ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

A LEARNING-BASED METHOD FOR DETECTING DEFECTIVE CLASSES IN OBJECT-ORIENTED SYSTEMS

M.Sc. THESIS

Çağıl BİRAY

Department of Computer Engineering

Computer Engineering Programme

MAY 2015

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

A LEARNING-BASED METHOD FOR DETECTING DEFECTIVE CLASSES IN OBJECT-ORIENTED SYSTEMS

M.Sc. THESIS

Çağıl BİRAY
(504111509)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Assoc. Prof. Feza BUZLUCA

MAY 2015

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

NESNEYE DAYALI YAZILIMLARDA HATALI SINIFLARIN ÖĞRENME TEMELLİ YÖNTEMLE BELİRLENMESİ

YÜKSEK LİSANS TEZİ

Çağıl BİRAY
(504111509)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Doç. Dr. Feza BUZLUCA

MAYIS 2015

Çağıl Biray, a **M.Sc.** student of ITU **Institute of Science and Technology of Computer Engineering** student ID 504111509, successfully defended the **thesis** entitled "**A LEARNING BASED METHOD FOR DETECTING DEFECTIVE CLASSES IN OBJECT-ORIENTED SYSTEMS**", which she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Feza BUZLUCA** İstanbul Technical University	.....
Jury Members :	**Assoc. Prof. Şule ÖĞÜDÜCÜ** İstanbul Technical University	.....
	Assoc. Prof. Alper ŞEN Boğaziçi University	.....

Date of Submission : 04 May 2015
Date of Defense : 29 May 2015

To my family,

FOREWORD

First and foremost I would like to express my sincere appreciation to my thesis advisor Assoc. Prof. Feza BUZLUCA for his suggestions and invaluable comments that he provided throughout this research and my graduate education. His appropriate guidance, constructive criticism and advices helped me to reach my goal.

I would also like to express my profound gratitude to my family due to their precious support and motivation that made this thesis all possible. I also want to thank to my friend Volkan ÖZTÜRK for his support and suggestions during my study.

Last but not least, I would like to thank Ericsson Turkey R&D Center for supporting me to develop myself and being sponsor for my M.Sc. study.

May 2015

Çağıl BİRAY
(Computer Engineer)

TABLE OF CONTENTS

Page

ABBREVIATIONS

EF : Error Frequency
ChC : Change Count
ErrC : Error Count
CR : Change Request
SCC : Software Customization Center
CVS : Concurrent Versions System
CK : Chidamber and Kemerer Metrics Suite
Ckjm : Chidamber and Kemerer Java Metrics
QMOOD : Quality Model for Object-Oriented Design
WMC : Weighted Methods per Class
AMW : Average Method Weight
LOC : Lines of Code
NOM : Number of Methods
NOA : Number of Attributes
NAS : Number of Added Services
WOC : Weight of a Class
LCOM : Lack of Cohesion in Methods
ATFD : Access to Foreign Data
CBO : Coupling Between Object Classes
Ca : Afferent Coupling
RFC : Response for a Class
FDP : Foreign Data Providers
FANOUT : Number of Called Classes
CC : Changing Classes
CM : Changing Methods
NPM : Number of Public Methods
NOPA : Number of Public Attributes
NOAM : Number of Accessor Methods
NOC : Number of Children
DIT : Depth of Inheritance
HIT : Height of Inheritance Tree
BUR : The Base class Usage Ratio
WEKA : Waikato Environment for Knowledge Analysis

LIST OF TABLES

Page

LIST OF FIGURES

Page

A LEARNING-BASED METHOD FOR DETECTING DEFECTIVE CLASSES IN OBJECT-ORIENTED SYSTEMS

SUMMARY

In today's competitive environment, increasing customer demands have led to changes in the traditional software development methods. In order to clearly determine and fulfill customer requirements, continuous customer interaction with the team is important. Following requests and feedbacks of customers necessitate modifications in software classes. Poorly designed classes are difficult to analyze, modify and test, so that maintenance costs for such classes are very high.

Code or design problems in software classes reduce understandability, flexibility and reusability of the system. Performing maintenance activities on defective components such as adding new features, adapting to the changes, finding bugs, and correcting errors, is hard and consumes a lot of time. Unless the design defects are corrected by a refactoring process these error-prone classes will most likely generate new errors after later modifications. Therefore, these classes will have a high error frequency (EF), which is defined as the ratio between the number of errors and modifications.

Predicting defective classes before releasing the software is an important issue for the software quality assurance. Early estimate of error-prone classes has two important benefits, firstly it helps testers to focus on faulty modules of software, thus it saves significant proportion of testing time; secondly developers can refactor classes to correct their design defects.

Software classes that include structural design defects mostly include one or more of the following properties. They are complex, highly coupled to other classes, their internal cohesion is low or they have an inappropriate position in the inheritance hierarchy. These properties can be revealed using software design and code metrics of the classes. However, it is difficult to work with metrics to create certain rules for detecting defects because of their various types, distributions and different minimum/maximum values. Also, different metrics should be used together to create a model for quality assessment; but it is difficult to determine the roles, weights and thresholds of metrics in creating such a model.

The aim of this work is to detect poorly designed classes, in order to have testers focus on them and/or have the developers refactor them. Poorly designed classes typically become error-prone when modified. In this thesis, a learning-based decision tree model for detecting error-prone classes with structural design defects is proposed. In learning-based systems the accuracy of the model strongly depends on the training set. The main novelty in the proposed approach is that the EFs and change counts (ChC) of classes to construct a proper data set is considered for the training of the model. The training set is built that includes design metrics of classes

by analyzing numerous releases of real-world software products and considering EFs of classes to mark them as error-prone or non-error-prone.

To train the model and evaluate its performance, two long-standing projects are studied, namely Project A and Project B. Several releases of the projects are examined and modifications in classes triggered by changes in customer use case scenarios, new feature implementations and/or bug fixes inherited from previous releases are identified. After this examination, classes with high ChC and error rate are identified and classified as “defective”. Using the tags (defective/healthy) of the classes and collecting their design metrics, the data set is constructed to use in training and testing of the proposed model. Empirical experiment results demonstrate that the proposed approach succeeds in finding error-prone classes that need refactoring. The proposed model succeeded in finding frequently changing defective classes with relatively high EFs. The model correctly predicted 80% of the most defective classes with the highest EFs of Project A and 83% of Project B. Exposing these risky classes automatically also decreases the test time and maintenance cost.

NESNE DAYALI YAZILIMLARDA HATALI SINIFLARIN ÖĞRENME TEMELLİ YÖNTEMLE BELİRLENMESİ

ÖZET

Günümüzün rekabetçi ortamında artan müşteri ihtiyaçları geleneksel yazılım geliştirme yöntemlerinin değişmesi gerekliliğini ortaya çıkarmıştır. Müşteri ihtiyaçlarını en iyi şekilde anlamak ve yerine getirmek için yazılım ekibi ile müşteri arasında devamlı bir karşılıklı etkileşim olması gerekmektedir. Müşteri ihtiyaçları projenin ilk safhasında iyi belirlenmeli ve yazılım altyapısı bu ihtiyaçlara yönelik tasarlanmalıdır. Projenin ilerleyen aşamalarında, ihtiyaca yönelik müşteri tarafından gelen yeni istekler veya geri bildirimler yazılımdaki sınıflarda değişiklik yapılmasına sebep olabilmektedir. Mevcut yazılımın değişen isteklere uyum sağlayacak esneklikte tasarlanması, yazılımın ilerleyen sürümlerinde bakım maliyetlerinin düşürülmesinde önemli bir role sahiptir.

Yazılımdan beklentiler arttıkça, yazılımlar yapısal olarak karmaşıklaşmakta ve bunun sonucunda yazılım geliştirme ve bakım maliyetleri artmaktadır. Yazılım dünyasında son zamanlarda üzerinde durulan en önemli konulardan biri yazılımda kalitenin sağlanmasıdır. Yazılımda kalite kavramı, yazılım projelerinin maliyetlerini önemli ölçüde iyileştiren unsurların başında gelmektedir. Yazılımın esnekliği, bakımının kolay olması, anlaşılabilirliği, gerçeklenebilirliği, kolay sınanabilir olması ve güvenilirliği gibi kriterler kaliteli bir yazılımdan beklenen özelliklerdir. Ancak, projelerdeki zaman kısıtı, müşteri isteklerinin projenin en başında iyi belirlenmemesi, müşteri taleplerinin sıkça değişmesi, yazılım ekibindeki iletişim eksiklikleri, yazılım geliştiren ekibin nesneye dayalı tasarımın gerekliliklerini bilmemesi ve uygulamaması sonucunda yazılımlardaki tasarım kalitesi düşmektedir.

Yazılım projelerinin tasarım kalitesini projenin erken safhalarında değerlendirmek, ilerleyen aşamalarda projenin sağlıklı bir şekilde devam ettirilebilmesini sağlar. Yazılımın mevcut kalitesinin değerlendirilmesinin yanısıra ileriki fazlarda sorun çıkarabilecek bileşenlerin önceden tespit edilmesi ile yazılımdaki kalitenin devamlılığını sağlanır. Kalitenin ölçülebilmesi ve değerlendirilebilmesi için bir kalite modeline ve ölçme metoduna gereksinim vardır. Elde edilen ölçümlerden yazılımın kalitesine ilişkin bir anlam çıkarılabilmesi için metrik-kalite ilişkisi incelenerek sonucun yorumlanması gerekmektedir.

Yazılımı ölçmek ve kalitesini değerlendirebilmek için yazılım ölçüm metrikleri kullanılır. Yazılım metrikleri, yazılımların belirli özelliklerine göre kalitelerini belirleyebilmek için sayısal değerler ile ölçülmesidir. Yazılım metrikleri, yazılım çalıştırılmadan yazılımın kaynak kodundan statik bir şekilde elde edilebileceği gibi, yazılımın çalışma anında dinamik bir şekilde de toplanabilir. Bu bağlamda yazılım metrikleri ile yapılan ölçümlerle elde edilen sayısal veriler kullanılarak yazılımın hataya açık, kusurlu bileşenlerini önceden kestirimi mümkündür. Yazılımda çıkacak olası hataların erken tespit edilmesinin iki önemli avantajı bulunmaktadır; *(i)* yazılımının kusurlu bileşenleri önceden belirlenip yazılım geliştiriciler önceden

bilgilendirildiği takdirde, ilgili sınıfların tasarımlarında iyileştirme ve düzeltmeler yapılabilir, *(ii)* yazılımı test edenler, yazılımın kritik kısımlarının sınanmasını önceliklendirebilir ve bu sayede yazılım sınama maliyetleri ve süresi önemli ölçüde azaltılabilir. Bu şekilde proje geliştirmek ve bakımının sağlanması için harcanan işgücü ve zamandan kazanç sağlanır.

Yazılım sınıflarındaki tasarımsal problemler, yazılımın anlaşılabilirliğini, esnekliğini ve tekrar kullanılabilirliğini azaltır. Tasarım kusuru olan yazılım bileşenlerinin analiz edilmesi, değiştirilmesi ve sınanması zordur. Bu sınıfların bakım maliyeti yüksektir ve bakımı zaman alır. Yapısal tasarım kusurları kodun derleme veya çalışma zamanında hata vermezler. Bu tarz sınıflar üzerinde değişiklik yapılmadığı müddetçe yazılım yaşam döngüsü boyunca kendilerini gizleyebilirler ve hata üretmeyebilirler. Yazılımın işlevselliğini değiştirmeden kodda iyileştirme yapılmak suretiyle sınıfların yapısal bozuklukları düzeltilmedikçe, tasarım kusuru olan sınıflar yazılıma yapılacak her değişiklikte yeni hatalar çıkarmaya eğilimli hale gelecektir. Yapılacak çoğu değişiklik bu sınıflarda ve bu sınıfın bağımlılığının yüksek olduğu sınıflarda hata çıkaracak ve yazılımın fonksiyonelliğini düşürecektir. Bu durumda, sınıfın hata sıklığı olarak da adlandırılabilecek, sınıfa yapılan değişiklik başına sınıfta çıkan hata sayısı artar. Hataya eğilimli bu sınıfları yazılımın erken safhalarında tespit etmek hem test, hem de geliştirme maliyetlerini azaltır.

Yapısal tasarım kusurları bulunan yazılım sınıfları çoğunlukla; çok karmaşık, yazılımın diğer sınıflarına olan bağımlılığı yüksek, kendi içerisinde uyumu düşük ve kalıtım hiyerarşisindeki uygun olmayan yerde olabilmektedir. Bu özellikler yazılımın iç özellikleri olarak adlandırılmaktadır. Bu iç özelliklere sahip yapısal bozukluklar, yazılımın tasarım ve kod metrikleri kullanılarak belirlenebilmektedir. Ancak, yazılım metriklerinin değişik özelliklerde ve dağılımlarda olması, aldıkları minimum ve maksimum değerlerin farklılaşması sebebiyle, yazılım kusurlarını belirlemek için yazılım metriklerini kullanarak kesin kurallar oluşturmak zordur. Bununla birlikte, yazılımda kalitenin tanımlanabilmesi için farklı metrikler bir arada kullanılarak modeller oluşturulabilir; fakat metrik ağırlıklarına, eşik değerlerine ve rollerine bağlı bir modelleme disiplini oluşturmak kolay değildir. Bu sebeple sınıflardaki kusur tahmini için öğrenme tabanlı yöntemler yaygın bir şekilde kullanılmaktadır.

Bu çalışmada, yapısal tasarım kusuru olan hataya eğilimli sınıfları tespit etmek için karar ağacı modelleri oluşturularak öğrenme tabanlı bir yöntem önerilmektedir. Öğrenme tabanlı yöntemlerde, örnek veri ya da geçmiş bilgiler kullanılarak, gelecekteki bir durumun kestirimini yapmak mümkündür. Belirli parametrelere bağlı olarak bir model tanımlanır ve bu model gelecekteki bir veri için öngörü yapmak için kullanılır. En iyi parametre değerleri bulunduğunda yöntemin kestirim başarısı artmaktadır.

Karar ağaçları modellemesi eldeki verinin sınıflandırılması için kullanılabilecek en etkili yöntemlerden birisidir. Karar ağacı, kök düğümden, iç düğümlerden, dallardan ve yaprak düğümlerden oluşmaktadır. Her bir iç düğümde ilgili kurala göre bir karar verilir ve kararın sonucuna göre dallardan biri seçilir. Karar denetimi kökte başlar ve yaprak düğümlere gelene kadar özyinelemeli olarak devam eder. Yaprak düğümler karar algoritmasının sonlandığı yerdir ve aynı yapraktaki örnekler aynı sınıfa mensupturlar. Karar ağaçları eğitim kümesinde yer alan niteliklerden yalnızca ihtiyaç duyduklarını kullanır ve boyut indirgeme işlevini kendisi gerçekleştirir. Karar ağacının seçtiği nitelikler, problemin tahmini için en belirleyici özelliklerdir.

Önerilen yöntemde kullanılan tasarım metrikleri ile nesneye dayalı yazılımların tasarım kalitesi nitelikleri ölçümlenerek, kusurlu sınıfların kestirimi yapılmıştır. Öğrenme tabanlı yöntemlerde modelin doğruluğu, kullanılan eğitim kümesi ile doğrudan ilişkilidir. Modelin eğitimi için sınıfların hata sıklığı ve değişim sayıları göz önüne alınarak yenilikçi bir yaklaşımla veri kümesi oluşturulmuştur. Yazılımdaki sınıfların ileriki sürümlerde hata çıkarıp çıkarmayacağının kestirimini yapabilmek için her bir sınıfın belirli bir kurala göre etiketlenmesi gerekmektedir.

Modeli eğitmek için kullanılacak eğitim kümesi oluşturulurken, telekom sektöründe kullanımda olan iki olgun yazılım projesinin kaynak kodları ardışıl sürümler boyunca incelenmiştir. Proje A yazılımı sektörde altı yıldır, Proje B ise sektörde dört yıldır kullanılmaktadır. Her bir sürümün hata raporları sürüm notları arasında yer almaktadır. Her projenin sürüm raporunda ilgili sürümde hangi hataların çözümlerinin yapıldığı, müşteri değişiklik isteklerinin neler olduğu ve o sürümde gelen yeni özelliklerle ilgili detaylar yer alır. Çalışma kapsamında kullanılan Proje A ve Proje B yazılımlarının hata raporları yazılımcı ekiple birlikte değerlendirilmiştir. Yapılan gözlemler sonucunda bazı önemli çıkarımlar yapılmıştır: *(i)* Yapısal kusuru olan sınıfların hemen hemen hepsi testler sırasında hata çıkarmaya meyillidir; buna ek olarak sağlıklı sınıfların bazıları da hata raporlarında yer alabilmektedir, *(ii)* Kusurlu sınıflar değiştirilmediği sürece hata çıkarmayabilirler; hatalar değişikliklerden sonra ortaya çıkmaktadır, *(iii)* Sağlıklı sınıflar çok sık değişime uğramazlar; ancak bu sınıflarda değişiklik olduğunda nadiren hata gözlenebilmektedir. Yazılım sınıflarında olan değişiklikler; müşteri kullanım senaryolarındaki değişikliklerle, yeni versiyonla gelen yeni özelliklerin gerçeklenmesi veya önceki sürümlerden kalıtılan hatalara gelen düzeltmelerden etkilenerek tetiklenebilmektedir.

Yazılım sınıflarında olan tüm değişiklikler gerek yazılım takımının desteği ile gerekse kod incelemeleri ile değerlendirilerek, yazılımlarda bulunan her bir sınıfın hata sıklığı hesaplanmıştır. Eğitim kümesindeki hata sıklıklarına ve değişim sayılarına göre sınıfları "kusurlu/sağlıklı" olarak etiketlemek için birçok eşik değeri ile denemeler yapılmıştır. Bu denemeler sonucunda sık değişime uğrayan sınıflar ve sık hata çıkaran sınıflar belirlenmiş ve değerleri belirlenen eşik değerlerinin üzerinde kalan sınıflar "kusurlu" olarak etiketlenirken, tersi özellikteki sınıflar "kusursuz" olarak etiketlenmiştir. Ancak, yapılan deneyler esnasında yazılımdaki bazı sınıfların hiç değişmediği ya da çok az değişime uğradığı gözlenmiştir. Bu tarz sınıflar yapısal olarak kusurlu olsalar dahi, yazılım yaşam döngüsü boyunca değişime uğramadıkları için, bu sınıfların doğrudan "sağlıklı" olarak sınıflandırılması gerçekçi bir yaklaşım değildir. Bu sınıfları etiketlemek için eğitim kümesinde "kusurlu" olarak belirlenmiş sınıfların belirleyici özelliği olan sınıf karmaşıklıkları göz önüne alınmıştır. Bu durumda, hiç değişim geçirmeyen ya da az değişen sınıflardan, sınıf karmaşıklığı eğitim kümesindeki "kusurlu" sınıfların sınıf karmaşıklığının belli oranda yukarısında olan sınıflar "kusurlu" olarak etiketlenip, eğitim kümesine dahil edilmiştir.

Her iki yazılım projesindeki sınıfların yazılım metrikleri, literatürde yer alan metrik elde etme araçları yardımıyla ile elde edilmiştir. Eğitim kümesinde her bir sınıf için toplanan metrikler ve sınıfın "kusurlu/kusursuz" etiketleri yer almaktadır.

Elde edilen verilerle oluşturulan eğitim kümesi kullanılarak öğrenme tabanlı bir model oluşturulmuş ve bu model sistemin önceden görmediği bir versiyonundaki sınıfların hata kestirimini yapmak üzere kullanılmıştır. Öğrenme algoritmasının

kusurlu olarak tahmin ettiği sınıflar belli sayıdaki sürümler boyunca gözlemlenmiş ve yöntemin kusurlu sınıfları tahmin başarısı elde edilmiştir.

Sonuçlar, düşük kaliteli yazılım sınıflarının sıklıkla değiştiğini ve hataya eğilimli olduklarını göstermektedir. Proje A' dan elde edilen eğitim kümesi oluşturarak eğitilen sistem, aynı projenin ilerleyen safhalarında bir sürümdeki sık hata üreten kusurlu sınıfların %80' ini başarılı bir şekilde belirlemiştir. Aynı eğitim kümesi hiç eğitilmediği Proje B yazılımının herhangi bir sürümündeki sınıflarından kusurlu olanlarının %83' ünü doğru tahmin etmiştir.

Geliştirilen öğrenme tabanlı yöntemin belirlediği sınıfların öncelikli olarak sınanması ve tasarımsal olarak kodlarında iyileştirilme yapılması gerekliliği ile ilgili yazılım proje ekibi bilgilendirilmiştir. Bu çalışma kapsamında önerilen öğrenme tabanlı yöntemin sonuçları, telekomünikasyon sektöründeki yazılımlarda sık hata üreten tasarım kusuru olan sınıfları önemli oranda tespit etmektedir. Yazılım ekibinin bile farkında olmadığı bazı sınıfların kusurları tespit edilmiş ve bu sayede projenin ilerleyen sürümlerinde kaliteli yazılım geliştirilerek, yazılım geliştirme süresi ve bakım faaliyetlerinden kazanç sağlanmasına katkıda bulunulmuştur.

1. INTRODUCTION

In today's competitive environment, increasing customer requirements have led to changes in traditional software development methods. In order to gain a competitive advantage, today's software development mainly focused on fast delivery with high quality software products. Continuous customer interaction with development team is important for clearly elaborating and fulfilling customer needs. The lack of understanding of customer requirements yields to deviations from the desired behavior of software product and consequently new software defects are introduced.

Following requests and feedbacks of customers necessitate modifications in software classes. Well-designed software classes easily adapt to changing customer demands, whereas poorly designed software classes are difficult to analyze, modify and test. A high-level design quality of a software system is ensured by applying object-oriented design techniques from the beginning of the project. Software classes that are developed ignoring object-oriented design principles are prone to be defective. Software classes having structurally design defects are generally complex, not cohesive and highly coupled to other classes. They also have inappropriate position in the inheritance hierarchy. If the design quality of these classes is not improved, further modifications cause new design defects and maintenance costs of these classes become higher. Early detection of defect-prone classes helps to reduce the software development costs and saves project time. Detecting and solving software problems after software project delivery costs 100 times more than the one predicted earlier in design and requirements phase [1].

Software quality is the degree of the software product that it complies with the specified conditions and meets the requirements [2]. Ease of integration, ease of installation, ease of use, stability, maintainability and reliability are some of the non-functional quality requirements that are expected from a well-designed software product [3]. Software quality measurement is challenging, as software design quality is more intuitive and based on experiences. Some attributes of software modules reflect information about the design defects, such as classes having high method

complexity are difficult to understand, modify and maintain. It is possible to evaluate the object-oriented design quality of software classes to some extent by using software design metrics. Software design metrics are used to quantify the features of software with numerical values for software quality assessment [4]. The difficulty behind quality measurement is to determine how to use metric combinations and interpret the results. It is difficult to create certain rules for detecting defects, because of various metric distributions, types and minimum/maximum values. In addition, various metric types can be used together to create quality models for quality assessment; but it is difficult to determine weights, roles and thresholds of metrics. Metric values only provide quantifiable expression of software classes thus; detection strategies, rules or learning-based approaches should be used in order to assess the quality of software product.

The evaluation process of the design quality of a software system must start in early stages of the development and continue until the end of development. Quality measurement methods should indicate the status of the software as well as predetermine the defective software artifacts that might lead to future defects. In this way, the certain level of software design quality is ensured at every stage of the project. As the importance of pre-estimation of defective software modules, it is one of the most studied topics of software quality. This section gives an overview of software design quality. The reasons, types, impacts and advantages of early prediction of software defects are described in the following subsections.

1.1 The Reasons for Software Design Defects

In a constantly changing environment, customer demands also vary accordingly. Trying to adapt software development process to these changes, software products become more defect-prone. Understanding the reasons behind the introduction of the defects in software helps both early indication of defects and preventing the future defects. Recent studies on defect prediction have revealed that the following substances are the main factors of software defects [5, 6]:

- Lack of object-oriented design experience: The usage of object-oriented design principles in on-going projects reduces the overall development effort, the complexity of software product and development costs [7]. Software products developed by considering object-oriented design principles are more

flexible to changes and reusable on other systems. System functionality is not affected by software changes, i.e. new feature implementations, customer change requests and bug corrections. High-quality software product is an inevitable result of well-designed object oriented software programming. The lack of knowledge of object-oriented design principles during the software product development may yield poor design of software in which all changes made to software affect another part of software and software becomes vulnerable to introduction of new design defects. Unless the design defects are corrected by a refactoring process, these structurally defective classes will most likely generate new errors after later modifications.

- Vague definition of customer requirements: At the beginning of the software development process, expectations from the software according to customer requirements are determined. Inadequately identified requirements may be the cause of introduction of new defects in software. In the first phase of the project, properly understanding customer demands allows for making high-quality software design. The ambiguity of customer requirements would lead to incorrect coding of developers. In this situation, customer expectations begin to deviate from actual results.

- Unrealistic code or schedule estimation for development: In today's continuously changing environment, increasing customer demands have led to changes in traditional software development methods. In order to gain competitive advantage, unrealistic project deadlines are planned. In this case, there is not enough time left for developers to design, develop and complete unit tests of software. As a natural consequence of unrealistic project time plans, the overall quality of software product reduces and introduction of software defects accordingly increases.

- Start code implementation before completing the design: Unrealistic time plans for software projects lead to ignorance of essential steps in software development process. Software development process starts with identification of requirements followed by requirements analysis. After software requirement analysis is completed, software design phase starts accordingly [8]. In this phase, software is designed to have high software quality attributes, such as reusability, reliability, maintainability of software

attributes are ensured. If code implementation starts before the fulfillment of the software design, software becomes inflexible to code changes according to new customer requirements. Poor software design yields software defects to increase exponentially.

- Communication breakdown: Major software projects are developed by a team composed of many developers. Unless a common coding standard is determined at the initial phase of software development, each developer may use their own coding style that leads to software defects. Miscommunication between team members may cause erroneous coding and ripple effect of these errors may result in introduction of structural software defects. Before implementation phase, there should be a consensus on object-oriented design principles between development team and each developer should develop considering these rules.

1.2 The Types of Software Design Defects

Long-term software products are modified in relation to the changes in requirements, new feature implementations and error corrections throughout the software lifecycle. In addition, software developers should refactor the low-quality parts of software in order to ensure the maintainability. Most of the poor-designed software parts reflect negatively on the metric values, which can be corrected with refactoring methods. However, some structural defects may not be determined by the metric analysis and remain undetected for software development lifecycle. Bad smell [9] definition is first introduced by Martin Fowler, which is considered as low-quality parts of software design. Bad smell code parts reveal the design defects of software product that need to be corrected by refactoring steps. Michele Lanza and Radu Marinescu [10] brought additional perspective to code smells by considering the harmonies of software design. They called design defects as structural disharmonies and grouped in three categories named identity, collaboration and classification disharmonies. Figure 1.1 shows the classification of code smells in detail.

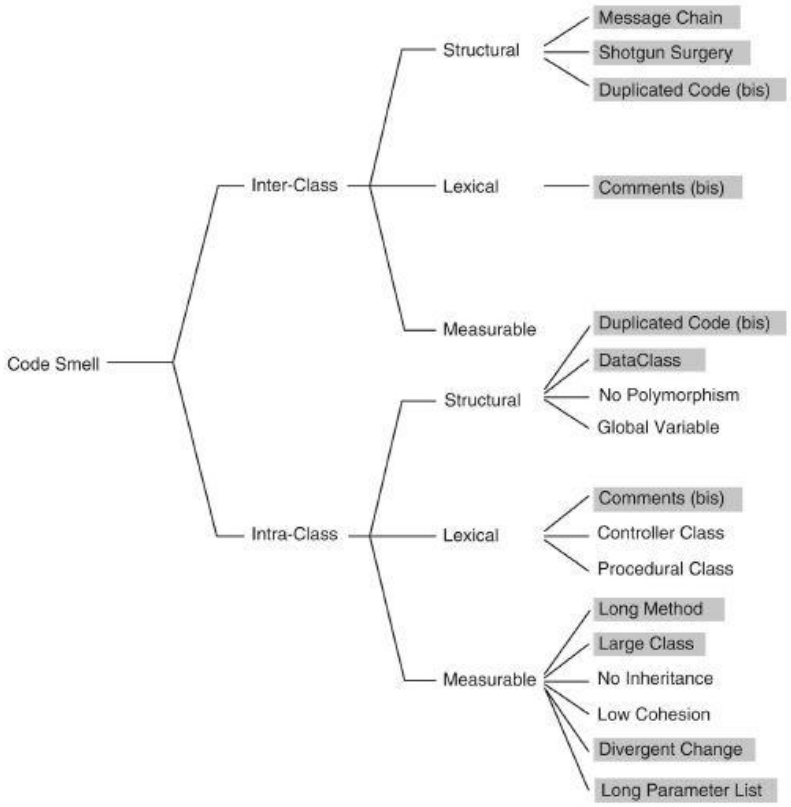

Figure 1.1 : Classification of code smells [9].

Some indicators of low-quality parts of software are listed below:

- Duplicated code fragments: The same piece of code is seen more than one place in software. In this case, the code fragment should be extracted as a method and be invoked from another part of software.
- Long method: Longer methods reduce the understandability of code. Shorter methods are preferred in object-oriented programming principles. In order to overcome the understandability problem, the longer method names should be given. Longer method names are easy to understand and decrease the need for additional comments in code. The defects caused by long methods can be detected by using design metrics.
- Large class: Classes having too many responsibilities include many instance variables, which may cause code duplications in class. Lanza and Marinescu address large class design defect as God Class disharmonies. Functionality of all system is given to many few classes, these classes become unmanageable and unmaintainable in further modifications. Such classes are generally

having high complexity, low cohesion between the methods of a class and using external data belongs to other classes [10], therefore they reveal themselves in metric values.

- Long parameter list: The usage and understandability of long list of parameters that are used in method definitions are difficult and continuous changes in requirements may trigger modifications on parameter lists. This design defect may be detected with metrics related with method cohesions.
- Data class: Such classes hold the data that are used by other classes and do not have any transactions on their own data. These classes are the indicators of design defects as they allow other classes to manipulate their data. Class complexity is low and exposes data with getter/setter methods to external classes. High value of public methods and attributes may reveal these design defects.
- Shotgun surgery: Making any modifications on one class may trigger changes on many other classes. This design defect occurs when responsibility of one class is distributed to many other classes. Coupling metrics may reveal the shotgun surgery design defect. Figure 1.2 shows the illustration of shotgun surgery design defect.

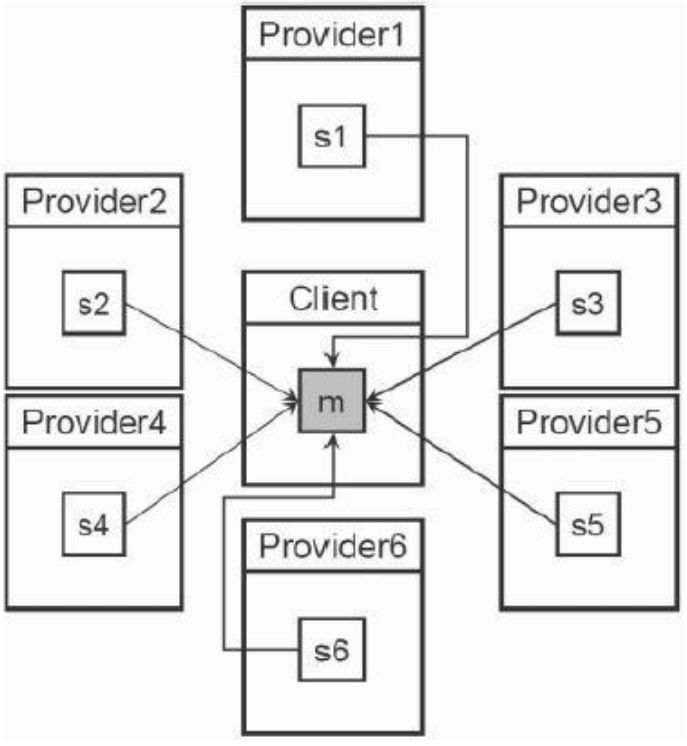

Figure 1.2 : Shotgun surgery design defect [10].

- Divergent change: This design defect occurs when many modifications made on a single class, unrelated methods should be changed accordingly. It is difficult to determine this design defect via metric values. Such classes have methods for many unrelated functionalities.
- Refused bequest: If the inherited interfaces are not used by subclasses, this design defect indicates an error in the hierarchy. Subclass functional complexity and size should be at least an average. Refused bequest design defect can be detected with measuring the base class usage ratio and overriding methods for subclass.
- Tradition breaker: The derived class should not break the tradition by providing unrelated new services are not included in the interfaces of its base class. Subclass should carry on the tradition by extending the services of its base class. Subclass functional complexity and the number of added services, methods may be signs of this design defect.

1.3 The Impacts of Software Design Defects

Design defects are considered as deviations from the design quality and they deflect the expectations from the software product. Structural design defects may remain undetected until they are triggered by software changes. It is not possible to understand design defects during compile or run-time of program code; they appear after the modifications made to the class. They reduce the quality of software as a cause the following design problems:

- Increased software maintenance costs: All modifications made on software product after software is released to the customer are called maintenance cost of software [15]. Finding bugs, fixing the errors for next release and refactoring the software are the main items of software maintenance costs. The development and maintenance costs are increased depending on the increased software complexity. Software maintenance costs are generally more than 50% of the total software lifecycle costs. The structural design defects adversely affect the progress of software development and make it difficult to maintain the software. Consequently, the software development costs, especially software maintenance costs are greatly increased. Early

detection of software design defects is an important task to reduce software lifecycle costs.

- Reduce code reusability: Code reuse is the use of existing assets in order to create new assets for increasing the productivity and improving the software quality [11, 12]. The advantage of the code reusability is making use of same components in various use cases, thus enables effective usage of time and resources. Software design defects reduce the time to adapt the software to existing system, thus slows down the software to keep up with the change requirements.

- Reduce software flexibility: Continuous changes applied on software systems increase the software structural complexity and degenerate the overall system functionality. It is not easy to adapt constant changes for the inflexible software structures and therefore software becomes more unmanageable in means of software modules and interconnections between them [13]. Modification tasks performed to adapt software to changes are related with the flexibility of software, i.e. more complex tasks means less flexible software structure [14]. Software products that allow fast and easy modifications are considered flexible. In order to handle continuously changing requirements, the software needs to be designed flexible to any modifications. New feature implementations applied on software may introduce new defects on other modules of software product. In this case, software tends to become more complex and difficult to maintain over time. As a result, software maintenance costs increase dramatically.

- Vulnerable to introduction of new errors: Most of the errors detected in tests are gathered from structurally defective classes. Structurally defective classes most likely generate errors after any modifications are made on a class. Adding new features, changing classes according to new customer requirements or fixing bugs will trigger new errors on other classes that structurally defective class is highly coupled. Unless the design defects are corrected by refactoring processes, maintenance costs of these classes will be higher.

- Reduce the reliability of software: The reliability of software is the ability of software product to function in a certain way when used in the given environmental conditions [16]. Software reliability is one of the important key factors that affect the quality of software. When the software size and complexity becomes higher and the software product is developed with the lack of object-oriented design principles, the software design defects are likely to occur. Software design defects increase the intensity of failures in the software product that reduces the reliability of software. As the outcome of software product differs from the expected output, software product deviates from the specified requirements.

1.4 The Advantages of Early Defect Prediction

The early detection of structurally defective classes in development and maintenance phases of software projects has many benefits for improving software quality and software maintanance. The most important benefits are listed below:

- Saving testing effort and time: Finding defects in software is costly and time taking activity to detect and fix them. Prediction of software defects in earlier phases of the project lifecycle helps testers to focus on faulty modules of software. The defective classes can be prioritized in testing procedures, thus it saves significant proportion of testing time and leverages software testing effort.

- Reduction in software maintenance costs: As software systems tend to become more complex, adversely understandibility of software decreases and this situation dramatically increases the software maintenance costs. Nowadays, maintenance costs are typically more than 90% of the total cost of software. Software lifecycle cost can be reduced by preventing introduction of software design defects from the beginning of the project. Studies show that the relative cost of fixing defects found in testing are 15 times more costly than found during the design phase and nearly 3 times more than found during implementation phase [18]. Figure 1.3 shows a study performed by the IBM System Science Institute to find the relative costs to fix defects.

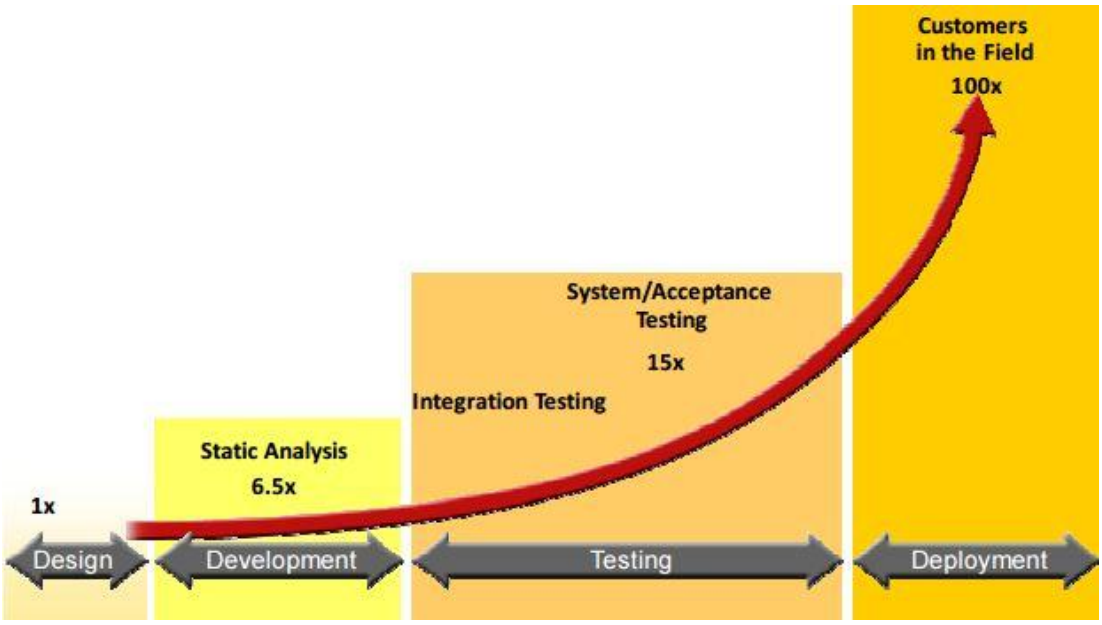

Figure 1.3 : Relative costs to fix software defects [18].

- Benefits refactoring of defective classes: Refactoring is all changes made on software structure without changing its functionality [9]. Software refactoring improves the quality of software as the software becomes more understandable and easier to modify. Getting familiar with defective classes helps developers to take precautions on these classes before these classes become difficult to maintain. Early estimation of defective components of software helps developers to refactor classes in order to correct their design defects and reduces the maintenance cost in further releases [17]. Regular refactoring processes made on software reduces the probability of the introduction of software defects and as a result the reliability of software product is increased.

1.5 Purpose of Thesis and Hypothesis

Although there have been many studies to predict defective modules software in the literature, there is no strict rules on defect prediction and none of the techniques dominates anothers. Defect prediction is still a matter of research with more and more susceptible to study aspects.

Most of the studies on defect prediction use the well-known public datasets in the literature [19]. These datasets consider each software fault as software defects in each release of software. For example, if a software module is erroneous on the

measured release, it is directly labeled in training set as defective; otherwise it is labeled as non-defective. These defects on the class may appear in that release by coincidence or may be triggered due to the fault of another class. On the other hand, design defects are not incidentally occurred; even if the classes are structurally defective these defects may remain undetected if the class is not changed. In poorly designed classes, unless the design quality of these classes is improved they generate errors after most of the modifications. Therefore, these classes will have high error frequency (EF), which is defined as the ratio between the number of errors and modifications.

Quality in use is the quality of software from users' perspective when it is used under specified conditions. External quality of software which is measured during the execution of software influences the quality in use. In the same way, the external behaviour of software is a reflection of the internal quality of software product. Class complexity, cohesion and coupling are the internal attributes of software product. In general terms, coupling is the degree of dependency between software modules, the cohesion is the consistency of software component functionality and the complexity indicates the degree of difficulty in understanding the complexity of the internal structure of the software [20]. Figure 1.4 illustrates the quality in software lifecycle [16].

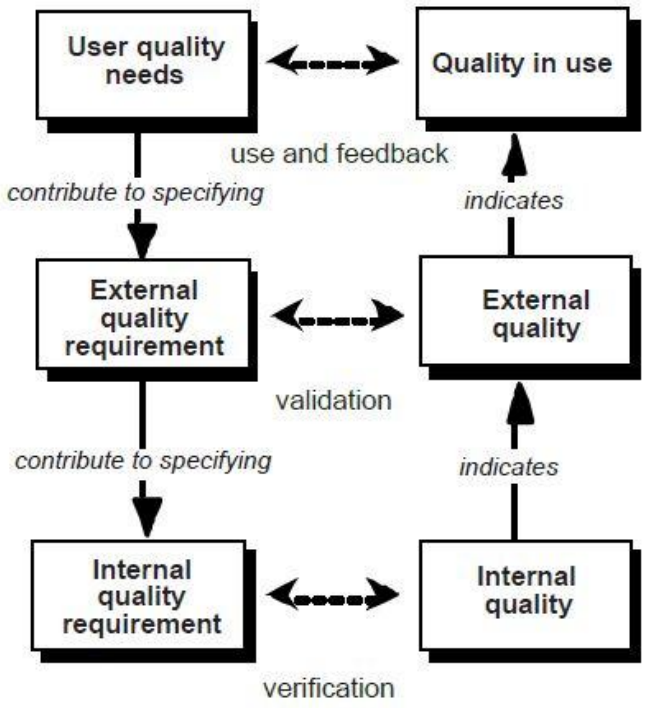

Figure 1.4 : Quality in the software lifecycle [16].

The commonly aggreed fact in literature is that high-quality software systems have high internal cohesion, low complexity and loose coupling [21]. Some internal properties of software can be revealed by using software design metrics have been widely accepted in literature [22-24]. However, it is difficult to work with metrics to create detection rules for detecting defects because of their various types, distributions and different minimum/maximum values. Also, different metrics should be used together to create a model for quality assessment; but it is difficult to determine the roles, weights and thresholds of metrics in creating such a model. Therefore, learning-based methods are used for defect prediction.

In learning-based systems the accuracy of the model strongly depends on the training set. To create a proper training set firstly several releases of the software projects are examined with the development teams and obtained the following observations:

- Structurally defective classes tend to generate most of the errors in tests, but healthy (non-defective) classes are also involved in some bug reports.
- Defective classes may not generate errors if they are not changed; errors arise after modifications.
- Healthy classes are not changed frequently and if they are modified they generate errors very rarely.

Considering these observations, one of the basic hypothesis at the beginning of the thesis is to predict software defects is software classes having structural design defects. Such classes mostly include one or more of the following attributes; they are complex, highly coupled to other classes, their internal cohesion is low or they have an inappropriate position in the inheritance hierarchy. Classes having too many functionality with too many methods become complex and in these classes tends to be more defect-prone. Deeper and large inheritance trees are not given preference in well-designed softwares. Also, well-designed software classes should have low coupling to others for reusability of modules.

Another hypothesis is that structurally defective classes are not indiscriminately occured; they may remain undetected if they are not changed and if they are changed they frequently generate errors. Based on this assumption, error counts (ErrC), change counts (ChC) and EFs for each classes of software product are taken into account for classification. ErrCs are the total number of bug fixes which are made on

a class in the observed *x* training releases and ChCs are the total number of changes in a class during the training releases. The ratio between ErrC and ChC of a class gives the EF of a class. Classes having low EFs can be tolerated and these classes can be thought as non-defective, whereas high error frequencies are most probably the indicators of structurally defective classes.

1.6 Contribution of the Thesis

Software design problems in software classes reduce understandability, flexibility and reusability of the system. Performing maintenance activities on defective components such as adding new features, adapting to the changes, finding bugs, and correcting errors, is hard and consumes a lot of time. Early estimate of error-prone classes helps developers to focus on faulty modules, thus reduces testing time and maintenance costs. In general, previous studies on defect prediction are generally conducted with publicly available datasets prepared based on defects on each release of software product. In this thesis, a learning-based decision tree model for detecting error-prone classes with structural design defects is proposed. The result of this thesis is provided following contributions:

- Instead of using publicly available datasets, datasets are constructed by examining real-world software project releases. Based on main observations, the EFs and ChCs of classes are considered to construct a proper data set for the training of the model.
- Classes having frequent errors above determined thresholds are considered as defective, whereas classes remaining under predetermined threshold values are labeled as healthy.
- Some classes are very rarely or never modified in the training releases and most of them have the ChC value zero. Even these classes need refactoring they remain undetected for a long time in the projects lifecycle as they stayed unchanged. The learning-based model created in this thesis also detects these classes with such insidious defects that do not appear until the class needs modifications.
- It has been shown that the proposed method succeeds in predicting frequently changing defective classes with relatively high EFs. Moreover, proposed

method also detects classes which neither generated errors nor were modified in the observation releases. Even software developers are not aware of the defectiveness of some of the predicted classes. Determination of such defective classes is an important contribution to the literature on the improvement of software quality and software maintenance activities.

- The decision tree model that was created with a dataset obtained from a project was applied to another project of the same company. We analyzed the results with the development team of the project and saw that the model was successful in finding defective classes in projects with similar characteristics.

The rest of the thesis is organized as follows: Next section shows basic and recent studies in the literature related with defect prediction. In the third section, technical background information about common defect prediction steps and the learning-based software defect prediction algorithms are explained. Section 4 provides information about proposed defect detection approach including basic steps, the source projects and how the dataset is created. Empirical studies on defect prediction and obtained results are given in Section 5. Final section summarizes the carried out study and provides recommendations for future work on defect prediction.

2. LITERATURE REVIEW

This section describes the basic studies conducted in the literature on defect prediction. Then, main defect prediction methods based on these studies are briefly described. Some of these methods are also used in some experiments for verification and comparison of results. Finally, recent studies are mentioned which are closely related thesis subject. Recent studies describing the study performed within the scope of the thesis, based on their advantages and differences have been demonstrated.

2.1 Main Defect Prediction Methods

As there have been many benefits of early prediction of defective software modules, several methods and algorithms have been developed in literature to identify software defects automatically. Publicly available datasets, private datasets and partial datasets created from the analysis of open-source projects are generally used to determine the candidates of defective software modules [32]. However, in these datasets each fault or bug is labeled as a possible defect. In contrast, this study explains how to use a machine learning technique to predict which classes in a system have structural design defects. Bugs or faults occurred during program compile or runtime are excluded, whereas software releases are examined to find frequently generated structural defects.

Some of the methods developed for detecting software defects are classified according to the technique used in related studies and described in below subsections.

2.1.1 Rule-based approaches

In this technique, detection strategies are used to detect and localize design flaws based on metric-based rules [25]. Some metrics may reveal the deviation from the good object-oriented design principles and metrics can be combined to create formulations. Potential bad smells based on poor design can be systematically determined by using detection strategies [10]. The sequence of detection strategy

begins with analysis of problem and followed by selection of problem-specific metrics. Detection strategies consist of logical statements and define thresholds for various metrics. Afterall, the candidates of design flaws are determined and examined. This technique is the basis of rule-based method and constitute the infrastructure of many similar method. An example representation of simplified detection strategy is illustrated in Figure 2.1.

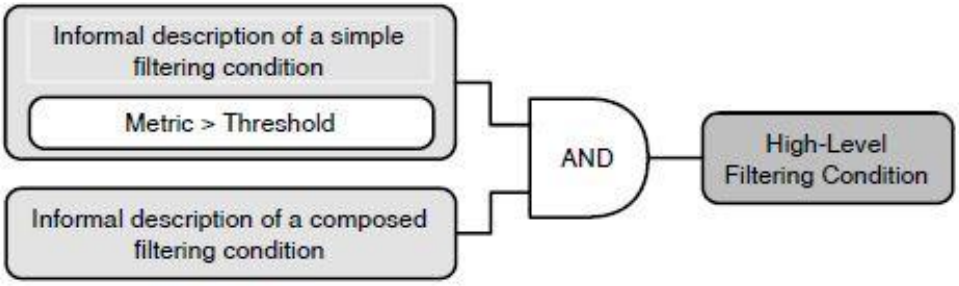

Figure 2.1 : A simplified representation of a detection strategy [10].

2.1.2 Machine learning-based approaches

In order to predict defective modules of software product, machine learning-based methods are broadly used in literature. Machine learning algorithms help creating prediction models based on dataset representation. These models are extracted from training of historical data and used to estimate unseen test data. In literature, widely used machine learning techniques to predict software defects are C4.5 for decision trees, Naïve Bayes for bayesian learners, Multilayer Perceptron for neural networks, Random Forest for ensemble learners and support vector machines [27].

2.1.3 Statistical-based approaches

Some studies in literature have been applied univariate and multivariate binary logistic regression statistical methods for software defect prediction [28, 29]. When traditional logistic regression models have been compared with models created by machine learning algorithms, machine learning models outperform to statistical logistic regression models [27].

2.1.4 Manual code review-based approaches

Manual inspection methods can be used for identification of software defects. A set of reading techniques can be used in order to help reviewers for inspections [30]. It is not an automated method to identify code smells and not easy to utilize for large

software projects [31]. Manually inspecting source code becomes unfeasible if the size and complexity of software increases.

2.1.5 Template-based approaches

As the size of the software increases manual inspection becomes a difficult task to perform. In order to locate design defects in the source code, identification of bad smells in source code should be automized. Template-based methods, which include the description of design flaw and characteristics of design problem are utilized automatically determination of bad smells within the software [33].

2.1.6 Visualization-based approaches

Suspicious modules that have design defects in large and complex software systems can be determined by using visualization-based detection techniques [34]. Software visualization helps identification of design flaws on large scale object-oriented software systems. Different colors can be assigned to indicate design problem on a class. An example illustration based on coupling metric (CBO) distribution filter on a class shown in Figure 2.2.

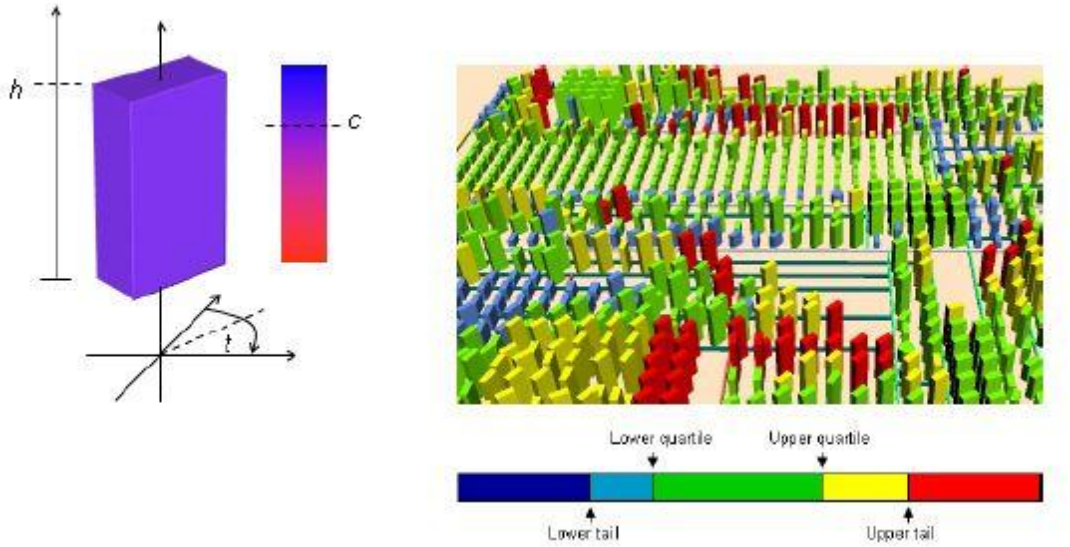

Figure 2.2 : Class representation and distribution filter [34].

2.2 Defect Prediction Tools

Several tools have been developed to automatically detect design flaws found in the software. These tools can greatly simplify the quality analysis tasks by automatically providing some measurements needed to improve quality of software. In this section,

some of the widely used open source and commercial defect detection tools are briefly mentioned.

FindBugs [35]: It is free software developed by The University of Maryland used for static analysis of Java source code to find different severities of bugs. It investigates common bug patterns and represents all of the potential bugs in software.

Coverity [36]: It is a commercialized tool for finding bugs in Java, C, C++ and C# source codes without running the software program. It handles comprehensive source code analysis for identification of critical defects in highly complex source codes by deeply examining each line of code in software.

PMD [37]: A source code analyzer utilized for finding possible source code flaws, complex classes and duplicated codes, which may cause potential problems in Java program. Even if the flaws found by PMD is corrected by refactoring process, there may be no hazards in functionality but the software will be vulnerable to identification of new errors.

Metrics [38]: An open source Eclipse plugin used for dependency graphs visualization as well as object-oriented metric attitude calculations. Widely used code or design metrics, such as number of methods (NOM), depth of inheritance (DIT), lack of cohesion in methods (LCOM) can be displayed in a dependency graph with different colors according to violations on metric values. An example illustration of dependency graph on Metrics tool is depicted in Figure 2.3.

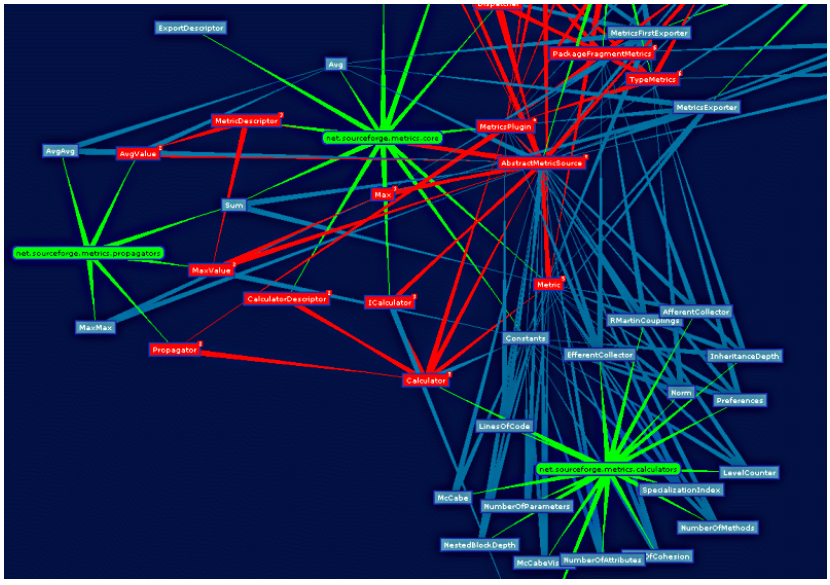

Figure 2.3 : Dependency graphs on metrics tool [38].

JDeodorant [39]: JDeodorant is an Eclipse plugin used for identification of four kinds of bad smells such as “Feature Envy”, “State Checking”, “Long Method” and “God Class” as well as providing suggestions on how to refactor these flaws. It detects bad smells with the help of ASTParser API and applies refactoring with ASTRewrite API of Eclipse [40].

inCODE [41]: It is an Eclipse plugin for automated detection of design problems and characterizes software system with the visual maps. Detection strategies [25, 26] are used to find design problem of class or method during the development of inCode. It also detects “God Class”, “Data Class”, “Feature Envy” and “Code Duplication” code smells. It isolates software developers by taking advantage of object-oriented metrics and continuously updates them to correct the design defects in development. An exemplary work of inCODE during the detection of design problems and process of marking the defect is illustrated in Figure 2.4.

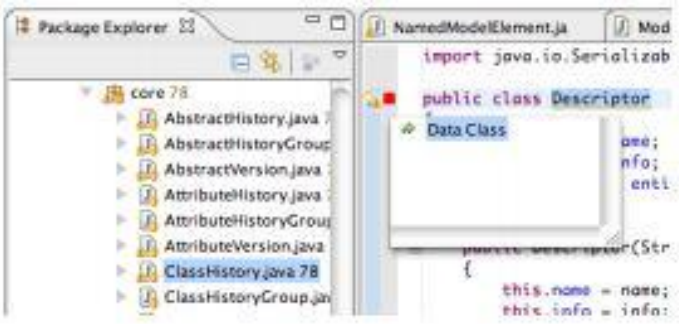

Figure 2.4 : An exemplary work of inCODE tool [41].

2.3 Recent Studies on Defect Prediction

Many studies have been carried out so far related with software defect prediction. As also stated in [32] that is the systematic review on defect prediction area, numerous studies have been conducted on this subject, whereas very few of them were applied on the software defects in the industrial software projects. In [42], the authors also mentioned about the bottlenecks of software defect prediction on industrial projects. Ref [43], the authors implemented their study on one of the large telecommunications company in Turkey for determination of defects earlier in project development lifecycle. They enhanced the performance of Naïve Bayes classifier [49] by adjusting the decision thresholds for better defect detection rates. They also utilized information content of data in terms of dependencies between

modules to decrease the introduction of false alarms, i.e. erroneously a defect-free module has been assumed defective. Similarly, in [44] the authors worked on two real-world projects to predict faults on files of industrial software. They used a regression model on both projects and predicted the majority of files with the highest error-densities that is the number of defects per line of code.

Many studies have benefited from the decision tree learners for bug prediction. Ref [45], the authors worked on seven Mozilla open source project releases for defect density prediction by using decision tree learners. They bent over many hypotheses in order to understand defect density of source code files by using source code metrics and the ability to predict them in the future. Their experiments showed that J48 tree learner [46] succeeded in predicting defect densities. In this thesis, the success of J48 classifier is also endorsed, but the main focus is the design defects of the classes and to categorize them ChCs and EFs of each class is calculated. In [47], the authors set up their experiments on Eclipse JDT project and investigated each commits, which may introduce defects. They performed commit-level defect prediction research by training the dataset with J48 model and periodically evaluated the performance of their prediction in a dynamic environment. In their experiments, J48 model was always the best performing algorithm for their dynamic approach. In addition, the authors in [59] inducted features of class-level dataset based on decision tree classifier for improving the performance of classifier. They applied three decision tree algorithms on dataset and provided relevant features to different classifiers for defect prediction. Their results showed that for defect prediction, classification based on new feature set has significant success when compared with the one all features is in the dataset. In this thesis, feature selection based on J48 classifier is also utilized during experiments.

In [10], [25] the authors presented detection strategies based on metric values to find software design problems. They introduced threshold values for better interpretation of metric values in order to determine problems. They combined design metrics and proposed relative thresholds for their values to identify design disharmonies. The authors of [48] also studied on threshold optimization for imbalanced software defect data. They reduced false alarms for better prediction of defective classes by applying decision threshold on Naïve Bayes classifier [49]. In this study, threshold values for EFs of classes are also utilized to tag them as “defective” or “healthy”.

Ref [50], the authors studied on the prediction of change-prone classes. They investigated the relation between object-oriented metrics and class change-proneness. Their results showed that structurally defective classes tend to change frequently. According to this information, in this study the number of changes as well as EFs of the classes are considered in categorizing and tagging them.

The change-classification of file-level software modules are investigated in [51] whether changes applied to source code is buggy or clean. They created their own corpus for text classification by reviewing 12 open source projects and they applied Support Vector Machine (SVM) [49] classifier on file changes. Their approach succeeded 78 percent accuracy on average and 60 percent average recall for buggy content changes. Similarly in [52], the authors also studied the possibility of bug occurrence related with changes in the source code file. They reduced the number of machine learning features for improving bug prediction performance. They performed experiments by using several classification-based feature selection algorithms on the training set obtained from software history. By applying this technique, they achieved successful enhancements for the performance of Naïve Bayes and SVM [49] classifiers. In order to obtain more appropriate metric set related with defect-proneness of software modules, the authors of [53] also applied F-score feature selection approach based on Linear Twin Support Vector Machine [54]. In this thesis, instead of providing all metrics for defect prediction, feature selection is applied and only most relevant metrics are given to the classifier for better results. Also, instead of using open source projects, industrial codes developed by Ericsson Turkey software center are examined and the comments or feedbacks of the development teams are considered during evaluation of results. After all reviews conducted on several releases of software systems by considering the EFs and ChCs of the classes, the individual dataset is created.

Ref [55], the authors proposed a general defect prediction framework including both evaluation of learning schemes and defect prediction based on the best performing scheme. They concluded that changes in datasets impact on prediction results and different learning schemes should be used for different datasets. In this study, it is also experienced that the proper representation of dataset is important for better prediction results.

Prior empirical studies [56], [57] were generally based on publicly available datasets, i.e. NASA Metrics Data Program (MDP) [58], whereas in this research data sets are constructed analyzing several incremental releases of two long-standing industrial software systems and evaluating bug reports to match them with the error-prone classes. Own training and test sets considering reasonable threshold values for EFs are created. The observations of this study showed that higher EFs are early indicator of structurally defective classes and if such classes remain unchanged, they generate errors after modifications. The aim of learning-based method used in this study is identifying such error-prone classes.

3. TECHNICAL BACKGROUND

In this section, the general defect prediction approach steps and some of the widely used algorithms in literature for defect prediction are given.

3.1 Common Defect Prediction Steps

Defect prediction models have been constructed using historical defect data from software version control repositories. Defect information of a software class is generally obtained from bug reports or release notes. Based on these informations, classes are labeled as “defective” or “healthy”. Prediction model is created by training of a dataset which includes software classes as instances, metrics associated with defect-proneness of a class as features and “defective/healthy” class tags for classification. The prediction model can be used for predicting the defect-proneness of an unseen test set. Afterall, the obtained results are evaluated for understanding the overall performance of utilized machine learning algorithm during training phase. The basic steps of learning-based defect prediction process are illustrated in Figure 3.1.

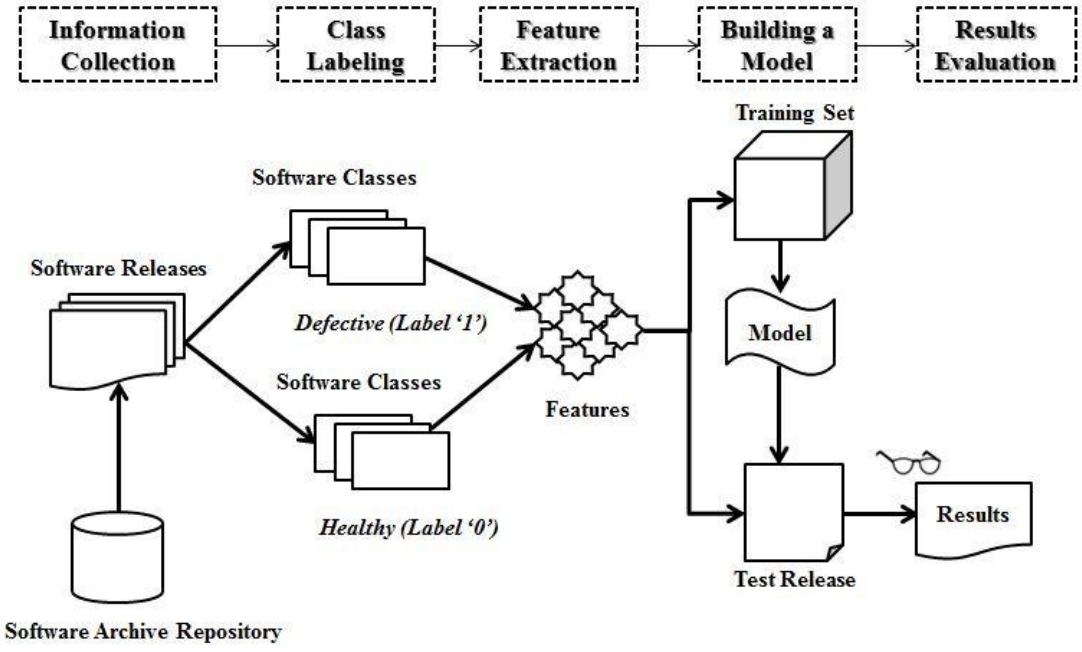

Figure 3.1 : Common steps of learning-based defect prediction methods.

- Information collection: Software versions or releases are steps in the software development process. Today, during development of both commercial software and open source software projects, version control systems are widely used. Software releases kept in archieve repositories, such as Concurrent Version Systems (CVS) [60]. The first step of the defect prediction process is to collect information from software system. In this phase, releases are investigated to clarify which strategy to follow for defect detection. Release notes for each software project include all modifications; i.e. bug fixes, new feature implementations, change requests made on each class. For each release, related information are gathered and examined to determine the defective classes.

- Class labeling: In order to determine the defectiveness of the collected instances; i.e. classes of software system, the process should be followed by labeling of the instances. Tagging each class as "defective/healthy" or "1/0", a labeling strategy or threshold definitions should be considered. Correctly deciding the labels of each instances will increase the success of the prediction model.

- Feature extraction and creating the training set: Features or attributes of each class instance can be extracted from various metric collection tools; such as iPlasma [61], Chidamber & Kemerer Java Metrics (ckjm) tool [62], Eclipse Metrics plugin [38], etc. These tools provide design and code metrics revealing size, coupling, cohesion, complexity, level of inheritance information about the object-oriented classes. When the defect prediction is considered as a classification problem, training set can be created by combining object-oriented software classes (instances), the metric values (attributes) and "defective/healthy" tags (labels of each instance).

A training set to be used by machine learning algorithm to build a prediction model can be constructed as shown in Figure 3.2.

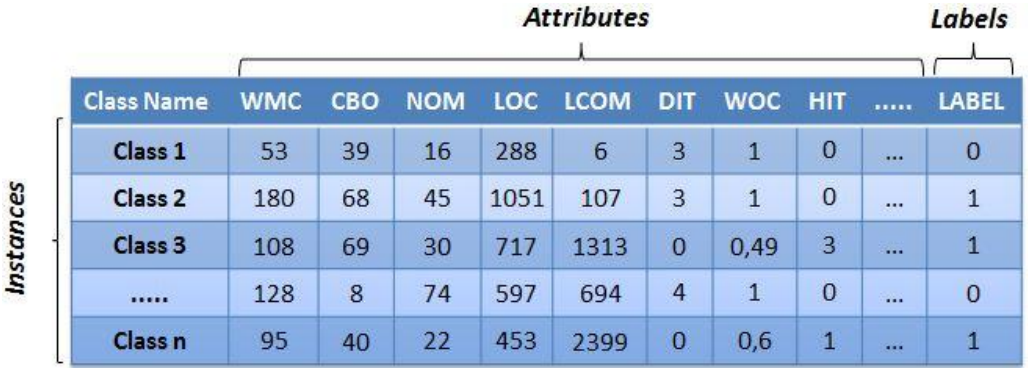

Figure 3.2 : An exemplary training set for defect prediction.

- Building a prediction model: Software design and code metrics may reveal some information about the object-oriented classes; however it is difficult to work with metrics to create certain rules for detecting defects because of their various types, distributions and different minimum/maximum values. Also, different metrics should be used together to create a model for quality assessment; but it is difficult to determine the roles, weights and thresholds of metrics in creating such a model. Therefore, a learning-based approach to form a prediction model is needed. Using the constructed training set, common machine learning algorithms in literature; such as bayesian learners, statistical regression algorithms, decision tree learners, support vector machines [49] can be utilized to create a defect prediction model. The prediction model is used for predicting the labels of unseen test dataset instances.

- Results evaluation: For the assessment of the learning-based model performance, a test dataset is needed besides the training dataset. The labels of class intances within the unseen test dataset are predicted by machine learning model. The overall success of the prediction outcome is calculated depending on the status of the results as “True Positive (TP)”, “False Negative (FN)”, “False Positive (FP)” and “True Negative (TN)” which constitute confusion matrix as represented in Table 3.1.

Table 3.1 : Confusion matrix.

Actual	Predicted	
	Defective	Healthy
Defective	TP	FN
Healthy	FP	TN

"TP rate" or "recall" is the measure of correctly classified defective software classes. Recall is the number of correctly classified "defective" classes divided by the number of actual "defective" classes and it can be calculated with the following equation **(3.1)**.

$$Recall = TP/(TP + FN) \quad \textbf{(3.1)}$$

"FP Rate" is the measure of false alarms, i.e. erroneously a "healthy" class has been assumed as "defective". The formulation of false alarms is given in **(3.2).**

$$FP\ Rate = FP/(FP + TN) \quad \textbf{(3.2)}$$

"Precision" shows the predicted positives that are actually positives, i.e. the number of correctly classified software classes as "defective" divided by the total number of classes labeled as "defective". The following equation represents the calculation of precision in **(3.3)**.

$$Precision = TP/(TP + FP) \quad \textbf{(3.3)}$$

"TN Rate" is the predicted negatives that are really negatives; i.e. the number of correctly classified healthy classes is divided by the actual healthy classes. The equation of "TN Rate" is formulated in **(3.4)**.

$$TN\ Rate = TN/(TN + FP) \quad \textbf{(3.4)}$$

The classification accuracy which shows the proportion of correctly classified classes in whole dataset is described in **(3.5)**.

$$Accuracy = (TP + TN)/(TP + TN + FP + FN) \quad \textbf{(3.5)}$$

If the number of defective class samples and healthy class samples are imbalanced in dataset, the classification accuracy would not be enough for evaluation of prediction model performance. In this case, both precision and recall values should be take into consideration.

3.2 Software Defect Prediction Algorithms

A learning-based defect prediction model can be created by using machine-learning algorithms in the literature. In machine learning approach, it is possible to learn from history and predict the future behavior of data. During the creation of learning-based model, typical machine-learning approach is followed in which model is trained by using a training set including object-oriented software classes with class metrics and class-level defect labels; then model is applied on an unseen test set whose defect labels are unknown.

In defect prediction literature various machine learning algorithms are used to understand the relationship between metrics and defectiveness of software modules; such as logistic regression, decision tree, artificial neural networks, Bayesian networks and support vector machines methods, etc. However, none of the algorithms in the literature outperform another for all datasets to solve defect prediction problem. In learning-based systems the accuracy of the model strongly depends on the training set. To create a proper training set several releases of software projects needs to be examined and characteristics of software defects should be identified.

In this thesis, a learning-based method is proposed to detect software classes that include structural design defects. Several experiments are conducted by using Naïve Bayes, logistic regression and decision tree algorithms to create the best performing learning model that is trained with data collected from on-going real-world projects. Background information for the used algorithms in this study is provided in the following subsections.

3.2.1 Naïve bayes learning

Naïve Bayes classifier is widely used for defect prediction in many studies in literature as it is known to be simple and robust machine learning approach that has good predictive classification performance [64], [65]. Naïve Bayes theorem is based on Bayes theorem, in which the posterior probability of $p(C_i|x)$, i.e. the probability of instance x belonging to C_i, is proportional to prior probability of C_i and the likelihood of $p(x|C_i)$ [49]. The Bayesian theorem which represents the posterior probability in terms of prior probabilities and likelihood given in **(3.6)**.

$$p(C_i \mid x) = \frac{p(x \mid C_i)\, p(C_i)}{p(x)} \quad \textbf{(3.6)}$$

As the predictor prior probability $p(x)$ is a constant, it can be ignored in calculation. In binary problems such as defect prediction problem, the sum of each probability is as shown in **(3.7)**.

$$p(C_1 \mid x) + p(C_2 \mid x) = 1 \quad \textbf{(3.7)}$$

In defect prediction problem, instances are object-oriented classes and attributes are referred as object-oriented design metrics. The posterior probabilities of each class is "defective" or "healthy" classification based on metric values. The instance x is assigned to C_i class which satisfies the rule $p(C_i \mid x) > 0.5$; i.e. if posterior probability is greater than 0.5, the class instance is assigned to "defective" class (C_1), otherwise it is assigned to "healthy" (C_2) class. However, if the number of "defective" instances and "healthy" instances are imbalanced in dataset, the default threshold may negatively affect the predictive outcome. In this case, the threshold needs to be tuned for better results [43].

Naïve Bayes classification is the simplified form of Bayesian theorem in which features are assumed to be independent [66]. Because of the conditional independence assumption, the class probabilities can be written as in **3.8** where x_t are the values for attributes of sample x.

$$p(C_i \mid x) = \prod_{t=1}^{n} p(x_t \mid C_i) \quad \textbf{(3.8)}$$

3.2.2 Logistic regression

Along with the machine learning models, statistical-based models are also widely used for defect prediction in literature [67-70]. Logistic regression measures the relationship between dependent variable (labels) and independent variables (features) [71]. The main purpose of logistic regression is to find the most simple model to predict the outcome of the dependent variable. Defect prediction problem refers to

binary logistic regression in which the number of dependent variable can only take two categories, i.e. “defective” and “healthy” class labels. The general formulation of logistic regression model is shown in **(3.9),** where x_i is the independent variable and π is the probability that the dependent variable belonging to a class C_i; i.e. C_1 represents a “defective” whereas C_2 represents “healthy” class in binary logistic regression. The β parameters are the coefficients predicted by maximizing the likelihood function on the set of independent observations.

$$\pi = \frac{1}{1 + e^{-\left(\beta_o + \sum_{i=1}^{k} \beta_i x_i\right)}} \tag{3.9}$$

3.2.3 Decision tree learning

There are several studies highlighted the successful results of decision tree algorithms to predict defective modules of software system. The decision tree approach is widely used in literature because of the robustness of the algorithms and understandable, interpretable results of the algorithm [26], [45], [53], [59], [73]. The decision tree can learn the characteristics of system from the provided metrics of software classes and generates rules based on the most relevant metric values with the defect-proneness of each object-oriented class instance. The general structure of decision tree includes a root node, depending on the decision rules zero or more internal nodes with one or more leaf nodes. The top most node of the decision tree is known as the root node. The internal nodes of the decision tree split into branches based on detection rules of class metrics. Leaf nodes indicate the tag of the object-oriented class instance, namely “1” represents “defective” classes and “0” represents “healthy” classes. The constructed decision tree model can be applied on an unseen test set to identify defective classes. Starting from the root node, attributes of the test instances are recursively evaluated according to rules defined in the internal nodes. Based on the results of these evaluations the test instance is assigned to one of the leaf nodes that shows whether the class belongs to “defective” class or not. The constructed decision tree model can be applied on an unseen test set to identify defective classes. Starting from the root node, attributes of the test instances are

recursively evaluated according to rules defined in the internal nodes. Based on the results of these evaluations the test instance is assigned to one of the leaf nodes that shows whether the class belongs to "defective" class or not. The general structure of decision tree is shown in Figure 3.3.

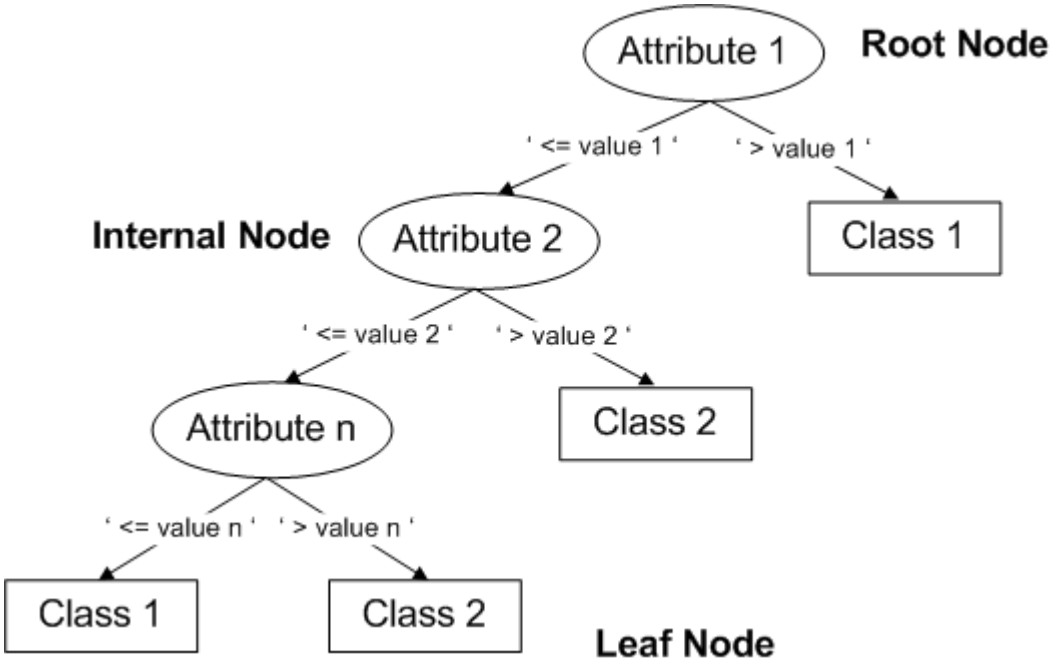

Figure 3.3 : The general structure of decision trees.

At each internal node the best attribute is selected for splitting the set of instances into subsets by using the "Information Gain" (change in entropy) measure [72]. The entropy E of the attribute is calculated by using the equation given in **(3.10)**.

$$E(T) = -\sum_{i=1}^{n} p_i \log_2 (p_i) \qquad \textbf{(3.10)}$$

T shows the set of samples in training data set and p_i is the probability of an instance belongs to class C_i. The highest entropy means the most useful information about the classification. The information gain reveals the importance of attributes and it is used for arrangement of attributes in internal nodes. The attribute with the maximum information gain is chosen as the root node in decision tree and in each iteration, the attribute with the highest information gain is preferred for discriminating the instances into branches. The information gain is the difference between entropy of the parent node and the weighted average entropy of children nodes as given in (**3.11**).

$$Information\ Gain = E_{parent} - AVG\left(E_{children}\right) \quad \textbf{(3.11)}$$

In this thesis, using the data in training set the decision tree learning-based model is constructed and the model provided promising results in predicting the defective classes in the further releases of the same project or in other projects with similar characteristics.

4. PROPOSED DEFECT DETECTION APPROACH

In this section, the main steps of the proposed defect detection approach and the used algorithms within this thesis are described in detail.

4.1 Basic Steps of the Approach

In this thesis, a learning-based approach is proposed to detect software classes that have design defects. Learning-based methods use historical data collected from software systems to predict problems in current and future systems. The performance of these methods depends greatly on the data sets that are constructed using the historical data. Therefore it is very important which metrics of classes are collected, how the bug reports are interpreted and how the classes are tagged. In this section main concepts of the proposed approach are briefly explained, then the details are given in the following sections.

The architecture of the approach is illustrated in Figure 4.1 and steps are summarized in below subsections.

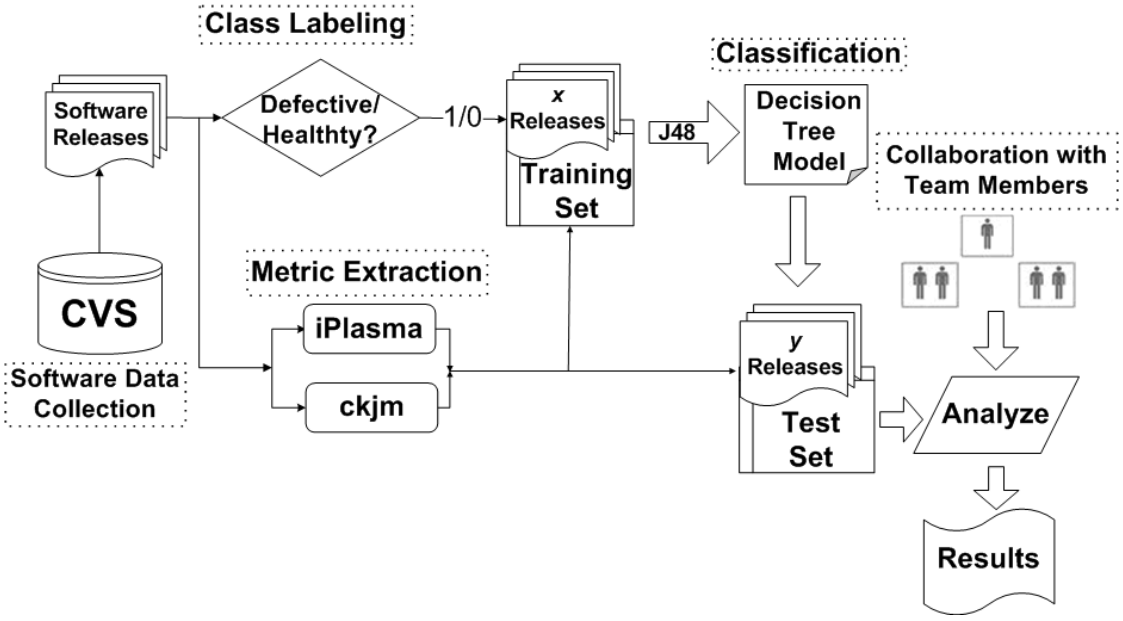

Figure 4.1 : Architecture of the proposed detection model.

4.1.1 Constructing the data set

At the beginning of this study, notes of several releases of a project are examined to gather bug fix or CR information for each class in all versions. These releases are called training releases of the proposed method. The number of training releases *x* may depend on the error density in the tests, however the experiences of this study show that at least five releases should be observed to collect reliable data in creating the data set.

After the bug fix and CR information of each class in training releases are collected, EFs (bug fix per change) of classes are calculated. Classes are labeled as "defective" or "healthy" according their EFs and ChCs. In order to determine the threshold of the EF correctly, several experiments are made on the real data and the results are discussed with development teams of each project. At the same time, the metric values of each class in all versions are extracted. These values constitute the attributes of the classes in the data set. For this purpose, the software design metrics are used that reflect properties of classes such as size, coupling, cohesion, complexity, level of inheritance.

The details about the creation of the data set are given in Section 4.3.

4.1.2 Feature selection and building the decision tree for classification

In order to have a rich data set with more information at the beginning a lot of metrics are collected. Then, to have a simple, understandable and proper classification model, the number of the attributes (metrics) of classes in the data set are reduced by a feature selection algorithm. This algorithm selects metrics that are strongly related to the error-proneness of the classes. After the elimination of the unrelated metrics the training set is constructed. The training set contains the metrics selected by the feature selection algorithm and categories of classes as defective or healthy. Using the training set a decision tree model is constructed and trained with the collected data.

4.1.3 Testing results and evaluation

The decision tree model is applied to a later release (test release) of the project that is not included in the training set and the defective classes are identified. Consecutive *y* releases following the test release (observation releases) are examined and the EFs of

the classes are calculated that the decision model classified as defective and healthy. With the cooperation of the project team, the success of the proposed method in detecting classes that have design defects and need refactoring is evaluated. The model is also applied to a different project than the reference project.

Training (*x* successive releases), test and observation releases (*y* successive releases) of the project are shown in Figure 4.2.

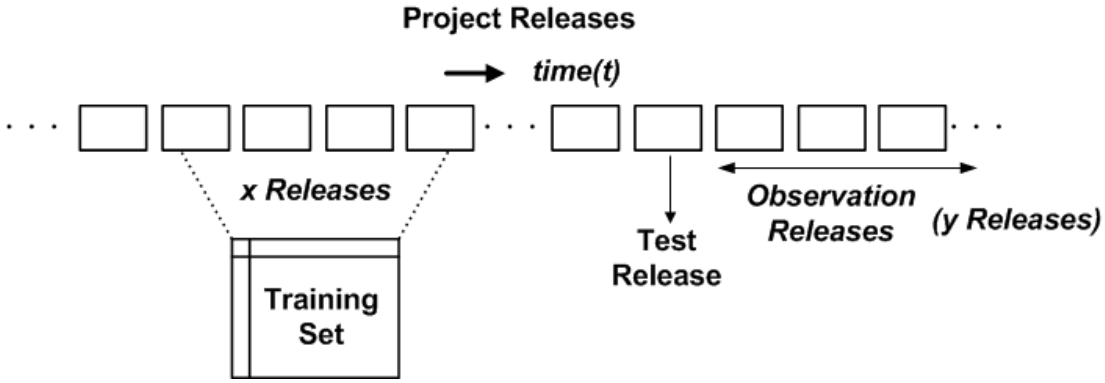

Figure 4.2 : Training, test and observation releases.

4.2 The Source Projects

In this thesis, the proposed approach is constructed and evaluated using data collected from ongoing real-world projects that are developed by Ericsson Turkey for two leading companies in the field of telecommunications. As these are long-standing software systems for almost four and six years, they reached a certain quality standard and the ratio of the defective classes is low. However, in today's fierce competition environment numerous opportunities arise in telecommunication industry. New levels of customer needs encourage operators to be innovative, thus service providers must react faster. In this agile environment, business rules become more variable and complex so that the software systems must continuously adapt to the new requirements. As a result software classes are modified and these modifications trigger errors if the classes are poorly designed.

Ericsson's Software Customization Center (SCC) aims to deliver high-quality, reusable solutions for end to end customer satisfaction in most efficient way. Identification of customer requirements, software design, implementation and testing are handled by this organization. Software solutions are developed in Java language

and code files include about thousands of lines. Existing projects of SCC department are maintained in a CVS repository [60].

Each release of software has its own release notes file which includes the changes, enhancements, bug fixes made to a specific version. The time between two releases is approximately two weeks. The examined first project (Project A) is being developed for six years and includes 810 software classes. The second project (Project B) is being developed for four years and includes 790 software classes.

The training set is constructed by examining classes from 46 successive releases of the Project A. Then the decision model is applied to a new release (test release) of the same project and structurally defective classes are detected to analyze the success of proposed learning-based method. After the test release, the errors and changes in classes are observed for 49 consecutive releases. The same model trained with data from Project A is also applied to a release of Project B. The performance of the proposed method is evaluated by observing 49 releases of Project B.

The collaboration with mentor group of both projects is performed during the study. Each mentor group consisted of two software developers, a solution architect and service project manager. In creating the dataset and evaluating results, their contributions are utilized.

4.3 Creating the Dataset

In learning-based systems creating proper data sets for training the model is the most critical part of the approach. Training the model with incorrect or insufficient data will clearly degrade the detection performance. As the aim of this thesis is detecting classes that have design problems and generate errors frequently, a method is developed to find such classes automatically in the later releases of the projects using historical data. Our proposed model is trained with the characteristics of these defective classes. To determine the characteristics of the classes, software design metrics are used and to categorize them automatically EFs and ChCs of each classes are calculated. At the end, a dataset is obtained in the matrix form. Each row of the matrix refers to a software class and contains *m* different metric values that represent the attributes of the class. The last element of the row is the predetermined category (tag) of that class, where 0 stands for healthy and 1 for defective. A class c_i is

represented as a vector with m+1 elements as follows: $c_i = (a_{1,i}\ a_{2,i}\ ....\ a_{m,i}, ti)$, where $a_{j,i}$ denotes the j^{th} attribute of the class c_i and t_i its category (healthy or defective). If the reference project includes n software classes than the data set $D_{n\ (m+1)}$ is a matrix with n rows and $m+1$ columns.

4.3.1 Collecting metric attributes

In order to determine the relation between error-proneness of a class and metric combinations, 23 static code and object-oriented design metrics defined in different studies are collected [10], [22], [24]. These metric values for each class is extracted using iPlasma [61] and the Chidamber and Kemerer (CK) Java metrics (ckjm) tools [62].

The rationale behind choosing these metrics was to find metric combinations which are the best predictors of defective classes.

The aforementioned metrics used in this thesis and the reasons for their usage is concisely defined in the following paragraphs.

4.3.1.1 Complexity and size metrics

WMC (Weighted Methods per Class) and AMW (Average Method Weight) is the measure of total and average complexity of all methods in a class, respectively [10]. It is expected that EF is higher for complex classes because of the difficulty of the class maintenance.

LOC (Line of Code) is the total nonempty, non-commentary lines of the class. Some studies showed the association with module size and defect-proneness [73]. Class with excessive size is expected to reveal more errors.

NOM (Number of Methods) is the functional size of a class in terms of the number of methods per class.

NOA metric shows the number of attributes of a class.

NAS (Number of Added Services) is the number of public methods added to a class.

WOC (Weight of a Class) shows the degree of how interface of a class implements methods that reflects the class behavior.

4.3.1.2 Cohesion metrics

LCOM (Lack of Cohesion in Methods) shows the relativeness of methods in a class. Low cohesion implies that the class comprises methods with different responsibilities, thus needs refactoring. Unless the class is not divided into subclasses, it is expected to have a high EF value is expected.

4.3.1.3 Coupling metrics

ATFD (Access to Foreign Data) is the number of external classes whose attributes are accessible from the measured class. High ATFD value shows the coupling of the class with other classes, i.e. class may be affected from the modification of other classes.

CBO (Coupling between Object classes) represents the number of coupled classes to measured class. If a given class becomes more coupled to others, it becomes unmanageable in terms of maintainability and testability.

Ca (Afferent Coupling) is the number of classes uses the measured class. Highly coupled classes are affected by changes in another.

RFC (Response for a Class) is the number of invoked methods in response to a message received by an object of the measured class. High value of RFC metric may indicate the complexity of given class, therefore indicates the error-proneness.

FDP (Foreign Data Providers) is the number of different foreign attributes accessed by a method. Low value may indicate the misplacement of a method.

FANOUT (Number of Called Classes) is the number of dispersed operation calls of a class [10]. CC (Changing Classes) is the number of classes that the methods in the classes call the given method of the class. CM (Changing Methods) is the number of different methods of calling the measured method. Amendments to such classes affect the classes called by measured class, thus this class-specific error spread to other classes.

4.3.1.4 Interface-related metrics

NPM is the number of public methods in a class. NOPA is the number of public attributes of the class. A high number of public members declared in a class may indicate the class may need to be split into subclasses.

NOAM (Number of Accessor Methods) is the number of getter / setter methods declared in the interface of class. Higher values indicate the misplacement of the method.

4.3.1.5 Inheritance-related metrics

NOC (Number of Children) is the number of directly derived classes from the given class. Classes having too many subclasses should be tested carefully, as the modifications related to such classes may have ripple effect on others.

DIT (Depth of Inheritance) shows the inheritance level of a given class in the class hierarchy. A deeper class in the hierarchy inherits more property of methods and classes, thus predicting the behavior of such complex class become difficult.

HIT (Height of Inheritance Tree) is the height in the hierarchy of a class to its deepest subclass. Deep class hierarchies may become more error-prone.

BUR (The Base class Usage Ratio) shows the ratio of dependency of a class to its base class. Low dependency to base class may indicate inheritance problem.

4.3.2 Assessment method for class labeling

In this thesis, the observations and experiences on software systems show that structurally defective classes tend to generate most of the errors in tests, but healthy classes are also involved in some bug reports. However, defective classes may not generate errors if they are not changed; errors arise after modifications. Healthy classes are not changed frequently and if they are modified they generate errors very rarely. Therefore, as it is also observed in experiments of this study, training the classifier only with the current release and testing the model with the next release do not provide reasonable results. To build a proper learning-based detection method the following points should be considered for categorizing classes.

1- Classes should be observed in long terms.

2- The status (erroneous/error-free) of a class after modifications is important.

3- The number of changes in a class should be considered.

The calculations based on these considerations are provided in detail in the following subsections.

4.3.2.1 Change count and error frequency calculation

In order to be sure about the validity of our proposed approach, 46 releases of Project A are reviewed and the reasons for changes in classes are determined. As it is been observed from release notes, classes are modified because of bug fix or CR of the customer. An exemplary template showing the total bug fixes, change requests made on a class along with 46 releases and EF values are given in Figure 4.3.

Release Number	Class Name	Is a BUG?	Is a Change Request?	Error Frequency
4_R0A_01	Class 1	NO	**YES**	12/18
4_R0A_02	Class 1	NO	NO	12/18
4_R0A_03	Class 1	NO	NO	12/18
4_R0A_04	Class 1	NO	NO	12/18
4_R0A_05	Class 1	NO	NO	12/18
4_R0A_06	Class 1	**YES**	NO	12/18
4_R0A_07	Class 1	**YES**	NO	12/18
4_R0A_08	Class 1	NO	NO	12/18
4_R0A_09	Class 1	NO	NO	12/18
4_R0A_10	Class 1	NO	NO	12/18
4_R0A_11	Class 1	NO	NO	12/18
4_R0A_12	Class 1	NO	NO	12/18
4_R0A_13	Class 1	NO	NO	12/18
4_R0A_14	Class 1	NO	NO	12/18
4_R0A_15	Class 1	NO	NO	12/18
4_R0A_16	Class 1	NO	NO	12/18
4_R0A_17	Class 1	NO	NO	12/18
4_R0A_18	Class 1	NO	NO	12/18
4_R1A_01	Class 1	NO	YES	12/18
4_R1A_02	Class 1	NO	NO	12/18
4_R1A_03	Class 1	**YES**	NO	12/18
4_R1A_04	Class 1	NO	NO	12/18
4_R1A_05	Class 1	NO	**YES**	12/18
4_R1A_06	Class 1	NO	**YES**	12/18
4_R1A_07	Class 1	**YES**	NO	12/18
4_R1A_08	Class 1	**YES**	NO	12/18
4_R1A_09	Class 1	**YES**	NO	12/18
4_R1A_10	Class 1	NO	NO	12/18
4_R1A_11	Class 1	**YES**	NO	12/18
4_R1A_12	Class 1	NO	NO	12/18
4_R1A_13	Class 1	**YES**	NO	12/18
4_R1A_14	Class 1	NO	**YES**	12/18
4_R1A_15	Class 1	NO	NO	12/18
4_R1A_16	Class 1	NO	NO	12/18
4_R1A_17	Class 1	NO	NO	12/18
4_R1A_18	Class 1	NO	NO	12/18
4_R1A_19	Class 1	NO	NO	12/18
4_R1A_20	Class 1	**YES**	**YES**	12/18
4_R1A_21	Class 1	NO	NO	12/18
4_R1A_22	Class 1	NO	NO	12/18
4_R1A_23	Class 1	NO	NO	12/18
4_R1A_24	Class 1	NO	NO	12/18
4_R1A_25	Class 1	**YES**	NO	12/18
4_R1A_26	Class 1	NO	NO	12/18
4_R1A_27	Class 1	**YES**	NO	12/18
4_R1A_28	Class 1	**YES**	NO	12/18

Figure 4.3 : An exemplary training set template.

The ErrC of each class is calculated as given in **(4.1)**, that is the total number of bug fixes which are made on this class in the observed *x* training releases. The ErrC of

class c is formulated as in **(4.1)**, where $e_{c,i} = 1$ if the class *c* is modified in release *i* because of a bug fix; otherwise $e_{c,i} = 0$.

$$ErrC_c = \sum_{i=1}^{x} e_{c,i} \quad \textbf{(4.1)}$$

Similarly, the CR count of each class that is the total number changes in the class made because of CRs of the customer is calculated as in **(4.2)**. The CR count of class *c* is formulated as in **(4.2)** where $r_{c,i} = 1$ if the class *c* is modified in release *i* because of a CR; otherwise $r_{c,i} = 0$.

$$CRcount_c = \sum_{i=1}^{x} r_{c,i} \quad \textbf{(4.2)}$$

The sum of the ErrC and CR count gives the total number of changes in a class *c* during the training releases as shown in **(4.3)**.

$$ChC_c = ErrC_c + CRcount_c \quad \textbf{(4.3)}$$

Finally, the EF of the class *c* is calculated as given in **(4.4)**, which is the most important value used to label classes as defective or healthy.

$$EF_c\,\% = \frac{ErrC_c}{ChC_c} * 100 \quad \textbf{(4.4)}$$

4.3.2.2 Threshold selection

To tag classes in the training set as defective and healthy two thresholds t_1 and t_2 considering both ChC and EF are needed, respectively. If both ChC and EF values of a class *c* exceed the threshold values this class is tagged as defective as shown in **(4.5)** where tag_c represents the label of the class *c*.

$$tag_c = Defective, if\ (ChC_c \geq t_1\ and\ EF_c \geq t_2) \quad \textbf{(4.5)}$$

The real values of t_1 and t_2 are assigned after examining the source projects with the help of the development teams as explained in the section "Empirical study and results". Since these values may depend on the characteristics of the project, different

threshold values can be selected by teams according the test results. These values should be assigned carefully, because selecting low thresholds increases the false positive detection rate of the method. On the other hand, assigning too high threshold values makes the method overlook many of the defective classes.

4.3.2.3 Considering rarely or never modified classes

Some classes are very rarely or never modified in the training releases and most of them have the ChC value zero. It is not correct to immediately tag these classes as healthy because defective classes may not generate errors if they are not modified. To find a method, the metric values of classes with high EFs are examined. The characteristics of these classes are investigated with development team and it has been realized that the EFs of them were highly correlated to their complexity (WMC). Therefore, classes with very low ChCs are also labeled as defective if their complexity metric (WMC_c) is higher than 1.5 times of the average of the classes previously tagged according **(4.5)**. The remaining classes with low ChC or EF are labeled as healthy. As a result, the rules in **(4.6)** is used for labeling the classes in the training phase.

$$tag_c = \begin{cases} Defective, if\ (ChC_c \geq t_1\ and\ EF_c \geq t_2), \\ Defective, if\ ((ChC_c < t_1\ or\ EF_c < t_2)\ and\ WMC_c \geq AVG * 1.5), \\ Healthy, otherwise. \end{cases} \quad \textbf{(4.6)}$$

Finally, a dataset $D_{n\ (m+1)}$ is obtained as a matrix with n rows and $m+1$ columns, where n is the number of classes, m is the number of attributes (metrics) and +1 stands for the tag. An example matrix form of dataset is shown in Figure 4.4.

Class Name	ATFD	BUR	CBO	CM	DIT	FDP	HIT	LOC	NOA	NOM	WMC	AMW	CC	ANOU	NAS	NOAM	NOPA	WOC	NOC	RFC	LCOM	Ca	NPM	LABEL
Class 1	19	0,52	18	0	4	7	0	294	3	23	62	2,7	0	60	1	0	0	1	0	138	194	0	2	0
Class 2	35	1	13	5	1	6	0	551	14	38	90	2,37	3	62	0	13	8	0,49	0	135	603	3	36	1
Class 3	26	0,71	19	0	4	7	0	471	0	25	103	4,12	0	92	1	0	0	1	0	162	179	0	2	0
Class 4	12	1	11	0	1	4	0	474	13	39	64	1,64	0	41	0	14	3	0,55	0	132	659	0	38	1
Class 5	48	1	4	8	0	3	0	368	23	48	62	1,29	1	56	0	41	1	0,09	0	126	1181	3	47	1
Class 6	112	1	21	0	0	20	0	845	1	11	50	4,55	0	124	0	0	1	0,5	0	171	66	0	2	0
Class 7	17	1	8	0	2	5	2	367	2	34	55	1,62	0	35	0	0	0	0,5	6	153	462	6	3	0
Class 8	29	0,23	21	0	4	13	0	357	5	19	55	2,89	0	58	0	0	0	1	0	155	193	0	2	1
Class 9	68	1	11	0	1	11	0	777	24	80	146	1,82	0	68	0	35	5	0,47	0	186	2438	0	72	0
Class 10	12	1	11	1	1	4	0	630	19	56	92	1,64	1	47	0	23	13	0,42	0	140	1314	1	52	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Class n	21	0,56	14	0	4	5	0	288	6	16	53	3,31	0	68	1	0	0	1	0	123	107	0	2	1

Figure 4.4 : An exemplary matrix form of dataset.

4.4 Constructing the Detection Model

Detection of the defective classes can be considered as a classification problem within the concept of machine learning [74]. A classification model is created by applying the labeled training set that contains the attributes (metrics) of classes to a machine learning algorithms in literature [49]. Later, the classification model can be utilized in the detection of the defective classes in the unseen test set.

After applying the feature selection algorithm *k* out of *m* metrics are kept, which are strongly related to the error-proneness of the classes. As a result we have training set $T_{n\,(k+1)}$ as a matrix with *n* rows and *k+1* columns.

One of the most important factors affecting the success of the learning-based method is the selection of the classification algorithm. For this purpose, J48 decision-tree learner is used that is an implementation of C4.5 algorithm [46] in Weka (Waikato Environment for Knowledge Analysis) open source tool [75]. J48 is widely preferred in literature [26] for ease of interpretation of the output and its robustness.

The dataset obtained in the training phase is applied to J48 algorithm. First, all metrics and "defective/healthy" class labels are provided to J48 algorithm. The algorithm iteratively selects *k* out of *m* metrics in the set, which are strongly related to the error-proneness of the classes. As a result, a training set $T_{n\,(k+1)}$ is obtained as a matrix with *n* rows and *k+1* columns. Using this training set the J48 algorithm constructs the classifier in the form of a decision tree.

In the classification model, internal nodes of the decision tree split into branches based on detection rules of class metrics. Leaf nodes indicate the tag of the class, namely "1" represents "defective" class and "0" represents "healthy" classes.

In this thesis, the decision tree model is applied to a test release of the project that includes unseen data and the defective classes are identified. In order to evaluate success of the model, the following releases of the test release is examined and the EFs of the classes are observed that our model classified as defective or healthy.

5. EMPIRICAL STUDY AND RESULTS

In this section, the experimental environments of the proposed methods are explained and the results of the application to industrial software projects are given. In order to reveal the benefits of proposed defect prediction approach more clearly, experimental results are analyzed with development teams. Some experiments are repeated with different parameters or different algorithms to analyze their effects on achievement and performance.

5.1 The Experimental Environment

The software projects used in the experiments are developed in Java development environment. The first project (Project A) is being developed for six years and includes 810 software classes. The second project (Project B) is being developed for four years and includes 790 software classes. To create a proper data set and evaluate the obtained results, two different intervals of the releases in the Project A, where EFs of classes are relatively high are selected. For training data, the interval with x=46 releases are used and for testing an interval with y=49 releases are observed in Project A. In Project B, 49 releases are observed for evaluation. To analyze our method deeply in this study, a lot of releases are used. However, in a real application of our proposed method, fewer releases (about x=10) would be sufficient for training.

Object-oriented design and code metric values are collected by using iPlasma [61] and ckjm metric tool [62].

iPlasma is used for quality assessment of object-oriented software projects starting from model extraction up to high-level metrics analysis and even code duplication detection. It directly receives C++ or Java source code and provides metric values of object-oriented classes [10]. It is used for analysis of large software applications including telecom systems.

Ckjm metric tool is used for extracting Chidamber and Kemerer (C&K) Java metrics and many other metrics. The program processes the bytecode of compiled Java files

and calculates C&K object-oriented metrics and two non-object-oriented metrics [76]. Java class files are provided as an input to command line in order to extract metric values with the full name of each class.

5.2 Determining the Threshold Values and Creating the Training Set

To tag the classes in the training set as given in **(4.6)**, the thresholds values of ChC and EF need to be determined. 46 training releases of the Project A are examined with the help of the development team.

5.2.1 Correlation-based training set experiment

At the beginning, the correlations between metric values and the class labels are calculated by using Pearson's correlation. Pearson's correlation results between metric values and class labels are provided in Table 5.1.

Table 5.1 : Project A Pearson correlation.

Metric Name	Correlation with Label
RFC	0,5764
CBO	0,5390
FANOUT	0,5387
WMC	0,5227
LOC	0,5034
FDP	0,4741
ATFD	0,4253
NOM	0,3765
LCOM	0,3349
AMW	0,3284
NPM	0,2278
NOA	0,2159
NAS	0,1971
DIT	0,1602
CM	0,1367
NOPA	0,1054
WOC	0,0991
NOAM	0,0964
NOC	0,0936
CC	0,0705
Ca	0,0233
HIT	-0,0902
BUR	-0,1782

The first 15 metrics with the highest correlation values are selected for constructing the training set attributes, whereas others are eliminated. After this investigation,

some experiments are conducted with different threshold values for both t_1 and t_2, such as t_1=2, t_1=3, t_1=4, t_2=0.2, t_2=0.25 and t_2=0.3. After examining the releases of the project the classes are labeled in the training releases as follows.

- Classes with ChC $\geq t_1$ and EF $\geq t_2$ are labeled as defective.
- As the RFC metric gives the highest correlation value with the defectiveness of the classes, classes having small ChC (ChC $< t_1$) or low EF (EF $< t_2$) but high RFC metric value ($RFC_c \geq AVG(RFC_{dc})$) are also labeled as defective. $AVG(RFC_{dc})$ shows the average RFC metric value of classes identified as defective using the first rule of **(4.6)**.
- The remaining classes with low ChC, EF and RFC metric value are considered as healthy. For simplicity 200 out of all classes are randomly selected and put into the data set. In a project with fewer classes can all healthy classes be included in the data set.

The logical expressions, which are used to categorize classes, are shown in Table 5.2.

Table 5.2 : Rules for categorizing classes in training set.

Expression	Quantity	Classification Label
ChC $\geq$ 5 and EF $\geq$ 0.3	44	Defective
(ChC < 5 or EF < 0.3) and $RFC_c \geq AVG(RFC_{dc})$	12	Defective
(ChC < 5 or EF < 0.3) and $RFC_c < AVG(RFC_{dc})$	200	Healthy

5.2.1.1 Results of correlation-based training set experiments

There are 256 classes in the data set with 15 object-oriented and code metrics. The constructed training set is given as an input to Naïve Bayes learner, logistic regression and J48 decision tree algorithms, respectively. The models are applied on a test release of the Project A that was not included in the training phase. Naive Bayes-based model labeled 87 of 807 classes of the project as defective. The method identified also 27 classes in Project A as defective, which neither generated errors nor were modified in the observation releases (ErrC=0, ChC=0). The model created by using logistic regression method labeled 49 of 807 classes of the project as defective. The method also detected 9 classes in Project A as defective with ErrC=0 and ChC=0.

J48 model labeled 42 classes of the project as defective. The method identified also 8 classes in Project A as defective with ErrC=0 and ChC=0. These classes are observed in 49 other releases following the Project A test release. The predictive performance of models are given in Table 5.3.

Table 5.3 : Predictive success of used algorithms.

Algorithm	# of Defective Classes (Predicted)	# of 0/0 EF Classes (Predicted)	Success Rate (%)
Naïve Bayes	87	27	86%
J48	42	8	76%
Logistic Reg.	49	9	71%

The model created with Naïve Bayes algorithm was predicted the majority of the most defective classes, on the other hand the false positive detection rate of the method was relatively high. The results obtained with logistic regression method is relatively low when compared to successive performance of other algorithms. Best results for this experiment were obtained by using J48 decision tree algorithm. Naive Bayes, logistic regression and J48 method results are presented in terms of ChCs and EFs of detected classes in Table 5.4, Table 5.5 and Table 5.6 for Project A, respectively.

Table 5.4 : Experimental results of Naïve Bayes algorithm for Project A.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
8 / 11 = 0.73	1	1
7 / 11 = 0.64	1	1
6 / 12 = 0.5	1	1
6 / 10 = 0.6	1	1
6 / 7 = 0.86	1	1
5 / 11 = 0.45	1	1
5 / 10 = 0.5	1	1
5 / 9 = 0.56	1	0
5 / 8 = 0.63	1	1
5 / 7 = 0.71	1	1
4 / 10 = 0.4	1	1
4 / 6 = 0.67	2	0
4 / 5 = 0.8	1	1
3 / 7 = 0.43	2	2
3 / 6 = 0.5	1	1
3 / 5 = 0.6	2	2
2 / 5 = 0.4	2	2

Table 5.5 : Experimental results of Logistic Regression algorithm for Project A.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
8 / 11 = 0.73	1	1
7 / 11 = 0.64	1	1
6 / 12 = 0.5	1	1
6 / 10 = 0.6	1	1
6 / 7 = 0.86	1	1
5 / 11 = 0.45	1	1
5 / 10 = 0.5	1	1
5 / 9 = 0.56	1	0
5 / 8 = 0.63	1	1
5 / 7 = 0.71	1	1
4 / 10 = 0.4	1	1
4 / 6 = 0.67	2	0
4 / 5 = 0.8	1	1
3 / 7 = 0.43	2	1
3 / 6 = 0.5	1	1
3 / 5 = 0.6	2	2
2 / 5 = 0.4	2	0

Table 5.6 : Experimental results of J48 algorithm for Project A.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
8 / 11 = 0.73	1	1
7 / 11 = 0.64	1	1
6 / 12 = 0.5	1	1
6 / 10 = 0.6	1	1
6 / 7 = 0.86	1	1
5 / 11 = 0.45	1	1
5 / 10 = 0.5	1	1
5 / 9 = 0.56	1	0
5 / 8 = 0.63	1	1
5 / 7 = 0.71	1	1
4 / 10 = 0.4	1	1
4 / 6 = 0.67	2	0
4 / 5 = 0.8	1	1
3 / 7 = 0.43	2	1
3 / 6 = 0.5	1	1
3 / 5 = 0.6	2	2
2 / 5 = 0.4	2	1

These tables show the following information about the classes: ErrC, ChC, EF, total number of the classes having the same ErrC/ChC in the observation releases and the number of the correctly detected classes.

Initially, all 15 metric values and “defective/healthy” class information were given to J48 classifier algorithm. For the given training set, the J48 algorithm iteratively selected 4 out of 15 metrics which were most significant for classification. Selected metrics were: RFC, CBO, CM and DIT. After the feature selection a training set was obtained that includes 256 classes which are represented with 4 attributes and one tag. Using this training set the J48 algorithm constructed a decision tree model as shown in Figure 5.1.

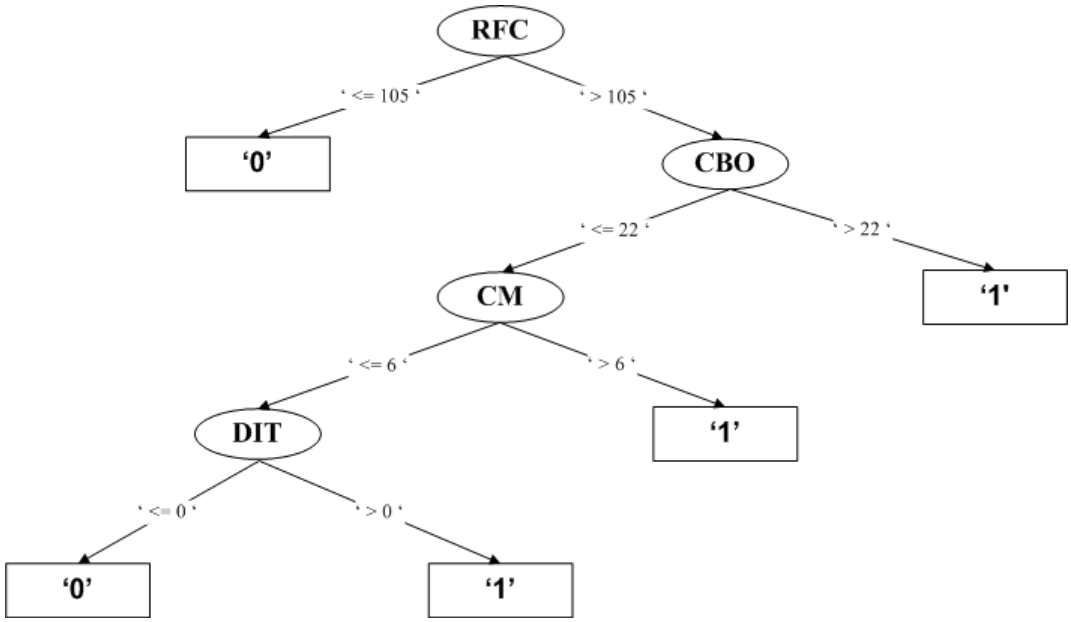

Figure 5.1 : Decision tree for correlation-based training set

Same experiment is also conducted on the Project B by using J48 algorithm. The method identified 27 defective classes where the total number of the classes was 789. Table 5.7 shows the experimental results conducted on Project B.

Table 5.7 : Experimental results of J48 algorithm for Project B.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
10 / 10 = 1	1	1
9 / 11 = 0.82	1	1
8 / 9 = 0.89	1	1
7 / 7 = 1	1	1
6 / 8 = 0.75	1	1
6 / 7 = 0.86	1	0
5 / 6 = 0.83	1	1
5 / 5 = 1	1	1
4 / 5 = 0.8	2	0
3 / 6 = 0.5	1	1
3 / 5 = 0.6	1	1

Results show that the method succeeds in finding frequently changing defective classes with relatively high EFs. The decision tree model predicted 76% of the most defective classes (i.e. 16 out of 21 defective classes) with the highest EFs of Project A and 75% of (i.e. 9 out of 12 defective classes) Project B correctly.

5.2.2 Complexity-based training set experiment

Based on previous experiments, the J48 decision tree model provided the best results for predicting the most defective classes. To enhance the overall prediction success of the proposed model, the classes are sorted according to their ChCs. After sorting based on ChCs, it is realized that structural defective classes tend to change at least 5 times and their EFs are higher than 0.25. The ErrC and ChCs of 20 classes with highest ChCs and EFs are given in Table 5.8.

Table 5.8 : Classes with high change counts and error frequencies.

Error Frequencies		
Change Count	Error Count	Error Frequency
18	12	0.66
17	12	0.7
14	9	0.64
13	10	0.76
11	5	0.45
10	4	0.4
10	6	0.6
9	5	0.55
9	4	0.44
9	6	0.66
9	7	0.77
8	4	0.5
8	5	0.62
8	3	0.37
8	2	0.25
7	5	0.71
7	4	0.57
6	3	0.5
6	4	0.66
6	5	0.83

After this investigation two threshold values are assigned for class labeling, namely t_1=5 and t_2=0.25. Some experiments with other threshold values such as t_1=4, t_2=0.25 and t_2=0.2 are also conducted, but the highest detection success results are obtained if the training set is constructed with t_1=5 and t_2=0.25. The metric values of classes

with high EFs are examined and it is been realized that the EFs of these classes were highly correlated to their complexity (WMC).

After examining the releases of the project, the classes are labeled in the training releases as follows.

- Classes with ChC ≥ 5 and EF ≥ 0.25 are labeled as defective. There are 45 classes in this group.
- Classes having small ChC (ChC < 5) or low EF (EF < 0.25) but high complexity (WMC ≥ AVG*1.5) are also labeled as defective. There are 2 classes in this group.
- The remaining 763 classes with low ChC, EF and complexity are considered healthy. For simplicity, 200 out of 763 classes are selected randomly and put into the data set. In a project with fewer classes can all healthy classes be included in the data set.

The logical expressions which are used to categorize classes are shown in Table 5.9. AVG (WMC_{dc}) shows the average complexity value of classes identified as defective using the first rule (ChC ≥ 5 and EF ≥ 0. 25).

Table 5.9 : Rules for categorizing classes in training set.

Expression	Quantity	Classification Label
ChC ≥ 5 and EF ≥ 0. 25	45	Defective
(ChC < 5 or EF < 0.25) and $WMC_c \geq AVG(WMC_{dc})*1.5$	2	Defective
(ChC < 5 or EF < 0.25) and $WMC_c < AVG(WMC_{dc})*1.5$	200	Healthy

There are 247 classes in the data set with 23 object-oriented and code metrics. Initially, all metric values and “defective/healthy” class information are given to J48 classifier algorithm. The algorithm iteratively selected 5 out of 23 metrics which are most significant for classification. Selected metrics are: CBO, LCOM, WOC, HIT and NOM. After the feature selection, a training set is obtained that includes 247 classes which are represented with 5 attributes and one class tag. Using this training set the J48 algorithm constructed a decision tree model as shown in Figure 5.2.

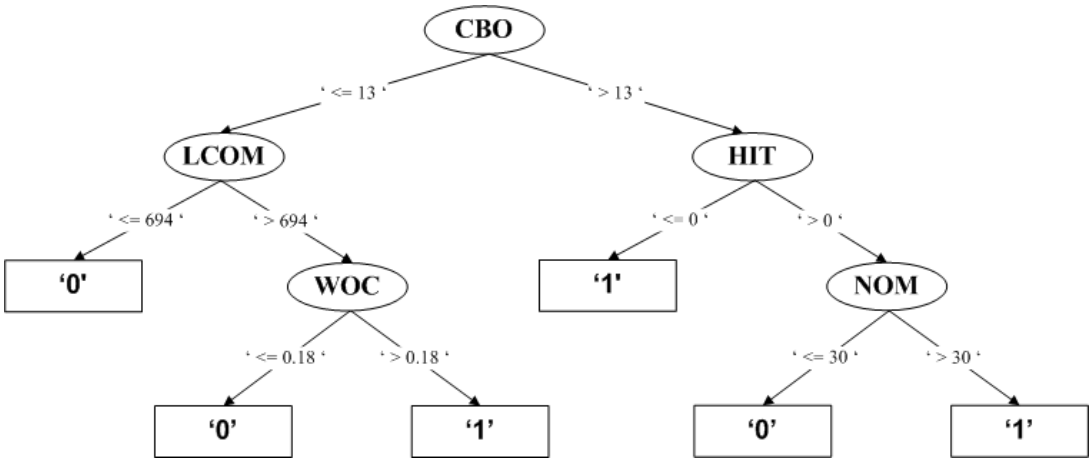

Figure 5.2 : The decision tree model for complexity-based training set.

5.2.2.1 Results of complexity-based training set experiment

The proposed decision tree model is applied on a test release of the Project A that was not included in the training phase. The proposed method labeled 53 of 807 classes of the project as defective. Then, these classes are observed in 49 other releases following the test release. The similar experiment is also conducted on the Project B. In this case, the proposed method identified 41 defective classes where the total number of the classes was 789. The experimental results are presented in terms of ChCs and EFs of detected classes in Table 5.10 and Table 5.11 for Project A and Project B, respectively.

Table 5.10 : Experimental results of J48 algorithm for Project A.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
8 / 11 = 0.73	1	1
7 / 11 = 0.64	1	0
6 / 12 = 0.5	1	1
6 / 10 = 0.6	1	1
6 / 7 = 0.86	1	1
5 / 11 = 0.45	1	1
5 / 10 = 0.5	1	1
5 / 9 = 0.56	1	0
5 / 8 = 0.63	1	1
5 / 7 = 0.71	1	1
4 / 10 = 0.4	1	1
4 / 6 = 0.67	2	0
4 / 5 = 0.8	1	1
3 / 7 = 0.43	2	2
3 / 6 = 0.5	1	1
3 / 5 = 0.6	2	2
2 / 5 = 0.4	2	2

Table 5.11 : Experimental results of J48 algorithm for Project B.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
10 / 10 = 1	1	1
9 / 11 = 0.82	1	1
8 / 9 = 0.89	1	1
7 / 7 = 1	1	1
6 / 8 = 0.75	1	1
6 / 7 = 0.86	1	0
5 / 6 = 0.83	1	1
5 / 5 = 1	1	1
4 / 5 = 0.8	2	2
3 / 6 = 0.5	1	0
3 / 5 = 0.6	1	1

These tables show the following information about the classes: ErrC, ChC, EF, total number of the classes having the same ErrC/ChC in the observation releases and the number of the correctly detected classes.

Results show that the proposed method succeeds in finding frequently changing defective classes with relatively high EFs. The decision tree model predicted 81% of the most defective classes (i.e. 17 out of 21 defective classes) with the highest EFs of Project A and 83% of (i.e. 10 out of 12 defective classes) Project B correctly.

The proposed model identified also 18 classes in Project A and 7 classes in Project B as defective, which neither generated errors nor were modified in the observation releases (ErrC=0, ChC=0). These classes are analyzed with the mentor groups of the projects and it is been confirmed that 13 of the classes in Project A and 4 classes in B have design defects and need refactoring. For example, some classes have long and complex methods; whereas others are highly coupled too many other classes and the business model of these classes are handling complex tasks. Even these classes need refactoring they remain undetected for a long time in the projects lifecycle as they stayed unchanged. This is another advantage of the proposed approach that it can detect classes with such insidious defects that do not appear until the class needs modifications.

5.3 Threads to Validity

The first threat to the validity of the proposed study is that a single project is used to obtain the attributes of classes in constructing the data set. However, the reference project used in the study is a large and long-standing commercial software system. To create a reliable training set, the classes were analyzed and the results were verified with the help of the development team. In a further study, several different projects may be used in the learning phase and evaluate their effects on the results.

Another threat could be caused by the selected threshold values of ChC and EF. These values are obtained by analyzing a large project developed by Ericsson Turkey and saw that they were also valid for another project of the same company. The proposed model will be tested also on other industrial and open-source projects. Besides, if users of the proposed method observe different threshold values for their projects they can easily create a new data set with their own propoer threshold values.

6. CONCLUSIONS AND FUTURE WORK

The aim of this work is to detect poorly designed classes, to enable testers focus on them and/or the developers refactor them. Poorly designed classes typically become error-prone when modified. In this thesis, a decision tree model is proposed to predict error-prone classes that have design defects in the software projects. In order to label a class as defective or not, ChC and EF values are calculated. ChC being the total number of changes made to a class, and EF (or error frequency) being the ratio of the number of changes made to fix a bug over ChC. Classes with high ChC and EF are deemed error-prone or defective.

The 23 features of a class used to predict the outcome of the model were based on properties of the given class such as size, coupling, cohesion, complexity, depth of inheritance. The model was trained with 200 healthy classes and 47 defective classes. The classifier selected only 5 out of the 23 features, namely, coupling between objects, lack of cohesion in methods, height in inheritance tree of a class, weight of a class and number of methods.

The trained decision tree can be used to detect defective classes in future releases of the same project or in other projects with similar characteristics. Two real-world telecommunication software projects, namely Project A and Project B are used to evaluate the performance and practical usability of the proposed approach. The model is trained with dataset from Project A and tested on both Project A and B.

The results of the proposed approach is promising since the model succeeded in finding frequently changing defective classes with relatively high EFs. The model correctly predicted 81% of the most defective classes with the highest EFs of Project A and 83% of Project B. The model also identified 18 classes in Project A and 7 classes in Project B as defective, which neither generated errors nor were modified. After further analysis, it is confirmed that 13 of the classes in Project A and 4 classes in B have design defects and need refactoring.

The structurally defective classes are detected and the experimental results are shared with the mentor groups of the projects. The teams also approved the accuracy of the method and started refactoring activities on the detected classes. The findings are also shared with the testers to make them test the detected classes deeply and primarily.

The proposed approach in this thesis ensures the early detection of defect-prone classes and provides benefits to the developers and testers. Firstly, it helps testers to focus on faulty modules of software, thus it saves significant proportion of testing time; secondly developers can refactor classes to correct their design defects and reduce the maintenance cost in further releases.

REFERENCES

[1] **Shull, F., Basili, V., Boehm, B., Brown, W. A., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M.** (2002). What We Have Learned About Fighting Defects. *Proceedings of the 8th International Symposium on Software Metrics*, Ottowa, Canada.

[2] **ISO/IEC 25000.** (2005). Software Engineering --- Software Product Quality Requirements and Evaluation (SQuaRE) --- Guide to SQuaRE.

[3] **Falessi, D., Cantone, G., Kazman, R., and Kruchten, P.** (2011). Decision-making Techniques for Software Architecture Design: A Comparative Survey. *ACM Computing Surveys.* Vol. ***43***, pp. 1-28.

[4] **Lincke, R., Lundberg, Y. and Löwe, W.** (2008). Comparing Software Metrics Tools. *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*, Seattle, WA, USA, 20-24 July.

[5] **Url-1** <*http://www.softwaretestingclass.com/top-10-reasons-why-there-are-bugs-defects-in-software*>, date retrieved 14.03.2015.

[6] **Url-2** <*http://www.klocwork.com/resources/research/improving-software-by-reducing-coding-defects*>, date retrieved 14.03.2015.

[7] **Buzluca, F.** (2014). Object-Oriented Programming in C++ (Lecture Notes). Date retrieved: 21.03.2015, address: http://www.ninova.itu.edu.tr/en/courses/faculty-of-computer-and-informatics/21/blg-252e/ekkaynaklar?g116559.

[8] **Royce, W. W.** (1987). Managing the Development of Large Software Systems: Concepts and Techniques. Originally published in *Proceedings WESCON*, Aug, 1970. Currently available in *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California, USA.

[9] **Fowler, M., Beck, K.** (2000). Bad Smells in Code. Refactoring: Improving the Design of Exiting Code, pp (75-88). Addison-Wesley.

[10] **Lanza, M., Marinescu, R.** (2006). Object-Oriented Metrics in Practice. pp 73-163. Springer.

[11] **Url-3** <*http://lombardhill.com/articles/software-reuse-101-what-is-software-reuse*>, date retrieved 28.03.2015.

[12] **ISO/IEC 25010.** (2011). Systems and Software Engineering --- Systems and Software Quality Requirements and Evaluation (SQuaRE) --- System and Software Quality Models.

[13] **Lindvall, M., Tesoriero, R., Costa P.** (2002). Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture. *In*

Proceedings of International Symposium on Software Metrics (METRICS '02), (pp. 77-86). Ottawa, Ontario, Canada, June.

[14] **Ackermann, C., Lindvall, M., Dennis, G.** (2009). Redesign for Flexibility and Maintainability: A Case Study. *13th European Conference on Software Maintenance and Reengineering (CSMR '09),* (pp. 259-262). Maryland, March 24-27.

[15] **Buzluca, F.** (2012). Object-Oriented Modeling and Design (Lecture Notes). Date retrieved 21.03.2015, address: http://ninova.itu.edu.tr/tr/dersler/bilgisayar-bilisim-fakultesi/2097/blg-468e/ekkaynaklar?g197953.

[16] **ISO/IEC 9126-1.** (2001). Information Technology --- Software Product Quality --- Part 1 Quality Model.

[17] **Tosun, A., Bener, A., Kale, R.** (2011). AI-Based Software Defect Predictors: Applications and Benefits in a Case Study. *AI Magazine.* Vol. ***32***, no. 2, pp. 57-68.

[18] **Kolodgy, J. C.** (2011). The Case for Building in Web Application Security from the Start. *White Paper Sponsored by IBM.* Date retrieved: 30.03.2015, address: ftp://ftp.software.ibm.com/software/pdf/dk/rational/everywhere/2_wp_the_case_for_building_web_app.pdf.

[19] **Url-4** <*http://promise.site.uottawa.ca/SERepository/datasets-page.html*>, date retrieved 30.03.2015.

[20] **Erdemir, U., Tekin, U., Buzluca, F.** (2008). Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi - Object Oriented Software Metrics and Software Quality. *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu.* Istanbul, Turkey.

[21] **Booch, G.** (1994). Object-Oriented Analysis and Design with Applications. 2nd Edition. Addison-Wesley. Santa Clara, California, pp. 277.

[22] **Chidamber, K., Kemerer, C.** (1999). A Metrics Suite for Object-Oriented Design. *IEEE Trans. Software Engineering.* Vol. ***20***, no. 5, pp. 476-493.

[23] **Brito e Abreu, F., Pereira, G., Soursa, P.** (2000). Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. *Proc. Euromicro Conf. Software Maintenance and Reengineering.* pp. 13-22.

[24] **Bansiya, J., Davis, C.** (2002). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transaction on Software Engineering***.** Vol. ***28,*** no. 1, pp. 4-17.

[25] **Marinescu, R.** (2004). Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. *20th IEEE International Conference on Software Maintenance (ICSM).* September 11-14.

[26] **Koru, A. G., Liu, H.** (2005). Building Effective Defect Prediction Models in Practice. *IEEE Software*. Vol. ***22***, no. 6, pp. 23-29.

[27] **Malhotra, R.** (2015). A Systematic Review of Machine Learning Techniques for Software Fault Prediction. Applied Software Computing. Vol. ***27***, pp. 504–518.

[28] **El Emam, K., Benlarbi, S., Goel, N., Rai, S.** (2001). The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Trans. Software Engineering*. Vol. ***27***, no. 7, pp. 630-650.

[29] **Olague, H. M., Gholston, S., Quattlebaum, S.** (2007). Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*. Vol ***33***, no. 6, pp. 402–419.

[30] **Travassos, H. G., Shull, F., Fredericks, M., Basili, V. R.** (1999). Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality. *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Vol. ***34***, no. 10, pp. 47-56.

[31] **Moha, N., Guéhéneuc, Y.-G., Duchien, L., 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.

[32] **Catal, C., Diri, B.** (2009). A Systematic Review of Software Fault Prediction Studies. *Expert Systems with Applications*. Vol. ***36***, no. 4, pp. 7346-7354.

[33] **Munro, M. J.** (2005). Product Metrics for Automatic Identification of Bad Smell Design Problems in Java Source-Code. *Proeedings of the 11th International Software Metrics Symposium*. September 19-22.

[34] **Dhambri, K., Sahraoui, H., Poulin, P.** (2008). Visual Detection of Design Anomalies. *Proceedings of the 12th European Conference Software Maintenance and Reengineering*. pp. 279-283.

[35] **Url-5** <*http://findbugs.sourceforge.net/*>, date retrieved 04.04.2015.

[36] **Url-6** <*https://www.coverity.com/*>, date retrieved 04.04.2015.

[37] **Url-7** <*http://pmd.sourceforge.net/*>, date retrieved 04.04.2015.

[38] **Url-8** <*http://metrics.sourceforge.net/*>, date retrieved 04.04.2015.

[39] **Url-9** <*http://jdeodorant.com/*>, date retrieved 04.04.2015.

[40] **Hamid, A., Ilyas, M., Hummayun, M., Nawaz, A.** (2013). A Comparative Study on Code Smell Detection Tools. *International Journal of Advanced Science and Technology*. Vol. ***60***, pp. 25-32.

[41] **Marinescu, R., Ganea, G., Verebi, I.** (2010). inCode: Continuous Quality Assessment and Improvement. *14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*. pp. 274-275.

[42] **Hryszko, J., Madeyski, L.** (2015). Bottlenecks in Software Defect Prediction Implementation in Industrial Projects. *Foundations of Computing and Decision Sciences*. Vol. ***40***, no. 1, pp. 17-33.

[43] **Tosun, A., Bener, A., Turhan, B., Menzies, T.** (2010). Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry. *Information and Software Technology*. Vol. ***52***, pp. 1242–1257.

[44] **Ostrand, T., Weyuker, E., Bell, R.** (2005). Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*. Vol. ***31***, no. 4, pp. 340–355.

[45] **Knab, P., Pinzger, M., Bernstein, A.** (2006). Predicting Defect Densities in Source Code Files with Decision Tree Learners. *Proceedings of the 2006 International Workshop on Mining Software Repositories*. Shanghai, China. May 22-23.

[46] **Url-10** <*http://en.wikipedia.org/wiki/C4.5_algorithm*>, date retrieved 21.01.2015.

[47] **Cavezza, G. D., Pietrantuono, R., Russo, S.** (2015). Performance of Defect Prediction in Rapidly Evolving Software. *Accepted Paper in 3rd International Workshop on Release Engineering (RELENG '15)*. Frenze, Italy. May 16-24.

[48] **Tosun, A., Bener, A.** (2009). Reducing False Alarms in Software Defect Prediction by Decision Threshold Optimization. *In Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*. pp. 477–480.

[49] **Alpaydin, E.** (2004). Introduction to Machine Learning. *MIT Press*.

[50] **Eski, S., Buzluca, F.** (2009). An Empirical Study on Object-Oriented Metrics and Software Evolution in order to Reduce Testing Costs by Predicting Change-prone Classes. *In 2011 IEEE Fourth International Conference on Software Testing, V&V Workshops*, pp. 566-571.

[51] **Kim, S., Whitehead Jr., E. J., Zhang, Y.** (2008). Classifying Software Changes: Clean or Buggy?. *IEEE Transactions Software Engineering*. Vol. ***34***, no. 2, pp. 181-196.

[52] **Shivaji, S., Whitehead Jr., E. J., Akella, R., Kim, S.** (2013). Reducing Features to Improve Code Change-based Bug Prediction. *IEEE Transactions on Software Engineering*. Vol. ***39***, no. 4, pp. 552-569.

[53] **Agarwal, S., Tomar, D.** (2014). A Feature Selection Based Model for Software Defect Prediction. *International Journal of Advanced Science and Technology*. Vol. ***65***, pp. 39-58.

[54] **Jayadeva, R., Khemchandani, R., Chandra, S.** (2007). Twin Support Vector Machine for Pattern Classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. ***29***, no. 5, pp. 905-910.

[55] **Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J.** (2011). A General Software Defect-proneness Prediction Framework. *IEEE Transactions on Software Engineering*. Vol. ***37***, no. 3, pp. 356-370.

[56] **Pelayo, L., Dick, S.** (2007). Applying Novel Resampling Strategies to Software Defect Prediction. *In Annual Meeting of North American Fuzzy Information Processing Society*. pp. 69–72. June 24-27.

[57] **Menzies, T., Greenwald, J., Frank, A.** (2007). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*. Vol. ***33***, no. 1, pp. 2-13.

[58] **Chapman, M., Callis, P., Jackson, W.** (2004). Metrics Data Program. *Technical Report NASA IV and V Facility*.

[59] **Gayatri, N., Nickolas, S., Reddy, V.** (2010). Feature Selection Using Decision Tree Induction in Class Level Metrics Dataset for Software Defect Predictions. *In Proceedings of the World Congress on Engineering and Computer Science*. Vol. ***1***, pp. 124-129.

[60] **Url-11** *<http://en.wikipedia.org/wiki/Concurrent_Versions_System>*, date retrieved 21.01.2015.

[61] **Marinescu, C., Marinescu, R., Mihancea, P., Ratiu, D., Wettel, R.** (2005). iPlasma:An Integrated Platform for Quality Assessment of Object-Oriented Design. *In Proceedings of the 21st IEEE International Conference on Software Maintenance*. pp. 77-80.

[62] **Spinellis, D.** (2005). Tool Writing: A Forgotten Art?. *IEEE Software*. Vol. ***22***, no. 4, pp. 9-11.

[63] **Okutan, A., Yildiz, O. T.** (2012). Software Defect Prediction Using Bayesian Networks. *Empirical Software Engineering*. pp. 1-28.

[64] **Turhan, B., Bener, A.** (2009). Analysis of Naive Bayes' Assumptions on Software Fault Data: An Empirical Study. *Data and Knowledge Engineering*. Vol ***68***, pp. 278-290.

[65] **Dejaeger, K., Verbraken, T., Baesens, B.** (2013). Toward Comprehensible Software Fault Prediction Models Using Bayesian Network Classifiers. *IEEE Transactions on Software Engineering*. Vol. ***39***, no. 2, pp. 237-257.

[66] **Catal, C., Sevim, U., Diri, B.** (2011). Practical Development of an Eclipse-based Software Fault Prediction Tool Using Naive Bayes Algorithm. *Expert Systems with Applications*. Vol. ***38***, pp. 2347-2353.

[67] **Emam, K. El, Benlarbi, S., Goel, N., Rai, S. N.** (2001). The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering*. Vol. ***27***, no. 7, pp. 630-650.

[68] **Olague, H. M., Etzkorn, L. H., Gholston, S., Quattlebaum, S.** (2007). Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*. Vol. ***33***, no. 6, pp. 402-419.

[69] **Schneidewind, N. F.** (2001). Investigation of Logistic Regression as a Discriminant of Software Quality. *7th International Software Metrics Symposium, METRICS '01*. pp. 328-337.

[70] **Denaro, G., Pezze, M.** (2003). Towards Industrally Relevant Fault-Proneness Models. *International Journal of Software Engineering and Knowledge Engineering*. Vol. ***13***, no. 4, pp. 395-417.

[71] **Url-12** <*http://en.wikipedia.org/wiki/Logistic_regression*>, date retrieved 09.04.2015.

[72] **Url-13** <*http://homes.cs.washington.edu/~shapiro/EE596/notes/InfoGain.pdf*>, date retrieved 11.04.2015.

[73] **Koru, A. G., Liu, H.** (2005). An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures. *ACM SIGSOFT Software Engineering Notes*. Vol. ***30***, pp. 1-5.

[74] **Li, M., Zhang, R., Wu, R., Zhou, Z.** (2012). Sample-Based Software Defect Prediction with Active and Semi-Supervised Learning. *Automated Software Engineering*. Vol. ***19***, no. 2, pp. 201-230.

[75] **Url-14** <*http://www.cs.waikato.ac.nz/~ml/weka/*>, date retrieved 14.04.2015.

[76] **Url-15** <*http://www.spinellis.gr/sw/ckjm/*>, date retrieved 15.04.2015.

CURRICULUM VITAE

Name Surname: Çağıl BİRAY

Place and Date of Birth: IZMIR / 27.09.1987

E-Mail: cb.biray@gmail.com

B.Sc.: Eskisehir Osmangazi University, Computer Engineering

Professional Experience: Çağıl BİRAY has been working at Ericsson R&D Turkey as an Experienced Integration Engineer since 2011.

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- **C. Biray**, F. Buzluca, "A Learning-Based Method for Detecting Defective Classes in Object-oriented Systems", *10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, April 17, Graz, Austria, 2015.