolationDetectors/DetectorFactory.java class which contains the names of the other 3 violation detector classes.
3. Then create the respective violation detector class in a similar manner as the other existing violation detector classes.

6.0 UNINSTALLING

To uninstall the plugin, navigate to settings panel by clicking *File->Settings*. The “Settings” panel will appear.



From there, choose the “Plugins” from the list on the left and then select “Framework for Secure Coding” from the plugin list. Then click on the “Uninstall” button and after uninstallation of the plugin, you will need to restart the IDE.



The End