Chapter 2

Related Literature

Contents

2.1 Introduction ............................................. 28

2.2 Extraction of Bad Smells with Software Metrics ... 29

2.2.1 Bad smell Extraction with Subjective Evaluation Approach ........................................... 30

2.2.2 Bad Smell Extraction with Automatic Evaluation Approach ............................................... 33

2.2.3 Bad Smell Extraction with Visualisation-Based Approaches 37

2.3 Software Metrics Threshold Design .................. 40

2.4 Relationship of Bad smell with Faulty Classes ..... 45

2.5 Problem Formulation ................................. 48

2.6 Objective of Study ........................................ 53
2. Related Literature

2.1 Introduction

There have been various studies by Hatton (1998) and Erlikh (2000) which suggest that post-release maintenance effort constitutes a major share of the entire software cost. Polo et al. (2003) has also analysed the results of various (Pressman (2005); Lientz & Swanson (1980); Pigoski (1996); Schach (1990); Frazer (1992)) empirical studies for software maintenance costs from the early 1970s till the late 1990s. The trend is for maintenance as a percentage of total software cost to increase (from 35% to 90%). The increasing trend shows that legacy code has been adapted and integrated with the novel technologies and platforms.

However, it has been observed that design quality deteriorates with continuous evolution and maintenance of the software due to insufficiencies in the initial design documents. The initial design documents become inconsistent with the original design as the software is continuously maintained. Deterioration in design quality results in design flaws in the code. Two types of work are determined to be design flaws in the software industry: antipatterns and bad smells. Antipatterns W.J. et al. (1998), result from solutions to recurring implementation and design problems that are poor in comparison to a design pattern. Bad smell (Fowler et al. (2001)) is a code fragment whose structure shows that design principles were not followed during implementation and whose design needs to be restructured. Although antipatterns and bad smells are related concepts, they have some major differences. Antipatterns are global design flaws whereas bad smells are local design flaws. Antipatterns are known to be an outcome of multiple bad smells existing in multiple parts of the code. Remedying a bad smell requires a single refactoring technique, transforming code to eliminate the specific bad smell while preserving external behaviour. Remedying antipatterns, on the other hand,
2.2 Extraction of Bad Smells with Software Metrics

requires several consecutive refactorings or the redesigning of more than one part of the software.

Identifying bad smells in code helps in refactoring the code, and thus facilitating the evolution of the projects. Research and practice have shown that there are no “silver bullets” to solve the design problems; each bad smell has a separate design problem and must be handled in a distinct manner. Furthermore, most research efforts focus solely on detecting design problems rather than solving them with different transformation techniques.

Studies using software metrics (Ciupke (1999); Simon et al. (2001); Gronback (2003); Kataoka et al. (2002); Marinescu (2004)) on decision making for refactoring have been carried out. According to Fowler et al. (2001), metrics for reviewing or refactoring code are not well defined. So, certain external attributes are needed for the code to be refactored for better review, understandability, maintainability, and evolution of the software. Metrics can help in guiding this discussion by providing solid information regarding certain aspects of object-oriented properties. Collection and analysis of metric data is time consuming. Furthermore, it is not foolproof and interpretation of the metric results is difficult (Fowler et al. (2001)). So, using the source code metric together with a code review is the best practice. Code metrics provide qualitative information about how well the code can evolve.

2.2 Extraction of Bad Smells with Software Metrics

Metrics can help by providing help to provide solid information regarding certain aspects of the software design. Concepts such as cohesion, coupling, inheritance, encapsulation and complexity are some properties that software practitioners may
need to measure with metrics. Several studies have been done with the source code metrics for the evaluation of software. Metrics are gathered using the program analysis, which played an important role in the maintenance of the software. Code metrics help to measure the maintenance of the code and its ability to evolve. Code metrics set have been used in a number of studies to produce a polynomial equation that can be further used to quantify the maintenance effort. This type of analysis offers a quantitative and objective way to determine the quality of the software. The automatic tool offers a better quantitative analysis than does subjective human evaluation. A number of tools are available for the research community to quantitatively evaluate the software, including commercial and open source tools and ones for academic purposes. In today’s world, the software development and research communities require tools that help developers locate areas where refactoring can be applied. When a potential area for refactoring is identified, the refactoring effect in the source code can be analysed effectively.

2.2.1 Bad smell Extraction with Subjective Evaluation Approach

A number of studies (Simon et al. (2001); Balazinska et al. (2000); Ducasse et al. (1999); Kataoka et al. (2001); Tourwe & Mens (2003); Kessentini et al. (2010)) have focused on automatically detecting poor structures in software or using historical data Maruyama & Shima (1999) to detect area where refactoring can be performed. Many of the studies mentioned are actually more focused on improving the software’s ability to evolve than on evaluating it. A limited number of studies of subjective evolvability evaluations have been found as apposed to automatic program evolution. To detect refactoring area in the software Fowler et al. (2001) proposes a list of twenty two bad code smells. The list is a more
concrete indication of refactoring area than some vague idea of programming aesthetic. Code smells are seen as a compromise between the source code metrics and unguided subjective evaluation of code. Following are some of the subjective evaluation criteria for detecting poor structure in the code:

Shneiderman (1980) reviews the software on the seven point Likert scale. He had asked one member of each of three teams of five professionals to review a piece of software subjectively, and the other four to review it on the basis of a questionnaire. The thirteen questions varied from blank line usage to best algorithm selection. The results of the study show that, on each team, three of the four persons agreed with the subjective evaluation. Shneiderman’s explanation for this difference is that the fourth analysts misunderstood the question or scale. In this, the researcher does not consider the possibility that the developers’ opinions on structural difference, design difference and style explain the result.

Kafura & Reddy (1987) studied the relationship between software maintenance and seven complexity metrics. An expert in the field of maintenance made subjective evaluation of the work that had been done on the three different versions of the medium-sized software system as it evolved over a period of three years. But the authors had not provided the expert with any criteria for the evaluation of the maintenance and no data of evaluation was included in the paper. Therefore, it is not possible to replicate the study. Kafura & Reddy (1987) concluded that software maintenance was in conformance with source code complexity metrics.

Shepperd (1990) performed an experiment of validation with software information flow metrics on software maintenance. An experiment data was collected from 89 modules of aircraft software with a total line of around 30,000 line of code. In their experiment the maintainer evaluated each module on a four-point ordinal scale of the perceived difficulty of hypothetical maintenance tasks. For 73% of task, the perceived difficulties was within one point in all the modules.
2. Related Literature

However, no detail were given about the collection of data and it would be difficult to repeat or replicate the study.

Oman in his various studies (Oman & Hagemeister (1994); Coleman et al. (1994a); Coleman et al. (1995b); Welker et al. (1997)) worked on various issues of software maintenance and constructed a maintainability index for industrial software.

Oman and his colleagues have developed the regression model for the maintainability index with the software metrics. The regression model of metrics was then correlated with the subjective evaluation of the software maintainer. The regression model was developed from the data set of eight software systems ranging from 1,000 LOC to 10,000 LOC. For the validation of the model the data set of six projects with length of code ranging from 1,000 to 8,000 was collected. The validation study of the subjective evaluation study found discrepancies in evaluation. For example, one evaluator is more lenient than another in evaluating the same system. This discrepancy was not seen in the other study by same researchers, where it seems that the same person both evaluated the calculation of metrics and performed the validation. After all these experiments work has concluded the result with the opinion that automatic tool will give more accurate picture than subjective evaluation.

Kataoka et al. (2002) studied the usefulness and quality of software after applying the refactoring. They also, compared human evaluation and software metrics approaches, concluding that with subjective evaluation the effectiveness of software refactoring correlated with an improvement in the coupling metrics. They had done the study with five refactoring cases and one developer evaluating the effectiveness of the software refactoring.

Marinescu (2004) has proposed a detection strategy for formulating metrics-based rules to check the deviation from good design principles and heuristics. In this case human intervention is required because the detection process is uncertain.
2.2 Extraction of Bad Smells with Software Metrics

Marinescu (2004) did not give any justification for the selection of metrics, threshold and combination of metrics defined in the detection strategies. It is essential to know whether the metrics chosen encapsulate the design problem or not. Also, the approach proposed did not address the uncertainty issues, quality analysts’ interpretations and context of the programs. Furthermore, in this work, the case studies were done on small projects and not on industrial projects. Of all the above studies, only Katako’s study is with the object-oriented language. The drawback in this study is that they had studied only five refactorings and had only one analyst analysing refactoring effectiveness.

2.2.2 Bad Smell Extraction with Automatic Evaluation Approach

Subjective evaluation approaches avoid the problem of uncertainty but cannot be used with large amounts of data. Automatic approaches, in contrast, allow large amounts of data to be evaluated for design problems. Further, an automatic approach provides the quality analysts with an unsorted set of candidate classes with no indication of which one should be inspected first. Following are some of the automatic techniques for the metrics-based evaluation of software for design problems or bad smells.

Ciupke (1999) proposes and develops a tool in his study for the identification of design problem in the object-oriented legacy system. Design problems are specified as queries on a design model and tool locate an occurrence of these specified design problems using a model derived from source code. Problem are detected using the tool developed by Ciupke to support this task with query-based analysis. This work uses case studies from both academia and industry. Experimental results in the study showed good results for deciding where to start when reorganising the code. The tool only runs on certain platforms with the requisite
2. Related Literature

software and it also requires the highly skilled users. This study did not touch complex design problems, such as simple conditions with fixed threshold values i.e. “Inheritance hierarchy should not exceed the values of six descendants, avoid multiple inheritance, unused inheritance etc”. However, legacy systems suffer from design problems which go beyond common problems.

Munro (2005) studies the metrics-based approach to detect the code smells from the programs. The template in this work has designed on the basis of metrics to detect the two kinds of bad smells (Lazy Class and Temporary Field) in the code. On the basis of this, template-base design heuristics study, it has proposed the prototype tool for evaluation of interpretation rule design in designed heuristics. The empirical study was done on the small projects and not on large industrial projects. With this empirical study, the work justified the choice of the metrics and bad smell. The proposed approach did not include the context of program and quality analyst. To validate the proposed tool, the study considered only two bad smells.

Simon et al. (2001) use metrics to identify the structure or code where refactoring can be applied. This study has developed a visualisation approach to identify the areas where refactoring can be applied. In the study the metrics-based visualisation approach supports the developer in judging the source code. Small software systems were analysed better than large ones with this study.

Tahvildari & Kontogiannis (2003) and Tahvildari & Kontogiannis (2004) propose a comprehensive framework for an object-oriented metrics suite (complexity, coupling and cohesion metrics) to detect design flaws at class level. The framework focuses on a class, called a key class, whose design is deteriorated because of high coupling causing lack of cohesion, a common flaw in object-oriented legacy systems. The approach also suggests a possible and appropriate meta-pattern transformation for the key class; this helps a maintenance team to focus on a
2.2 Extraction of Bad Smells with Software Metrics

particular part of a large system. The other design heuristic, they examined is called one class-one concept. This concept violation takes two forms: the implementation of more than one concept in a single class, which results into low cohesion, and the implementation of no single concept in one class, typically a class that is tightly coupled with other classes. The study was done on only one Java-based system (Java Expert System Shell).

Trifu & Marinescu (2005) develop strategies to detect instances of a structural anomaly, which authors call a ‘symptom’ of bad design. A detection strategy developed to identify a bad smell (the authors called them a ‘problem’) is compose of various metrics, each selected with its appropriate threshold. The selected metrics are combined with AND/OR operators into a single rule that expresses a design heuristic or bad design. The threshold values used for the defining metric rules were defined based on statistical data collected from more than sixty Java and fifty C++ projects. The identified design problems can be eliminated using corresponding restructuring strategies (the authors called them a ‘treatment’) which are informally described in textual form as the actions required for the elimination procedure. The case study involved only one open source system, AgroUML. No valid criteria was detailed, of how metrics threshold was derived by the authors, instead of the line that threshold was derived from sixty Java and fifty C++ projects. Detail information about these Java and C++ projects was also not given (i.e. size of projects, open source or commercial projects, availability of these projects etc.).

Salehie et al. (2006) proposed a metric-based heuristic framework to detect design flaws in object-oriented systems. The framework has three main components: a) A generic object-oriented design knowledge base that stores a set of design heuristics, metrics and flaws along with the relationships among them. b) A hot spot indicator that uses primitive classifier to discover the entities most likely to be
2. Related Literature

called namely hot spots. c) A design flaw detector that uses composite classifier to locate possible design flaws in the predetermined hot spots. The study uses the coupling, cohesion and complexity metrics along with the God class and Shotgun Surgery bad smell to validate the proposed methodology of framework. The author had collected the metrics and bad smell data from a case study of the Java-base application Jboss Application Server. The framework proposed in the study begins the process by defining the design flaws in terms of well-defined principles, features and metrics threshold values instead of any formula for it. This systematic approach locates and isolates the design flaws to improve the maintenance activities. The proposed framework would improve two maintenance activities perfective and preventive maintenance, but was not validated for corrective maintenance. The study identified only two bad smells and considered only three properties of programming: coupling, cohesion and complexity. Reddy (2008) uses the Design Change Propagation Probability (DCPP) matrices to detect two bad smell or antipatterns (Shotgun Survey and Divergent Change). A DCPP matrix is a NxN matrix where N represents the artifact or class. In the matrix, the value at column A, row B represents the probability that a design change in artifact A will require changes in B to preserve the overall functionality. The study does not use the threshold values for the detection of the antipatterns, but compare candidates of design defects (Source Code or Class, called as artifacts in the study) under certain specified conditions. The study checked the specified conditions (listed in the study of Reddy (2008)) iteratively and correct the design defects with refactoring techniques representing the bad smell (e.g. Move Method for Shotgun Surgery).

Tsantalis (2010) tries to identify three bad smells (Feature Envy, State checking and Long Method) with the case study of Java-based software. Due to the conceptual differences among the three design problems or bad smells, authors identi-
2.2 Extraction of Bad Smells with Software Metrics

Bad smells are identified using three separate methods. The study takes a semi-automatic approach to identifying the refactoring opportunity that can resolve the bad smell in the code. The study follows all the activities designed by Tourwe & Mens (2003) for refactoring the process. The results of the identifications of refactoring points were evaluated by an expert independently to determine whether identified points for refactoring are sound, and useful and that applying refactoring transformations on them will give positive results. The study not only pre-evaluates the effects of refactoring but also provides a ranking mechanism to help the maintainers to prioritise their efforts on the most crucial part(s) of program. The developed methodology was implemented as a tool in the Eclipse plug-in JDeodorant. The study has designed the Entity Placement Metrics, on the basis of cohesion and coupling metrics, to rank the possible refactoring solutions. The lower the Entity Placement Metrics value, the more the effective refactoring solution. The results were evaluated, on the basis of various cohesion and coupling metrics after the refactoring transformations were applied.

2.2.3 Bad Smell Extraction with Visualisation-Based Approaches

Emden (2002) developed a bad smell browser visualisation tool for java source code, jCOSMO. jCOSMO is used for code smell browsing and visualisation of bad smell within the source code. The graphical representation helps the maintainer to identify where the bad smell is and thus what area the maintainer needs to concentrate on to remove or reduce its effects. The tool, used for the automatic inspection and visualisation of bad smell, was tested with a case study on the CHARTOON system, which has 147 classes. The approach followed for the development of the tool has two steps: first, design consideration for the software inspection tool is based on code smell detection. Second, code smells are broken
2. Related Literature

or split into number of parts, so that it’s become easy for the tool to detect the smelly classes. The tool jCOSMO is an extended version of the tool Rigi. Tourwe & Mens (2003) along with his student Bravo (2003) defined an automatic approach with logic meta-programming to identify the refactoring area in a source code. The Tourwe & Mens (2003) study proposes a framework by using the information extract from approach designed above (logic meta-programing) and propose the adequate refactoring technique at required area. The proposed approach is automatic but not metrics based; it is a logical programming approach. The work was done in the SOUL programming environment, which is a logic meta-programming environment and a variant of PROLOG. The work has identified the two type of bad smells in the source code. After identification of the bad smell a list of refactoring was proposed. The developer then applied the suggested refactoring one by one to get the optimum results. This approach is suitable for the small-to-medium-sized projects but not for large projects. A potential disadvantage of this approach for large projects is the long list of refactoring techniques that would be proposed in some cases.

Dhambri et al. (2008) present a semi-automatic visualisation approach to detect six design anomalies in the source code. The authors designed a visual framework called VERSO and used the features it provided to extracts design anomalies from the source code. The study uses the low level architecture to represent the 3D graphical approach on 2D plane. This work extracts the structural information using the reverse engineering tool PADL (Albin-Amiot & Yann-Gael (2001)) and the quantitative information about metrics using POM (Gueheneuc et al. (2004)). In this approach the detection strategy extracts four types of information: quantitative, relational, architectural and semantic. Quantitative information is measured with various object-oriented metrics of a class. Each type of metrics is associated with three measures: height, colour and twist. As per the authors’ quantitative
2.2 Extraction of Bad Smells with Software Metrics

Analysis technique size of metrics is measured with the height of the 3D box. The colour of a class indicates its position, e.g. red colour indicates an extremely high metrics value and blue colour indicates towards the lower tail. Architectural information refers to the low-level information i.e. the structural information of the program in terms of modules and packages. This provides such information as which package belongs to which class. Relational information was measured with reverse engineering relationship approaches including associations (‘in’, ‘out’, and ‘in/out’), aggregation, generalisation, implementation, invocation, etc. To represent relationship the study uses structure filters, such as queries. Semantic information about the code of a class was measured by browsing a code of a class with a mouse click over the visualised component of the specific class. This will help to improve the accuracy in measuring Blob anomaly. The authors have done a case study with a concrete example of one design anomaly (Blob or God Class) on two software systems PMD 1.8 (286 classes) and Xerces 1.0.1 (296 classes). Dhambri et al. (2008) propose the visual technique for the identification of smelly classes as a compromise between manual and automatic technique, but this technique is unable to determine when a smelly class is actually smelly and provides a futilely long list of candidates.

The major disadvantage of the metrics-based approach, as reported by Tsantalis (2010) is the definition of threshold values, which is a matter of subjective study. As per Tsantalis (2010), threshold value is based on the arbitrary choice and on the statistical analysis of historical data. One of the aims of this study is to remove this limitation by defining the threshold values of some significant software metrics with the help of historical data. The following section presents the literature related to the design and selection of software metrics threshold.
2. Related Literature

2.3 Software Metrics Threshold Design

Users required a software that conforms to some certain external quality factors. To uphold these quality standards, quality attributes must be fulfilled for the execution of the software projects. Attributes including understandability, maintainability and others can be measured with the help of software metrics. The following are some of the studies that take different approaches to derive the threshold values of software metrics.

In early days the metrics threshold value was calculated on the basis of experience. Some of the studies that uses this on the basis of experience include McCabe (1976); Nejmeh (1988); Coleman et al. (1995a). McCabe proposes a threshold value of 10 for the McCabe Cyclomatic Complexity measure (McCabe (1976)). For example, for a metrics NPATH define by Nejmeh (1988) proposed 200 as a threshold value on the basis of experience and past data. Coleman et al. (1995a) has proposed a threshold value for a metrics Maintainability Index. That study has classified the data into three categories on the basis of threshold values of 65 and 85. Methods with a maintainability index greater than 85 are highly maintainable; between 85 and 65, moderately maintainable; and less than 65, difficult to maintain. None of the above threshold values are scientifically supported, making them difficult to defend objective and leading to dispute about the values. The problem of objectivity has lead to the solution of a theoretical foundation based on various statistical analyses.

Erni & Lewerentz (1996) define a threshold on the basis of mean and standard deviation of the data. This work has defines two threshold values for each metrics, i.e. a minimum and a maximum value. The minimum value was calculated by subtracting the standard deviation from the mean \((\mu - \sigma)\) and the maximum by adding the standard deviation to the mean \((\mu + \sigma)\) value. The major require-
2.3 Software Metrics Threshold Design

ment for deriving threshold values with this technique was that metrics data be normally distributed at the outset. French (1999) proposes a threshold value of software metrics that eliminate the restriction of the normal distribution found in Erni & Lewerentz (1996). In this work, the threshold value is determined using the mean and standard deviation values as well as, study had used the Chebyshev’s inequality theorem. In the study, the threshold $T$ is calculated as $T = \mu + k \times \sigma$ where $k$ is the number of standard deviation. The paper does not mention the design flaws in the code. It is important to know the design flaws in order to plan the solution in term of refactoring or correction strategies. Further, this study is not valid for measures which has high variation or high range.

Rosenberg (1997) extracts the software metrics threshold against the error probability using the histogram analysis technique. This study has extracted the threshold for CBO, RFC and WMC software measures. This study provides clear evidence to relate the error probability with the software metrics.

Benlarbi & El-emam (2000) and El-emam et al. (2002) were the first researchers to try to estimate the metrics threshold values of object-oriented metrics. These works present a theoretical and empirical approach to evaluate the Goldilock conjecture (i.e. defective content and size have U-shaped curve relation, meaning that components that are too small or too large are more prone to error). The authors of the studies made clear that optimal size theory had significant implications for object-oriented design. They used the logistic regression model suggested by Ulm (1991) for designing threshold values. The studies demonstrate that the fact that smaller components are more faulty than larger components is merely a consequences of mathematical artifacts. To test the threshold theory authors had done an experiments on three object-oriented applications (two C++ systems and one Java system). El-emam et al. (2002) checked the threshold of four size measures (Statement, Line Of Code, Number of Methods and Number of
2. Related Literature

Attributes) where as Benlarbi & El-emam (2000) tried to determine the threshold for the Chidamber and Kemerer metrics (NOC, DIT, WMC, RFC and CBO) set. However, Benlarbi & El-emam (2000) study found that the threshold values that were calculated from the logistic regression were not valid for all the three systems (i.e., there was no statistical difference between the two models the no threshold model and the threshold model). Later on Ulm model was revised by Bender (1999) and applied his model in the field of epidemiology. Also, El-emam et al. (2002) tested the threshold model only for the size measures and not on any other object-oriented properties.

Mihancea (2003) and Mihancea & Marinescu (2005) designed the threshold values for a set of software metrics for the two design flaws God Class and Data Class. The metrics used for quantifying the God Smell are Access to Foreign Data (ATFD), Weighted Method per Class (WMC) and Tightly Class Cohesion (TCC) whereas Data Class was quantify with the Weight of a Class, Number of Public Attributes and Number of Accessor Methods Metrics. In this study, the authors define the approach “tuning machine” to automatically find the threshold values of software metrics. It tunes the detection strategies to get a better threshold value, which should improve accuracy in prediction of smelly classes. The strategy was tuned by configuring the genetic algorithm. This study first built a database of design flaws, then tune the threshold value so as to correctly identify a maximum flaws. The major drawback of this approach is that, it requires a large database for optimum tuning of the strategies, and collecting these samples is not easy.

Shatnawi (2010) and Sanchez-Gonzlez et al. (2011) have applied Bender (1999) model based on logistic regression in the field of software engineering. Shatnawi (2010) uses this model to find the threshold for software metrics. This study has taken up the database of Eclipse three versions for identifying the threshold.
The threshold was identified successfully for the CBO, WMC and RFC. Threshold values were found on the basis of logistic regression. The threshold value of software metrics was found against the fault analysis (known as corrective maintenance). From the logistic regression the metrics significantly associated with the fault were shortlisted. After the equation of logistic regression for faulty classes was defined, threshold values with Value at Acceptable Risk Level (VARL) were identified at suggested probability \( p_o = 0.050, 0.060, 0.065, 0.075, \) and 0.10 and is calculated with coefficient of logistic regression. The threshold designed with this study was not validated in third party software.

Shatnawi & Li (2010) design the threshold values of software metrics using the ROC curve analysis technique. With this methodology, metrics value were shortlisted on the basis of the location of a point having the highest sensitivity and specificity value. This critical point was a point at the maximum distance from the 0.5 diagonal. The threshold values obtained from this analysis were measured for their performance with Area Under Curve (AUC). For the analysis, three versions of Eclipse were used and with this methodology, threshold values were calculated for the CBO, RFC, WMC, CTM and NOO metrics. The work calculated the threshold of the software metrics on the basis of faulty classes and the severity level of faults. The threshold values derived in this analysis were more accurate at identifying high and medium severity levels of fault (AUC > 0.70) than at simply determining whether the class is faulty or not (AUC < 0.70). In this work, the authors have found the threshold in relation to the corrective maintenance (faulty classes) and validated it with the same.

Herbold et al. (2011) presented a machine learning and data mining approach for the calculation of software metrics thresholds. The contribution of this work includes: the design of a machine learning-based model for defining the threshold; the optimisation of an already existing metrics set with its threshold; the
2. Related Literature

effective replacement of the existing classification method and reducing its complexity; and finally, the calculation of threshold of metrics set in the environment where no earlier metrics existed. To validate these objectives of the work, four case studies with different large size open-source software (written in C, C++, C# and Java) were taken up. The metrics analysed in the case studies were control flow structuredness, coupling, size, method complexity, inheritance and staticness. The machine learning methodology used in this study was data driven and independent of the selection of the metrics set. With the above methodology, the study was able to determine the optimal metrics set for replicating the quality attributes. This work also highlights that: how the proposed methodology be applied to improve the efficiency of existing metrics set along with the threshold, reduction of complexity of used classifier and with the proposed methodology new metrics set can be introduced. Out of four case studies, two shows that the proposed methodology of machine learning can improve the efficiency of a metrics set. In the other two case studies, a complex classification can be reproduced successfully with the simple threshold. The only limitation of the study is that it checked only whether or not each class is faulty and made not attempt to check the threshold values for further categories of faults. A category check could give more precise information about the threshold values.

Some of the above studies (Rosenberg (1997); Benlarbi & El-emam (2000); Shatnawi (2010); Shatnawi & Li (2010); Herbold et al. (2011)) found the threshold for the Chidamber and Kemerer metrics (NOC, DIT, WMC, RFC and CBO) set. Out of these Benlarbi & El-emam (2000) show that there is no difference between faulty and not faulty models. Herbold et al. (2011) recommend the Rosenberg (1997) threshold values for some metrics. None of the above studies has identified the threshold for the design flaws and check its validity to determine the faulty classes or faulty classes with its severity levels. From among all of the
above studies, two methodologies (ROC Curves and Benders Methodology) were selected for identifying the threshold of software metrics. Sánchez-González et al. (2012) had supported and compared these two approaches for the identification of threshold. And, the authors found that ROC curve obtained more accurate results.

2.4 Relationship of Bad smell with Faulty Classes

The above literature review has shown that there is a relationship between bad smell and software metrics, both with the automatic and subjective approaches. Apart from metrics-based approaches, some other approaches, to the identification or prediction of bad smell in a code are also listed, but it has been seen that none of the above approaches designed for preventive maintenance is evaluated or validated with another maintenance type, e.g. validation with corrective maintenance by predicting the error-prone area. This must be evaluated as per the definition of preventive maintenance, which is as follows (Tsantalis (2010)):

“The long term effect of corrective, adaptive and perfective maintenance is an increase in the complexity of the system. This increase in the complexity deteriorates the system structure, this will continue until the work should be done to maintain it or to reduce it. The required work to be done is known as preventive maintenance.”

As per the above definition, to remove the side effects (increase in complexity, structure deterioration, etc.) of corrective, adaptive and perfective maintenance, preventive maintenance of the system is required at regular intervals. Further, preventive maintenance will help to make the system more maintainable and understandable (Vliet (2007)). Some empirical evidence is required to validate the above theoretical approach. So, any code area that, according to object-oriented
2. Related Literature

metrics, is likely to need preventive maintenance, should also be validated for its need for other types of maintenance. This type of validation will increase the users’ confidence in well-designed and fault-free software. Various researchers have studied this in two separate studies: First study is to check the relationship of metrics with faulty classes and second study is to check the impact of code smell on the faulty/error prone classes. As per the paragraph title, few studies below (belongs to second part) have shown the relationship between design error or bad smell and the error-prone classes.

Li & Shatnawi (2007) investigated the relationship between bad smell and the probability of fault occurrence. This study evaluated three versions of Eclipse (2.0, 2.1 and 3.0). Out of twenty two bad smell authors showed that three bad smells (God Class, Shotgun Surgery, Long Method) are able to predict faults in the classes. The study is based on the logistic regression model. They start with univariate binary logistic regression to check for significant (p-value <.05) relationships between the smelly and faulty classes. After finding the statistically significant (having p-value <.05) smell they use multivariate binary regression to check whether or not the selected bad smell is able to predict whether a class is faulty or not. After this they check the shortlisted bad smell to predict for three different categories (High, Medium and Low) of fault occurrence. From their study they found that the bad smell Shotgun Surgery was associated with all severity levels of errors in all releases, whereas God Class and Long Method are partially associated with some releases. Their study does not validate the metrics model for the bad smell identification. Moreover their study is based on three different versions of one software i.e. Eclipse. The study is of limited use because of twenty two bad smell that have been defined, they selected just six bad smells and of those six only three smells (Shotgun Surgery, God Class and Long Methods) were able to identify faulty classes. Out of these three smells,
2.4 Relationship of Bad smell with Faulty Classes

Shotgun Surgery is best at predicting the faulty classes at all three severity levels for all three versions. The other two smells were partially associated with faulty classes in some releases. Li & Shatnawi (2007) concluded by pointing out the need for broader study in this area to find the association between bad smell and faulty classes.

Khomh et al. (2012) focus on finding the impact of antipatterns on changes and fault-proneness of the classes. The authors developed a six hypothesis for their study. Two hypotheses relates to check the impact of antipatterns in predicting fault-proneness and impact of each type of antipatterns in predicting fault-proneness. To check these hypothesis, the authors analysed 54 releases of four different types of software system: Rhino, AgroUml, Eclipse and Mylyn. The study detected thirteen antipatterns from the source code including ten bad smells defined by Fowler et al. (2001). The study concluded with the result that classes participating in the antipatterns are prone to errors but not with all types antipatterns i.e. four antipatterns types out of thirteen are able to predict the fault proneness. The authors did not develop either a standard object-oriented metrics model or found metrics threshold values to identify antipatterns classes. Study only check the impact of antipatterns (on the basis of rule cards designed by Moha (2008)) and their types to predict the faulty classes.

As in the above literature, the relationship of metrics to bad smell and role of bad smell to detect the faulty classes was seen separately. Further, a number of empirical studies (Basili et al. (1996); Shatnawi & Li (2008); Gyimothy et al. (2005); Briand et al. (1998)) have shown a relationship between the metrics and error prone classes. None of the above approaches have checked the metrics model or its threshold, designed for locating the restructuring/refactoring area (solution of preventive maintenance) was further used or validate to predict the faulty classes (corrective maintenance). In this study, main objectives are to
2. Related Literature

evaluate or validate the preventive maintenance metrics model and the threshold values for corrective maintenance. If maintenance activities are designed on the basis of avoiding bad smells, then as per the above discussion confidence is primarily based on common sense that the incidence of faulty classes will be reduced.

2.5 Problem Formulation

As the developer community has suggested, it is more generic to have a metrics model that can identify the specific bad smell at required area and which helps to select a design solution, such as refactoring. Further, this metrics model for bad smell identification, should also identify or predict the probability of design error. These ideas indicate that the effective use of bad smell is hindered by lack of empirical evidence for error probability and error-prone areas. Bad smells that can be identified using a metrics model or metrics threshold values provide information that helps the developer to manage or analyse the code on daily basis. Though previous empirical studies show that there is a relationship between software metrics and smelly classes, but no studies from above literature review have validated the metrics model of bad smell identification for the further prediction of error-prone classes. This validation is not obvious and requires empirical analysis, which should show that an empirical metrics models and threshold values designed for preventive maintenance, can help in corrective maintenance also. On the basis of this thought the problem has been formulated into the following two parts, similar to the way the metrics literature was surveyed for predicting the refactoring area:

1. To design the efficient empirical metrics model to identify the smelly areas or classes which further can predict the faulty classes and its various sever-
2.5 Problem Formulation

2.5.1 Problem Formulation

The prediction accuracy of the metrics model for smelly and categorised smelly classes should be measured on the basis of prediction accuracy for faulty classes and its severity level.

2. To identify the threshold values of the software metrics against the smelly classes. Further, a specific threshold value is to be picked from the shortlisted metrics on the basis of its high accuracy in predicting the faulty classes and the severity level of the fault.

The work aims at determining the relationship between software metrics and bad smell in the classes. To solve the above problem following research issues (RI) along with their corresponding null hypotheses are considered.

RI1. Is there any relationship between bad smell and software metrics? And with what probability of accuracy can a metrics model of smelly classes can predict the smelly classes? An experiment can be performed to check whether class metrics can predict smelly class probability. To address the above research issue with the proposed methodology null hypotheses for the experiment can be considered are: a) software metrics can not predict any relationship with bad smell classes, b) software metrics model can not show any significant relationship with bad smell classes and c) an empirical model of software metrics can not predict the smelly classes accurately.

RI2. Is there any relationship between software metrics and smelly classes in each of the bad smell categories? And with what probability of accuracy can a metrics model for categorised smelly classes can predict the respective categorised smelly classes? This can be determined by developing an empirical model of software metrics to predict smelly classes in various bad smell categories. This research issue can be addressed by considering the following two null hypotheses: a) A software metrics cannot predict bad smell
2. Related Literature

categories, b) software metrics empirical model cannot predict bad smell
categories and c) Software metrics empirical model for bad smell categories
cannot predict the respective bad smell categories accurately.

RI3. With what probability of accuracy can a metrics model for smelly classes
predict the error-prone classes? The the metrics model for bad smell
can be investigated to predict whether the class is faulty or not, by testing
the null hypothesis: a software metrics model for bad smell cannot predict
the faulty classes.

RI4. With what probability of accuracy can a metrics model for categorised
smelly classes can predict the error-prone classes? This questioned can be
answered by investigating the accuracy of categorised metrics model for bad
smell to predict the faulty classes. In this study the null hypothesis can be
considered as: a software metrics model for various categories of bad smell
identification cannot predict the faulty classes.

RI5. With what probability of accuracy can a metrics model for categorised
smelly classes which predict the error-prone classes at RI4 more accurately
can predict the error-prone classes at various severity levels? An exper-
iment can be performed on the shortlisted metrics models which predict
the faulty classes more accurately. This can be done by investigating the
ability of the shortlisted metrics model for bad smell categories to predict
the faulty classes at various severity level. Metrics model can be shortlisted
on the basis of the best performance to predict the faulty classes. The null
hypothesis tested for this can be: the shortlisted software metrics model
from RI4 for various categories of bad smell identification cannot predict
the faulty classes at various severity levels.

RI6 With what probability of accuracy can a metrics model for smelly classes of
2.5 Problem Formulation

one system can predict the smelly and faulty classes for other open source software? This can be investigated by applying the design metrics model of smelly classes of one open source software to the other open source software for predicting the smelly and faulty classes. The null hypothesis can be: software metrics model designed for one software system cannot be applied to other software system to detect the smelly and error-prone classes for second open source software.

RI7. Can metric-prediction models based on categorised smelly classes of one system can predict the categorised smelly for other open source software? This can be investigated using the MMLR metrics model of various bad smell categories on the second open source software. The second open source software will be different from the original software through in terms of which bad smell binary and categorical model can be developed. The Null hypothesis can be: a software metrics model designed for one software system cannot be applied to second software system to detect the smelly categories.

RI8. Can a smelly categorised metrics model predict the faulty classes and their severity levels in another open source software? This can be answered by investigating the smelly categorised models of one software using the other open source software for error-prone classes and its error severity level. The Null hypothesis can be: a smelly categorised metrics model designed for one software system cannot be applied to other software system to detect the faulty classes and their severity levels.

RI9. What are the threshold values of different software metrics to identify the refactoring area or smelly classes and improve the design? One metrics value can be identified to decide whether the class is smelly or not. The
null hypothesis can be: there are no practical and effective threshold values for the object-oriented metrics that can predict smelly classes.

RI10. Can the threshold value of different software metrics found for predicting smelly classes being able to identify the faulty classes? A threshold metrics value found on the basis of smelly classes can be validated by using it to try to predict whether the class is faulty or not. The null hypothesis can be: practical significant threshold metrics values which predict the smelly classes will not predict the faulty classes or separate the classes into two modules i.e. faulty and no fault.

RI11. Can the threshold value of different software metrics derived for predicting smelly classes be used to identify the faulty classes at various severity levels? A threshold metrics value found on the basis of smelly classes can be validated by using it to try to predict the faulty classes for different categories. The null hypothesis can be: a practical significant threshold metrics value which predicts the smelly classes will not predict the faulty classes at three level of severity i.e. low, medium and high.

Hypotheses 1, 3, 4, 6, 9 and 10 described above are one tail because software metrics model or threshold values will investigate whether metrics can predict whether the class is smelly or error-prone. Hypothesis 2, 5, 7, 8 and 11 are two tailed which investigate whether the metrics model or threshold values is related to different kind of smelly categories and error severity levels. Research issue from 1 to 8 along with the corresponding null hypothesis are tried to be resolved in chapter five. Research issue from 9 to 11 along its null hypotheses are addressed in chapter six.
2.6 Objective of Study

On the basis of the above problem formulation and research issues, the following were set up as objectives of the present work:

1. To develop a metrics model to identify or predict the smelly classes.

2. To develop a metrics model to predict the probability of smelly classes in various bad smell categories.

3. Validation of metrics model for smelly classes to predict the faulty classes.

4. Validation of metrics model of various bad smell categories to predict the faulty classes.

5. To find the effective threshold values of metrics to predict the smelly classes.

6. To determine the practically significant threshold values that predict the smelly classes and can also differentiate the fault and no fault classes.

7. For the practically significant threshold value identified in previous point to also be able to predict the faulty classes under three (High, Medium and Low) severity level of faults.

These problems objectives have been dealt with in this work and resolved upto certain extent. All these issues and objectives are elaborately discussed in the following chapters.
2. Related Literature