Code sniffer: a risk based smell detection framework to enhance code quality using static code analysis

Ahmad Tahmid, Md. Nurul Ahad Tawhid*, Sumon Ahmed and Kazi Sakib

Institute of Information Technology,
University of Dhaka,
Dhaka, Bangladesh
Email: a.tahmiid@gmail.com
Email: tawhid@iit.du.ac.bd
Email: sumon@du.ac.bd
Email: sakib@iit.du.ac.bd
*Corresponding author

Abstract: To maintain software and enhance its code quality, code smell, i.e., undesired design flaws need to be detected. However, as the system size increases, manual smell detection becomes difficult. In this paper, a static code analysis framework, named *code sniffer*, is proposed to detect code smells with predicting their risk severity. This has been calculated using code metrics, and defined as low, moderate and high. The system consists of three main components: *parser* extracts a syntax tree from the given source code to identify the code structure. The syntax tree is searched against the syntax of class and method. *Analyser* searches found classes and methods against various code syntax to identify key features like line of code (LOC), number of properties (NOP), etc. In *smell and risk detector*, code smells, risky code segments and their severity are detected as a set of quantitative values (using LOC, NOP, etc.) using which classes and methods are judged. A comparative case study of this risk based static analyser is performed with a dynamic analyser, FxCop, and the comparison results support each other.

Keywords: code smell; static analysis; risk based smell detection; code quality.

Reference to this paper should be made as follows: Tahmid, A., Tawhid, M.N.A., Ahmed, S. and Sakib, K. (2017) 'Code sniffer: a risk based smell detection framework to enhance code quality using static code analysis', *Int. J. Software Engineering, Technology and Applications*, Vol. 2, No. 1, pp.41–63.

Biographical notes: Ahmad Tahmid is a young researcher, who is currently doing his Master's at the Institute of Information Technology (IIT), University of Dhaka (DU). He completed his Bachelor of Science in Software Engineering from the same institute in 2014. He started to work on code smell detection under the supervision of Kazi Muheymin-Us-Sakib during his undergraduate studies. Currently, his research focus is code smell analysis and re-factoring.

Md. Nurul Ahad Tawhid completed his Bachelor and Masters from the Department of Computer Science and Engineering, at the University of Dhaka. His research interest includes image processing, artificial intelligence and software engineering. He is currently working as an Assistant Professor in Institute of Information Technology, at the University of Dhaka.

Sumon Ahmed completed his Bachelor and Masters of Science from the Department of Computer Science and Engineering at the University of Dhaka. His research interest covers computational biology and bioinformatics, machine learning, large-scale modelling and simulation, software engineering, agile software development and software test case prioritisation. Currently, he is working as an Assistant Professor in the Institute of Information Technology, at the University of Dhaka.

Kazi Sakib completed his Bachelor and Masters of Science from the Department of Computer Science at the University of Dhaka. He achieved his PhD from the RMIT University, Australia. His research interest includes distributed systems and software engineering. Currently, he is working as a Professor in Institute of Information Technology, at the University of Dhaka.

1 Introduction

Undesired design flaws in source code which degrades readability, updatability and overall maintainability of software are known as code smells. The value of modern day software depends much on the maintainability of its source code. However, the design flaws of source code have strong negative impact on software maintainability (Tahmid et al., 2016). So the detection of code segments with flaws is important for software evaluation and improvement.

To maintain the quality of a code by eliminating code smells, it is important to re-factor it on regular basis. Identifying the code segments to re-factor requires more expertise, effort and time. It is believed that human intuition is the best way to detect code smells (Roperia, 2009). However, manual detection becomes more difficult as system size increases. This creates the need for automated detection, especially for larger systems. The need of code smell detection mechanism with lower time cost and complexity is increasing day by day (Fard and Mesbah, 2013). There are few automated smell detection tools available in the market such as FxCop (Kresowaty, 2008), JSNose (Fard and Mesbah, 2013), but most of these tools require running dynamic analysis on compilable projects. So detection is incomplete without considering non-compilable projects.

Studies that concentrated on code smell detection and re-factoring, followed mainly three approaches. In rule-based detection approach, a list of rules is manually designed to detect various flaws of object oriented design at method, class and subsystem levels (Fard and Mesbah, 2013), but the main problem is to define the threshold value for metrices manually. In correction based approach, source code is re-factored to improve the certain quality properties, which is considered as an optimisation problem (O'Keeffe and Cinnéide, 2008). In visual-based detection approach, solutions try to draw attention of the user to the suspicious code segments (Parnin et al., 2008), but it offers minimal service as an automated code smell detection tool. None of these approaches can identify code segments with risk probability of having code smells. Rule-based detection approach and visual-based detection approach are combined in another study (Abdelmoez et al., 2014) where the authors have introduced the concept of risk on detected code smells. However, it runs only with method level code segments therefore overall code quality cannot be judged.

The proposed framework ‘code sniffer’ performs static analysis to detect code smells such as God class (GC), lazy class (LC), long method (LM), long parameter list (LPL) and switch statement (SS). It uses metrics based approach to detect code smells in method and class level. It also categorises the detected code smells into low, moderate and high-based on the severity of risk associated to those. The framework works in two phases: firstly it parses the given source code and calculates various code metrics using an ‘analyser’. Some smell detection criteria are generated from previous studies because those represents design and implementation problems with data, size, complexity and features provided by the classes and methods. Based on these, a ‘smell and risk detector’ detects code smells with risk severity in the second phase.

The tool is tested on some open source projects from ‘github.com’ and code blocks of those projects are classified as low, moderate or highly risky-based on the code metrics. The results are then compared with a dynamic analyser, FxCop (Kresowaty, 2008). Although for LC the output is indecisive, for other four selected smells, the tool and FxCop supports each other. The code blocks, which are identified as highly risky by the tool, also have the low maintainability index (MI) given by FxCop. Similarly, the code blocks that are identified as low risky have the high MI in FxCop.

This paper is organised in the following sections: Section 2 will give a relative overview of code smell, source code analysis and different approaches of code smell detection. Section 3 will describe some theoretical background related to code analysis, code quality and; Section 4 will give an architectural overview of the proposed tool which will be followed by a case study of the tool in Section 5; in Section 6, some threats to the validity of the tool will be discussed and finally in Section 7, some conclusive remarks regarding this work and possible future enhancements will be mentioned.

2 Related works

According to Fowler (2006), “a code smell is a surface indication that usually corresponds to a deeper problem in the system”. In other words, smells are certain code structures that violate the fundamental design principles. Code smells are neither bugs nor technically incorrect and also do not stop the program functioning.

Many different approaches have been considered to automatically identify code smells. According to Fowler et al. (1999), there are 22 code smells in object-oriented source codes. A large number of studies has concentrated on code smell detection and re-factoring over the last decade. Those studies include both static and dynamic source code analysis.

2.1 Types of code smell detection

From detailed analysis of previous works, there are four main approaches to detect code smells. Those are:

2.1.1 Rule based detection approach

In this approach, a list of rules is designed to detect various flaws of object oriented design at method, class and subsystem levels (Marinescu, 2004). In study (Erni and Lewerentz, 1996), the authors have proposed a framework for evaluating source code where the concept of multi-metrics is used to express a quality criterion. Multi-metrics is *n*-tuple of metrics such as a multi-metric (MM) for coupling:

MM coupling = {numberof references to other classes,
number of bidirectional references,
number of references to abstract classes}.

In this approach the main problem is to define the threshold value for matrices manually. To resolve this problem, in another study, Alikacem and Sahraoui (2006) have used fuzzy logic to define metrics. For example, the research has determined the smell detection rules as small, medium and large. Although there is no pre-defined threshold, the question is how to define the membership functions, which is associated to a given fuzzy set that maps an input value to its appropriate membership value. For detecting defected codes, Moha et al. (2010) used DÉCOR methodology. They have formulated an algorithm by compiling defect symptoms into an abstract language. Because of unavailability of accepted symptom-based definition of design defects, translating the symptoms into rules is the main problem (Brown et al., 1998).

2.1.2 Correction-based approach

The studies that follow this approach do not try to detect defects or code smells. Rather those attempt to re-factor source code to improve certain quality properties. So this is basically an optimisation problem.

In study (O'Keeffe and Cinnéide, 2008), the authors have applied a sequence of refactoring in source code. Then 12 metrics are used to measure improvements. Comparing results before and after re-factoring is presented as achievements. The aim of that study was to improve certain quality attributes of the source code. So the source quality improved only in terms of metrics. The relation between change and its effect on code smells is not clearly defined. Although this approach shows significant improvement, it is difficult to understand and use in real life situation because of determining the useful metrics for a given system.

2.1.3 Visual-based detection approach

It is believed that human intuition is the best way to detect code smells (Roperia, 2009). The visual-based detection solutions take advantage of human intuitions, as human can easily integrate contextual information with smell detection rule set. The best way to detect code smells is to involve human in detection process.

In study, Kothari et al. (2004) have proposed to represent potential defects by detecting software anomalies. The user is presented with some automatically detected symptoms to analyse and finally detect code smells in another study (Dhambri et al., 2008). Although this is more accurate approach, but it is manual. In addition, the success depends on user's expertise and also it is time intensive.

2.1.4 Risk-based code smell detection tool

In this study, the focus is on the impact of detected code smells. Abdelmoez et al. (2014) have proposed a tool that assesses risk issues in source code and informs users the severity of the issue. It targets only four code smells and tries to detect those using static analysis. Risk factor levels (high, low and medium) have been qualitatively associated with each code smells based on the frequency of occurrences and the severity of each code smell. The targeted smells are: empty catch, LPL, message chain, and LM. It identifies the most risky components and informs users about those. The approach is a combination of rule based and visual-based detection and hence, it is not fully automated.

The review of the existing research shows that, none of the processes considered the categorisation of code blocks according to their risk severity for certain code smells. This will help the developer to re-factor code during the development phase of the project and reduce effort of maintenance in the future. Furthermore, most of the code flaw detection tools are developed for dynamic analysis, causing the tool unusable for non-compliable code in the middle of the development. So, a tool for static source code analysis and detecting their risk severity is needed, so that some re-factoring can be done in the development phase and as well as in the maintenance phase.

3 Background study

This section describes some theoretical background related to software code analysis, code quality and maintainability. Also some key feature definitions related to those are given.

3.1 Static and dynamic analysis

To detect the code smell, the source code has to be analysed. This can be done in two ways: static and dynamic analysis.

Static program analysis is the process of analysing the computer software without executing programs (Gomes et al., 2009). In most cases, the analysis is performed on some version of the source code. It is the simplest way of code analysis as it gives an overall idea of the software quality and can identify many common coding problems before the software is released.

Dynamic program analysis is the process of analysing the computer software by executing programs on a real or virtual processor (Gomes et al., 2009). Dynamic analysis is more accurate but has high cost of computing, time and complexity. It is used to identify specific quality issues like, cyclomatic complexity (CC), MI etc.

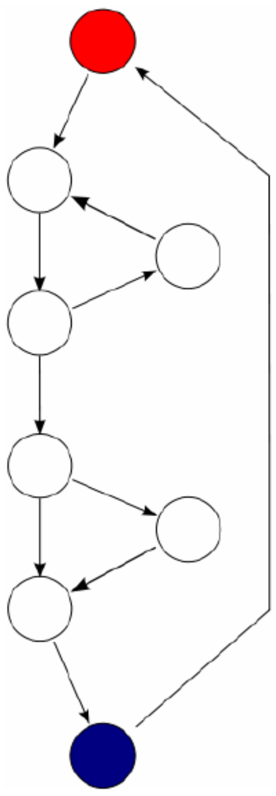
3.2 Software quality measurement

There are various measures to analyse the quality of a source code (Heitlager et al., 2007). Among those the most common measures are CC and MI. Both can be generated by dynamic code analysis.

3.2.1 Cyclomatic complexity

CC is the directional complexity of a code segment (McCabe, 1976). A code is more complex to understand if it has many different flows in it. It has no relation with real complexity. CC is calculated by generating flow graphs.

Figure 1 Code flow diagram to detect CC (see online version for colours)



After generating the flow graph, number of nodes (*N*) and number of edges (*E*) are calculated from it. There are several formulas to calculate *CC*, *C*. One of the formulas is given in equation (1) (McCabe, 1976).

$$\left.\begin{array}{l} C = E - N + P \\ Or, C = E - N + 2P \\ Or, C = E - N + 2 \end{array}\right\} \quad (1)$$

Here *P* is the number of connected components in the code flow graph.

3.2.2 Maintainability index

MI is calculated differently by different tools. Microsoft computes the maintainability of code segment based on the volume metric (Halstead, 1977), the CC (McCabe, 1976) metric and the average number of Lines of Code per module, using the following formula (Conorm, 2007).

$$MI = MAX\begin{pmatrix} 0, (171 - 5.2 * ln(Halstead\ Volume) \\ -0.23 * (Cyclomatic\ Complexity) \\ -16.2 * ln(Lines\ of\ Code)) * 100 / 171 \end{pmatrix} \quad (2)$$

It results a value between 0–100. MI has three ranges of values:

$$\begin{array}{ll} a & 0-9 \text{ } is \text{ Re}d \\ b & 10-19 \text{ } is\ Yellow \\ c & 20-100 \text{ } is\ Green \end{array} \tag{3}$$

Here, *green* indicates that the code has good maintainability. A *yellow* indicates that the code is moderately maintainable. A *red* indicates low maintainability.

3.3 Software maintainability

Software maintenance does not start from the initial stages of software development life cycle (SDLC) (Pigoski, 1996). This process starts when the software begin to take shape. This process deals with bug and defect fixes. It is called corrective maintenance (Pigoski, 1996). When this maintenance process handles changes such as updates, new features or requirements, etc., it is called adaptive maintenance (Pigoski, 1996). Maintenance requires developer’s skill and expertise. It can often be a costly process.

The cost of software maintenance is estimated 60%–80% of the overall software system cost. Corrective and adaptive maintenance occupy 78%–83% of the maintenance effort (Glass, 2001; Pigoski, 1996).

4 Code sniffer

To help the developer in development time, the proposed framework performs static code analysis on software source code and predicts the amount of risk for different code segments. This framework takes source code file(s) as input, parses the file(s) to extract source code and identifies some code metrics like line of code (LOC), number of properties (NOP) and number of parameters (PAR), etc. Then using those metrics, it performs a code smell analysis and categorises the classes and methods into three risk categories (low, moderate and high) depending on their probability of being smelly. Developer can use those risk categories for identifying the code segments that need re-factoring and improve the code quality at the development time.

As this study focuses on static code analysis, a metric based approach is followed for smell detection. Different researchers have defined smells using their own set of criteria. Based on those criteria a code segment is judged as either smelly or smell-free. However, in reality the concept of code smell is a bit vaguer. For example, according to Martin (2009), a 750 liners class is a GC but a 749 liners class in not. Although in reality there are not many differences in understandability of those classes. That is why, instead of identifying smells, this research have focused on identifying risks associated to those. To do so, the criteria defined by previous researches are analysed and compiled those as threshold for different level of risks (low, moderate and high). Although this levels are defined using the same metric values, the main merit of this approach is, it eliminates some confusions. For example, if a method matches the criteria of LM in high risk level, it can be said that, the method has high risk of having a LM smell.

4.1 Selection of metrics and criteria

Code metrics are a set of quantitative values derived from code, using which classes and methods are judged to provide better insight into the code. In Table 1, some code metrics are listed, which have been identified from previous studies on code smell (Fowler et al., 1999; Mäntylä et al., 2003; Marinescu and Lanza, 2006) and used in this framework.

Table 1 Code metrics

Metric	*Description (class)*	*Metric*	*Description (method)*
LOC	Lines of code	MLOC	Method lines of code
NOP	Number of properties	PAR	Number of parameters
NOM	Number of methods	NOC	Number of cases in switch

Some criteria for the given metrics are gathered to detect the targeted smells. Those are discussed below:

a God class

According to Williams et al. (Software Metrics in Eclipse, 2015), a GC contains more than 750 lines of code or more than 20 methods or more than 20 properties. So from Table 1, if a class has LOC > 750 or NOM > 20 or NOP > 20 is a GC.

According to ‘rule of 30’ by Lippert and Roock (2006), a class should contain less than 30 methods or less than 30 properties or less than 900 lines of code (Abdelmoez et al., 2014). So a class with LOC > 900 or NOM > 30 or NOP > 30 is a GC.

b Lazy class

In Fard and Mesbah (2013), the authors have identified LC where number of properties is less than 3. So, if NOP < 3, it is a LC. In more definite LC, both number of properties and number of methods are less than 3. So, if NOP < 3 and NOM < 3 then it is a LC.

c Long method

In clean code, Martin (2009) claimed a method should be less than 20 lines long. So, a method with LOC > 20 is a LM. On the other hand, in code complete, McConnell and Johannis (2004) gives a reference to a study that says routines 65 lines or longer are cheaper to develop. So, method with LOC > 65 is a LM.

d Long parameter list

Martin (2009) in book clean code referred that the number of parameters in a method should not be more than three. So, method with PAR > 3 has long parameter smell. In the study (Fard and Mesbah, 2013), more than four parameters in a method are considered tougher to maintain. So, PAR > 5 is a long parameter smell.

e Switch statements

According to Fard and Measbah (2013), if the number of case statements in switch block goes past three it is considered as a smell. So, a SS with NOC > 3 is smelly. More than five is considered highly risky, i.e., NOC > 5.

From the above analysis, threshold for detecting various code smells have been defined. For each code smell three levels of threshold (low, moderate and high risky) are defined in Table 2.

Table 2 Risk criteria for different smells

Code smell	*Detection criteria*		
	Low risk	*Moderate risk*	*High risk*
God class	LOC < 751 && NOP < 21 && NOM < 21	LOC > 750 or NOP > 20 or NOM > 20	LOC > 900 or NOP > 30 or NOM > 30
Lazy class	NOP > 2	NOP < 3	NOP < 3 && NOM < 3
Long method	LOC < 21	LOC > 20	LOC > 65
Long parameter list	PAR < 4	PAR > 3	PAR > 5
Switch statement	NOC < 4	NOC > 3	NOC> 5

4.2 Architecture of code sniffer

The proposed system parses the source code and categorises those into low, moderate or highly risky using the metrics defined in Table 2 of Section 4.1. The system consists of four components:

1 parser

2 analyser

3 smell and risk detector

4 report generator.

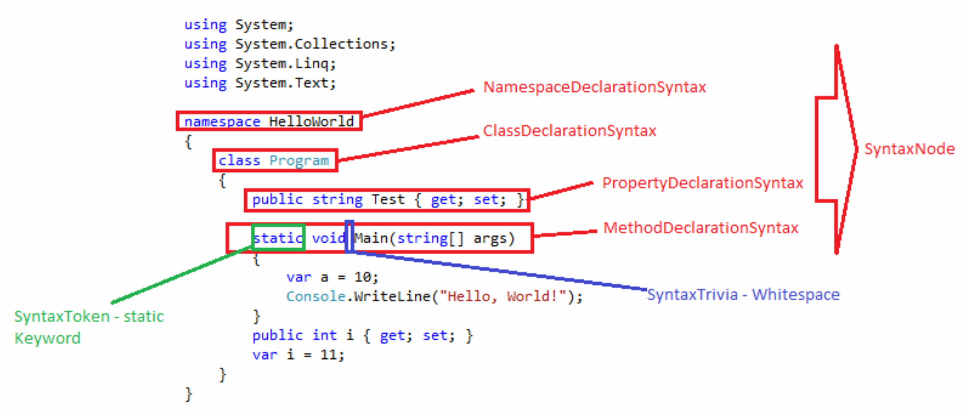
4.2.1 Parser

This module parses the file and extracts a syntax tree for the source code. To do so, .NET compiler platform *Roslyn* (https://github.com/dotnet/roslyn) is used. The syntax tree generated from *Roslyn* contains all the information from the source text like grammatical construct, lexical token, white space, comments, and preprocessor directives (Getting Started C# Syntax Analysis, 2017). Each syntax tree has following building blocks:

- *SyntaxNode* represents syntactic constructs such as declarations, statements, clauses, and expressions
- *SyntaxToken* represents an individual keyword, identifier, operator, or punctuation
- *SyntaxTrivia* represents syntactically insignificant bits of information such as the white space between tokens, preprocessing directives, and comments

Trivia, tokens, and nodes are composed hierarchically to form a tree that completely represents everything in a fragment of C# code. A sample of those three blocks in a code is shown in Figure 2, and Figure 3 shows the syntax tree structure.

Figure 2 Sample code with syntax node, token and trivia (see online version for colours)



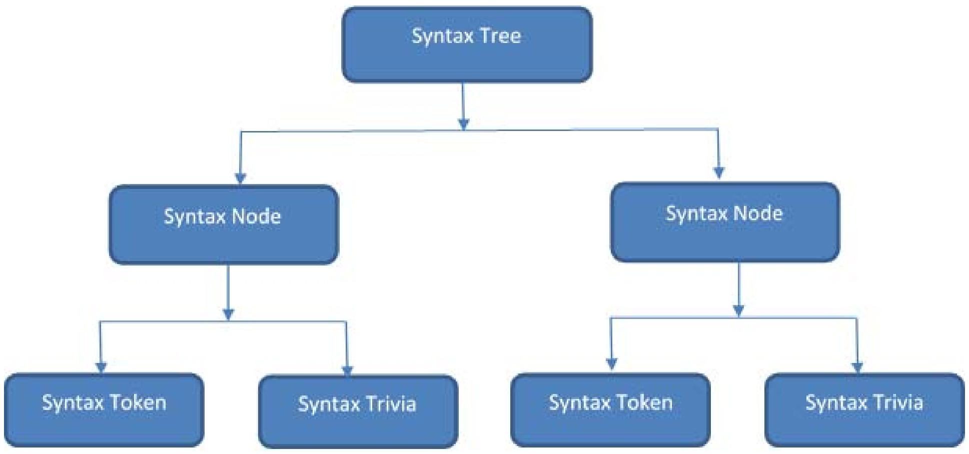
```
using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        public string Test { get; set; }

        static void Main(string[] args)
        {
            var a = 10;
            Console.WriteLine("Hello, World!");
        }
        public int i { get; set; }
        var i = 11;
    }
}
```

NamespaceDeclarationSyntax
ClassDeclarationSyntax
PropertyDeclarationSyntax
MethodDeclarationSyntax
SyntaxNode
SyntaxTrivia - Whitespace
SyntaxToken - static Keyword

Figure 3 Sample syntax tree structure (see online version for colours)



This syntax tree is then passed through a search component that searches for the syntax of class (*ClassDeclarationSyntax* from Figure 2) and method (*MethodDeclarationSyntax* from Figure 2) in the tree and identifies the classes and methods from the source code. Found classes and methods are sent to the next component named analyser which analyses the classes and methods found in this step for different code metrices. Figure 4 shows the architecture of the parser.

4.2.2 Analyser

Analyser is used to extract the source code metrices from the classes and methods found from the parser module. Analyser has two parts: one part handles the classes and other part handles the methods found from the syntax tree. In the class handler part, it searches the found classes against statement, property and method identification syntax, and identifies LOC, NOP and NOM. On the other hand, in the method handler, parameter, statement, switch and variable identification syntax are searched on found methods and PAR, MLOC, NOC and NOV are identified. It gathers the metric values for each class and method, and sends to the smell detector. Figure 5 shows the architecture of the analyser.

Figure 4 Parser architecture

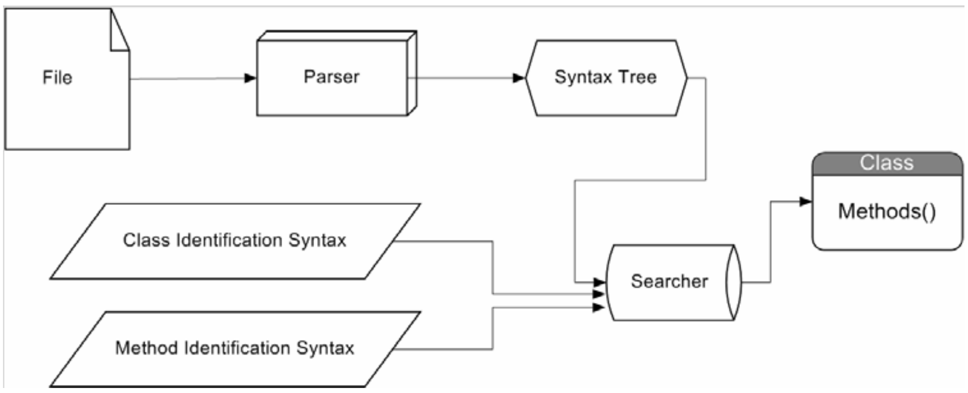
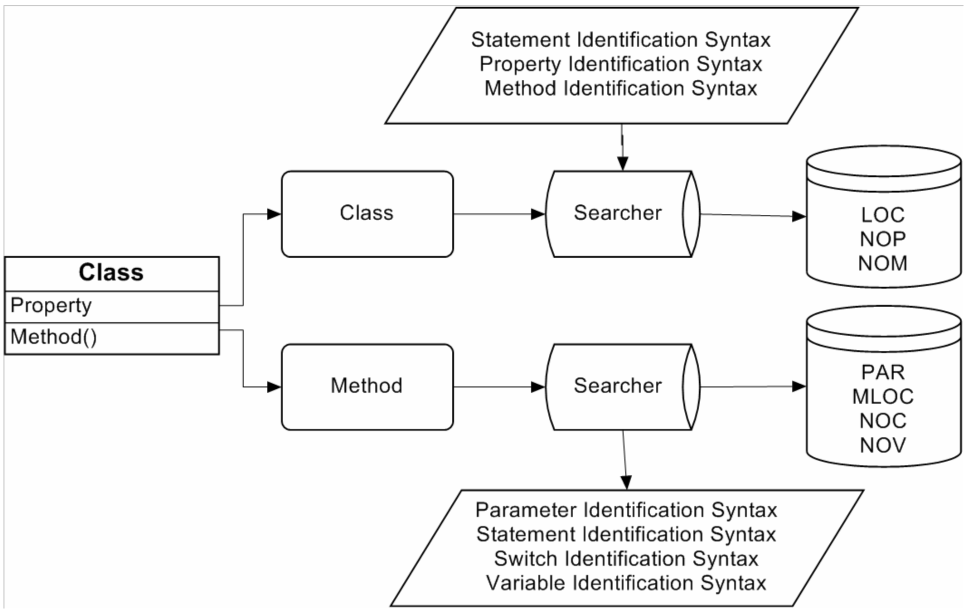


Figure 5 Analyser architecture


Statement Identification Syntax
Property Identification Syntax
Method Identification Syntax
Class
Searcher
LOC
NOP
NOM
Class
Property
Method()
Method
Searcher
PAR
MLOC
NOC
NOV
Parameter Identification Syntax
Statement Identification Syntax
Switch Identification Syntax
Variable Identification Syntax

4.2.3 Smell and risk detector

In smell and risk detector, the classes and methods identified in the parser module are categorised into one of three risk categories. To do so, it uses the code metrices identified in the analyser step and matches those metric values against the threshold value defined in the Table 2. Smell and risk detector has two parts: in class risk detector, class properties (LOC, NOP and NOM) are checked against GC and LC filtration criteria (second and third row of Table 2) and their severity of risk are detected. Similarly, in method risk detector, method properties (PAR, MLOC, NOC and NOV) are checked against LM, LPL and SS filtration criteria (fourth, fifth and sixth row of Table 2) and risk severity of those are detected. Figures 6 and 7 show the architecture of the smell and risk detector.

Figure 6 Smell and risk detector architecture for classes

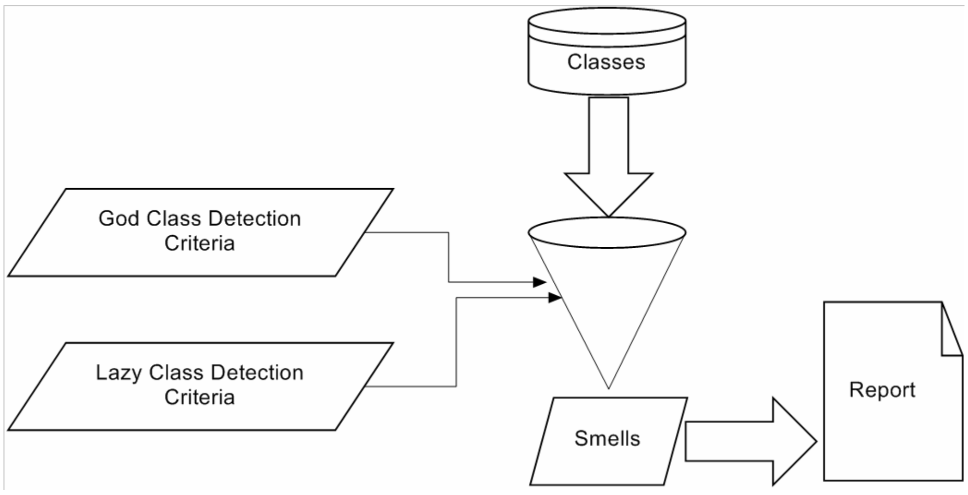
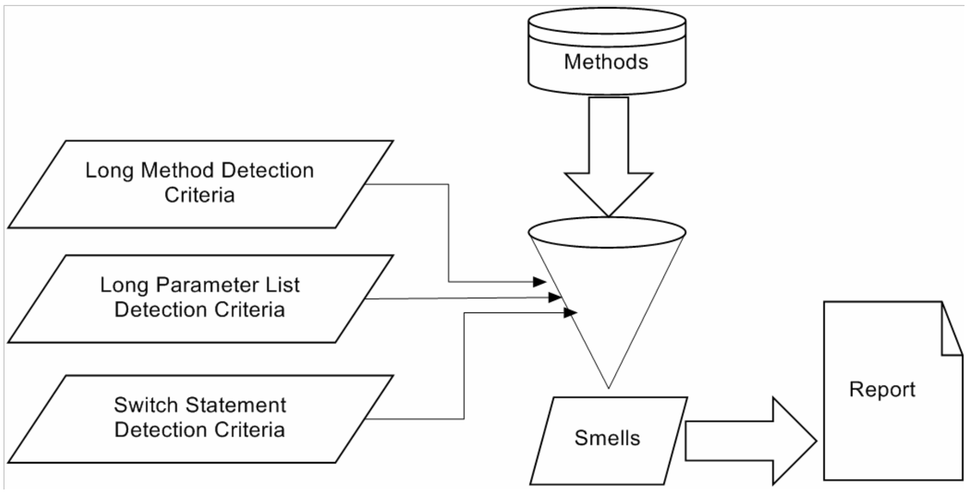


Figure 7 Smell and risk detector architecture for methods

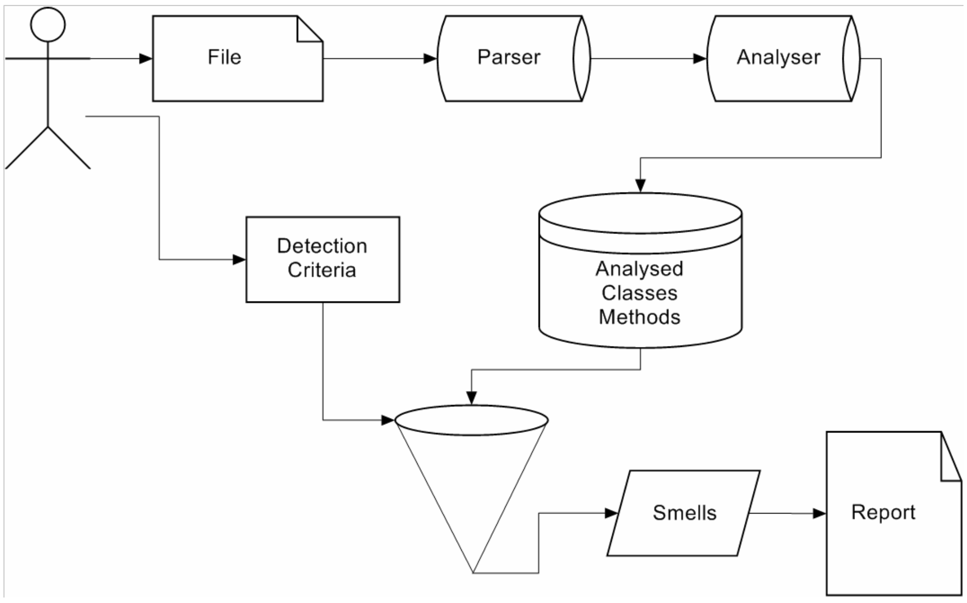

Methods
Long Method Detection Criteria
Long Parameter List Detection Criteria
Switch Statement Detection Criteria
Smells
Report

4.2.4 Report generator

Report generator visually and graphically presents the code smells, risky code segments and risk severity to the user.

Full architecture of the code sniffer system is given in Figure 8.

Figure 8 Code sniffer: full architecture



5 A case study of code sniffer

The proposed system is tested on four real world projects to evaluate the effectiveness and relevance of it. The study is designed to address the following research question:

RQ is there a relation between detected code smells' risk severity with the CC and MI associated to those?

5.1 Experimental objects

Four open source C# popular projects are selected for experiment from 'github.com', like *Microsoft ASP.NET MVC, Web API and Web Pages* source code (https://github.com/ASP-NET-MVC/aspnetwebstack). Another is *IdentityServer* (https://github.com/IdentityServer/IdentityServer3) which is a .NET/Katana-based framework used for single sign-on and access control for web applications and APIs using protocols like OpenID Connect and OAuth2. The experimental projects with their source code metrics are shown in Table 3. Here number of files, classes, methods and lines are extracted using the .NET Compiler Platform *Roslyn* (https://github.com/dotnet/roslyn). Experimental data along with the implementation of *Code Sniffer* are available for download (https://github.com/Tawhid-iitdu/CodeSniffer).

Table 3 Experimental projects overview

Project no.	*Name*	*#Files*	*#Classes*	*#Methods*	*#Lines*	*Description*	*Resource*
1	aspnetwebstack	2,988	3,918	24,119	417,075	Microsoft ASP.NET MVC, web API, and web pages source code.	https://github.com/ASP-NET-MVC/aspnetwebstack
2	Crisis check in	215	180	506	12,043	Crisis check in is an application meant to capture, share and integrate the data around volunteers, organisations and resources actively deployed into a disaster.	https://github.com/HTBox/crisischeckin
3	Identity server 3	477	498	1,974	57,519	.NET/katana-based framework used for single sign-on and access control for web applications and APIs using protocols like open id connect and Oauth2	https://github.com/IdentityServer/IdentityServer3
4	Sharp repository	371	406	2,369	30,840	C# generic repository for use with entity framework, raven DB and more with built-in caching options	https://github.com/SharpRepository/SharpRepository

5.2 Experimental setup

The proposed system is tested against a dynamic analyser FxCop (Kresowaty, 2008), an integrated tool in visual studio that calculates CC and MI of methods and classes. CC is a code metric that denotes the number of linearly independent paths through a programs source code. MI is a value between 0 to 100 which is derived from CC, LOC and Halstead volume to present the overall maintainability of a program.

The evaluation process of the relation between detected code smells and the CC and MI for those code segments, i.e., RQ are as follows:

- For detected code smells, MI will be high for low risky code segments as because those code segments are well designed. MI will decrease for medium and highly risky code segments because of smelly codes have low maintainability.
- On the other hand, CC will be low for low risky code segments and will increase for medium and highly risky code segments as because of smelly code have high CC value than non-smelly code.

5.3 Result analysis

Proposed tool is used to detect five different code smells on the four different projects from 'github.com'. Table 4 shows the results of the tool, where first column lists the five smell types, second column lists the risk severity for each smell type. Last four columns show the number of detected smells for four projects in different severity level such as among 24,119 methods of project 1, 19,321 methods are identified as low risky 4,758 methods are identified as moderate risky and 167 methods are identified as high risky. Similarly for project 2, among 506 methods, 361, 141 and four are respectively low, moderate and high risky LMs and so on. For example, in project 1, 'Serialise(AntiForgeryToken)' method in 'AntiForgeryWorker' class in 'src\System.Web.WebPages' project has LOC = 2, that is why it has low risk severity with respect to LM. On the other hand, 'Serialise(AntiForgeryToken)' method in 'AntiForgeryTokenSerialiser' class of 'src\System.Web.WebPages' project has LOC = 32, that is why it has moderate risk severity with respect to LM. In the same way 'OnAuthorisation(AuthorisationContext)' method in 'FacebookAuthoriseFilter' class of 'src\Microsoft.AspNet.Facebook' project has LOC = 110, that is why it has high risk severity with respect to LM.

CC and MI for the methods and classes of the four projects are calculated using FxCop. Then average CC and MI for the different code smells are calculated based on their severity. Table 5 shows the project wise detected code smells, risk vs. MI and risk vs. CC analysed by FxCop.

From Table 5, for each type of detected code smells except LC, CC is low for low risky code segments detected by the tool and it increases for moderate and highly risky code segments for every project. Similarly, project wise MI is high for low risky code segments except for LC and it decreases for moderate and highly risky code segments. Since LC have too little functionality, those have almost reverse relationship with CC and MI compared to other four code smells. A combined result for all four projects is given in Table 6.

Table 4 Analysis result by the proposed tool

Smell type	*Severity*	*Project 1*	*Project 2*	*Project 3*	*Project 4*
Long method	Low	19,321	361	1,495	2,232
	Moderate	47,58	141	429	183
	High	167	4	50	8
Long parameter list	Low	22,967	479	1,870	2,273
	Moderate	1,023	17	84	149
	High	256	10	20	1
Switch statement	Low	24,203	505	1,953	2,414
	Moderate	14	0	4	6
	High	29	1	17	3
God class	Low	3,670	173	476	398
	Moderate	141	7	9	7
	High	134	0	13	11
Lazy class	Low	2,241	102	274	223
	Moderate	809	45	88	90
	High	895	33	136	103

Figure 9 LM result comparison (see online version for colours)

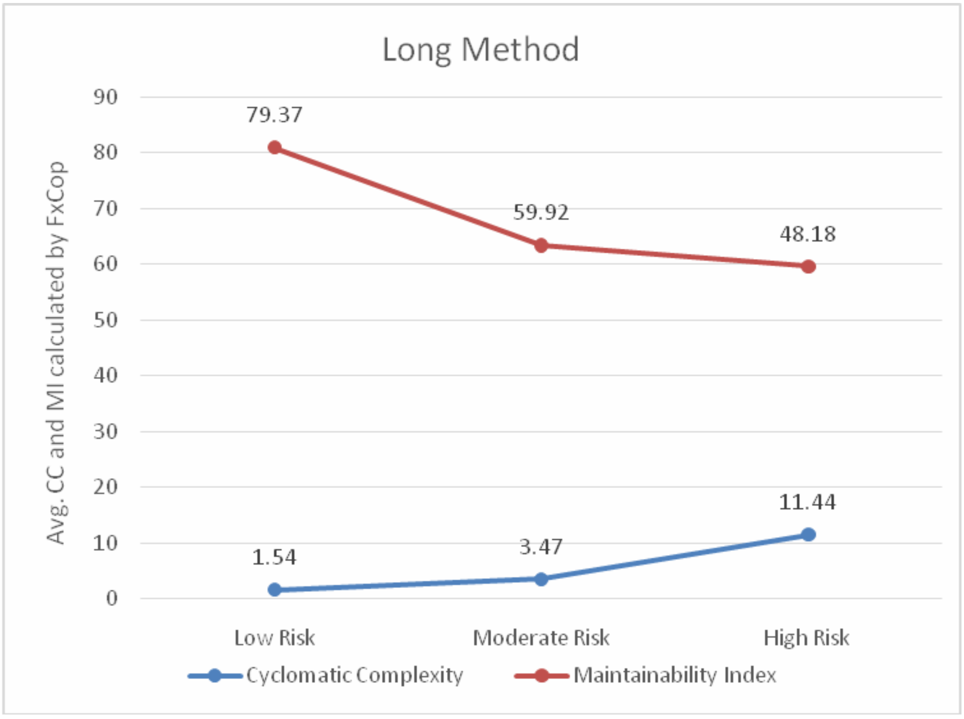


Table 5 Average CC and average MI calculated by FxCop (project wise)

Smell		*Project 1*			*Project 2*			*Project 3*			*Project 3*		
		Low risk	*Moderate risk*	*High risk*	*Low risk*	*Moderate risk*	*High risk*	*Low risk*	*Moderate risk*	*High risk*	*Low risk*	*Moderate risk*	*High risk*
LM	CC	1.53	3.49	12.71	1.48	2.7	20.33	1.63	3.26	6.23	1.57	3.85	18.88
	MI	79.37	59.51	45.36	76.24	55.14	33	76.67	66.21	57.29	81.63	59.99	44.5
LPL	CC	1.93	2.59	4.35	1.94	2.03	2.7	2.1	2.71	3.45	1.79	1.95	1
	MI	75.38	74.75	68.88	70.15	70	63.8	74.12	71.78	64.15	79.99	78.84	69
SS	CC	1.96	10	14.39	1.93	NILL	11.00	2.07	3	16.67	1.73	10.33	15.67
	MI	75.33	60.71	55.75	70.03	NILL	63	73.96	69	53.33	80.05	55.17	48.67
GC	CC	11.09	53.48	96.15	9.24	41.57	NILL	8.52	37.25	100.54	13.6	68	135.45
	MI	82.07	70.71	67.28	83.04	67	NILL	81.04	76.88	73.46	83.04	73.43	75.82
LC	CC	31.70	21.30	3.61	13.76	19.21	3.15	18.73	18.22	3.37	38.43	23.25	7.64
	MI	80.52	74.39	87.92	91.89	64.54	86.57	89.59	72.77	83.84	85.72	77.25	86.26

Figure 10 LPL result comparison (see online version for colours)

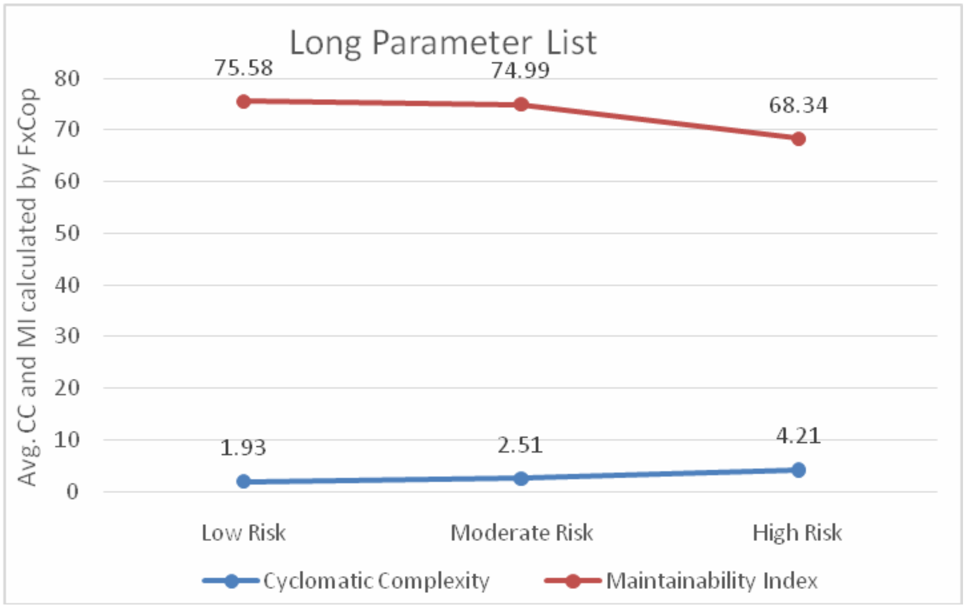


Figure 11 SS result comparison (see online version for colours)

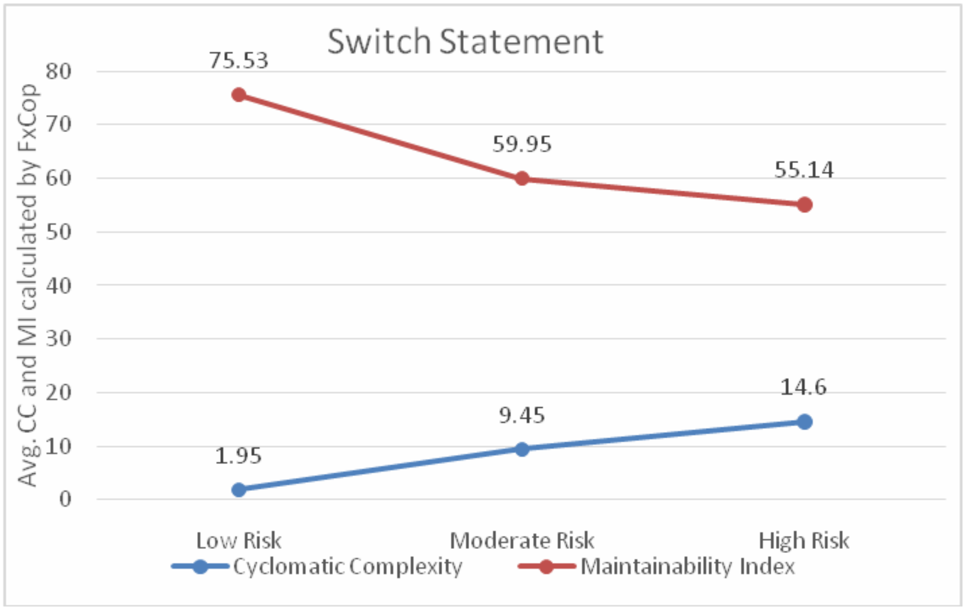


Figure 12 GC result comparison (see online version for colours)

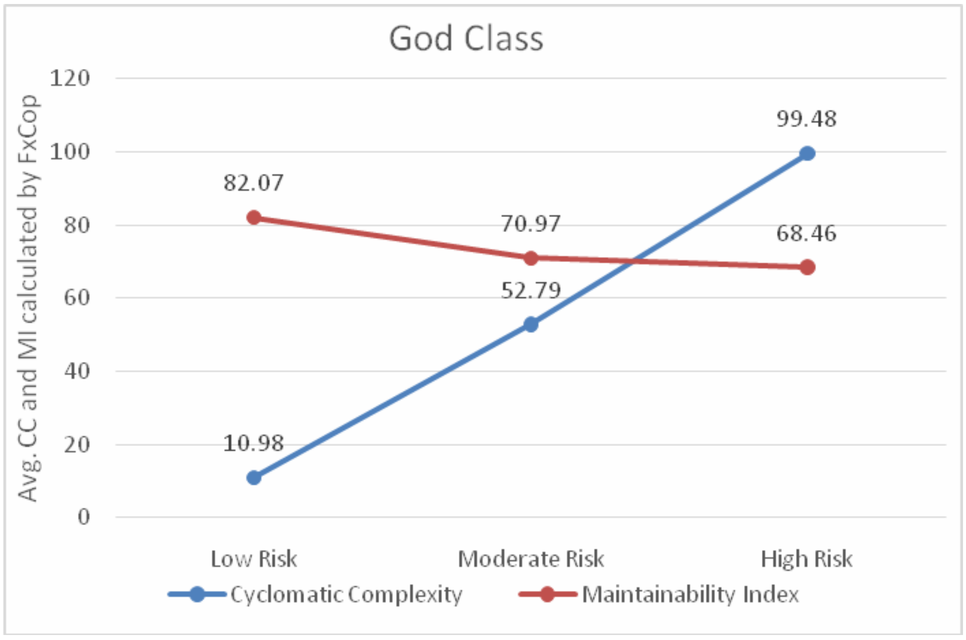
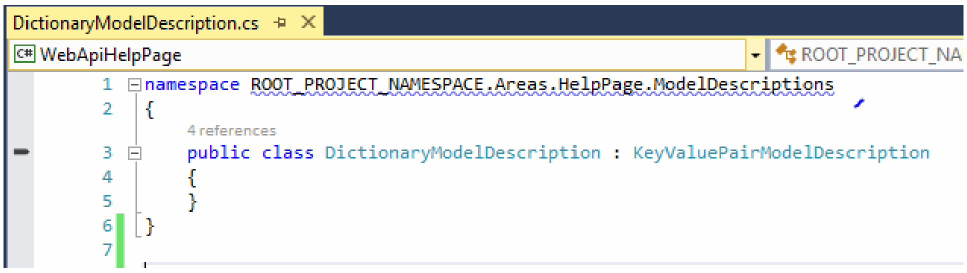


Figure 13 Sample highly risky LC of project 1 (see online version for colours)

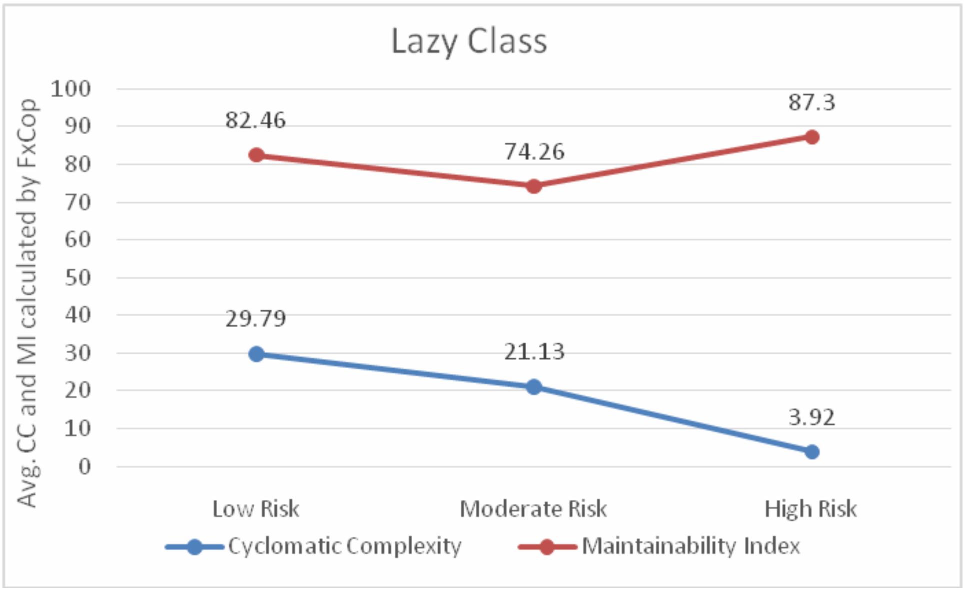


```
namespace ROOT_PROJECT_NAMESPACE.Areas.HelpPage.ModelDescriptions
{
    public class DictionaryModelDescription : KeyValuePairModelDescription
    {
    }
}
```

Table 6 Average CC and average MI calculated by FxCop (combined)

Detected smell		*Low risk*	*Moderate risk*	*High risk*
Long method	CC	1.54	3.47	11.44
	MI	79.37	59.92	48.18
Long parameter list	CC	1.93	2.51	4.21
	MI	75.58	74.99	68.34
Switch statement	CC	1.95	9.45	14.6
	MI	75.53	59.95	55.14
God class	CC	10.98	52.79	99.48
	MI	82.07	70.97	68.46
Lazy class	CC	29.79	21.13	3.92
	MI	82.46	74.26	87.3

Figure 14 LC result comparison (see online version for colours)



To check the relevance of detected code smells' severity with CC and MI, code smell wise average CC and average MI from Table 6 is plotted and given.

1 *LM:* Figure 9 shows the graph of average CC vs. average MI for LMs. Methods with lower risk of LM smell have average MI value 79.37, which decreases to 59.92 for moderate risky code and 48.18 for higher risky code. On the other hand, for lower risky code segments average CC value is 1.54, which increases to 3.47 for moderate risky code segments and 11.44 for higher risky code segments. Thus the result of this dynamic analysis supports the static analysis.

2 *LPL:* methods with high risk of LPL smell have high average CC (75.58) and low average MI (1.93). Moderate risky methods have lower average MI (74.99) and higher average CC (2.51) than low risky ones and highly risky code segments have similar behaviour. Figure 10 shows the graph of average CC vs. average MI for LPL.

3 *SS:* Figure 11 shows the graph of average CC vs. average MI for SS. Results of SS smells is similar to those of LM and LPL smells. Thus average MI decreases and average CC increases with risk severity as shown in the figure.

4 *GC:* Figure 12 shows the graph of average CC vs. average MI for GC. Here also, average MI decreases from 82.07 to 70.97 for moderate risky code segments and from 70.97 to 68.46 for highly risky code segments. Similarly, average CC increases from 10.98 to 52.79 for moderate risky code and increases to 99.48 for highly risky code segments.

5 *LC*: from row 5 in Table 6 it can be observed that for LC, the results are almost reverse of the previous four detected smells. An explanation for that behaviour can be given from the definition of LC which states that a LC is a class with too little

functionality. From test data analysis, it has found that, most of the highly risky LC has one or two lines of code and those are basically interfaces or classes with few lines. For example, Figure 13 shows a highly risky LC from project 1, which has only the class declaration.

That is why those have high average MI and low average CC and average MI decreases for moderate and low risky code segments because those have more lines of code, but this is not always the same. Improvement in detection criteria can resolve this deviation. Figure 14 shows the graph of average CC vs. average MI for LC.

After comparing the results of FxCop and code sniffer, it can be inferred that the proposed tool can provide a good insight of code quality without doing dynamic analysis.

6 Threats to validity

In this section, possible threats to the validity of this study are discussed. *Construct validity* depends on the assumption that the metrices used in this research are actually capture the intended characteristics, e.g., that LOC or the number of properties or NOM, accurately model class size. In this study, multiple metrices are used for each law (Table 2) to reduce this threat.

Since there is no standard C# dataset for code smell detection, the *content validity* is in threat here. To reduce this threat, four popular open source projects from ‘github.com’ have used as test data.

Internal validity focuses on how sure that the action actually caused the outcome. Since the tool uses some code metrics for detecting code smells and their risk severity, which are independent properties of code, internal validity is reserved here.

External validity is also threatened in this study. The tool only works on software written in C#. Therefore, the results do not generalise to software written in other languages.

7 Conclusions

Code smells are the most common anti patterns related to bad programming practices. These lead to deeper problems in maintaining the software. Furthermore, if a developer can check his/her written code how much risky of containing some code smells, he/she can take action to mitigate those issues at the development time. This will ultimately reduce the code re-factoring time and help the future maintenance process.

A tool, code sniffer, is proposed for detecting code smells in static code and analyses the risk factor associated to those for five different types of code smells. The tool is used to identify problems in a C# based case study. Smells such as large class, LC, LM, LPL and SS have been detected. High, moderate and low risk components of source code are presented as results. The comparison of code sniffer with a dynamic analyser, FxCop proves that similar judgement on code quality can be achieved without dynamic analysis.

The list of code smell can be extended for further research. Also, the language support for the tool can be extended so that it supports other popular languages.

References

Abdelmoez, W., Kosba, E. and Iesa, A.F. (2014) 'Risk-based code smells detection tool', in the *International Conference on Computing Technology and Information Management (ICCTIM2014)*, The Society of Digital Information and Wireless Communication, pp.148–159.

Alikacem, E.H. and Sahraoui, H.A. (2006) 'Détection d'anomalies utilisant un langage de règle de qualité', in *LMO*, pp.185–200.

Brown, W.J., Malveau, R.C., McCormick III, H.W. and Mowbray, T.J. (1998) *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed., John Wiley and Sons, USA.

Conorm (2007) 'Maintainability index range and meaning', *MSDN* [online] https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/ (accessed 14 September 2015).

Dhambri, K., Sahraoui, H. and Poulin, P. (2008) 'Visual detection of design anomalies', in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering, CSMR*, IEEE, pp.279–283.

Erni, K. and Lewerentz, C. (1996) 'Applying design-metrics to object-oriented frameworks', in *Software Metrics Symposium, Proceedings of the 3rd International*, IEEE, pp.64–74.

Fard, A.M. and Mesbah, A. (2013) 'JSNOSE: detecting JavaScript code smells' in source code analysis and manipulation (SCAM)', *2013 IEEE 13th International Working Conference on*, pp.116–125.

Fowler, M. (2006) *Code Smell* [online] http://martinfowler.com/bliki/CodeSmell.html (accessed 4 January 2015).

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, USA.

Getting Started C# Syntax Analysis (2017) [online] https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Syntax-Analysis (accessed 20 January 2017).

Glass, R.L. (2001) 'Frequently forgotten fundamental facts about software engineering', *IEEE Software*, Vol. 18, No. 3, pp.110–112.

Gomes, I., Morgado, P., Gomes, T. and Moreira, R. (2009) *An Overview on the Static Code Analysis Approach in Software Development*, Faculdade de Engenharia da Universidade do Porto, Portugal.

Halstead, M.H. (1977) *Elements of Software Science*, Vol. 7, p.127, Elsevier, New York.

Heitlager, I., Kuipers, T. and Visser, J. (2007) 'A practical model for measuring maintainability', in *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology, QUATIC*, IEEE, pp.30–39.

Kothari, S.C., Bishop, L., Sauceda, J. and Daugherty, G. (2004) 'A pattern-based framework for software anomaly detection', *Software Quality Journal*, Vol. 12 No. 2, pp.99–120.

Kresowaty, J. (2008) *FxCop and Code Analysis: Writing your Own Custom Rules* [online] http://www.binarycoder.net/fxcop/index.html (accessed 26 January 2016).

Lippert, M. and Roock, S. (2006) *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, John Wiley and Sons, USA.

Mäntylä, M., Vanhanen, J. and Lassenius, C. (2003) 'A taxonomy and an initial empirical study of bad smells in code', in *Proceedings of International Conference on Software Maintenance, ICSM*, pp.381–384.

Marinescu, R. (2004) 'Detection strategies: metrics-based rules for detecting design flaws', in *Proceedings of 20th IEEE International Conference on Software Maintenance, ICSM*, pp.350–359.

Marinescu, R. and Lanza, M. (2006) *Object-Oriented Metrics in Practice*, Springer, Germany.

Martin, R.C. (2009) *Clean Code: A Handbook of Agile Software Craftsmanship*, Pearson Education, USA.

McCabe, T.J. (1976) 'A complexity measure', *IEEE Transactions on Software Engineering*, Vol. 4, pp.308–320.

McConnell, S. and Johannis, D. (2004) *Code Complete*, Vol. 2, Microsoft Press Redmond.

Moha, N., Gueheneuc, Y.G., Duchien, L. and Le Meur, A.F. (2010) 'DECOR: a method for the specification and detection of code and design smells', *IEEE Transactions on Software Engineering*, Vol. 36 No. 1, pp.20–36.

O'Keeffe, M. and Cinnéide, M.Ó. (2008) 'Search-based refactoring: an empirical study', *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 20 No. 5, pp.345–364.

Parnin, C., Görg, C. and Nnadi, O. (2008) 'A catalogue of lightweight visualizations to support code smell inspection', in *Proceedings of the 4th ACM Symposium on Software Visualization*, ACM, pp.77–86.

Pigoski, T.M. (1996) *Practical Software Maintenance: Best Practices for Managing your Software Investment*, John Wiley and Sons, Inc.

Roperia, N. (2009) *J Smell: A Bad Smell Detection Tool for Java Systems*, California State University, Long Beach.

Software Metrics in Eclipse (2015) [online] http://realsearchgroup.org/SEMaterials/tutorials/metrics (accessed 21 October 2015).

Tahmid, A., Nahar, N. and Sakib, K. (2016) 'Understanding the evolution of code smells by observing code smell clusters', in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*, IEEE, pp.8–11.