Risk-Based Code Smells Detection Tool

W. Abdelmoez, Essam Kosba, Ali falah Iesa
Department of Computer Science
Arab Academy for Science, Technology, and Maritime Transport
ALEXANDRIA, EGYPT P.O. BOX1029
walid.abdelmoez@aast.edu, ekosba@aast.edu, Ali_falah_eassa@yahoo.com

Abstract

In order to maintain quality of software products, we need to perform refactoring regularly. Code smells detection is an important step to be able to successfully conduct software refactoring. In this paper, we propose to use risk concept in assessing code smells detected in the code. As a proof of concept, we developed an automated risk based code smells detection tool. We use the tool to identify problems in a C# case study. Code smells, such as Long Method, Long Parameter list, Message Chain and Empty Catch have been detected in the case study. Furthermore, risk factor level has been qualitatively associated (hi, low, Medium) with each code smell based on the frequency of occurrence and the severity of each code smell.

Keywords: code smell detection, risk factors, risk- based detection tools

1. INTRODUCTION

Code smells are code that is written in bad patterns. They are inherited from bad programming practices [1]. They could be as characteristics or patterns that serve as indicators of degraded code quality, which could prevent comprehensibility and modifiability. During the last 10 years, code smells have become a symbol of software that may cause problems in maintenance, performance [2],[3], and software quality. They can lead to a variety of maintenance problems, and thus makes it difficult to understand or maintain the code [4].

Risk estimation is an important part in the software life cycle management. Performing risk estimation in the early phases of software development can enhance allocation of resources. Moreover, it could provide ways to find defects in software components that requires careful development and testing efforts. In the maintenance phase, you can identify some risk types such as project risk, usability risk, reliability and maintainability risk [5]. These risks could be used to detect problem that would affect the software performance and quality.

In this paper, we propose to use the risk concept to assess detected code smells. In this context, risk factor is estimated by determining the frequency of occurrence and the severity of each code smell. Taking into concern the usefulness of automated detection of code smells, we developed a prototype for an automated risk based code smells detection tool. The tool estimate the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity of detected code smells.

The tool will allow identifying bad code smells such as Long Method, Long Parameter list, Message Chain, and Empty Catch. Also, it will allow the developer to estimate the risk-level of each code smell (high, medium, low) based on counting the number of occurrence of the smell in analyzed files, and the severity of each smell category. To estimate the severity of detected code smells, the analyzer determine the thresholds of metrics used in detecting code smells. Special attention should be given to this step as there are no accepted standard thresholds that are used in identifying code smells. Moreover, code evaluation code quality is an open area for dispute.

The paper organization is as follows: section 1 discusses an introduction to the paper, section 2 briefly discusses background. In section 3, we discuss related work and literature review. In section 4, we represent our proposed tool. In section 5, we represent and discuss the results from the case study. In section 6, we conduct a comparison among code smells detection tools. Finally, we conclude in Section 7 and discuss directions for future research.
2. BACKGROUND

This section provides technical background to software maintenance process; code smells definition and detection in the source code and risk assessment.

2.1. Software Maintenance

The cost of software maintenance accounts for 60% - 80% of the overall software system cost and enhancements (perfective/adaptive) account for 78%-83% of the maintenance effort [6]. The types of software maintenance are corrective maintenance dealing with error corrections, adaptive maintenance concerning system changes as requirements and environment change and perfective maintenance trying to improve the quality of the system.

Software maintainers in most cases do not join the software development cycle from the beginning. They must learn how a program functions before they can change it. They often interact with complex and difficult to comprehend systems. Maintenance process is affected by programmer skills, experience, system documentation and the nature of the system itself.

2.2. Code Smells

Code smells are usually not bugs—they are not technically incorrect and don't currently prevent the program from functioning. Instead, code smells could be considered as software weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. We are concerned with the following code smells: [7]

1. **Empty catch**: when programmer uses the try and catch blocks sometimes they left the catch part empty with no code inside it. Either code can handle the exception, then the catch clause shouldn't been empty, or the code cannot handle the exception, then there should not be a try/catch block at all.

2. **Long parameter list**: if the developer creates a method with parameters, he should know that the longer the parameters list, the more complex it becomes to maintain this method. This code smell is defined as many parameters passed into a method, this is different in OOP, and long parameter list method can be replaced by passing an object instead of the parameters because long parameter list method is harder to read, maintain and change.

3. **Message Chain**: occurs when a series of method calls is needed to perform a task. So, chain message bad smell is an emanation where class A needs information from class D. To get to this information, class A needs to recover objects C from item B (A and B have an immediate reference). When class A gets object C it then asks C to get object D. When class A at last has a reference to class D, A approaches D for the information it needs. The issue here is that A gets to be unnecessarily coupled to classes B, C, and D, when it just needs some bit of information from class D. This creates undesired dependencies which makes such intermediaries not recommended to have.

4. **Long Method**: is a method, a function, or a procedure that has grown too large. The longer the method, the harder it becomes to read, to maintain, to change and to troubleshoot.

2.3. Risk Assessment

To plan the development for a software system, the project manager should assess the risks facing the development effort. There are several risk assessments techniques so, they required human involvements depends on system attributes according to the system models itself. NASA-STD-8719.13a standard characterized risk as a function of the expected recurrence of event of an undesired event and the consequences associated with it. The standard characterizes a few sorts of risks, for example, accessibility risks, acknowledgement risk, execution risk, expense risk, plan risk, and so on. Risk assessment helps project managers to avoid high risk issues, wrong resource allocation and avoiding wrong decisions without proper information. Risk assessments is an exceptionally essential part in the administration development process[8].

3. RELATED WORK

Recent studies have concentrated on developing different techniques for code smells detecting and fixing. These developed techniques vary from putting guides for manual inspection to full automation of detection and correction for code smells. There are three types of code smell defects
detection: rules-based detection-correction, 
detection and correction combination, and visual-
based detection,

First, for rules-based detection-correction category: In [9], the author outlined a list of rules to detect 
design flaws of OO design at method, class and 
subsystem levels. These rules are based on OO 
metrics. In [10], the authors introduced the concept 
of multi- metrics, n-tuples of metrics to express a 
quality criterion. In order to improve this 
quality criterion, they used metrics within an 
evaluation framework. The main difficulty in these 
contributions is that you have to manually define 
threshold values. In [11], the authors tried to avoid 
this problem. They used fuzzy rules, with fuzzy 
labels for metrics, e.g., small, medium, large to 
express defect detection rules. Even though there is 
no need to define crisp thresholds, it is not clear 
how to come up with the membership functions.

In [12], the authors proposed their DÉCOR 
methodology. They used an abstract rule language 
to describe defect symptoms. Then, these 
descriptions are transformed into algorithms to 
detect defects. In [13], the authors extended 
DÉCOR methodology. This methodology takes into 
account uncertainty by using Bayesian belief 
networks to implement the detection rules. It sorts 
the defect candidates, as the detection outputs are 
probabilities that a class is an incidence of a defect 
type.

Rules that can be expressed as assertions, or graph 
transformation are the base for existing approaches 
of automatic refactoring activities. In [14], the 
authors proposed the use of invariants to detect 
parts of program that need refactoring. In [15], the 
author tried to preserve the behavior of the system 
by using pre- and post-condition with invariants.. In 
[16], the author proposed the use of graph 
production rules as refactoring activities.

A good alternative is using search-based techniques. 
In [17], the authors proposed an approach for the 
automatic detection of potential design defects 
using search-based techniques.

Second, for detection and correction combination: 
These approaches do not detect defects explicitly. 
Rather, they attempt to refactor a system by 
detecting elements for change that improve a 
system quality attribute. In [18], the authors 
considered defect detection as an optimization 
problem. They applied sequences of simple 
refactorings and used a combination of 12 metrics 
to measure the improvements achieved. 
Determining the sequence maximizing a function 
that captures the variations of a set of metrics is the 
goal of the optimization[19]. Changes resulting 
from this process do not need to make sense; it only 
Improves the quality in terms of metrics. Thus, the 
relation between the detection and correction is not 
clear, so it’s a difficult task for software 
maintainers.

Third, the visual-based detection solutions take 
advantage of the human ability to integrate complex 
contextual information in the detection process. In 
[20], the authors proposed to detect software 
anomalies by representing potentials defects with 
different colors within a pattern-based framework. 
In. [21], the authors proposed a visualization-based 
approach by automatically detecting some 
symptoms and letting the human analyst to detect 
the rest. The visualization metaphor attempts to 
reduce the complexity when dealing with a large 
amount of data. Finally, visual-based techniques is 
limited to the detection step only.

4. RISK-BASED CODE SMELLS 
DETECTION TOOL

In this section, we discuss the code smells detection 
tool which is based on the risk concept. The 
detection methodology depends on comparing the 
code line by reserved word, in case that the code is 
method declaration so the program will search for 
Long Method and Long parameter List, then the 
program runs to check each line in the specified 
code to find any Empty catch or message chain.

Figure (1) shows the user interface of the tool. The 
following gives brief description of the UI:

- **In the upper green area:** there are 2 
  buttons the first is used to Load File and the 
  other is used to show all smells

- **In the second green area:** the code is loaded

- **In the blue area:** there are: the four 
  filtration buttons

- **In the yellow area:** the Smells is loaded 
  and the code segment is colored with the same 
  color of detected smells
The first red border area contains the threshold while the second area contains the result as graphic view.

Accordingly to code entered to the software, the program will classify it to 4 code smells as follows:

1- Long Method: the program counts the method line, comparing lines to the threshold.
2- Long Parameter: the program counts the parameters count.
3- Message chain: the program counts the message chain, comparing parameter to the threshold.
4- Empty catch: find the empty catch statement, comparing number to the threshold

The System component diagram is shown in Figure (2), it presents the architecture of risk-based code smell detection tool. To use the tool, the user will choose the required file(s) then the software start calling its components as shown below the following sequence:

1- Detect error component is the most complicated component in the software it includes the following tasks
   a. Count files: is responsible of counting uploaded files from the user into the program.
   b. Create smells tree view: is responsible of creating tree view according to the number of files to upload the syntax tree to it.
   c. Create tab counts: create tabs according to the file count to hold the textboxes inside it.
   d. Add textbox: this component is responsible of creating Rich textbox to load the code inside it according to the file uploaded in the system.
2- Detect errors without message: it resets counter
3- C-sharp Parsers Component: is the main Component in the program, this component main task is to convert the c# code into c# syntax tree which uses the imported Assembly “ICSharpCode.NRefactory.dll” by calling CSharpParser class.
   o Make tree node: this component is responsible of the looping on the syntax tree node one by one.
     ▪ Get node title: check the node title to specify if the node identified that there are errors. If the node title has "NodeType", "IsNull", "IsFrozen" or "HasChildren" this title will be ignored, and if the node title has PropertyType it gets the node property type to check if the type is incorrect.
     ▪ _check smells: have two options either to get all smells in the file or get smell by smell.
       o _check smells to detect all code smells
       o _check smells for specific error

---

![Figure (1) Interface of developed software](image-url)
Figure (2) Software architecture of risk-based code smell detection tool

Figure (3) discuss the functionalities of the tool presented in the form of a use case model. from the user point of view:

- If the default threshold value of a code smell is not suitable, the user can change it and perform a new deduction.
- The user can load the C# file that he wants to detect smells on it, then press the Show all smells button which calls the C-Sharp Parser Library to convert the C# file into C# syntax tree.
- Detect all the code smells in the added file according to the values of the threshold values this step will deduct all the smells in the four type code smells, which are (Long methods, long parameter list, empty catch and message chain).
- If the user wants to detect Long Method code smell so he will press Show Long Method button, this will call the C-Sharp Parser to check for the Long method only and display it.
- If the user wants to detect the Long parameter list code smell, so he will press Show Long Parameter button, this will call the C-Sharp Parser to check for the long parameter list only and display it.
- If the user wants to detect Empty catch code smell so he will press Show Empty catch button, this will call the C-Sharp Parser to check for the Empty catch only and display it.
- If the user wants to detect Message Chain code smell so he will press Show Message chain button, this will call the C-Sharp Parser to check for the Message chain only and display it.

Figure (4) shows the result with blue font and highlight code with blue color, also figure (5) has the same action, and figure (6) and figure (7) changed due to the selected button.
**Figure (3)** Use case diagram according to the action form user perspectives

**Figure (4)** Example showing detection smell of Long Method
Figure (5) Example showing detection smell of Parameter list

Figure (6) Example showing detection smell of Message chain

Figure (7) Example showing detection smell of Empty catch
5. CASE STUDY
Case study Class Diagram: The case study program is Diet care c# Program, we will deduct errors in 3 classes using code smell detector, the code samples as Appointment.cs, Jobs.cs and Person.cs. figure (8) shown the case study class diagram.

A. Appointments.cs:
The Code Smell detector get 4 low risk long method smells, 3 low risk long parameter Method and 4 low risk message chains as shown in Figure(9).

B. Jobs.cs:
The Code Smell detector get 1 low risk long method smells, 1 low risk long parameter Method and 1 low risk message chains.

C. Person.cs
The Code Smell detector get 16 low risk long method smells, 14 low risk long parameter Method and 14 low risk message chains as shown in Figure (11).
The methods with line $\leq$ low risk threshold beside long method area are located at the left beside the green legend, the result is 16 low risk long method smells. Method with 7 lines beside the yellow Legend the result is 0 Mid Risk long method and method with lines more than 10 lines will be placed beside the red legend the result is 0 high risk long method. If the user changes the threshold the result will change immediately.
Figure (9) smells detected in Appointment.cs

Figure (10) smells detected in Jobs.cs

Figure (11) Smells detected in Person.cs
6. TOOLS COMPARISON:

In this section, we compare some code smells tools each of them have different features.

6.1. ClockSharp:
ClockSharp is a code checker tool for C# Programming language integrated with visual studio2005, 2008, 2010 and 2012, it checks code using more than 100 programming rules, it can be executed as command line tool.[22]

6.2. FindBugs:
FindBugs is an open source programs works on java byte code looks for bugs in java code using static analysis to identify four potential types of errors scariest, scary, troubling, of concern. [23]

6.3. PMD:
Programming Mistake Detector based on Java source code analyzer that identifies problems in five categories: bugs such as Copied/pasted code, Duplicate code, empty try, empty catch, empty finally, empty switch, dead code, parameters and private methods, string usage, string buffer usage, Wasteful overcomplicated expressions, Suboptimal code, unused local variables, Dead code, unnecessary if statements, for loops and while loops [24].

In Table (1), we discuss the comparison among these different tools taking also in account our developed tool. In this comparison we take into consideration the comparison criteria.

Table (1) shows comparison among developed software and clock sharp

<table>
<thead>
<tr>
<th>comparison criteria</th>
<th>Developed Software</th>
<th>Clock Sharp</th>
<th>Find Bugs</th>
<th>PMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool description</td>
<td>standalone</td>
<td>Plugin Tool</td>
<td>Stand alone</td>
<td>Plugin Tool</td>
</tr>
<tr>
<td>Program rule</td>
<td>No Rule</td>
<td>Predefined Rules setup</td>
<td>Predefined Rules setup</td>
<td>static rule</td>
</tr>
<tr>
<td>Threshold</td>
<td>Threshold for each code smell</td>
<td>No threshold</td>
<td>No threshold</td>
<td>No threshold</td>
</tr>
<tr>
<td>Smells filtration</td>
<td>Can view all error or Filter code smell one by one</td>
<td>View all the errors at the output</td>
<td>View all the errors at the output</td>
<td>View all the errors at the output</td>
</tr>
<tr>
<td>User Interface</td>
<td>User Friendly</td>
<td>Not User friendly</td>
<td>User Friendly</td>
<td>User Friendly</td>
</tr>
<tr>
<td>Can work on project / Files</td>
<td>Can work on single file or multi files</td>
<td>Work only on projects cannot work on single code file</td>
<td>Work only on single code file</td>
<td>Work on projects</td>
</tr>
<tr>
<td>The output result</td>
<td>The output is presented graphic</td>
<td>The output result is too long to read</td>
<td>Can be filtered By classes, By package, By bug type, Summery</td>
<td>PMD errors are not true errors, but rather inefficient code</td>
</tr>
</tbody>
</table>
7. CONCLUSIONS

Code smells are the most common bad patterns related to bad programming practices which lead to deeper problems in maintaining the software. Software products that contain code smells can be difficult to maintain. In this research effort, we propose a tool for detecting code smells which uses the risk concept. As a proof of concept, we developed an automated risk based code smells detection tool. We used the tool to identify problems in a C# case study. Code smells, such as Long Method, Long Parameter list, Message Chain and Empty Catch have been detected in the case study. Furthermore, risk factor level has been qualitatively associated (hi, low, Medium) with each code smell based on the frequency of occurrence and the severity of each code smell. We plan to extend our developed software to detect other code smells and test the tool using larger case studies.

ACKNOWLEDGMENT

This publication was made possible by NPRP grant # [09-1205-2-470] from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors

8. REFERENCES


[17] M. Kessentini, S. Vaucher, and H. Sahraoui, “Deviance from perfection is a better criterion than closeness to evil when identifying risky code”, in Proceedings of the IEEE/ACM international conference on


