INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Detection of Faulty Components in Object-Oriented Systems using Design Metrics and a Machine Learning Algorithm

Stefan V. Ikonomovski

School of Computer Science
McGill University, Montréal

November, 1998

A THESIS SUBMITED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

To Montréal,
a city with a Lovely Sky,
and a Lust for Life.

À Montréal,
une ville au superbe ciel,
et où il fait bon vivre.

Abstract

Object-Oriented (OO) technology claims faster development and higher quality of software than the procedural paradigm. The quality of the product is the single most important reason that determines its acceptance and success. The basic project management problem is "delivery of a product with targeted quality, within the budget, and on schedule". We propose a state-of-the-art approach that gets closer to the solution by improving the software development process used. An important objective in all software development is to ensure that the delivered product is as fault-free as possible. We proposed three hypotheses that relate the OO design properties—inheritance, cohesion, and coupling—and the fault-proneness as software's quality indicator. We built classification models that predict which components are likely to be faulty, based on an appropriate suite of OO design measures. The models represent empirical evidence that the aforementioned relationships exist. We used the C4.5 machine learning algorithm as a predictive modeling technique, because it is robust, reliable, and allows intelligible interpretation of the results. We defined three new measures that quantify the specific contribution of each of the metrics selected by the model(s), and also provide a deeper insight into the design structure of the product. We evaluated the quality of the predictive models using an objective set of standards. The models built have high quality.

Résumé

La technologie Orientée Objet (OO) apporte une rapidité de développement et une qualité du logiciel supérieures à ce qu'elles étaient dans le paradigme procédural. Aujourd'hui, à fonctionnalités égales, la qualité d'un produit logiciel est le critère majeur pour son acceptation et son succès. Le problème de base en gestion de projets est de «fournir un produit logiciel en respectant le budget, les échéances et un certain degré de qualité». Nous proposons une méthodologie pertinente afin d'atteindre ces objectifs, en se basant sur un objectif important sous-jacent : que le produit soit exempt d'erreurs.

Nous proposons trois hypothèses faisant le lien entre des propriétés conceptuelles OO que sont, héritage, cohésion et couplage – et la propension d'avoir des erreurs, considérée comme un indicateur de qualité du logiciel. Nous générons à partir des mesures des propriétés conceptuelles, des modèles de classification prédisant quels composants sont susceptibles d'êtres erronés. Ces modèles prouvent empiriquement la relation précitée. Pour ce faire, nous avons exploité C4.5, un algorithme d'apprentissage robuste, fiable et produisant des modèles prédictifs intelligibles. D'autre part, nous avons défini trois nouvelles mesures pour quantifier la contribution de chacune des métriques retenues par le modèle, donnant ainsi un éclairage nouveau sur la structure du produit. Enfin, nous avons évalué les modèles produits à partir de grandeurs standards reconnues. Il en ressort que nos modèles sont de grande qualité.

Acknowledgements

This thesis research is dedicated to the city of Montréal. With a reason, and without any reason at all. Montréal is the city where I was reborn. It helped me recuperate my energy after sleepless nights engaged with a research. Montréal is jazz, Place Des Arts, the World Film Festival, McGill University, Mont Royal, the old port, Basilique Notre-Dame... It is the smell of the coffee on the streets, the after-hours, and the fashion. But above all, it is the Spirit. Joie de vivre. A basic mixture of the two dominant cultures—French and English—flavored with people from allover the world. The planet Earth on a small scale. It brought to me people that do not have a "best before" date. I pay my deepest respect to it, and I love it. Montréal, c'est toi ma ville.

During my studies I have been financially supported by Ontario Student Aid Program loan. I was also, partially supported for my thesis research by a grant from CRIM.

Many people contributed in numerous ways to the quality of this work.

Chronologically, everything started with Professor Nazim Madhavji, with whom I decided to take my multiterm, special topic in computer science course, in the area of software process engineering. I learned a lot from his inquisitive and straight to the point approach, in those rare brainstorming occasions. I expect to continue the collaboration with him in the field of our mutual interest.

My deepest thanks go to Saïda Benlarbi. She defined the initial framework of the joint project between CRIM and IESE, and offered me the execution of its largest part. She also, provided me with a very pleasant working environment.

Hervé Marchal, the LALO team leader, helped me to understand the structure of LALO, and the design decisions behind it. He always found the time to satisfy my curiosity, and provided me with an access to the strategic parts of the system. He is a friend.

Jean-François Rizand helped me to collect the OO design measures by translating the LALO source code into its AT&T C++ compiler counterpart. By doing so, he reduced the time for that part of my thesis research and its cost, so I could concentrate mostly on the quality aspect.

Sébastien Émard from the technical support at CRIM, was the healer whenever I experienced severe technical problems.

Houari Sahraoui, senior researcher at CRIM, was a valuable source of information during the several brainstorming sessions that we had.

Yida Mao, my colleague and friend, worked with me on various aspects regarding software process engineering. We had long discussions that were mutually stimulating.

Hakim Lounis, my thesis co-supervisor at CRIM, provided me with a guidance that was both easygoing and very pleasant, throughout the milestones of the CRIM-IESE joint effort, and later, the parts of my thesis research. The discussions that we had, generated many ideas in my mind and made the completion of this thesis easier. He gave me prompt responses to my all inquires, and always had enough time for me. He also assisted me by translating the abstract. We plan to continue our fruitful collaboration.

People have always inspired me. Professor Gerald Ratzer, my thesis supervisor, whom I address as "dear Sir", is a very inspiring and truly deserving individual. He is a model mentor, a model teacher—which rounds up to a model professor—and a model person. A founding member of McGill's School of Computer Science, he is still full of curiosity and enthusiasm while trying to improve the educational process. He pays wonderful attention to people, and he was always prompt when reviewing my thesis writing. He does the things just for the sake of good things happening. I am fortunate to have him in my life.

Finally, I would like to acknowledge the support I have been receiving from my friends throughout my studies. They are my hidden source of energy and power. The Apostolov family—Aleksandar, Snezana, and Mark—always provided me with a pleasant after-hours surrounding. Maxim Andreev was a wonderful companion, as well as Ashkan Pourafzal—who was also, a very pleasant target of my management-related discussions and ideas. Carleen Joseph, Vesna Trajkov, Daniela Orso, and Senami Apithy, provided me with various aspects of support, that helped me to "survive" more easily. Dionis Hristov, Robin De Lorey, and Vida Dujmovic, were my roommates during the first, and the second year of my studies, and the latest while, respectively. We discussed various aspects of life. I would also like to acknowledge the support of the rest of my friends whom I did not mention here.

The School of Computer Science at McGill University was a valuable source of a world class research, as well as party time.

I would like to thank all of you mentioned, and not mentioned here. You have warmed my heart, and the heart never forgets.

At the end—which is just a new beginning, existentially speaking—I thank God for letting all of this happen.

Montréal, November 11, 1998

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The production of high quality, low cost software has become a fixation for many software development organizations [46]. Many of the current applications involve use of software in safety critical systems, where the high quality of the product is essential to the success of the mission [2], [34]. Furthermore, in a highly competitive market for software products with comparable feature sets and cost, within today's production environment where reducing the time to market has become prerequisite for survival, the quality of the product becomes the single most important reason that determines its acceptance and success [5], [6], [45].

Software engineering involves the study of the means of producing high quality software products with predictable costs and schedules [36]. The term was coined by Friedrich Bauer in 1967 at a NATO pre-conference meeting in Germany on issues in developing large-scale software systems [47, p.17]. According to his definition:

> *Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

1.1 The Big Picture

Pressman in [47, p.30] describes software engineering as a layered technology. Referring to Figure 1.1, any engineering approach must rest on an organizational commitment to quality. Total quality management fosters a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is *a quality focus.*

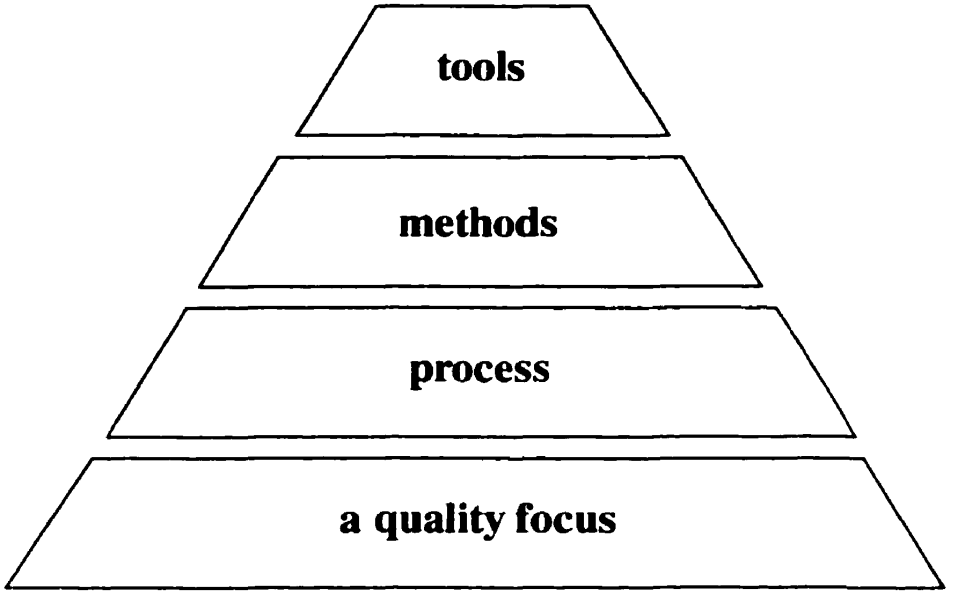


Figure 1.1. Software engineering layers

Process layer is the foundation for software engineering. It is the glue that holds technology layers together and enables rational and timely development of computer software. The key process areas form the basis for management control of software projects, and establish the context in which technical methods are applied, deliverables (models, documents, data reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical "how to's " for building software. Methods encompass a broad array of tasks that include requirement analysis, design, program construction, testing, and maintenance. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities, and other descriptive techniques.

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering (CASE), is established.

1.1.1 Software Development Process

The *software development process* comprises software engineering activities, including technical and managerial ones, that are carried out in the production of software. The scope of these activities includes determination and specification of system and software requirements; analysis and management of risk; software prototyping; design; implementation; verification and validation; software quality control and assurance; integration of components; documentation; management of software configurations and versions; management of data; evolution of software; project management; software evaluation; software contracting; software acquisition etc. [38].

Software engineering incorporates a development strategy that encompass the process, methods, and tools layers described in the previous section. This strategy is often referred to as a *process model* or a *software engineering paradigm* [47, p.31]. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. Several classes of process models have been widely discussed and debated, such as:

- Waterfall model
- Incremental model
- Evolutionary model
- Prototyping model
- Spiral model
- Concurrent model

Just as a human life cycle model (for example, infant, child, adolescent, adult, senior citizen) helps us understand the basic activities and characteristics of humans as they progress, the process model, also known as the *software life cycle model*, helps us understand the basic activities and characteristics of software as it develops [47, p.105]. It is a view of the activities that occur during software development. The discussion about the pros and cons regarding different classes of process models is beyond the scope of this work. However, in order to identify the most important phases of software development process we will describe briefly the waterfall model, Figure 1.2.

The waterfall model—sometimes called the *classic life cycle*, or linear sequential model—was originally documented for software in 1970 by Royce. It is the most basic of all life cycle models, and in fact serves as the building block for most other life cycle models. The waterfall view of software development is very simple; it says that software development can be thought of as a sequence of phases. Each phase has a set of well-defined goals, and the activities within any phase contribute to the satisfaction of that phase's goals or perhaps a subsequent phase's goals. The forward arrows show the normal flow of information among the phases; the backward arrows represent feedback. As Davis describes in [47, p.106], Royce did not say "only requirements activities may occur during the requirement phase" nor that "only design activities may occur during the design phase", and so forth.

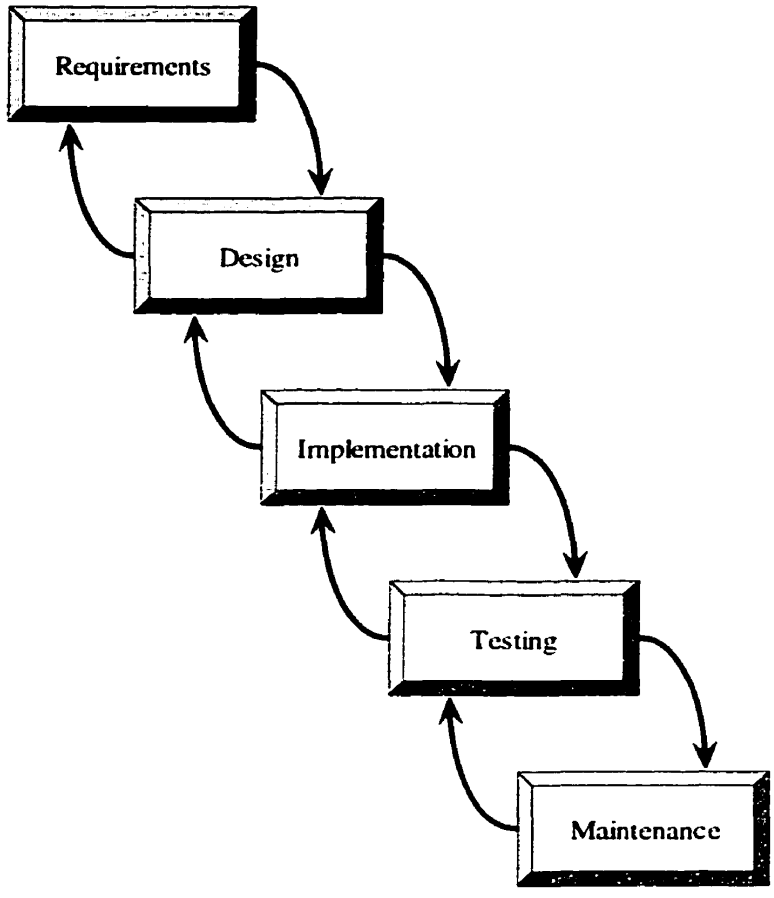


Figure 1.2. The waterfall model

There is nothing sacred about the names of the phases. The requirements phase has been called elicitation, system analysis, requirements analysis, or requirements specification; the preliminary design phase has been called high-level design, top-level design, software architectural definition, design specification or just *design*; the detailed design phase is often called program design, module design, lower level design, algorithmic design, or just *design*, and so on.

One of the most important contributions of the waterfall model is for management. It enables managers to track development progress, although on a very gross scale.

1.2 General Problem Statement

It has been widely recognized that an important component of *process improvement* is the ability to measure the process. *Software metrics* provide a quantitative means to control the software development process and the quality of software products [36]. Metrics can help address the most critical issues in software development and provide support for planning, predicting, monitoring, controlling, and evaluating the quality of both software processes and products [21]. Moreover, the development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different activities that take place during the software development. Software metrics are, thus, necessary to identify where the resources are needed, and are a crucial source of information for decision-making [8].

1.2.1 Motivation and Aim

Briand et al. point out in [20] that the production of better specifications and better designs reduces the need for extensive review, modification, and rewriting not only of code, but of specifications and designs as well. As a result, this allows the software organization to save time, cut production costs, and raise the final product's quality. Early availability of metrics is a key factor to a successful management of software development, since it allows for:

- early detection of problems in the artifacts produced in the initial phases of the life-cycle (specification and design documents) and, therefore, reduction of the

cost of change—late identification and correction of problems are much more costly than early ones;

- better software quality monitoring from the early phases of the life-cycle;
- quantitative comparison of techniques and empirical refinement of the processes to which they are applied;
- more accurate planning of resource allocation, based upon predicted *error-proneness* of the system and its constituent parts.

Software maintenance is generally recognized to consume the majority of resources in many software organizations [7]. It is one of the most difficult and costly tasks in the software development process [36]. Numerous factors can affect software maintenance quality and productivity, e.g., the maintenance personnel experience profile and training, the way knowledge about the maintained systems is managed and conveyed to the maintainers and users, the maintenance organization, processes and standards in use, the initial quality of the software source code and its documentation [16]. We must be able to characterize, asses, and improve the maintainability of software products in order to decrease maintenance costs [4]. Maintenance involves activities such as: correcting errors, functional enhancement, migrating software to new technologies and adapting software to deal with new environmental requirements [4], [7].

Corrective maintenance is the part of software maintenance devoted to correcting errors. Mostly, when software maintainers have to correct a faulty software component, they rely almost exclusively on their previous experience in order to estimate the effort they will spend to do it. Even though highly experienced software maintainers may make accurate predictions, the estimation process remains informal, error-prone, and poorly documented, making it difficult to replicate and spread throughout the organization [4].

As Rombach describes in his measurement experience based study [44], design measures can be used in order to predict maintainability. There are two different design steps: architectural, or high-level design, and algorithmic, or low-level design. Architectural design involves identifying software components and their interconnection; algorithmic design involves identifying data structures and the control flow within the architectural components. Rombach also supports the belief that the architectural design information has more influence on maintainability than algorithmic design information.

Testing of a large system is an example of a time- and resource-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive [8]. Therefore, we must build predictive models able to identify fault-prone components. The fault-proneness predictive models will help project managers to accurately allocate the testing and verification resources, as well as to concentrate the testing and verification effort on the parts of the system likely to be faulty. These models will also help software maintainers better assess the maintainability of software products [4]. For instance, estimation models can help maintainers optimize the allocation of resources to corrective maintenance activities. Evaluation models can help developers make decisions about when to re-structure or re-engineer a software component in order to make it more maintainable. Understanding models can help maintainers know better the underlying reasons about the difficulty of correcting specific kinds of errors.

The overall software process improvement will actually reduce the cost and time associated with the development of the product, thus, enabling the project management achieve its ultimate goal—delivery of a software product with the targeted quality within the budget and on schedule.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In the next chapter we present fault-proneness as a software product's quality indicator. Then we define the specific problem with respect to OO metrics and predictive modeling. Chapter 3 describes the solution strategy. In Chapter 4 we present the related work in two domains: OO metrics, and predictive modeling. Chapter 5 presents the concepts behind the predictive power of C4.5 machine learning algorithm. We describe the experimental framework in Chapter 6 with respect to the investigated C++ system, selected suite of OO metrics, and method used. The results are presented and discussed in Chapter 7. Finally, Chapter 8 gives the conclusion of the thesis and points to the future works.

Chapter 2

Fault-proneness as a Quality Indicator

It is often noted that a small number of software components are responsible for a disproportionately large number of faults during software development [33], [34], [39], [15], [22]. A major research effort is underway to try to determine, a priori, which modules are likely to contain a significant number of errors so that the testing and verification process might be focused in the most productive direction. This allows us to optimize the reliability of the system with minimum cost. In order to meet this goal, we build quantitative models that predict which components are likely to contain the highest concentration of faults. Once these *high-risk* components have been identified, the software development process can be optimized to reduce risk. There are two different aspects to be treated when one builds a risk model [15], [22], [33], [46]:

1. Metrics that are good predictors of risk should be identified and validated.
2. A suitable (in terms of underlying assumptions) modeling technique should be used so that the prediction is accurate and interpretation possible.

Risk reduction analysis can be performed from various perspectives of risk, e.g., number of errors, error density, associated cost of change during either testing or maintenance. For

example, additional testing can be applied to those components that have been determined to be likely to contain a high density of defects [22]. However, building fault-proneness predictive models is a difficult task: it is often the case in software engineering that the data which is collected are minimal, incomplete, and heterogeneous [15]. This presents several problems for model construction and interpretation (e.g., small data sets, inaccurate models, etc.). Therefore, we need a modeling process that is robust to these problems, allows for reliable classification of high-risk components, and aids in the understanding of the causes of this high risk. This understanding is important because it can give us insight into the software development process, allowing us to take remedial actions and make better process decisions in the future.

2.1 Software Process Improvement

Process improvement in terms of prediction of defects in the delivered product is one area that has received a significant amount of attention [39], [22], [29]. Recent studies have focused on the identification of problem areas during the design phase, noting that the software architecture is a major factor in the number of errors and effort rework found in later phases [44], [3], [29]. Decisions made during design can affect software reliability, maintainability, flexibility, and other quality factors. A shortcoming of most large-scale software development projects is the lack of information concerning the consequences of these design decisions until much later in the development process [3]. For example, it may become clear that a system is not flexible in adapting to changing requirements only after extensive investment of time and effort to implement the design, and to integrate, test, and use the system.

Greater capability is needed during the design phase to assess the design itself for indications that, when implemented, the resulting system will have particular quality characteristics. Rombach observes that "... most of the important structural decisions had

been made irreversibly by the end of architectural design" [44]. Computer industry related organizations have an especially strong need for such early design assessment capabilities because they expect delivered systems to be reliable and supportable over long operational lifetimes.

Traditional approaches to design assessment—for example, through attending design reviews and inspecting design documents—are highly subjective and don't scale up well to large, complex systems [3]. Thus, a strong motivation exists to develop a more analytical and repeatable technology to enhance design assessment capabilities. If potential problem areas regarding software product development can be detected during the design, as opposed to during the phases of implementation and testing, the development organization may have more options to mitigate the risk [22]. For example, rather than intensively testing the "problem components", one might restructure the system to avoid potential problems entirely. While this may be an option during the design phase, it is very unlikely scenario late in the implementation phase.

Thus our goal is to use measures of the design phase to determine potential problem areas in the delivered product, and allow for a wide range of preventive/corrective actions to be taken. Examples of these types of actions include increasing testing, providing additional documentation, re-designing a part of the system, and providing additional training.

2.1.1 Benefits of Process Improvement

A key requirement for the success of a new software development process is the accurate evaluation of how effective it is in reducing the bottom-line cost of getting the job done. Dion described in [29] how Raytheon—a diversified, international, technology based company, one of the largest corporations in the USA—approached measuring how changes in its software process resulted in reduced development costs.

As Figure 2.1 illustrates, Raytheon's process improvement paradigm is based on a three phase cycle of stabilization, control, and change, which applies the principles of W. E. Deming and Joseph Juran—the real process improvement must follow a sequence of steps, starting with making the process visible, then repeatable, and then measurable.

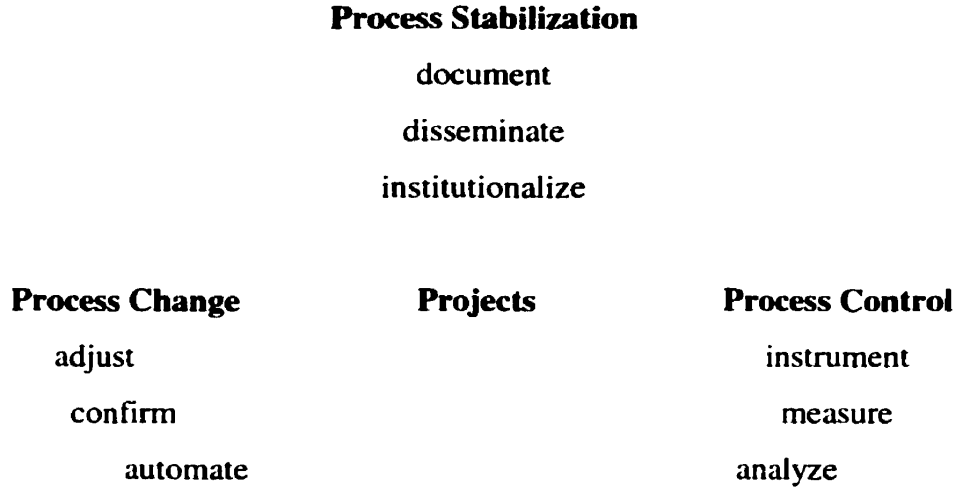


Figure 2.1. Raytheon's three phase process improvement paradigm (clockwise).

In the *process stabilization* phase, the emphasis is on distilling the elements of the process actually being used (achieving visibility) and progressively institutionalizing it across all projects (providing repeatability). In the *process control* phase, the emphasis shifts to instrument projects to gather significant data (measurement) and analyze the data to understand how to control the process. In the *process change* phase, the emphasis is on determining how to adjust the process as a result of measurement and analysis, and how to diffuse the new methods among practitioners (technology transition). Improvement is continuous, thus, completion of the third phase signals a beginning of the first [29].

At Raytheon, the staff used Philip Crosby Associates' approach [26] to quantify the benefit of improvements made to ongoing projects. Crosby's approach differentiates the cost of *doing it right the first time* from the cost of *rework* and categorizes the costs associated with any process as:

- *Performance.* The costs associated with *doing it right the first time*, including elements such as developing the design and generating the code.
- *Appraisal.* The costs associated with testing the product to determine if it is *faulty.*
- *Rework (nonconformance).* The costs associated with *fixing defects* in the code or design.
- *Prevention.* The costs incurred in attempting to *prevent the fault* from getting into the code.

The sum of appraisal, rework, and prevention costs is what Crosby calls "the cost of quality". The total project cost is simply the performance cost plus the cost of quality.

It is now obvious that the fault-proneness is built in the very foundation of the cost of quality. That is perhaps, the main motivation for assessment of likely to be faulty components using predictive modeling techniques, based on the metrics obtained in the phase of design, thus early in the software development process. One of the side benefits, according to Dion is that "...individual engineers when made aware of error-data categorization, tend to be more careful in areas with the highest frequency of errors".

At the time the Raytheon's process improvement initiative started, rework costs averaged about 41 percent of total project cost. By reducing the rework its costs have shrunk to about one forth of the original value (from 41 to 11 percent) [29]. Rework savings are often achieved at the expense of a small increase in the cost of other process stages. In Raytheon's case, the cost of design and coding rose slightly because formal inspections replaced informal reviews. However, that change enabled rework savings in uncovering source code problems before software integration and eliminating unnecessary re-testing. The attention paid to uncovering errors up front is guaranteed to save money in the end.

The cost associated with fixing source code problems found during integration has been identified as the largest contributor to rework costs. The integration cost has decreased to

about 20 percent of its original value. Raytheon's staff believes that the savings can be credited to the design and code inspections, training, and requirements stability. The greater gain by making the integration phase more efficient is due to the fact that fixing problems this late in the process costs so much more. The cost of re-testing decreased to about half of its original value. The saving indicates that far fewer problems were found during the first test (all the software is tested at least once), a direct effect of removing design and coding errors found in inspections [29].

2.2 Object-Oriented Paradigm

Object-oriented technology is a common practice for many software development companies. Given the potential of object-oriented programming (OOP) to systematically facilitate the reuse of code, we observe OOP rising in popularity as the industrial programming methodology of the next century [46]. The object-oriented paradigm claims a faster development pace and higher quality of software than the procedural paradigm [36]. OO programming, OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The study of the OO paradigm results in OO concepts such as [14]:

- *Object*. OO programming uses *objects*, not algorithms, as its fundamental logical building blocks.
- *Class*. Each object is an instance of some *class*.
- *Inheritance*. Classes are related to one another via *inheritance* relationship (the "kind of" hierarchy).

The OO programming style is based on its own conceptual framework—the *object model*. There are four major elements of this model: (1) abstraction, (2) encapsulation, (3) modularity, and (4) hierarchy.

The programming behaviors exhibited in the OO paradigm differ from those of the procedural paradigm. For example, the creation of classes in the OO programming languages is a programming behavior distinguished from the creation of procedures/functions in the procedural languages [36]. The insertion of OO technology in the software industry has created new challenges for companies which use product metrics as a tool for monitoring, controlling, and improving the way they develop and maintain software [8]. So, it is not surprising that researchers have begun to explore software measures unique to the OO paradigm and their application to software quality engineering [46]. Some studies have concluded that "traditional" product metrics are not sufficient for characterizing, assessing, and predicting the quality of OO software systems, as reported in [1].

To address this issue, OO metrics have been proposed in the literature [24], [2], [17], [10], [11], [37], [40]. Metrics proposed in [10] and [11] are code based measures and hence cannot be considered early in the software life cycle. However, Chidamber and Kemerer have proposed a set of metrics which can be evaluated from the design documents [24], [25], [32]. The Chidamber and Kemerer metrics have been empirically validated by Basili et al. in [8]. Briand et al. have proposed and empirically validated a set of class coupling metrics in [27]. Abreu et al. have defined MOOD metrics that were empirically validated by Abreu and Melo in [2]. All of these empirical validation have been conducted in laboratory on a small sized projects. We will investigate the empirical validation of some of these metrics on an industrial product.

Empirical validation aims at demonstrating the usefulness of a measure in practice, and is, therefore, a crucial activity to establish the overall validity of a measure [8]. A measure may be correct from a measurement theory perspective but of no practical relevance to the problem at hand. On the other side, a measure may not be entirely satisfactory from a theoretical perspective but can be a good enough approximation and work fine in practice. We will investigate the usefulness of selected OO design metrics to predict fault-proneness as a quality indicator.

Chapter 3

Specific Problem Statement

An important objective in all software development is to ensure that the delivered product is as fault-free as possible. One way to achieve this is to utilize a metric suite that can be applied to the design phase, early in the life cycle, to identify which modules are likely to be fault prone. These modules can then be redesigned before the product is implemented. Metrics of this kind have a second use, as well. If a product has already been constructed, these metrics can be utilized to predict future maintenance effort. That is, because they can detect fault-prone components, they can identify which components are likely to require corrective maintenance in the future [18].

- *Which OO design product measures are good predictors of fault-proneness as a software product's quality indicator?* What are the desirable properties of the OO system that need to be measured? *Coupling* is one of the design properties that characterize the quality of software products. Stevens, Myers, and Constantine define coupling as "the measure of the strength of association by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules". Coupling with regard to modules is applicable to OO design, but

coupling with regard to classes is equally important [14]. However, there is a tension between the concepts of coupling and inheritance. On the one hand, weakly coupled classes are desirable; on the other hand, *inheritance*—which tightly couples superclasses and their subclasses—helps us to exploit the commonality among abstractions. *Cohesion* is another design property that comes from structured design. Cohesion measures the degree of connectivity among the elements of a single module (and for OO design, a single class or object). The least desirable form of cohesion is coincidental cohesion (i.e., class comprising the abstractions of dogs and spacecraft, whose behavior are quite unrelated). The most desirable form of cohesion is functional cohesion, in which the elements of a class or module all work together to provide some well-bounded behavior [14].

- *What is a suitable technique for predictive modeling and empirical validation of the selected OO design metrics?* Several different modeling techniques are commonly cited in the literature. Linear regression has been used in [7], [2], [27], [33], logistic regression in [2], [8], [15], [20], [22], discriminant analysis in [39], [46], optimized set reduction in [15], [22], machine learning algorithms in [4], [9], [37]. The prediction models, though, are based on the assumption that the OO design metrics are potential predictors for the fault-proneness of classes in OO systems.

- *What are the objective criteria against which the models built should be evaluated with respect to their predictive quality?* In other words, how to validate the models and claim research success?

3.1 Solution Strategy

In order to provide adequate answers on the questions from the previous section we will take the following steps:

1. *Fault-proneness hypotheses proposal.* We propose hypotheses that relate design properties of OO components and fault-proneness as their quality indicator. The properties quantified with the corresponding product measures are coupling, cohesion, and inheritance.

2. *Metrics selection.* We select and empirically validate an appropriate suite of metrics with respect to the hypothesis from the previous step. Adequate candidates are those metrics that capture the OO design properties stated in the hypothesis. We also provide suitable definition(s) for fault-proneness as a dependent variable.

3. *Predictive modeling.* We build predictive models that capture different dependency forms in an OO design and allow relating them to the number of defects detected. We use these models as a means for empirical validation of the metrics set used. We selected the C4.5 machine learning algorithm [43] as a modeling technique. In order to choose the best models, we investigate different options of C4.5 algorithm during the model construction process.

4. *Evaluation and verification.* The predictive quality of the constructed models is evaluated against the objective set of standards—correctness, completeness, accuracy, and goodness of fit of data. Predicted fault-proneness is compared with the actual fault proneness extracted as defect data.

Chapter 4

Background and Related Work

In this chapter we describe where the inspiration for our work came from ? We organize the information presented in two sections, corresponding to two different domains essential for our research. In the first section, we define some of the OO design metrics available in the software engineering community using the terminology and formalism introduced by Briand et al. in [18] and [19]. In the first part of the second section we discuss some of the works regarding various predictive modeling techniques for various software quality related dependent variables. The second part of the second section is dedicated to the research work that relates software product measures and software quality indicators with respect to machine learning algorithms as predictive modeling technique.

4.1 Some Object-Oriented Design Metrics

Automated tools supporting OO metrics data collection and analysis are essential for the use of product metrics on a regular basis in order to monitor and improve software [6]. For metrics to be actively used in the real world, the support in terms of simple and easy to use tools is a must. The use of metrics collection tools can significantly help in

improving a practitioner's ability to identify, analyze, fix, and improve quality characteristics of software design and implementations.

In order to meet those objectives, such a tool, M-System [40], has been designed and developed at Fraunhofer IESE[1]. M-System supports the collection of 49 plus OO design product metrics. The definitions of the measures come from [24], [17], [20], [25], [36], [12], [31], [30], [35]. M-System itself, is based on AT&T Gen++ [27] which is a language tool that processes C++ source code. Firstly, the source code of the investigated C++ system needs to be analyzed by Gen++. Then, M-System uses the Gen++ Query Results as an input in order to compute the OO design metrics in question. Further discussion about M-System or Gen++ is beyond the scope of this work. For any additional information refer to [40] and [27] accordingly. We decided to use M-System in our research work, since it provides automated data collection support for the measures of targeted design properties as required by steps 1 and 2 in the section 3.1 *Solution Strategy*.

Typically, the OO approach to software development is iterative in nature and the phases of the development process highly overlap. While the basic set of objects, operations, attributes, and relationships that fulfill given requirements are identified in the analysis phase, the details of a class's methods, parameters, data declarations, relationships, and algorithms are resolved during the design. The results of an analysis and design process are hierarchies of well defined classes that represent a blueprint for an implementation [6].

Metrics assess the internal and external structure, relationships, and functionality of software components. The most basic components of an OO system are its classes. OO systems are built around classes and their interrelationships. The interdependence of classes upon each other defines the external structure of the system. Relationships among classes define the paths of communication between objects of classes. Organization of classes using relationships such as "is-a" and "consists-of" allows for the sharing of

[1] Fraunhofer IESE (the Institute for Experimental Software Engineering), Kaiserslautern, Germany.

functionality and attributes. The member functions of a class define the services a class supports, and its interactions with other objects, while the member data of a class define the internal structure of the class' objects.

In the past, research within the area of software measurement has suffered from a lack of (i) standardized terminology and (ii) a formalism for defining measures in an unambiguous and fully operational manner (that is, a manner in which no additional interpretation is required on behalf of the user of the measure) [18], [19]. As a consequence, development of consistent, understandable, and meaningful software quality predictors has been severely hampered. To remedy this situation it is necessary to reach a consensus on the terminology, define a formalism for expressing software measures, and, most importantly, to use this terminology and formalism.

To express the OO design metrics extracted by M-System consistently and unambiguously the following terminology and formalism based on set and graph theory are presented.

4.1.1 Terminology and Formalism

a) System, classes, inheritance relationships

An OO system S consists of a set of classes C. There can exist inheritance relationships between classes such that for each class $c \in C$ let

- $Parents(c) \subset C$ be the set of parent classes of class c
- $Children(c) \subset C$ be the set of children classes of class c
- $Ancestors(c) \subset C$ be the set of ancestor classes of class c
- $Descendants(c) \subset C$ be the set of children classes of class c

In C++, a class c can declare a class d its friend, which means that d is thus granted access to non-public elements of c. We must be able to specify for a class c, which are its friends, and which classes declare class c their friend. Therefore, let

- $Friends(c) \subset C$ be the set of friend classes of class c
- $Friends'(c) \subset C$ be the set of classes that declare class c their friend
- $Others(c) \subset C$ be the set of other classes to class c,

where

$$Others(c) = C \setminus \left(Ancestors(c) \cup Descendants(c) \cup Friends(c) \cup Friends'(c) \cup \{c\} \right)$$

b) Methods

A class c has a set of methods $M(c)$. A method can be either virtual or non-virtual, and either inherited, overridden, or newly defined. Thus, $M(c)$ can be decomposed into the set of methods declared and set of methods implemented in c. For each class $c \in C$ let

- $M_D(c) \subseteq M(c)$ be the set of methods *declared* in c, i.e., methods that c inherits, but does not override or virtual methods of c
- $M_I(c) \subseteq M(c)$ be the set of methods *implemented* in c, i.e., methods that c inherits, but overrides or non-virtual non-inherited methods of c,

where

$M(c) = M_D(c) \cup M_I(c)$ and $M_D(c) \cap M_I(c) = \varnothing$.

Furthermore, for each class $c \in C$ let

- $M_{INH}(c) \subseteq M(c)$ be the set of *inherited* methods of
- $M_{OVR}(c) \subseteq M(c)$ be the set of *overriding* methods of c
- $M_{NEW}(c) \subseteq M(c)$ be the set of new, or, non-inherited, non-overriding methods of c,

where

$M(c) = M_{INH}(c) \cup M_{OVR}(c) \cup M_{NEW}(c)$.

The access specification of the different methods can be taken into consideration to split up *M(c)* into the set of public and private methods. Since in C++, following OO notation, there are *public, protected* and *private* methods, $M(c)$ is split up into public, and not public methods

$$M(c) = M_{pub}(c) \cup M_{npub}(c) .$$

For each method of a class there exist a set of parameters *Par(m).*

To measure the cohesion, and coupling of a class, c, it is necessary to define the set of methods that $m \in M(c)$ invokes and the frequency of these invocations. Method invocations can be either static or dynamic,. Consequently, for each method $m \in M(c)$ the following sets are defined:

- *SIM(m)* the set of statically invoked methods of m.

  Let $c \in C$, $m \in M_I(c)$ and $m' \in M(c)$. Then $m' \in SIM(m) \Leftrightarrow \exists\, d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

- *NSI(m, m')* the number of static invocations of m' by m.

  Let $c \in C$, $m \in M_I(c)$ and $m' \in SIM(m)$. *NSI(m, m')* is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$.

- *PIM(m)* the set of polymorphically invoked methods of m.

  Let $c \in C$, $m \in M_I(c)$ and $m' \in M(c)$. Then $m' \in PIM(m) \Leftrightarrow \exists\, d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism, be invoked for an object of dynamic type class d.

- *NPI(m, m')* the number of polymorphic invocations of *m'* by *m*.

  Let $c \in C$, $m \in M_I(c)$ and $m' \in PIM(m)$. *NPI(m, m')* is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$. As a result of polymorphism, one method invocation can contribute to the *NPI* count of several methods.

c) Attributes

Classes have attributes which are either inherited or newly defined. For each class $c \in C$ let *A(c)* be the set of attributes of class *c*.

$$A(c) = A_D(c) \cup A_I(c), \quad \text{where}$$

- $A_D(c)$ is the set of attributes declared in class c (i.e., inherited attributes).
- $A_I(c)$ is the set of attributes implemented in class c (i.e., non-inherited attributes).

Methods may reference attributes. It is sufficient to consider the static type of the object for which an attribute is referenced because attribute references are not determined dynamically. We define:

- *AR(m)* as the set of attributes referenced by method m, where $m \in M(c)$
- *NAR(m, a)* as the number of references of method m to attribute a, where $a \in A(d)$ for some $d \in C$.

d) Predicates, interactions, derived sets

To account for coupling of classes $c, d \in C$ we define the predicate

$$uses(c,d) \Leftrightarrow \left(\exists m \in M_I(c) : \exists m' \in M_I(d) : \exists m' \in SIM(m)\right)$$
$$\vee \left(\exists m \in M_I(c) : \exists a \in A_I(d) : \exists a \in AR(m)\right)$$

To account for certain interactions between classes $c, d \in C$ we define:

- *ACA(c, d)* as the number of actual *class-attribute* interactions from server c to client d
- *ACM(c, d)* as the number of actual *class-method* interactions from server c to client d
- *AMM(c, d)* as the number of actual *method-method* interactions from server c to client d

For the sets of methods *SIM(m)* and *PIM(m)*, mentioned previously, we also define the sets of indirect method invocations

$$SIM^{*}(m) \quad \text{and} \quad PIM^{*}(m)$$

that are transitive closures of *SIM(m)* and *PIM(m)*.

In the continuation, we present definitions of OO design metrics grouped around properties of the OO system which they measure—inheritance, cohesion, and coupling.

4.1.2 Inheritance

Let $c \in C$. We define:

1. Depth of inheritance tree $DIT(c)$

$$DIT(c) = \begin{cases} 0, \ if \ \ Parents(c) = 0 \\ 1 + \max\{DIT(c') : c' \in Parents(c)\}, \ \ else. \end{cases}$$

2. Average inheritance depth $AID(c)$

$$AID(c) = \begin{cases} 0, \ if \ Parents(c) = 0 \\ \dfrac{\sum_{c' \in Parents(c)} (1 + AID(c'))}{|Parents(c)|}, \ else. \end{cases}$$

3. Class to leaf depth $CLD(c)$

$$CLD(c) = \begin{cases} 0, if \ Descendants(c) = 0 \\ \max\{DIT(c') - DIT(c) : c' \in Descendants(c)\}, \ else. \end{cases}$$

4. Number of children $NOC(c)$

$$NOC(c) = |Children(c)|$$

5. Number of parents $NOP(c)$

$$NOP(c) = |Parents(c)|$$

6. Number of descendants $NOD(c)$

$$NOD(c) = |Descendants(c)|$$

7. Number of ancestors $NOA(c)$

$$NOA(c) = |Ancestors(c)|$$

8. Number of methods overridden $NMO(c)$

$$NMO(c) = |M_{OVR}(c)|$$

9. Number of methods inherited $NMI(c)$

$$NMI(c) = \left| M_{INH}(c) \right|$$

10. Number of methods added, new methods $NMA(c)$

$$NMA(c) = \left| M_{NEW}(c) \right|$$

11. Specialization index $SIX(c)$

$$SIX(c) = \frac{NMO(c) \cdot DIT(c)}{\left| M(c) \right|}$$

The development of OO system can be analyzed at two levels: class level, and system level. We present the definitions of the following system level metrics:

12. Average/Maximum class depth of system $ACD(S) / MCD(S)$

$$ACD(S) = \frac{\sum_{c \in C} DIT(c)}{|C|}$$

$$MCD(S) = \max\left\{ DIT(c') : c \in C \right\}$$

13. System's average inheritance depth $SAID(S)$

$$SAID(S) = \frac{\sum_{c \in C} AID(c)}{|C|}$$

14. Total base classes of system $TBC(S)$

$$TBC(S) = \left| \left\{ c : c \in C \wedge Parents(c) = 0 \right\} \right|$$

15. Maximum breadth of inheritance tree in system $MBIT(S)$

$$MBIT(S) = \max\left\{\left|\{c : c \in C \wedge DIT(c) = n\}\right| : n = 1, 2, 3, \ldots\right\}$$

16. Maximum number of children in system $MNOC(S)$

$$MNOC(S) = \max\left\{\left|Children(c)\right| : c \in C\right\}$$

17. Number of inheritance links in system $NIL(S)$

$$NIL(S) = \left|\{[c,d] : c,d \in C \wedge d \in Children(c)\}\right|$$

18. Reuse ratio of system $U(S)$

$$U(S) = \frac{\left|\{c : Children(c) \neq 0\}\right|}{|C|}$$

19. Specialization ratio of system $SR(S)$

$$SR(S) = \frac{\left|\{c : Parents(c) \neq 0\}\right|}{\left|\{c : Children(c) \neq 0\}\right|}$$

20. Total/maximum overload count of system $TOC(S)$ / $MOC(S)$

$$TOC(S) = \sum_{c \in C} NMO(c)$$

$$MOC(S) = \max\left\{NMO(c) : c \in C\right\}$$

21. Method inheritance factor in system $MIF(S)$

$$MIF(S) = \frac{\sum_{c \in C} NMI(c)}{\sum_{c \in C} \left|M(c)\right|}$$

22. Attribute inheritance factor in system $AIF(S)$

$$AIF(S) = \frac{\sum_{c \in C} |A_D(c)|}{\sum_{c \in C} |A(c)|}$$

23. Polymorphism factor of system $POF(S)$

$$PF(S) = \frac{\sum_{c \in C} NMO(c)}{\sum_{c \in C} NMA(c) \cdot |Descendants(c)|}$$

4.1.3 Cohesion

Let $c \in C$. We define:

1. Lack of cohesion in methods $LCOM_1(c)$

$$LCOM_1(c) = \left| \left\{ [m_1, m_2] : m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = 0 \right\} \right|$$

Note that this definition only considers methods implemented in class c, and that only references to attributes *implemented* in class c are counted. *LCOM* is an inverse cohesion measure. A high value of *LCOM* indicates low cohesion and vice versa. It can be interpreted [30] as the number of pairs of methods in class c having no common attribute reference.

2. Lack of cohesion in methods $LCOM_2(c)$

Let $G_c = (V_c, E_c)$ be an undirected graph with vertices $V_c = M_I(c)$ and edges $E_c(c) = \left\{ [m_1, m_2] : m_1, m_2 \in V_c \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq 0 \right\}$. Then

$LCOM_2(c)$ is the number of connected components of G_c.

3. Lack of cohesion in methods $LCOM_3(c)$. Like $LCOM_2(c)$, but using the following definition of edges

$$E_c(c) = \{[m_1, m_2] : m_1, m_2 \in V_c \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq 0$$
$$\vee m_1 \in SIM(m_2) \vee m_2 \in SIM(m_1)\}$$

In the case where G_c consists of only one connected component ($LCOM_3(c) = 1$), the number of edges $|E_c|$ ranges between $|V_c| - 1$ (minimum cohesion) and $|V_c| \cdot (|V_c| - 1)/2$ (maximum cohesion). Hitz and Montazeri [31] define a measure *Co* "connectivity" which further discriminates classes having $LCOM_3(c) = 1$ by taking into account the number of edges of the connected component (entry 6, on our list).

4. Lack of cohesion in methods $LCOM_4(c)$. Let

$$P = \begin{cases} 0 \ \ if \ \ AR(m) = 0 \ \forall \ m \in M_I(c) \\ condition \ \ from \ \ LCOM_1(c), \ \ else. \end{cases}$$
$$Q = E_c \ \ from \ \ LCOM_2(c)$$

Then, we define

$$LCOM_4(c) = \begin{cases} 0, \ \ if \ |P| < |Q| \\ |P| - |Q|, \ \ else. \end{cases}$$

This measure was originally defined by Chidamber and Kemerer in [24] as the number of pairs of methods in a class having no common attribute references, $|P|$, minus the number of pairs of similar methods, $|Q|$.

5. Lack of cohesion in methods $LCOM_5(c)$

$$LCOM_5(c) = \frac{|M_I(c)| - \frac{1}{A_I(c)} \sum_{a \in A_I(c)} |\{m : m \in M_I(c) \wedge a \in AR(m)\}|}{|M_I(c)| - 1}$$

This measure was originally defined by Henderson-Sellers [30] in order to satisfy the following properties:

- The measure yields 0, if each method of the class references every attribute of the class ("perfect cohesion" according to Henderson-Sellers).
- The measure yields 1, if each method of the class references only a single attribute.
- Values between 0 and 1 are to be interpreted as percentages of the perfect value.

6. Lack of cohesion in methods $Coh(c)$—a variation on $LCOM_5(c)$

$$Coh(c) = \frac{\sum_{a \in A_I(c)} \left| \{ m : m \in M_I(c) \wedge a \in AR(m) \} \right|}{\left| M_I(c) \right| \cdot A_I(c)}$$

7. Connectivity $Co(c)$

$$Co(c) = 2 \frac{\left| E_c \right| - \left(\left| V_c \right| - 1 \right)}{\left(\left| V_c \right| - 1 \right) \left(\left| V_c \right| - 2 \right)}$$

where V_c and E_c are from $LCOM_3(c)$.

The original name of this measure in [HM95] is C, not Co. Briand et al. use Co in order to avoid the name conflict with the set C of all classes in the system S.

8. Tight class cohesion $TCC(c)$

$$TCC(c) = 2 \frac{\left| \left\{ \left[m_1, m_2 \right] : m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cau(m_1, m_2) \right\} \right|}{\left| M_I(c) \cap M_{pub}(c) \right| \left(\left| M_I(c) \cap M_{pub}(c) \right| - 1 \right)}$$

The predicate $cau(m_1, m_2)$, common attribute usage, is true, if $m_1, m_2 \in M_I(c)$ directly or indirectly use an attribute of class c in common.

$$cau(m_1, m_2) = \left(\bigcup_{m \in \{m_1\} \cup SIM^*(m_1)} AR(m) \right) \cap \left(\bigcup_{m \in \{m_2\} \cup SIM^*(m_2)} AR(m) \right) \cap A_I(c) \neq 0$$

This definition reflects the approach by Bieman and Kang [12] to measure cohesion. They also consider pairs of methods which use common attribute. A method m uses an attribute a directly, if $a \in AR(m)$. Method m uses attribute a indirectly, if m directly or indirectly invokes a method m' which uses attribute a: $\exists m' \in SIM^*(m) : a \in AR(m')$. Two methods are called "connected", if they directly or indirectly use a common attribute. The measure *TCC* is defined as the percentage of pairs of public methods of the class, with common attribute usage.

9. Loose class cohesion $LCC(c)$

$$LCC(c) = 2\frac{\left|\left\{\left[m_1, m_2\right] : m_1, m_2 \in M_I(c) \cap M_{pub}(c) \wedge m_1 \neq m_2 \wedge cau^*(m_1, m_2)\right\}\right|}{\left|M_I(c) \cap M_{pub}(c)\right|\left(\left|M_I(c) \cap M_{pub}(c)\right| - 1\right)}$$

where the predicate $cau^*(m_1, m_2)$ is the transitive closure of $cau(m_1, m_2)$ defined at $TCC(c)$.

10. Information flow based cohesion $ICH(c)$

$$ICH^c(m) = \sum_{m' \in M_{NEW}(c) \cup M_{INR}(c)} \left(1 + \left|Par(m')\right|\right) NPI(m, m')$$

$$ICH(c) = \sum_{m \in M_I(c)} ICH^c(m)$$

$$ICH(U) = \sum_{c \in U} ICH\ (m), \quad U \subset C$$

This set of cohesion measures is based on information flow through method invocations within a class [35]. For a method m implemented in class c, the cohesion of m is the number of invocations to other methods implemented in class c, weighted by the number of parameters of invoked methods. The more parameters an invoked method has, the more information is passed, the stronger the link between the invoking and invoked method. The cohesion of a class is the sum of the cohesion of its methods. The cohesion of a set of classes simply is the sum of the cohesion of the classes in the set.

4.1.4 Coupling

Let $c \in C$. We define:

1. Coupling between object classes $CBO(c)$

$$CBO(c) = \left|\left\{d \in C \setminus \{c\} : uses(c,d) \vee uses(d,c)\right\}\right|$$

This measure was originally defined by Chidamber and Kemerer in [24] as the number of classes to which class c is coupled. $CBO(c)$ takes a binary approach to coupling between classes: two classes are either coupled or not and the number of such "class couples" is counted.

2. Coupling between object classes $CBO'(c)$

$$CBO'(c) = \left|\left\{d \in C \setminus \left\{\{c\} \cup Ancestors(c)\right\} : uses(c,d) \vee uses(d,c)\right\}\right|$$

$CBO'(c)$ measures non-inheritance based coupling.

3. Response for class $RFC(c)$

$$RFC(c) = RFC_1(c)$$

RFC counts the number of methods invoked by the class. Originally, it represents the size of the *response set of a class*, defined as the set of methods in the class together with the set of methods called by the class's methods [24].

4. Response for class $RFC'(c)$

$$RFC'(c) = RFC_\infty(c)$$

$RFC'(c)$ is the number of methods that can possibly be invoked by sending a message to a class c. This includes methods of c, methods invoked by the methods of c, and so on.

5. Response for class $RFC_{\alpha}(c)$

$$RFC_{\alpha}(c) = \left| \bigcup_{i=0}^{\alpha} R_i(c) \right|, \quad \text{where}$$

$$R_0(c) = M(c) \quad \text{and} \quad R_{i+1}(c) = \bigcup_{m \in R_i(c)} PIM(m)$$

$RFC_{\alpha}(c)$ counts nested method invocations, those that $RFC'(c)$ stands for, up to a specified level α.

6. Message passing coupling $MPC(c)$

$$MPC(c) = \sum_{m \in M_I(c)} \sum_{m' \in SIM(m) \setminus M_I(c)} NSI(m,m')$$

Originally defined by Li and Henry in [36] as a number of send statements in a class, this measure counts the number of methods invocations.

7. Data abstraction coupling $DAC(c)$

$$DAC(c) = \sum_{\substack{d \in C \\ d \neq c}} ACA(c,d)$$

Originally defined by Li and Henry in [36] as a number of *abstract data types* defined in a class, this measure counts the number of attributes and parameters having a class type.

8. Data abstraction coupling $DAC'(c)$

$$DAC'(c) = \sum_{\substack{d \in C \\ d \neq c}} \left(ACA(c,d) > 0 \right)$$

This measure counts the number of classes used as a type for an attribute.

Briand et al. observe in [18] that there is an important difference between the "number of attributes" having a class as its type and the "number of classes" used as types for attributes. If a class c has 10 attributes of type class d, $DAC(c) = 10$ whereas $DAC'(c) = 1$ is quite possible.

9. Information flow based coupling $ICP(c)$

$$ICP^c(m) = \sum_{m' \in PIM(m) \setminus (M_{NEW}(c) \cup M_{OVR}(c))} \left(1 + \left|Par(m')\right|\right) NPI(m,m')$$

$$ICP(c) = \sum_{m \in M_I(c)} ICP^c(m)$$

$$ICP(U) = \sum_{c \in U} ICP(c), \quad U \subset C$$

ICP measures, introduced by Lee et al. in [35], aim at measuring the amount of information flow between methods. Information flow occurs between a method m and any method m' that is possibly invoked by method m which includes methods $m' \in PIM(m)$. Lee et al. acknowledge the need to differentiate between inheritance-based and non-inheritance-based coupling by proposing corresponding measures: *NIH-ICP* and *IH-ICP*. *ICP* is simply the sum of both types of coupling.

10. Information flow based non-inheritance coupling $NIH\text{-}ICP(c)$

$$NIH - ICP^c(m) = \sum_{m' \in PIM(m) \cap \left(\bigcup_{c' \in Ancestors(c)} M(c')\right)} \left(1 + \left|Par(m')\right|\right) NPI(m,m')$$

$$NIH - ICP(c) = \sum_{m \in M_I(c)} NIH - ICP^c(m)$$

$$NIH - ICP(U) = \sum_{c \in U} NIH - ICP(c), \quad U \subset C$$

11. Information flow based inheritance coupling *IH-ICP(c)*

$$IH-ICP^{c}(m)=\sum_{m'\in PIM(m)\cap\left(\mathbf{U}_{c'\in C\setminus\{c\}\setminus Ancestors(c)}M(c')\right)}\left(1+\left|Par(m')\right|\right)NPI(m,m')$$

$$IH-ICP(c)=\sum_{m\in M_I(c)}IH-ICP^{c}(m)$$

$$IH-ICP(U)=\sum_{c\in U}IH-ICP(c),\quad U\subset C$$

Briand et al. created a metrics suite [17] designated to investigate the quality impact of the different design mechanisms in C++ (i.e., specialization, generalization, aggregation, and friendship). There are three different facets, or modalities, of coupling between classes in OO systems developed in C++. They are referred as: *locus*, *type*, and *relationship*:

- ***Relationship*** refers to the type of relationship: inheritance, friendship, or other (neither). Clearly, a class c is most closely coupled with its descendants, ancestors, friends.

- ***Locus*** refers to expected locus of impact; i.e., whether the impact of change flows towards a class (import) or away from a class (export). Thus changes to an ancestor flows towards a class (import) and changes to a class flows towards its descendants (export). During import class c acts as a client, while during export class c acts as a server.

- ***Type*** refers to the type of interactions between classes (or their elements). It may be:

  a) *Class-Attribute* interaction: there is a class-attribute interaction between classes c and d, if class c is the type of an attribute of class d.

  b) *Class-Method* interaction: there is a class-method interaction between classes c and d, if class c is the type of a parameter of method m_d; or if class c is the return type of method m_d.

c) *Method-Method* interaction: let m_c and m_d be methods of class c and class d respectively. There is a method-method interaction between classes c and d, if m_d directly invokes m_c, or m_d receives via parameter a pointer to m_c thereby invoking m_c indirectly.

Coupling between classes in C++ can be due to any combination of these facets. Using measures that can account for all different types of interactions, we can evaluate the actual impact of each coupling dimension on the quality of the resulting artifact. There are three types of relationships, two loci, and three types of interactions described in [17], that comes to 18 different possible coupling measures. They are listed in the continuation.

12. Inverse friends class-attribute import coupling $IFCAIC(c)$

$$IFCAIC(c) = \sum_{d \in Friends^{-1}(c)} ACA(c,d)$$

13. Ancestors class-attribute import coupling $ACAIC(c)$

$$ACAIC(c) = \sum_{d \in Ancestors(c)} ACA(c,d)$$

14. Others class-attribute import coupling $OCAIC(c)$

$$OCAIC(c) = \sum_{d \in Others(c)} ACA(c,d)$$

15. Friends class-attribute export coupling $FCAEC(c)$

$$FCAEC(c) = \sum_{d \in Friends(c)} ACA(d,c)$$

16. Descendants class-attribute export coupling $DCAEC(c)$

$$DCAEC(c) = \sum_{d \in Descendants(c)} ACA(d,c)$$

17. Others class-attribute export coupling $OCAEC(c)$

$$OCAEC(c) = \sum_{d \in Others(c)} ACA(d,c)$$

18. Inverse friends class-method import coupling $IFCMIC(c)$

$$IFCMIC(c) = \sum_{d \in Friends^{-1}(c)} ACM(c,d)$$

19. Ancestors class-method import coupling $ACMIC(c)$

$$ACMIC(c) = \sum_{d \in Ancestors(c)} ACM(c,d)$$

20. Others class-method import coupling $OCMIC(c)$

$$OCMIC(c) = \sum_{d \in Others(c)} ACM(c,d)$$

21. Friends class-method export coupling $FCMEC(c)$

$$FCMEC(c) = \sum_{d \in Friends(c)} ACM(d,c)$$

22. Descendants class-method export coupling $DCMEC(c)$

$$DCMEC(c) = \sum_{d \in Descendants(c)} ACM(d,c)$$

23. Others class-method export coupling $OCMEC(c)$

$$OCMEC(c) = \sum_{d \in Others(c)} ACM(d,c)$$

24. Inverse friends method-method import coupling $IFMMIC(c)$

$$IFMMIC(c) = \sum_{d \in Friends^{-1}(c)} AMM(c,d)$$

25. Ancestors method-method import coupling $AMMIC(c)$

$$AMMIC(c) = \sum_{d \in Ancestors(c)} AMM(c,d)$$

26. Others method-method import coupling $OMMIC(c)$

$$OMMIC(c) = \sum_{d \in Others(c)} AMM(c,d)$$

27. Friends method-method export coupling $FMMEC(c)$

$$FMMEC(c) = \sum_{d \in Friends(c)} AMM(d,c)$$

28. Descendants method-method export coupling $DMMEC(c)$

$$DMMEC(c) = \sum_{d \in Descendants(c)} AMM(d,c)$$

29. Others method-method export coupling $OMMEC(c)$

$$OMMEC(c) = \sum_{d \in Others(c)} AMM(d,c)$$

The usefulness of the selected OO design metrics, with respect to their ability to predict likely to be faulty components through appropriate modeling technique, is discussed in Chapter 7 where we present the experimental results of our study.

4.2 Predictive Modeling

According to the Webster's dictionary, a ***model*** is a system of postulates, data, and inferences presented as a (mathematical) description of an entity or state of affairs. From that perspective, ***predictive model*** is a model where one to few important Y-variables are predicted from some number of X-variables that can be obtained (easily) from standard maps or standard monitoring programs. We call the Y-variables *dependent*, or predicted variables, as opposed to the X-variables, which are *independent*, or predictor variables.

Statistical models are derived from purely statistical considerations about the parameters in the system and their relationships. It is often very difficult to establish causal relationships among the factors and the effect variables, over certain range of conditions.

Deterministic models are based not solely on statistical analysis of the available data, but also on the presuppositions of the model designer. This often means that the model designer decides how the model should behave.

Step-by-step predictive models predict a desired Y-variable in several steps in such a way that one or more of the model variables X used to predict Y are themselves predicted variables.

Models can be described by:

- *mathematical links* among the parameters of the system, or
- *decision trees* or *rules* based on heuristic data analysis, or a knowledge of human experts.

4.2.1 Various Modeling Techniques

a) Linear Regression Analysis

Khoshgoftaar and Munson used *linear regression analysis* as a modeling technique in order to predict software development errors based on their assumed relationship with software complexity metrics [33]. The general notion of linear regression is to select from a set of independent variables a subset of these variables, which will explain the most amount of variance in a dependent variable. The key to model development is to choose the subset of independent variables in such a manner as to not introduce more variance (or noise) in the model than might be contributed by the independent variable itself. A major problem in the development of linear regression model centers around multicollinearity. The basic regression model is based on the assumption that the independent variables of the analysis are not linear compounds of each other nor do they share an element of common variance. Two variables sharing an element of variance are said to be collinear. To meet this assumption of nonmulticollinearity, another statistical procedure called *factor analysis* has been used. The specific value of factor analysis is the reduction of the complexity metric space to a set of orthogonal complexity dimensions, that are, in fact, noncollinear. Khoshgoftaar and Munson demonstrated with their models that there is a relationship between program errors and complexity domains of program structure and size.

Li and Henry [36] used the same linear regression analysis in order to build predictive models that relate OO metrics and maintainability. They have analyzed two commercial systems built in Classic-Ada, using Chidamber and Kemerer's metric suite [24], as well as several newly developed metrics. Their models proved Rombach's indication [44] that software metrics can predict maintainability. Basili et al. [7] though, developed predictive cost model for maintenance releases that were primarily composed of enhancements. They have analyzed 25 complete releases from 10 different projects at Flight Dynamics Division of the NASA Goddard Space Flight Center. Their linear regression model related

total release effort with total lines of code added, changed, and deleted. Finally, Abreu and Melo [2] validated a suite of OO design metrics called MOOD by developing linear regression models that predict defect and failure density (reliability measures), and normalized rework (maintainability measure). They have used data from a C++ controlled study performed by graduate and senior level students at the University of Maryland.

b) Logistic Regression Analysis

Agresti and Evanco [3] used log-linear regression analysis as a predictive modeling technique in order to relate design complexity metrics with the defect density in Ada programming language. Data has been obtained on three Ada projects consisting of 16 subsystems and 149K $SLOC^2$, from the NASA Goddard Space Flight Center. Ada systems have been viewed as being composed of design units ("parts"), and design relations ("connections"). Agresti and Evanco have shown that, of the five possible connections, the *context coupling* has been the most related to defect density. Briand et al., used two different modeling techniques in [15], Optimized Set Reduction (OSR), and logistic regression analysis—usually used when the dependent variable is binary in nature. OSR, developed at the University of Maryland, is based on both statistical and machine learning principles. Given a historical data set, OSR automatically generates a collection of logical expressions referred to as "patterns" which characterize the trends observable in the data set. The goal of the study was to relate design complexity and system architecture with *high/low fault frequency* components. The data set has been created using data from 146 components of a 260 KLOC Ada system. The components have been classified in two groups, high-risk and low-risk, using two-group predictive models based on each of the modeling techniques. Logistic regression has been used by Briand et al. [20] in another study with similar goal. Several high level design metrics have been defined and then used as predictors of high/low fault frequency components. The data used in the study are based on three NASA projects developed in Ada, with average of 100 KLOC each. In another study based on OSR and logistic regression as modeling techniques [22], Briand et al. defined a notion of a high risk component based on a combination of software quality factors. They defined two classes, *high isolation cost*, and *high*

completion cost, and built models for each. From change report form data, a component would be placed in the high isolation cost class if there is a defect in the component that requires more than certain amount of time to isolate an understand. Similarly, a component would be placed in the high completion cost class if there is a defect associated with it that would required more than certain amount of time to complete the error correction, once it had been isolated. The reason for the use of these two models has been to better understand the major influences in error isolation difficulty and error completion difficulty, which were likely to be different. The data in the study have come from the NASA Goddard Space Flight Center Flight Dynamics Division. Characteristics of the design have been used as explanatory variables in order to build classification models of 150 Ada components from three different systems. Both modeling techniques, OSR and logistic regression, have been evaluated with respect to the accuracy of the models, and ease of interpretation. Basili et al. [8] performed an empirical validation of the OO metrics defined in [24] with regard to their ability to identify fault-prone classes, as opposed to maintainability in the study by Li and Henry [36]. Data have been collected from eight university related projects developed in C++ programming language. Two two-group classification models have been built using logistic regression analysis, one based on OO metrics and the other one on classical code metrics. The results have shown that Chidamber and Kemerer's [24] OO metrics were better predictors than the best set of "traditional" code metrics, which can only be collected during later phases of the software development process. Finally, logistic regression has been used as a modeling technique in a study performed by Devanbu et al. [28], with a goal to relate reuse oriented measures with quality factors of productivity and defect density. The data in the study has been collected from seven C++ projects from the University of Maryland. The results indicate that different reuse metrics can be used as predictors of different quality attributes.

b) Discriminant Analysis

Munson and Khoshgoftaar [39] used another statistical modeling technique called discriminant analysis. It is basically a classification procedure. The underlying principle of

---

[2] SLOC stands for source lines of code.

the technique is that an operational hypothesis is formulated that there exists an *a priori* classification of multivariate observations into two or more groups or sets of observations. Further, the membership in one of these supposed groups is mutually exclusive. A criterion variable will be used for this group assignment. Thus a program, for example, might be classified with a code of 0 if it has been found to have no faults, or with a code of 1 if it has more than one fault. In the application of discriminant analysis in their study, Munson and Khoshgoftaar used non-correlated measures of program complexity (domain metrics) as independent variables in an attempt to classify programs into a group whose programs contain relatively few faults, or to a group whose programs contain relatively large number of faults. The data for the study have been collected from two commercial systems, one developed in Pascal and FORTRAN with approximately 40 KLOC, and the other one developed in Ada. Built predictive models have shown accuracy of 75% and 62% respectively. The authors concluded that discriminant analysis as a modeling technique did show promise for use in the circumstances of noise and certainly non-normally distributed data. In another study, Szabo and Khoshgoftaar [46] applied discriminant analysis to build a model that classifies program modules as either high, medium, or low risk. The quantitative measurements upon which classification has been based (independent variables) were principle components derived from a set of software product measures extracted from the source code. The classification (dependent) variable, *faults*, was a measure of the number of errors detected at the end of a specific development phase. The modules were divided into three groups based on the cutoff values. The cutoff values clearly determine the size of each group and will vary from environment to environment. Typically, cutoff values are determined based on the past history of projects developed in similar environment. The data in the study has been collected from a commercial system which contained 68 program modules (files) developed in C++. The software metrics used were divided into two groups, procedural metrics (apply equally to procedural and OO languages), and OO metrics (apply only to OO paradigm). Two three-group predictive models have been developed with achieved accuracy of 64.7% and 69.1% respectively.

4.2.2 Machine Learning Algorithms

Almeida et al. [4] performed an empirical study in which they investigated different machine learning (ML) algorithms with regard to their capabilities to generate accurate corrective maintenance estimation/evaluation models. In general, ML algorithms use an attribute-value representation language that allows the use of statistical properties on the learning set. Nevertheless, others use the first order language that permits the expression of relations between objects, thus having better expressive capabilities than the attribute-value language. In their study, Almeida et al. used four very well known, public-domain ML algorithms (NewID, CN2, C4.5, FOIL) belonging to three different families of ML techniques (divide and conquer, covering family, inductive logic programming). Data used in the study have been collected from the a library of reusable components, known as, Generalized Support Software (GSS) reuse asset library, located at the Flight Dynamics Division of the NASA Goddard Space Flight Center. The asset library consisted of 1K Ada83 components totaling approximately 515 KSLOC. The results have shown that the predictive models generated by FOIL and C4.5 were far better than the models produced by NewID, CN2, and multivariate logistic regression combined with principle component analysis.

Another study has been performed by Basili et al. [9] approximately on the same data. They have analyzed GSS library of reusable components in order to construct a model for predicting the cost of rework. The C4.5 ML algorithm, has been used as a modeling technique. This logical classification technique has been chosen "because the models are straight forward to build and are also easy to interpret" [9]. The C4.5 algorithm partitions continuous attributes, in this case the internal product metrics, finding the best threshold among the set of training cases in order to classify them on the dependent variable, in this case *costly to rework / not costly to rework* classes. As well as being useful for prediction, the generated decision tree provides production rules characterizing components that fall into each one of the rework cost categories. Basili et al. evaluated the constructed two-

group model with respect to its correctness (69% *costly to rework* class) and completeness (71% *costly to rework* class).

A suite of 24 coupling measures have been proposed by Lounis et al. in [37], to quantify the level of coupling in modular software systems. The metrics then have been empirically validated through fault-proneness predictive modeling based on C4.5 ML algorithm. The data for the study have been collected from a medium size, industrial OO system written in C++, with approximately 47 KSLOC. The quality of the model built has been evaluated with respect to the overall accuracy, correctness, and completeness of the predictions. Evaluation of the model's accuracy points out how good the model is expected to be as a predictor. The achieved accuracy of the model is 78.82%, while the correctness and completeness for the fault-prone classes are 74% and 64% respectively.

The research works in [37], [9], and [4], mentioned in this subsection have mostly inspired our work. We compare our findings with the results from these three studies in Chapter 7. A summary of the background and related work in a table format can be found in the Appendix A. The foundations and intrinsic characteristics of C4.5 are described in more details in the next chapter.

Chapter 5

C4.5 Machine Learning Algorithm

Most applications of artificial intelligence to tasks of practical importance are based on constructing a model of the knowledge used by a human expert. In some cases, the task that an expert performs can be thought of as *classification*—assigning things to categories or classes determined by their properties. In a classification model, the connection between classes and properties can be defined by something as simple as a flowchart or as complex and unstructured as a procedures manual. There are two very different ways in which executable models—those that can be represented as computer programs—can be constructed. The model might be obtained by interviewing the relevant expert(s); most knowledge-based systems have been built this way. Alternatively, numerous recorded classifications might be examined and a model constructed inductively, by generalizing from specific examples.

C4.5 [43] is a set of computer programs based on a machine learning algorithm that constructs classification models of the second kind, i.e., by discovering and analyzing patterns found in records of heuristic data. The key requirements of the particular induction methods embodied in C4.5 program(s) are:

- *Attribute-value description*: The data to be analyzed must be what is sometimes called a flat file—all information about one object or *case* must be expressible in terms of a fixed collection of properties or *attributes*. Each attribute may have either discrete or numeric values, but the attributes used to describe a case must not vary from one case to another.

- *Predefined classes*: The categories to which cases are to be assigned must have been established beforehand. In the terminology of machine learning, this is *supervised* learning, as contrasted with unsupervised learning in which appropriate groupings of cases are found by analysis.

- *Discrete classes*: The classes must be sharply delineated—a case either does or does not belong to a particular class—and there must be far more cases than classes.

- *Sufficient data*: Inductive generalization proceeds by identifying patterns in data. Sometimes, robust patterns cannot be distinguished from chance coincidences. As this differentiation usually depends on statistical tests of one kind or another, there must be sufficient cases to allow these tests to be effective. The amount of data required is affected by factors such as the numbers of properties and classes, and the complexity of the classification models; as these increase, more data will be needed to construct a reliable model.

- *Logical classification models*: C4.5 constructs only classifiers that can be expressed as *decision trees* or sets of *production rules*. These forms essentially restrict the description of a class to a logical expression whose primitives are statements about the values of particular attributes.

5.1 Divide and Conquer Method

Most of the works done in machine learning have focused on supervised ML algorithms [4]. In general, these algorithms use an attribute-value representation language. One of these attributes represents the class of the case. Two algorithm methods emerge in the attribute-value based family: the divide and conquer method, and the covering method.

C4.5 belongs to the family of algorithms that use divide and conquer method. In this family, the induced knowledge is, generally, represented by a decision tree. It is the case of algorithms like ID3 [42] (the direct ancestor of C4.5), ASSISTANT [23], etc. The principle of this approach, can be summarized by the following algorithm:

```
If      all the examples are of the same class
Then    - create a leaf labeled by the class name;
Else    - select a test based on one attribute,
        - divide the training set into subsets, each associated to
          one of the possible values of the tested attribute,
        - apply the same procedure to each subset;
Endif.
```

The key step of the algorithm above is the selection of the "best" attribute to obtain compact trees with high predictive accuracy. Information-based heuristics have provided effective guidance for the division process.

C4.5 induces *classification models*, also called *decision trees*, from data. ID3, C4.5's ancestor, works with a set of examples where each example has the same structure, consisting of number of attribute/value pairs. The problem is to determine a decision tree that on the basis of answers to questions about the non-class attributes correctly predicts the value of the class attribute. Usually, the class attribute takes only the values {true, false}, or {success, failure}, or something equivalent.

5.2 Decision Tree

A decision tree represents a structure that is either

- a *leaf*, indicating a class, or
- a *decision node* that specifies some test to be carried out on a single attribute value, with one branch and sub-tree for each possible outcome of the test.

A decision tree can be used to classify a case by starting at the root of the tree and moving through it until a leaf is encountered. At each nonleaf decision node, the case's outcome for the test at the node is determined and attention shifts to the root of the subtree corresponding to this outcome. When this process finally (and inevitably) leads to a leaf, the class of the case is predicted to be that recorded at the leaf.

A measure of entropy is used to measure how informative a node is [4]. In general, if we are given a probability distribution $P = (p_1, p_2, \ldots, p_n)$, then the *Information* conveyed by this distribution, also called the *Entropy of P*, is:

$$I(P) = -(p_1 * \log(p_1) + p_2 * \log(p_2) + \ldots + p_n * \log(p_n))$$

This notion is exploited to rank attributes and to build decision trees where at each node the attribute with greatest gain is located—not yet considered in the path from the root.

C4.5 introduces a number of extensions of the original ID3 algorithm. C4.5 accounts for unavailable values, continuous attribute value ranges, pruning of decision trees, rule derivation, and so on.

When building a decision tree, we can deal with training sets that have cases with unknown attribute values by evaluating the gain for an attribute by considering only the

cases where that attribute is defined. When using a decision tree, cases that have unknown attribute values are classified by estimating the probability of the various possible results.

If there is an attribute, say A_i, with continuous range, C4.5 proceeds as follows. Firstly, it examines the values for this attribute in the training set. Say, they are in increasing order, $V_1, V_2, ..., V_m$. Then, for each value V_j, $j = 1, ..., m$, the algorithm partitions the cases into those that have A_i values up to and including V_j, and those that have values greater than V_j. For each of these partitions, C4.5 computes the gain, and chooses the partition that maximizes the gain.

5.2.1 Pruning Decision Trees

The recursive partitioning method of constructing decision trees described in sections 5.1 and 5.2 will continue to subdivide the set of training cases until each subset in the partition contains cases of a single class, or until no test offers any improvement. The result is often a very complex tree that "overfits the data" by inferring more structure than is justified by the training cases. These complex trees are not only hard to comprehend, but also can actually have a higher error rate than simpler trees. Namely, when the training set has been split many times so that tests are selected from examination of a small subset of cases, several tests may appear equally promising and choosing a particular one of them has elements of randomness.

A decision tree is not usually simplified by deleting the whole tree in favor of a leaf. Instead, the idea is to remove parts of the tree that do not contribute to classification accuracy on unseen cases, producing something less complex and thus more comprehensible.

There are basically two ways in which the recursive partitioning method can be modified to produce simpler trees:

- deciding not to divide the training set any further, or
- removing retrospectively some of the structure built up by recursive partitioning.

The former approach, called *stopping* or *prepruning*, does not waste time assembling structure that is not used in the final simplified tree. The typical approach is to look at the best way of splitting a subset and to assess the split from the point of view of statistical significance, information gain, error reduction, or whatever. If this assessment falls below some threshold, the division is rejected and the tree for the subset is just the most appropriate leaf. However, such stopping rules are not easy to get write—too high a threshold can terminate division before the benefits of subsequent splits become evident, while too low a value results in little simplification. C4.5 follows the second approach in which an overfitted tree is pruned after it is grown.

Decision trees are usually simplified by discarding one or more subtrees and replacing them with leaves; as when building trees, the class associated with a leaf is found by examining the training cases covered by the leaf and choosing the most frequent class. In addition, C4.5 allows replacement of a subtree by one of its branches. Suppose that it was possible to predict the error rate of a tree and of its subtrees (including leaves). We can trim the tree using the following method: start from the bottom of the tree and examine each nonleaf subtree. If replacement of this subtree with a leaf, or with its most frequently used branch, would lead to a lower predicted error rate, then prune the tree accordingly, remembering that the predicted error rate for all trees that include this one will be affected. Since the error rate for the whole tree decreases as the error rate of any of its subtrees is reduced, this process will lead to a tree whose predicted error rate is minimal with respect to the allowable forms of pruning.

Two families of techniques can be used in order to predict the error rate. The first family predicts the error rate of the tree using a new set of cases that is distinct from the training

set. Since these cases were not examined at the time the tree was constructed, the estimates obtained from them are clearly unbiased and, if there are enough of them, reliable. The drawback associated with this family of techniques is that some of the available data must be reserved for the separate set, so the original tree must be constructed from a smaller training set. This may not be a disadvantage when the data are abundant, but can lead to inferior trees when data are scarce.

The approach taken in C4.5 belongs to the second family of techniques that use only the training set from which the tree was built. The raw resubstitution estimate of error rate is adjusted to reflect this estimate's bias. When a leaf covers N training cases, E of them incorrectly, the resubstitution error rate for this leaf is E/N. We could look at this as "observing E events in N trials". If we look at this set of N training cases as a sample, we could ask what this result tells us about the probability of an event (error) over the entire population of cases covered by this leaf. The probability error cannot be determined exactly, but has itself a posterior probability distribution. For a given confidence level CF, the upper limit on this probability can be found from the confidence limits for the binomial distribution; here written as $U_{CF}(E, N)$. Then, C4.5 simply equates the predicted error rate at a leaf with this upper limit, on the argument that the tree has been constructed to minimize the observed error rate. To simplify the accounting, error estimates for leaves and subtrees are computed assuming they were used to classify a set of unseen cases of the same size as the training set. So, a leaf covering N training cases with a predicted error rate of $U_{CF}(E, N)$ would give rise to predicted $N \times U_{CF}(E, N)$ errors. Similarly, the number of predicted errors associated with a (sub)tree is just the sum of the predicted errors of its branches.

Figure 5.1 shows a decision tree[3] derived from congressional voting data[4] before pruning.

[3] This example is included in the sample data directory of the original C4.5 software package.
[4] This set of data, collected by Jeff Schlimmer, records the votes of all United States congressmen on 16 key issues selected by the *Congressional Quarterly Almanac* for the second session of 1984.

```
Decision Tree:

physician fee freeze = n:
|   adoption of the budget resolution = y: democrat (151.0)
|   adoption of the budget resolution = u: democrat (1.0)
|   adoption of the budget resolution = n:
|   |   education spending = n: democrat (6.0)
|   |   education spending = y: democrat (9.0)
|   |   education spending = u: republican (1.0)
physician fee freeze = y:
|   synfuels corporation cutback = n: republican (97.0/3.0)
|   synfuels corporation cutback = u: republican (4.0)
|   synfuels corporation cutback = y:
|   |   duty free exports = y: democrat (2.0)
|   |   duty free exports = u: republican (1.0)
|   |   duty free exports = n:
|   |   |   education spending = n: democrat (5.0/2.0)
|   |   |   education spending = y: republican (13.0/2.0)
|   |   |   education spending = u: democrat (1.0)
physician fee freeze = u:
|   water project cost sharing = n: democrat (0.0)
|   water project cost sharing = y: democrat (4.0)
|   water project cost sharing = u:
|   |   mx missile = n: republican (0.0)
|   |   mx missile = y: democrat (3.0/1.0)
|   |   mx missile = u: republican (2.0)
```

Figure 5.1. Original decision tree before pruning

The original decision tree consists of 25 branches and 17 leaves. The number (N / E) appearing after a leaf indicates that the leaf covers N training cases, E erroneously. For example, the following path:

```
physician fee freeze = y:
|   synfuels corporation cutback = y:
|   |   duty free exports = n:
|   |   |   education spending = n: democrat (5.0/2.0)
```

Figure 5.2. A path and a leaf—associated with the number (N / E)

covers 5 training cases to the final leaf, of which 2 has been classified wrongly, *republican* instead of *democrat*.

The pruned decision tree, presented in Figure 5.3 has only 7 branches and 5 leaves. The number (N / E) appearing after a leaf indicates, as before, that the leaf covers N training

cases, but *E* is not the exact number of erroneously classified cases. E is rather *estimated pessimistic error rate* calculated during the pruning process. As can be seen, most of the values of E are float numbers.

```
Simplified Decision Tree:

physician fee freeze = n: democrat (168.0/2.6)
physician fee freeze = y: republican (123.0/13.9)
physician fee freeze = u:
|   mx missile = n: democrat (3.0/1.1)
|   mx missile = y: democrat (4.0/2.2)
|   mx missile = u: republican (2.0/1.0)


Evaluation on training data (300 items):

       Before Pruning           After Pruning
       ----------------   ---------------------------
       Size      Errors   Size      Errors   Estimate

         25    8( 2.7%)      7   13( 4.3%)    ( 6.9%)   <<

Evaluation on test data (135 items):

       Before Pruning           After Pruning
       ----------------   ---------------------------
       Size      Errors   Size      Errors   Estimate

         25    7( 5.2%)      7    4( 3.0%)    ( 6.9%)   <<
```

Figure 5.3. A decision tree after pruning with estimated error rates

The summary of the results for both training and test data is also presented in Figure 5.3. The *Error* is simply a number of wrongly classified cases; the percentage regarding overall population is included in the brackets. The sum of the predicted errors at the leaves, divided by the number of cases in the training set, provides an immediate estimate of the error rate of the pruned tree on unseen cases. For this tree, the sum of the predicted errors at the leaves is 20.8 for a training set of 300. By this estimate, then, the pruned tree will misclassify 6.9% of unseen cases. For this particular set of data, the error rate of the pruned tree is higher than that of the original tree for the training data, but, as hoped, the pruned tree has lower error rate than the original tree on the unseen cases (test data).

5.3 Production Rules

The development of accurate predictors—although is certainly a key concern—is not the only purpose of constructing classification models. Another principle aim is that the model should be intelligible to human beings.

As we saw in the previous section, it is often possible to prune a decision tree so that it is both simpler and more accurate. The simplified decision tree in Figure 5.3 is so compact that it can be readily understood. However, when classification tasks become more intricate, even simplified trees can grow to unwieldy proportions. Large trees are difficult to understand because each node has a specific context established by the outcomes of tests at antecedent nodes. Consider the leaf in the path of Figure 5.2, which is derived from the last node of the second subtree of Figure 5.1. That node tests *education spending*, giving class *democrat* if the answer is *n*, but it does not suggest if this test is self sufficient to decide the outcome of the answer. This test makes sense only when read in conjunction with the outcomes of earlier tests along the path. Every test in the tree has a unique context that is crucial to its understanding and it can be very difficult to keep track of the continually changing context while scanning a large tree.

C4.5 avoids this comprehension barrier by re-expressing a classification model as *production rules*, a format that appears to be more intelligible than decision trees. In any decision tree, the conditions that must be satisfied when a case is classified by a leaf can be found by tracing all the test outcomes along the path from the root to the leaf. In the Figure 5.2 the *democrat* leaf is associated with the outcomes *physician fee freeze* = *y*, *synfuels corporation cutback* = *y*, *duty free exports* = *n*, and *education spending* = *n*. Therefore, we can write the rule:

If physician fee freeze = y
synfuels corporation cutback = y
duty free exports = n
education spending = n
Then class *democrat*

with the understanding that the conditions making up the rule antecedent are to be considered as a conjunction. We say that a rule *covers* a case if the case satisfies the rule's antecedent conditions.

```
Rule 1:
      physician fee freeze = n
      ->  class democrat  [98.4%]

Rule 2:
      synfuels corporation cutback = y
      duty free exports = y
      ->  class democrat  [97.5%]

Rule 3:
      water project cost sharing = y
      physician fee freeze = u
      ->  class democrat  [70.7%]

Rule 4:
      physician fee freeze = y
      ->  class republican  [88.7%]

Rule 5:
      physician fee freeze = u
      mx missile = u
      ->  class republican  [50.0%]

Default class: democrat
```

Figure 5.4. Production rules for the decision tree from Figure 5.1

Rewriting the tree to a collection of rules, one for each leaf in the tree, would not result in anything much simpler than the tree, since there would be one rule for every leaf. However, the antecedents of individual rules may contain irrelevant conditions.

```
F = 0:
|   J = 0: no
|   J = 1:
|   |   K = 0: no
|   |   K = 1: yes
F = 1:
|   G = 0: no
|   G = 1:
|   |   J = 0: no
|   |   J = 1:
|   |   |   K = 0: no
|   |   |   K = 1: yes
```

Figure 5.5. Simple decision tree for F=G=1 or J=K=1

In the tree of Figure 5.5 the deepest *yes* leaf is associated with the outcomes *F=1*, *G=0*, *J=1*, and *K=1*; any case that satisfies these conditions will be mapped to that *yes* leaf. So, we can write the rule:

If F = 1
G = 0
J = 1
K = 1
Then class *yes*

In the rule above—taking the tree of Figure 5.5 into consideration—the conclusion is unaffected by the values of *F* and *G*. The rule can be generalized by deleting these superfluous conditions without affecting its accuracy, leaving the more appealing and understandable rule

If J = 1
K = 1
Then class *yes*.

Let rule *R* be:

if A then class C,

and a more general rule R^-:

if A^- *then class C*

where A^- is obtained by deleting one condition *X* from the set of conditions *A*. In section 5.2.1 we described how C4.5 prunes decision trees and estimates the *pessimistic error rate*. The same approach is used for simplification of production rules. If the pessimistic error rate of rule R^- is not greater than that of the original rule *R*, then it makes sense to delete condition *X*. Rather then looking at all possible subsets of conditions that could be deleted, the system carries out a straightforward greedy elimination: the condition whose removal produces the lowest pessimistic error rate is deleted first. As with all greedy searches, there is no guarantee that minimizing pessimistic error rate at each step will lead to global minimum. However, this technique works reasonably well in practice, and is relatively fast [43].

Let us summarize, it is easy to derive a rule set from a decision tree by writing a rule for each path in the decision tree from the root to a leaf. The left-hand side (LHS) of the rule contains all the conditions (nodes) established by the path, and the right-hand side specifies the class at the leaf. The resulting rule set can be simplified: let LHS' be obtained from LHS by eliminating some of its conditions. LHS' can certainly replace LHS of the rule if the subsets of the training set that satisfy LHS and LHS' respectively, are equal [4]. The Figure 5.4 presents the production rules set for the decision tree of Figure 5.1. The system chooses as the default that class which contains the most training cases not covered by any rule, resolving ties in favor of the class with the higher absolute frequency.

5.4 Conducting Experiments—Modeling

C4.5 programs come with several options that can enhance not only the process of model construction, but also the predictive power of the model. The question: "What is the best combination of options that fine tune the predictive models?" is rather subtle. For what kinds of tasks are decision tree methods in general, and C4.5 system in particular, likely to be appropriate? The answer is related to the application domain. Of course, a trial and error method never hurts. However, a better understanding of the relationships between tasks and methods can reduce the time for model construction and refine model's predictability.

5.4.1 Windowing (-t)

A subset of the training set called a *window* can be selected randomly in order to grow a decision tree. This tree can then be used to classify the training cases that have not been included in the window, usually some of them would be misclassified. A selection of

these *exceptions* would be added to the initial window, and a second tree, constructed from the enlarged training set, would be tested on the remaining cases. This cycle is repeated until a tree built from the current window, correctly classifies all the training cases outside the window.

C4.5 improves the windowing strategy described above—originally applied in ID3—in several ways. Firstly, C4.5 biases the choice of training cases for the initial window so that the distribution of classes is as uniform as possible. This type of sampling leads to better initial trees when the distribution of classes is unbalanced. Secondly, C4.5 includes at least half of the exceptions in the next window, thereby speeding the convergence on a final tree. Thirdly, the program stops before the tree correctly classifies all cases outside the window if the sequence of trees is not becoming more accurate.

The principle benefit of the windowing option is that it can lead to more accurate trees. Recall that the initial window is still selected randomly, although subject to making the class distributions as uniform as possible. Different initial windows generally lead to different initial trees. So, even though the training set is not changed, windowing allows different final trees to be constructed. This provides the following two features:

- Growing several trees and selecting as "the" tree the one with the lowest predicted error rate, and
- Generating production rules for each of the trees grown, then constructing a single production rule classifier from all the available rules.

The final classifier obtained in this way is often more accurate than one obtained via a single tree. The downside is that growing n trees requires n times as long as developing one of them; same for production rules.

5.4.2 Grouping Attribute Values (-s)

When the divide-and-conquer method discussed in section 5.1 chooses to split the training set on a discrete value attribute, it generates a separate branch and subtree for each possible value of that attribute. In order to reduce the number of outcomes from testing a multivalued attribute, one or more outcomes must be associated with a collection of attribute values rather than a single value. Collections of attribute values associated with a single branch will be referred to as *value groups*, not to be confused with subsets of training cases. In some domains, appropriate groups of attribute values can be determined from domain knowledge. Consider the case of an attribute denoting an employee:

- Grouped by title; employees can be *administrators*, *analysts*, *managers*, *directors*, and so on.
- Some employees have a *masculine* gender, the others have the *feminine* one.
- Employees can work *full time*, *part time*, on a *contract*, etc.

Any or all of these groupings of elements can be relevant to the classification task at hand. Where such well-established groupings are known beforehand, it makes sense to provide this information to the system by way of additional attributes, one for each potentially relevant grouping. So, in this case, we might add a multivalued attribute to indicate the title of the employee, another true-false attribute to indicate his/her gender, another multivalued attribute to indicate job status, etc.

5.4.3 Weight Option (-m)

Near-trivial tests in which almost all the training cases have the same outcome can lead to odd trees with little predictive power. To avoid this situation, C4.5 requires that any test used in the tree must have at least two outcomes with a minimum number of cases. The default minimum is 2; higher value should be tried where there is a lot of noisy data.

5.4.5 Confidence Factor (-c)

The *confidence factor* (*CF*) value affects decision tree pruning. The default value (25%) works well for many tasks [42], but should be changed to a lower value if the actual error rate of pruned trees on test cases is much higher than the estimated pessimistic error rate—indicative of underpruning. Small values cause heavier pruning, with a higher effect on small sets of data.

5.4.6 Cross-validation

The obvious method for estimating the reliability of a classification model is to divide the data into a training and test set, build the model using only the training set, and examine its performance on the unseen test cases. This is satisfactory when there are plenty of data, but in the more common circumstance of having less data than we would like[5], two problems arise. First, in order to get a reasonably accurate fix on error rate, the test set must be large, so the classification power of the training set becomes worse. Second, when the amount of data available is moderate, different divisions of the data into training and test sets can produce surprisingly large variations in error rates on unseen cases.

A more robust estimate of accuracy on unseen cases can be obtained by *cross-validation*. In this procedure, the available data is divided into *N* blocks so as to make each block's number of cases and class distribution as uniform as possible. *N* different classification models are than built, in each of which one block is omitted from the training data, and the resulting model is tested on the cases in that omitted block. This way, each case appears in exactly one test set. Provided that *N* is not too small—10 is a common

[5] Our research is based on a set of 83 cases, classes from the investigated C++ OO system.

number—the average error rate over the *N* unseen test sets is a good predictor of the error rate of a model built from all the data[6].

5.4.7 Building the Best Model

It is necessary to conduct a lot of experiments in order to find the combination of options that eventually leads toward the best predictive model. As mentioned before, the best models are strongly related with the application's domain. A combination of options that builds the best model for application *A* could lead toward the worst model for application *B*. We investigate the results of both decision trees and production rule classifiers for each of the system's runs. The quality of the model construction process is captured with the following measures:

a) Decision Tree

- *Size* of the decision tree; denoted with two values:
  - i) *before*: number of branches in the tree before the pruning, and
  - ii) *after*: number of branches in the tree after the pruning
- *Error rate*: number of cases that were misclassified. Three types of errors are considered:
  - i) *Train set error rate*: is the average error rate of the training set.
  - ii) *Unseen error rate*: is the average (observed) error rate of the test set.
  - iii) *Predicted error rate*: is the estimated pessimistic error rate for the unseen cases.

[6] This gives a slight overestimate of the error rate, since each of the *N* models is constructed from a subset of the data.

b) Production Rules

- *Error rate*: number of cases that were misclassified. Two types of errors are considered:
    i) *Train set error rate*: is the average error rate of the training set.
    ii) *Unseen error rate*: is the average error rate of the test set.

We are interested in the model with the lowest error rates, both observed and predicted.

We have applied the following 4-step procedure while searching for the best models related to our work:

1. Use ten-way cross-validation with all default options in the first run of the system, as a sighting shot.
2. Evaluate the quality of the constructed model with respect to the above criteria.
3. Run the system with options based on specific knowledge domain for the application and/or your intuition. We used the options in Table 5.1.
4. Repeat steps 2 and 3 until satisfied. Then build the actual production rule classifier by running the system with the selected best combination of options.

Option	Meaning
default	
s	Attribute value grouping option.
m5	Weight option. Any test used in the tree must have at least 5 outcomes.
c15	Confidence Factor option. CF values affect decision tree pruning, small values cause heavier pruning. The default value is 25%.
c10	CF option.
c15 m5	A combination of pruning (CF) and weight options.
t10	Windowing option. The best tree will be chosen from 10 previously grown decision trees.
t15	Windowing option.
t5	Windowing option.
t5 c15	A combination of windowing and pruning (CF) options.
t15 c10	A combination of windowing and pruning (CF) options.
m10	Weight option.
m15	Weight option.

Table 5.1. Options for constructing the best predictive models

In order to evaluate and compare the results from each system run we use Table 5.2. The columns present the measures needed to identify the quality of the model construction process, while the rows present the results regarding the options from Table 5.1.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate

Table 5.2. A form for evaluation and comparison of model construction results

We explain the data preparation and the format of C4.5 input files in the next chapter within the context of the experimental framework for our research.

Chapter 6

Case Study Framework

6.1 The C++ System

In our study we have used the data from a medium sized, industrial, open multi-agent system development environment, called LALO[7]. The system has been developed and maintained since 1993 at CRIM[8]. It contains 90 C++ components/classes totaling approximately 57K source lines of code (SLOC). So far, the system had five releases. The last one—Release 1.3—has been delivered on July 1997. At the time of this thesis writing, LALO had more than 100 registered users in 21 countries worldwide.

The developers of the investigated system were very experienced programmers. Each one of them had already worked on large industrial software development projects, had had significant experience regarding OO analysis/design methods, and had been very familiar with C/C++ programming language.

---

[7] Langage d'Agents Logiciel Objet (Agent Oriented Programming Language). At the time of the writing, further information could be found at the URL: http://www.crim.ca/sbc/english/lalo.

[8] Centre de recherche informatique de Montréal (Computer Science Research Institute of Montreal), Montréal, Québec, Canada.

6.2 Hypotheses

Fault-proneness is a software quality attribute that is domain dependent. It can be, also, heavily influenced by numerous factors such as corporate environment, experience of the developers, processes, methods, and tools used, and training [7]. Unfortunately, the complexity of the phenomena frequently obscures the identity and impact of such factors in any given software development organization. The resulting uncertainty about productivity and quality in the next software release gives rise to unreliable cost and schedule estimates.

The aim of our research is not to reveal a general method that universally relates all the software development process aspects with fault-proneness as the final product's quality attribute. Our goal is to investigate how much fault-proneness is influenced by internal (e.g. cohesion) and external (e.g., coupling, inheritance) design characteristics of OO classes, developed in C++ programming language. Do these (assumed) relationships have a practical meaning? Can we use them as a forecasting means in order to improve the overall software development process? In the continuation, we propose three hypotheses that relate OO design properties—inheritance, cohesion, and coupling—with fault-proneness as a software quality indicator.

6.2.1 Inheritance vs. Fault-proneness

In section 2.2 OO Paradigm, we pointed out, that the four major elements of the *object model*—the conceptual framework for the OO programming style—are (1) abstraction, (2) encapsulation, (3) modularity, and (4) hierarchy. Hoare suggests that "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the

time being the differences" [14]. Encapsulation helps manage the complexity captured by the abstractions, by hiding their inside views. Modularity helps also, by giving us a way to cluster logically related abstractions. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design we greatly simplify our understanding of the problem. The two most important hierarchies in a complex OO system are its class structure (the "kind of" hierarchy) and its object structure (the "part of" hierarchy).

Inheritance is the most important "kind of" hierarchy, and it is an essential element of OO systems. Inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (*single inheritance* and *multiple inheritance* respectively). Inheritance represents a hierarchy of abstractions, in which a subclass inherits from one or more superclasses. As we evolve our inheritance hierarchy, the structure and behavior that are the same for different classes will tend to migrate to common superclasses. Superclasses are generalized abstractions, and subclasses represent specializations in which fields and methods from the superclass are added, modified, or even hidden.

It seems logical that the lower a class is in the inheritance tree, the more superclass properties this class may access because of its inheritance, thus more opportunities a fault to be introduced in the class. On the other hand, the more children, or descendants a class has, the more (other) classes it may potentially affect because of inheritance. For example, if there are many subclasses of the class that are dependent on some methods or instance variables defined in the superclass, any changes to these methods or variables may affect the subclasses. Thus it becomes harder to maintain the class [36]. We propose the following hypothesis:

Hypothesis 1: *The position of the component in the class hierarchy of the OO system affects its fault-proneness.*

6.2.2 Cohesion vs. Fault-proneness

High quality software design, among many other principles, should obey the principle of high cohesion [24], [19]. Stevens, Myers and Constantine, who first introduced cohesion in the context of structured development techniques, define cohesion as "a measure of the degree to which the elements of a module belong together". Cohesion refers to the internal consistency within parts of the design. The higher the cohesion of a module, the easier the module is to develop, maintain, and reuse, and the less fault-prone it is [19], [20]. Cohesion captures the extent to which, in a software part, each group of data declarations and subroutines that are conceptually related belong to the same module. A high degree of cohesion is desirable because information related to declaration and subroutine dependencies should not be scattered across the system and among irrelevant information [20]. Data declarations and subroutines, which are not related to each other, should be encapsulated to the extent possible into different modules. With such a strategy, we expect the software components to be less fault-prone. We propose the following hypothesis:

Hypothesis 2: *The degree to which the component is cohesive affects its fault-proneness.*

6.2.3 Coupling vs. Fault-proneness

Coupling refers to the degree of interdependence between parts of the design [24]. Originally, Stevens, Myers and Constantine have defined it as "a measure of the strength of association established by a connection from one module to another". High quality software design should obey the principle of low coupling [14], [24], [18], [17], [37]. The stronger the coupling between modules, i.e., the more inter-related they are, the more

difficult these modules are to understand, change and correct, and thus the more complex the resulting software system.

In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum [24]. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore the class is likely to be more fault-prone.

If a large number of methods of a class can be invoked in response to a message received by object of that class, the understanding of that class becomes more complicated. This, also, implies greater complexity of the class thus the class is more likely to be fault-prone.

The higher the export coupling of class c, the greater the impact of a change to c on other classes [17]. Many classes depend critically on the design of c, thus there is greater likelihood of failures being traced back to faults in c. The higher the import coupling of a class c, the greater the impact of change in other classes on c itself. Thus class c depends critically on many other classes, and therefore: (1) understanding c may be more difficult hence more fault-prone, (2) coupled classes are more likely to be misunderstood and therefore misused by c. We propose the following hypothesis:

Hypothesis 3: *The degree of interdependence between the component and the environment within which it is defined affects its fault-proneness.*

6.3 Data Collection

The actual data required for our study were collected directly from the source code. The data preparation has consisted of the extraction of the selected suite of measures related to inheritance, cohesion, and coupling OO design properties, as well as, the extraction of various defect data attributes. Note that the measures were derived purely by static analysis of the investigated C++ system. Only the classes developed by the LALO team, have been utilized (83 components). Classes generated by software tools (seven components) have not been used in the study due to the impact that software reuse and code generators have on software quality [8].

6.3.1 Selected OO Design Metrics

We have already discussed the OO design metrics whose extraction is supported by M-System automated tool (section 4.1 "Some OO Design Metrics"). Therefore, herein we only enumerate those metrics selected as suitable for the evaluation of our hypotheses.

Hypothesis 1: *The position of the component in the class hierarchy of the OO system affects its fault-proneness.* All the class level, inheritance metrics—defined in the subsection 4.1.2—have been selected.

- ***DIT*(*c*), depth of inheritance tree.**
- ***AID*(*c*), average inheritance depth.**
- ***CLD*(*c*), class to leaf depth.**
- ***NOC*(*c*), number of children.**
- ***NOP*(*c*), number of parents.**

- ***NOD*(*c*), number of descendants.**
- ***NOA*(*c*), number of ancestors.**
- ***NMO*(*c*), number of methods overridden.**
- ***NMI*(*c*), number of methods inherited.**
- ***NMA*(*c*), number of methods added, new methods.**
- ***SIX*(*c*), specialization index.**

Hypothesis 2: *The degree to which the component is cohesive affects its fault-proneness.* All metrics related to cohesion —defined in the subsection 4.1.3—have been selected.

- $LCOM_1(c)$ **, lack of cohesion in methods.**
- $LCOM_2(c)$ **, lack of cohesion in methods.**
- $LCOM_3(c)$ **, lack of cohesion in methods.**
- $LCOM_4(c)$ **, lack of cohesion in methods.**
- $LCOM_5(c)$ **, lack of cohesion in methods.**
- ***Co*(*c*), connectivity.**
- ***TCC*(*c*), tight class cohesion.**
- ***LCC*(*c*), loose class cohesion.**
- ***ICH*(*c*), information flow based cohesion.**

Hypothesis 3: *The degree of interdependence between the component and the environment within which it is defined affects its fault-proneness.* All coupling related metrics—defined in the subsection 4.1.4—have been selected.

- ***CBO*(*c*), coupling between object classes.**
- $CBO'(c)$ **, coupling between object classes.**
- $RFC_1(c)$ **, response for class.**
- $RFC_\infty(c)$ **, response for class.**
- ***MPC*(*c*), message passing coupling.**
- ***DAC*(*c*), data abstraction coupling.**

- $DAC'(c)$, **data abstraction coupling.**
- ***ICP(c)*, information flow based coupling.**
- ***NIH-ICP(c)*, information flow based non-inheritance coupling.**
- ***IH-ICP(c)*, information flow based inheritance coupling.**
- ***IFCAIC(c)*, inverse friends class-attribute import coupling.**
- ***ACAIC(c)*, ancestors class-attribute import coupling.**
- ***OCAIC(c)*, others class-attribute import coupling.**
- ***FCAEC(c)*, friends class-attribute export coupling.**
- ***DCAEC(c)*, descendants class-attribute export coupling.**
- ***OCAEC(c)*, others class-attribute export coupling.**
- ***IFCMIC(c)*, inverse friends class-method import coupling.**
- ***ACMIC(c)*, ancestors class-method import coupling.**
- ***OCMIC(c)*, others class-method import coupling.**
- ***FCMEC(c)*, friends class-method export coupling.**
- ***DCMEC(c)*, descendants class-method export coupling.**
- ***OCMEC(c)*, others class-method export coupling.**
- ***IFMMIC(c)*, inverse friends method-method import coupling.**
- ***AMMIC(c)*, ancestors method-method import coupling.**
- ***OMMIC(c)*, others method-method import coupling.**
- ***FMMEC(c)*, friends method-method export coupling.**
- ***DMMEC(c)*, descendants method-method export coupling.**
- ***OMMEC(c)*, others method-method export coupling.**

6.3.2 Defect Data

In our study we collected error and fault data about the investigated C++ system. An *error* is a human action that results in a software product that contains a *fault*. Errors are defects in the thought process made while trying to solve a problem, based on the understanding of the given information and personal level of expertise. Faults are concrete manifestations of errors within the product. As a consequence of one or more failures of the software, a fault would be discovered and a physical *change* in one or more defective system components would be made. In order to document the error and effect the change the developers would fill a single software Change Request Form (CRF).

The CRF was used to gather data on: (1) error identification—including short description about the nature of the problem, (2) the names and version numbers of the C++ faulty components affected by the change, (3) type, location and the origin of the maintenance change, (4) actual parts of the components' source code modified, and (5) the overall effort taken to repair them.

Each CRF has been registered via Microsoft Visual Source Safe program. We built our own set of tools in order to classify the defect data on the component level as well as to generate corresponding size and effort metrics. The following development/maintenance change types have been considered [15]:

- **Error correction:** correct faults in developed/delivered system.
- **Enhancement:** improve performance or other system attributes, or add new functionality.
- **Adaptation:** adapt system to a new environment, such as new operating system.

Given the fact that we are interested in building predictive models that will identify which components are likely to have faults, we defined our dependent variable fault-proneness as "a number of *corrective* type of changes".

6.4 Dependent and Independent Variables

The terms *dependent* and *independent* variable apply mostly to experimental research where some variables are registered/manipulated (selected OO design metrics), and in this sense they are "independent" from the initial reaction patterns, features, etc. of the cases. Some other variables are expected to be "dependent" (fault-proneness) on the manipulation or experimental conditions. These terms are also used in studies where the researcher does not literally manipulate independent variables, but only assign cases to "experimental groups" based on some preexisting properties of the cases.

We built two types of predictive classification models—two-group and three-group models—in order to satisfy step 3 of the section 3.1 "Solution Strategy". To build two-group classification model(s) we had to dichotomize the components regarding their fault-proneness into two categories:

- *non-faulty*—there was not any change of corrective type for the component, and
- *faulty*—there were one or more changes of corrective type during the development/maintenance phase.

Similarly, to build three-group classification model(s) we dichotomized the components regarding their fault-proneness into three categories:

- *non-faulty*—there was not any change of corrective type for the component,
- *low-risk*—there were at least one, but less than four changes of corrective type during the development/maintenance phase, and
- *high-risk*—there were more than four changes of corrective type during the development/maintenance phase

Our decisions, with respect to the dichotomies mentioned above, have been guided by the distribution of faults (corrective type of changes) in the investigated C++ system, presented in Figure 6.1.

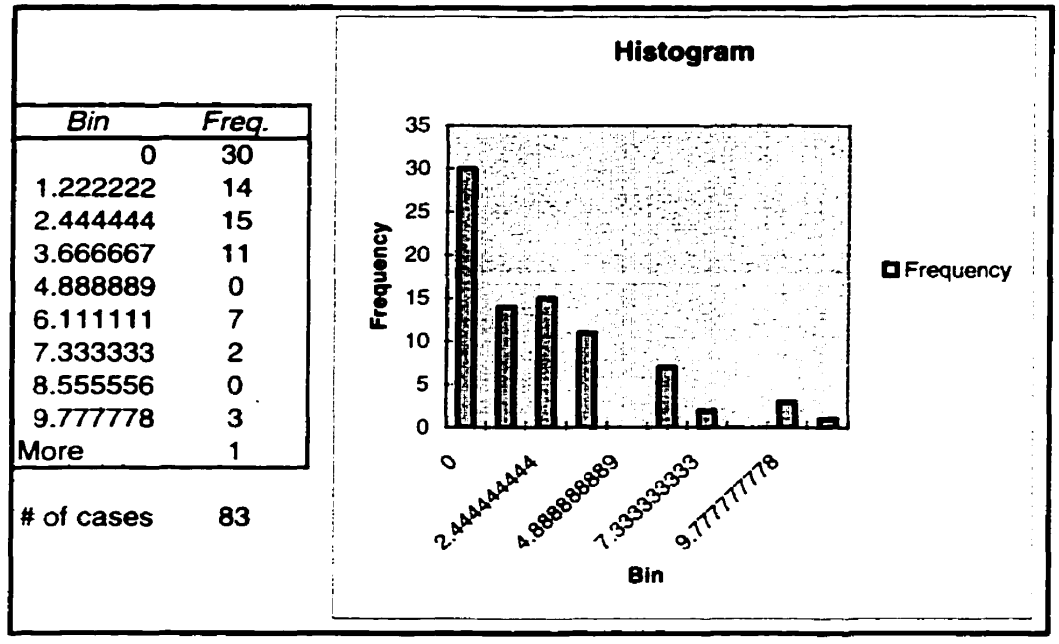


Figure 6.1. Defining fault-proneness (a histogram of faults in LALO).

We constructed eight predictive classification models regarding fault-proneness of the components. We built two types of models (two-group and three-group), related to three design properties of OO systems (inheritance, cohesion, and coupling) on their own, and finally, we built multivariate models combining all three design properties together.

The models that we developed identify C++ components that are likely to be faulty, rather than trying to predict the exact number of faults in the components.

In our research we, also, built predictive classification models for another dependent variable—*defect density*. Generally, defect density can be defined as a number of changes in the component divided by the number of lines of code in the component. We will not analyze and comment the results regarding the defect density, since our thesis is devoted to fault-proneness. However, in order to provide the software engineering community with additional empirical evidence regarding the modeling technique we have chosen, selected OO design metrics and dependent variables, we will incorporate the results related to defect density in the Appendix B.

We defined defect density in three different ways:

- *Defect density A*: a number of changes in the component divided by the total number of source lines of code in the component,
- *Defect density B*: a number of changes in the component divided by the number of statements[9] and the number of comments in the component, and
- *Defect density C*: a number of changes in the component divided by number of statements in the component.

We constructed 24 defect density predictive models, based on: two dichotomy types, three definitions of defect density, and three OO design properties and their combination.

6.5 Evaluation and Validation of the Models

In order to evaluate the quality of the classification models built, we need formal measures that comprise objective set of standards. Evaluating model accuracy tells us how good the model is expected to be as a predictor [4]. The high accuracy of the predictive model means that the selected OO design measures have been useful for identification of likely to be faulty components. In such a case, the model serves as a means for empirical validation of the selected metrics as independent (predictor) variables. Two criteria for evaluating the accuracy of predictions are the measures of *correctness* and *completeness*.

- *Correctness*: is defined as the percentage of components that were predicted as belonging to certain classification group (i.e., non-faulty, low-risk, high-risk) and actually did belong to that classification group. We want to maximize the correctness because if correctness is low, the model is identifying, for example,

[9] The term statement denotes the executable lines of code.

more components as being non-faulty when they really are faulty (or low-risk, high-risk) [9].

- *Completeness*: is defined as the percentage of those components that belonged to certain classification group (i.e., non-faulty, low-risk, high-risk) and were identified by the model. We want to maximize the completeness because if completeness is low, then more components that were likely to be faulty will not be identified.

		Predicted			
		class 1	class 2	class 3	Completeness
Real	class 1	n_{11}	n_{12}	n_{13}	$n_{11} / \sum_{j=1}^{3} n_{1j}$
	class 2	n_{21}	n_{22}	n_{23}	$n_{22} / \sum_{j=1}^{3} n_{2j}$
	class 3	n_{31}	n_{32}	n_{33}	$n_{33} / \sum_{j=1}^{3} n_{3j}$
	Correctness	$n_{11} / \sum_{i=1}^{3} n_{i1}$	$n_{22} / \sum_{i=1}^{3} n_{i2}$	$n_{33} / \sum_{i=1}^{3} n_{i3}$	

Table 6.1. Three-group classification model.

On the other hand, the overall *accuracy* of the model measures how correct is the model. It can be calculated by the following formula:

$$Accuracy = \frac{\sum_{i=1}^{3} n_{ii}}{\sum_{i,j=1}^{3} n_{ij}}$$

Another measure that can be used to evaluate the overall appropriateness of the model is a *goodness-of-fit* of data. This measure can be obtained via a *Chi-square* test. This test evaluates whether the expected cell frequencies under the respective model are

significantly different from the observed cell frequencies. We have used 2×2 tables[10] under "Nonparametric Statistics" menu of *STATISTICA*[11] software package in order to perform the test. The higher the value of Chi-square test the better the fit of data and the model. The corresponding *p-value* points out to the statistical significance of the test. The statistical significance of the result is an estimated measure of the degree to which it is "true", in the sense of "representative of the population". The p-value represents a decreasing index of the reliability of the result. The smaller the p-value the more significant the test—the more you can believe that the observed relation between variables in the sample is a reliable indicator of the relation between the respective variables in the population.

In order to calculate the values for the objective set of standards in question—correctness and completeness (described in Table 6.1), accuracy, and goodness-of-fit of data—we used a V-fold cross-validation procedure [37], [9], [4], [15], [46]. This validation procedure is commonly used when the data sets are small. In our study, we have used 10-way cross validation procedure, as it was described in the subsection 5.4.6. Furthermore, each of the predictive models has been built with the windowing option selected, as it was described in the subsection 5.4.1. It appeared that, growing 15 different decision trees in order to generate production rules for each of them, then constructing the final, single production rule classifier from all the available rules, was the best option for predictive modeling regarding our data set[12]. This way, each of the predictive models presented in the next chapter, is a result of 150 corresponding models.

[10] This option is often used as an alternative to correlation when the variables of interest are dichotomous.
[11] STATISTICA is a trade mark of StatSoft Inc.
[12] Empirical evidence for this statement can be found in the next chapter.

6.6 C4.5 Input Files

The Table 6.2 presents a portion of the input data regarding hypothesis 1, needed for the process of predictive model construction.

	Inheritance Design Metrics											Fault-proneness	
class	DIT	AID	CLD	NOC	NOP	NOD	NOA	NMO	NMI	NMA	SIX	2 X 2	3 X 3
Action	0	0	0	0	0	0	0	0	0	7	0	1	2
ActionsList	0	0	0	0	0	0	0	0	0	8	0	1	2
Agenda	0	0	0	0	0	0	0	0	0	12	0	1	2
ArgsSet	0	0	0	0	0	0	0	0	0	15	0	0	1
BasicAgent	0	0	2	1	0	2	0	0	0	37	0	1	3
Belief	0	0	0	0	0	0	0	0	0	19	0	1	3
BeliefAction	1	1	0	0	1	0	1	4	3	11	0.2222	0	1
BeliefCond	2	2	0	0	1	0	2	4	4	9	0.4706	1	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...

Table 6.2. Independent, and dependent variables.

In order to be meaningful to the C4.5 system, this information has to be converted into two files—*names* and *data*. The *names* file provides names for classes[13], attributes[14], and attribute values. It consists of a series of entries, each starting on a new line and ending with a period. The vertical bar character (|) appearing anywhere on a line causes the rest of that line to be ignored (useful for comments). The first entry in the *names* file gives the class names, separated by commas. There must be at least two class names (classes), and their order is not important. The rest of the file consists of a single entry for each attribute. An attribute entry begins with the attribute name followed by a colon and then a specification of the values that the attribute can take [43]. Four specifications are possible:

- *Ignore*: causes the value of the attribute to be disregarded.
- *Continuous*: indicates that the attribute has numeric values (integer, or float).

[13] Groups, or categories (dependent variable). Not to be mistaken with the term class used in OO systems.
[14] An attribute refers to an independent variable.

- *Discrete N*, where *N* is a positive integer: specifies that the attribute has discrete values, and there is no more than *N* of them.
- *A list of names separated by commas*: also indicates that the attribute has discrete values, and specifies them explicitly (i.e., cold, moderate, warm, hot, etc.).

```
1,2,3. | Fault-proneness 3x3

DIT: continuous. | Depth of Inheritance Tree
AID: continuous. | Height of Inheritance Tree
CLD: continuous. | Class-to-Leaf Depth
NOC: continuous. | Number of Children
NOP: continuous. | Number of Parents
NOD: continuous. | Number of Descendants
NOA: continuous. | Number of Ancestors
NMO: continuous. | Number of Methods Overridden
NMI: continuous. | Number of Methods Inherited
NMA: continuous. | Number of Methods Added
SIX: continuous. | Specialization Index
```

Figure 6.2. A *names* file (classes, attributes, and attribute values) for hypothesis 1.

The corresponding *data* file is used to describe the training cases from which the decision trees and/or production rules are to be constructed. Each line describes one case, providing the values for all the attributes (i.e., inheritance design metrics) and then the class of the case (i.e., fault-proneness), separated by commas, Figure 6.3. The attribute values must appear in the same order that the attributes were given in the *names* file. The order of the cases does not matter.

```
0,0,0,0,0,0,0,0,0,7,0,2
0,0,0,0,0,0,0,0,0,8,0,2
0,0,0,0,0,0,0,0,0,12,0,2
0,0,0,0,0,0,0,0,0,15,0,1
0,0,2,1,0,2,0,0,0,37,0,3
0,0,0,0,0,0,0,0,0,19,0,3
1,1,0,0,1,0,1,4,3,11,0.222222,1
2,2,0,0,1,0,2,4,4,9,0.470588,2
.....
```

Figure 6.3. A portion of *data* file, that corresponds to Table 6.2 and Figure 6.2.

Chapter 7

Experimental Results

In this chapter we present three types of experimental results. The first type of results is related to the search for the combinations of options that eventually lead towards the best predictive models, as it is discussed in the subsection 5.4.7. The second type of results presents actual predictive models (the C4.5 production rule classifiers). We discuss the relationships between the fault-proneness as software's quality indicator and each of the investigated OO design properties—inheritance, cohesion, and coupling. Finally, the third type of results helps us to evaluate, and to validate the quality of the constructed models, using the approach described in section 6.5. We also present the discovery, which we run across, while searching for the appropriate interpretation of the production rule classifiers. Namely, we found a way to quantify the contribution of each metric within certain model, and MLA, regarding its power to discriminate/classify the case on the dependent variable.

7.1 Selecting the Best Model Building Option

We constructed our predictive models after we selected the best model building options, by comparing the results of each system run presented in a format proposed in Table 5.2. The subsequent pairs of tables Table 7.1, 7.2, 7.3, 7.4, 7.5, and 7.6, record data for the two-group and the three-group models regarding hypotheses 1, 2, and 3, respectively.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate
default	20.8	16.4	9.3(12.5%)	*1.9(22.9%)*	27.90%	14.60%	*24.00%*
s	20.8	16.4	9.3(12.5%)	*1.9(22.9%)*	27.90%	14.60%	*24.00%*
m5	11.8	8.6	14.9(19.9%)	2.8(34.0%)	30.60%	20.90%	33.90%
c15	20.8	11.4	12.1(16.2%)	2.2(26.8%)	32.80%	15.40%	27.60%
c10	20.8	7.4	15.2(20.3%)	2.4(29.0%)	35.50%	15.80%	28.90%
c15 m5	11.8	7.8	15.3(20.5%)	2.8(34.2%)	34.50%	20.60%	35.10%
t10	20.8	16	*8.2(11.0%)*	2.0(24.2%)	26.20%	12.20%	27.60%
t15	**20.4**	**15.4**	**8.5(11.4%)**	**2.1(25.3%)**	**26.20%**	**11.90%**	**25.10%**
t5	21.2	16.2	8.4(11.2%)	2.1(25.4%)	26.50%	13.00%	26.50%
t5 c15	19.4	12.8	10.9(14.6%)	2.2(26.8%)	32.40%	13.10%	*24.00%*
t15 c10	19.8	12	10.9(14.6%)	2.5(30.1%)	34.60%	12.50%	27.80%
m10	6.2	4.4	19.1(25.6%)	2.9(35.3%)	33.10%	25.30%	36.40%
m15	5	3.6	21.6(28.9%)	3.1(37.6%)	35.90%	28.50%	31.80%

Table 7.1. Two-group model building options for hypothesis 1.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate
default	28.4	22.2	15.8(21.1%)	3.7(44.3%)	40.70%	24.90%	44.40%
s	28.4	22.2	15.8(21.1%)	3.7(44.3%)	40.70%	24.90%	44.40%
m5	14.6	7.4	26.0(34.8%)	4.3(51.5%)	45.10%	34.90%	53.80%
c15	28.4	16.2	18.7(25.0%)	*3.3(39.4%)*	46.00%	25.60%	44.30%
c10	28.4	14.8	19.5(26.1%)	*3.3(39.4%)*	49.20%	25.60%	44.30%
c15 m5	14.6	6.4	26.8(35.9%)	4.3(51.5%)	49.20%	34.90%	52.60%
t10	28.2	24.8	12.9(17.3%)	3.6(43.6%)	38.60%	21.40%	45.60%
t15	**27.6**	**24.8**	**12.9(17.3%)**	**3.4(41.1%)**	**38.50%**	**20.50%**	**41.80%**
t5	28.4	23.8	13.7(18.3%)	4.0(48.2%)	39.00%	22.60%	43.30%
t5 c15	28	15.8	17.9(24.0%)	3.5(42.2%)	44.80%	22.00%	43.00%
t15 c10	28	16.6	17.3(23.1%)	3.6(43.5%)	47.70%	22.20%	47.10%
m10	6.2	5.6	28.2(37.7%)	3.8(45.6%)	46.80%	37.70%	45.60%
m15	5.2	4.4	29.9(40.0%)	4.0(48.2%)	48.10%	39.80%	48.20%

Table 7.2. Three-group model building options for hypothesis 1.

The grouping option *s* obviously does not have any influence on building better models than the *default* models, for our data set. The weighting option *m*[*n*] produced even worse models, which leads to the conclusion that our data set has not been disposed to noise.

The pruning (CF) options *c*[*n*] performed better than the weighting options, but still worse than the *default*. For example, in hypothesis 1 three-group model the CF options *c15* and

c10 have the best decision tree unseen error rate, but are worse than the default models taking other parameters into account. The combination of the CF, and the weighting option *c15m5* gives better results than the weighting option alone, yet worse than the CF option. The clear winner is the windowing option *t*[*n*]. Obviously, growing *n* trees in order to produce the best production rule classifier is a well justified decision.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate
default	13.6	12.2	12.5(16.8%)	3.3(39.9%)	28.80%	18.20%	43.30%
s	13.6	12.2	12.5(16.8%)	3.3(39.9%)	28.80%	18.20%	43.30%
m5	7.8	6.4	16.8(22.5%)	3.6(43.3%)	31.30%	23.40%	42.10%
c15	13.6	8.8	14.2(19.0%)	3.3(39.9%)	32.80%	17.90%	42.10%
c10	13.6	8	14.7(19.7%)	3.3(39.9%)	35.30%	17.90%	42.10%
c15 m5	7.8	6.4	16.8(22.5%)	3.6(43.3%)	34.80%	23.40%	42.10%
t10	15.4	13.6	10.0(13.4%)	3.1(37.6%)	26.60%	15.00%	38.70%
t15	**15.8**	**14.4**	**9.4(12.6%)**	**3.2(38.8%)**	**26.30%**	**14.50%**	**36.40%**
t5	14.8	13	10.4(13.9%)	*3.0(36.4%)*	26.80%	16.20%	38.60%
t5 c15	14.6	10.4	11.8(15.8%)	3.4(41.0%)	31.30%	15.70%	34.70%
t15 c10	12.6	9.2	12.3(16.5%)	3.6(43.5%)	33.40%	14.80%	36.00%
m10	6.2	5.2	18.1(24.2%)	3.2(38.9%)	32.40%	23.70%	42.10%
m15	4.4	4	19.0(25.4%)	2.8(33.8%)	32.80%	25.20%	*33.80%*

Table 7.3. Two-group model building options for hypothesis 2.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate
default	28.2	25	10.8(14.5%)	4.1(49.4%)	35.50%	24.50%	48.30%
s	28.2	25	10.8(14.5%)	4.1(49.4%)	35.50%	24.50%	48.30%
m5	15.8	10	21.0(28.1%)	4.4(53.1%)	40.10%	28.30%	54.60%
c15	28.2	16.4	15.3(20.5%)	4.2(51.0%)	41.20%	24.90%	49.60%
c10	28.2	14.6	16.5(22.1%)	4.2(51.0%)	44.40%	25.10%	49.60%
c15 m5	15.8	7.4	22.7(30.4%)	3.8(45.8%)	44.60%	29.60%	52.10%
t10	30.2	26.4	8.8(11.8%)	4.0(48.1%)	33.50%	15.90%	51.80%
t15	**28.8**	**25.2**	**8.9(11.9%)**	**4.2(50.6%)**	**33.00%**	**14.60%**	**49.50%**
t5	30.6	27	*8.6(11.5%)*	4.1(49.3%)	33.70%	18.60%	48.20%
t5 c15	28.8	21	11.2(15.0%)	4.1(49.2%)	39.50%	17.60%	43.50%
t15 c10	27.6	18.6	12.0(16.1%)	4.3(51.5%)	42.00%	15.40%	48.20%
m10	5.2	3.6	26.2(35.1%)	*3.7(44.7%)*	42.50%	34.20%	48.30%
m15	3.8	3.6	26.2(35.1%)	*3.7(44.7%)*	42.50%	34.90%	*44.70%*

Table 7.4. Three-group model building options for hypothesis 2.

Although, in some of the models for different hypothesis, *t10* or *t5* values might perform better than *t15* for some of the parameters, it can be easily concluded that the windowing option with 15 trees gives the best results overall. The values for this option throughout the tables appear in bold, while the minimal error rates in each column are italicized.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate
default	24.8	15.4	9.7(13.0%)	3.1(37.6%)	27.80%	16.40%	*30.10%*
s	24.8	15.4	9.7(13.0%)	3.1(37.6%)	27.80%	16.40%	*30.10%*
m5	14	9.4	13.3(17.8%)	*2.9(35.0%)*	29.00%	17.80%	32.20%
c15	24.8	14	10.3(13.8%)	3.2(38.7%)	32.80%	16.80%	36.40%
c10	24.8	10	13.1(17.6%)	3.1(37.6%)	35.90%	16.80%	36.40%
c15 m5	14	9.4	13.3(17.8%)	*2.9(35.0%)*	33.40%	18.20%	31.00%
t10	23.8	16.6	6.0(8.0%)	3.3(39.9%)	23.40%	*9.60%*	36.20%
t15	**24.6**	**16.8**	**5.8(7.8%)**	**3.3(39.9%)**	**23.20%**	**9.80%**	**33.90%**
t5	23.8	17.4	6.1(8.2%)	3.6(43.6%)	24.10%	11.40%	36.40%
t5 c15	25.8	12.8	8.7(11.7%)	3.2(39.0%)	29.60%	10.20%	34.00%
t15 c10	24.4	12	8.5(11.4%)	3.4(41.2%)	31.40%	9.90%	36.40%
m10	8	6.6	15.1(20.2%)	3.0(36.2%)	29.60%	20.10%	37.50%
m15	5	5	20.3(27.2%)	2.8(33.9%)	35.50%	27.20%	33.90%

Table 7.5. Two-group model building options for hypothesis 3.

Options	Decision Trees					Rules	
	Size before	Size after	Train set Error Rate	Unseen Error Rate	Predicted Error Rate	Train set Error Rate	Unseen Error Rate
default	32.8	24.8	9.4(12.6%)	4.5(53.9%)	33.70%	21.50%	58.80%
s	32.8	24.8	9.4(12.6%)	4.5(53.9%)	33.70%	21.50%	58.80%
m5	17.4	13.4	17.6(23.6%)	*4.0(47.6%)*	38.10%	28.00%	48.20%
c15	32.8	17.6	13.3(17.8%)	5.1(61.1%)	40.00%	22.10%	56.40%
c10	32.8	17.2	13.6(18.2%)	5.1(61.1%)	43.50%	21.70%	55.10%
c15 m5	17.4	13.2	17.7(23.7%)	*4.0(47.6%)*	43.50%	27.30%	48.10%
t10	33.8	26.6	8.0(10.7%)	4.7(56.1%)	32.90%	16.50%	53.90%
t15	**34**	**27.4**	**7.5(10.0%)**	**4.6(55.6%)**	**32.70%**	**13.40%**	**52.60%**
t5	33.8	26	8.7(11.6%)	4.4(52.5%)	33.50%	18.50%	55.00%
t5 c15	33.6	18.6	12.1(16.2%)	*4.0(48.2%)*	39.60%	16.70%	51.30%
t15 c10	33.2	17.8	12.5(16.7%)	4.7(56.1%)	42.90%	14.70%	*47.80%*
m10	8.6	7.4	24.5(32.8%)	4.2(50.1%)	43.30%	33.10%	53.80%
m15	4.6	4.4	28.1(37.6%)	4.2(50.6%)	45.80%	37.50%	50.60%

Table 7.6. Three-group model building options for hypothesis 3.

7.2 The Predictive Models

The ultimate motivation for our thesis research was the assumed contribution toward software development process improvement—which helps solving the basic project management problem "delivery of a software product with targeted quality, within the budget and on schedule". An investigation of the relationships among OO design properties and fault-proneness per se, although noble from the scientific point of view, was not an option. We wanted to build predictive models of practical value/use: more accurate allocation of testing and verification resources during the development, and/or better maintainability while managing the change during the maintenance phase. The following predictive models represent an empirical evidence of our success.

7.2.1 Hypothesis 1

A production rule classifier consists of a collection of rules that discriminates/classifies a case on a dependent variable with certain confidence factor. We will comment some of those rules.

Rule 1: NMO > 1 NMI <= 22 SIX <= 0.222222 -> class 0 [75.8%]	Rule 2: NOC > 1 NOD <= 8 -> class 0 [72.2%]	Rule 3: DIT > 1 NMA <= 7 -> class 0 [70.0%]	Rule 4: NMI > 10 NMI <= 22 -> class 0 [63.0%]
Rule 5: CLD <= 0 NMA > 7 SIX > 0.222222 -> class 1 [91.2%]	Rule 6: NOC <= 1 NMO <= 0 NMI <= 6 -> class 1 [79.9%]	Rule 7: NMI > 22 -> class 1 [75.8%]	Default class: 1

Figure 7.1. Two-group predictive model for hypothesis 1.

INHERITANCE Descriptive Statistics					
variable	mean	median	minimum	maximum	std.dev.
DIT	0.838	1.0	0	3.00	0.8634
AID	0.838	1.0	0	3.00	0.8634
CLD	0.288	0.0	0	3.00	0.6202
NOC	0.613	0.0	0	9.00	1.4364
NOP	0.588	1.0	0	1.00	0.4954
NOD	0.863	0.0	0	12.00	2.2711
NOA	0.838	1.0	0	3.00	0.8634
NMO	2.563	0.0	0	13.00	3.6208
NMI	6.563	2.5	0	127.00	16.2620
NMA	12.638	9.0	1	104.00	12.2962
SIX	0.172	0.0	0	1.18	0.2584
COHESION Descriptive Statistics					
variable	mean	median	minimum	maximum	std.dev.
LCOM1	143.500	45.0000	0	5437.00	609.1501
LCOM2	6.050	5.0000	1	58.00	6.5948
LCOM3	5.850	5.0000	1	51.00	5.8528
LCOM4	103.100	24.0000	0	4988.00	557.7284
LCOM5	0.626	0.6250	0	1.25	0.2592
Coh	0.424	0.4330	0	1.00	0.2330
Co	0.106	0.0962	-1	0.50	0.1992
LCC	0.474	0.5000	0	1.00	0.2975
TCC	0.373	0.3258	0	1.00	0.2484
ICH	9.675	0.0000	0	343.00	43.3046
COUPLING Descriptive Statistics					
variable	mean	median	minimum	maximum	std.dev.
CBO	7.388	5.0	0	31	6.6950
CBO'	6.825	4.0	0	31	6.6804
RFC_1	49.175	33.0	3	358	58.7032
RFC_oo	109.075	43.0	3	669	145.0171
MPC	16.788	5.0	0	274	38.0272
ICP	51.038	13.5	0	769	114.8645
IHICP	7.688	2.0	0	190	22.7037
NIHICP	43.350	10.0	0	579	99.8522
DAC	1.150	1.0	0	8	1.4764
DAC'	0.950	1.0	0	7	1.2002
ACAIC	0.025	0.0	0	2	0.2236
OCAIC	1.125	1.0	0	8	1.4788
DCAEC	0.025	0.0	0	2	0.2236
OCAEC	1.213	0.0	0	16	2.6419
ACMIC	0.838	0.0	0	8	1.4964
OCMIC	8.450	4.0	0	205	23.1746
DCMEC	0.838	0.0	0	38	4.5128
OCMEC	9.738	1.0	0	135	23.6112
AMMIC	2.163	1.0	0	24	4.0890
OMMIC	14.625	4.0	0	250	35.8994
DMMEC	2.188	0.0	0	69	8.6786
OMMEC	14.350	4.0	0	124	24.1132

Table 7.7. Descriptive statistics for selected OO design metrics.

Rule 7 in Figure 7.1 reads "if the number of inherited methods in the component is greater than 22, then that component is likely to be faulty[15] (with confidence factor of 75.8%)". Not surprisingly, NMI > 22 implies that the component is in a lower[16] portion of the tree, so, higher the possibility a fault to be introduced through inheritance. In addition, rule 5 reads "if the component does not have descendents, and the number of methods added (not inherited, nor overridden) is greater than 7, and the specialization index of the component is greater than 0.222222, then the component is likely to be faulty (with a confidence factor of 91.2%!)". It means that even small numbers of new methods, and overridden methods (SIX is a function of NMO and DIT, refer to subsection 4.1.2) can provoke faults. The possible explanation lies in the experience of the developers. Namely, inheritance implies code reuse. Of course, nobody wants to reuse a superclass that is faulty. So, the developers would inherit from a component that is safe to the best of their knowledge. But, even with such an approach, the possibility of an error introduction remains "in the air"—and it is higher if the position of the component which inherits is lower in the tree. This finding confirms our hypothesis 1, again. Finally, rule 2 reads "if the number of children is greater than 1, and the number of descendents is lesser or equal to 8, then the component is not faulty (with a confidence factor of 72.2%)". The meaning of rule 2 comes somewhat unexpected, comparatively to the results of the related studies (refer to the section 4.2) based upon student/university settings. And again, the explanation depends on the experience of the programmers. The second rule is mostly about the components in the upper portion of the tree. The value of NOD = 8 is very close to the maximum which is 12 (refer to Table 7.7), while the corresponding mean and the median values (0.863 and 0, respectively) infer that the component is indeed in the upper part of the tree, so it is more reusable through inheritance. We confirmed this finding by checking the actual data.

Rule 7 in Figure 7.2 is the same as the second rule in Figure 7.1. The first rule in the three-group model is more related to the complexity of the component than an inheritance.

[15] Class 0 and class 1 denote non-faulty and faulty components, respectively.
[16] Closer look on the data confirms that, the lower the position of the component within the tree the higher the number of the methods inherited.

Namely, it is about those components at the root level, which are (obviously) not reusable. The faults there are introduced through the methods implemented. The value of NMA > 17 is greater than the corresponding mean and median values (12.638 and 9, respectively). This rule yields a high-risk[17] fault-prone component.

Rule 1: NOC <= 1 NOP <= 0 NMA > 17 -> class 3 [63.0%]	Rule 2: DIT > 1 DIT <= 2 NMI > 8 -> class 3 [63.0%]	Rule 3: NMO > 5 NMI > 0 SIX <= 0.380952 -> class 3 [54.6%]	Rule 4: NOC <= 1 NMA <= 5 -> class 3 [31.4%]
Rule 5: NMO > 1 NMO <= 3 -> class 1 [82.0%]	Rule 6: DIT > 1 NMA <= 7 SIX > 0.421053 -> class 1 [82.0%]	Rule 7: NOC > 1 NOD <= 8 -> class 1 [72.2%]	Rule 8: DIT <= 1 CLD <= 0 NMI > 6 NMI <= 28 -> class 1 [61.2%]
Rule 9: CLD <= 0 NMI <= 6 -> class 2 [58.6%]	Default class: 2		

Figure 7.2. Three-group predictive model for hypothesis 1.

Finally, rule 9 implies that the component is low-risk fault-prone, if the inheritance relation with the superclass is rather weak, and it does not have any descendants. Namely, those components do not inherit a lot of methods (the value of NMI = 6 is lesser than the corresponding mean value), in a sense, "a sort of" complexity measure.

Our findings justify the hypothesis 1.

[17] Class 1, class 2, and class 3 denote non-faulty, low-risk, and high-risk fault-prone components, respectively.

7.2.2 Hypothesis 2

Rule 1 in Figure 7.3 can be interpreted as "if the connectivity within the component is low and the loose class cohesion is greater than medium, then the component is likely to be faulty (with a confidence factor of 95.2%!)". If we take the mean and the median values for the connectivity (*Co*) measure into consideration (0.1 and 0.09, respectively), then we can interpret Rule 3 as "if the connectivity is rather high, as well as, the loose class cohesion, then the component is not faulty."

Rule 1:	Rule 2:	Rule 3:	Rule 4:
Co <= 0.28758 LCC > 0.52381 -> class 1 [95.2%]	LCOM1 <= 8 -> class 1 [82.0%]	Co > 0.28758 Co <= 0.41818 LCC > 0.82251 -> class 0 [66.2%]	LCOM3 > 3 LCC <= 0.52381 -> class 0 [54.7%]
Default class: 1			

Figure 7.3. Two-group predictive model for hypothesis 2.

Rule 2 in Figure 7.3 reads "if the number of pairs of methods in the component having no common attribute reference is lesser or equal to eight, then the component is likely to be faulty". $LCOM_1(c)$ is an inverse measure—the higher the value the lower the cohesion. The corresponding mean and median values (Table 7.7) are 143.5 and 45, respectively. So, the second rule is unexpected regarding the hypothesis 2, for it points to the components with high cohesion. The explanation about this finding came from the data set. Namely, there are very few components (within already rather small data set of 83 cases) with such a low value for $LCOM_1(c)$. The C4.5 system, while trying to produce the most elegant rules, has picked up the most appearing measure, and biased its decision.

Rule 5 in Figure 7.4 reads "if $LCOM_3(c)$ is greater than three, and *LCC* is lesser or equal to 0.52 (approximately, half of the population), and there is no information flow based

cohesion, then the component is not faulty (with a confidence factor of 63.3%)". The *ICH* metric though, has a median value of 0, which means that it can be hardly useful as a predictor. Rule 6 reads "if *LCC* is greater than 0.52381 then the component is low-risk likely to be faulty (with a confidence factor of 64.3%)". Finally, rule 1 says that if the connectivity increases and the tight class cohesion is lesser than 0.27 (which is already a small value), then the component is high-risk fault-prone.

Rule 1: Co > 0.06593 TCC <= 0.27272 -> class 3 [70.7%]	Rule 2: LCOM3 > 2 LCC > 0.75483 -> class 3 [63.0%]	Rule 3: LCOM5 <= 0.625 Co <= 0.02222 LCC <= 0.183 -> class 3 [63.0%]	Rule 4: Co > 0.35714 Co <= 0.41818 -> class 1 [75.8%]
Rule 5: LCOM3 > 3 LCC <= 0.52381 ICH <= 0 -> class 1 [63.3%]	Rule 6: LCC > 0.52381 -> class 2 [64.3%]	Default class: 2	

Figure 7.4. Three-group predictive model for hypothesis 2.

Our findings, generally, confirm the hypothesis 2.

7.2.3 Hypothesis 3

Not surprisingly, the production rule classifiers based on coupling measures produced the best models[18] of the three OO design properties in question. Coupling has been proven to be the most useful software product property as a predictor of its quality (refer to the related works).

[18] See the model evaluation and validation results in the next section (7.3).

Rule 1: CBO > 14 -> class 1 [88.2%]	Rule 2: IH-ICP > 16 -> class 1 [87.1%]	Rule 3: DAC' <= 2 OCAIC > 0 OMMEC > 9 -> class 1 [83.3%]	Rule 4: OCAIC <= 0 DMMEC <= 0 -> class 1 [81.9%]
Rule 5: MPC <= 6 DAC > 0 OMMEC <= 15 -> class 0 [77.0%]	Rule 6: CBO <= 14 DMMEC > 0 -> class 0 [70.0%]	Rule 7: ICP <= 31 IH-ICP > 9 -> class 0 [63.0%]	Default class: 1

Figure 7.5. Two-group predictive model for hypothesis 3.

Rule 1 in Figure 7.5 reads "if the coupling between the objects is greater than 14, then the component is likely to be faulty (with a confidence factor of 88.2!)". The rule confirms that high coupling means high fault-proneness. On the other hand, rule 7 reads "if information flow based coupling is lesser or equal to 31, and its inheritance based counterpart is greater than 9, then the component is non-faulty (with certainty of 63%)". This means that although low coupling fosters a healthy component, some forms of inheritance based coupling are desirable. The previous statement comes as no surprise, since the inheritance is the most significant property in any truly OO system.

Rule 1: ACMIC <= 1 OCMIC > 4 OMMEC <= 0 -> class 3 [63.0%]	Rule 2: CBO > 14 CBO' <= 17 -> class 3 [61.2%]	Rule 3: ICP > 59 OMMIC <= 16 -> class 3 [50.0%]	Rule 4: CBO <= 14 OCAIC <= 0 DCMEC <= 0 OMMEC > 0 -> class 2 [82.3%]
Rule 5: CBO' > 17 -> class 2 [75.8%]	Rule 6: MPC <= 7 OCAIC > 0 OMMEC <= 15 -> class 1 [77.0%]	Rule 7: ICP <= 31 IH-ICP > 5 -> class 1 [70.0%]	Default class: 2

Figure 7.6. Three-group predictive model for hypothesis 3.

Rule7 in Figure7.6 is almost the same as the corresponding rule in Figure 7.5. Rule 5 says that if the non-inheritance based coupling belongs to the upper portion of the corresponding value domain, then the component is low-risk fault-prone.

We may conclude that the models have confirmed the hypothesis 3.

7.2.4 Multivariate[19] Models

In order to build models with the highest accuracy—thus practical value—possible, we combined all OO design metrics in a same model (two-group and/or three-group). As expected, the results obtained are indeed the best. We will not comment the rules upon which these classifiers are based. However, we point to the selected metrics and their corresponding usefulness degrees in the section 7.4.

Rule 1: NOC > 1 NOD <= 8 LCC <= 0.52381 -> class 0 [87.1%]	Rule 2: LCOM2 <= 6 LCOM4 > 22 ACMIC > 0 -> class 0 [75.8%]	Rule 3: NMI > 9 ICP <= 31 -> class 0 [75.8%]	Rule 4: OCAIC > 0 ACMIC > 1 OMMEC <= 9 -> class 0 [75.8%]
Rule 5: Co > 0.28758 DAC > 1 -> class 0 [75.8%]	Rule 6: Co <= 0.28758 LCC > 0.52381 -> class 1 [95.2%]	Rule 7: CLD <= 0 NMI <= 3 DAC <= 0 -> class 1 [93.0%]	Rule 8: CLD <= 0 OMMEC > 15 -> class 1 [90.6%]
Rule 9: IH-ICP > 16 -> class 1 [87.1%]	Rule 10: CBO' > 14 -> class 1 [85.7%]	Rule 11: MPC > 7 DAC' <= 1 AMMIC <= 3 -> class 1 [85.7%]	Rule 12: RFC_1 <= 11 -> class 1 [80.9%]
Default class: 1			

Figure 7.7. Two-group multivariate predictive model.

[19] The term multivariate here points to models based on a combination of all three OO design properties.

Rule 1: LCC > 0.52381 OCAIC <= 1 -> class 2 [89.8%]	Rule 2: CBO' > 17 -> class 2 [75.8%]	Rule 3: NOC <= 1 CBO' <= 17 ICP > 59 OCAIC > 0 -> class 3 [79.4%]	Rule 4: NMO <= 9 Co > 0.06593 OCMEC > 7 -> class 3 [75.8%]
Rule 5: NOC <= 1 LCC <= 0.18181 ACMIC <= 1 -> class 3 [61.2%]	Rule 6: NOC > 1 CBO <= 14 RFC_1 > 11 -> class 1 [88.2%]	Rule 7: Co > 0.35714 Co <= 0.41818 -> class 1 [75.8%]	Rule 8: LCC <= 0.52381 ICP <= 59 DAC' > 0 OCMEC <= 4 -> class 1 [70.4%]
Default class: 2			

Figure 7.8. Three-group multivariate predictive model.

7.3 Evaluation and Validation of the Models

The following eight tables present the empirical evidence of the quality of the models built. Their mutual characteristics are:

- High overall accuracy achieved. The average value (excluding the two, multivariate models) is 85.74% across the six models.
- Very high values for the Chi-square (X-sqr) tests. The average value across the models is 40.73, with perfect statistical significance (p-value <= 0.0000).

Tested 83, errors 9 (10.8%)			
	predicted 0	predicted 1	Completeness
real 0	**24**	**6**	**80.00%**
real 1	**3**	**50**	**94.34%**
Correctness	**88.89%**	**89.29%**	
Accuracy=	**89.16%**		
X-sqr=	48.2352		
p <=	0.0000		

Table 7.8. Evaluation of the two-group predictive model for hypothesis 1.

Tested 83, errors 12 (14.5%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**25**	**5**		**83.33%**
real 2	**3**	**35**	**2**	**87.50%**
real 3		**2**	**11**	**84.62%**
Correctness	**89.29%**	**83.33%**	**84.62%**	
Accuracy=	**85.54%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	37.8501	46.1791	36.00	31.3711
p <=	0.0000	0.0000	0.0000	0.0000

Table 7.9. Evaluation of the three-group predictive model for hypothesis 1.

Tested 83, errors 16 (19.3%)			
	predicted 0	predicted 1	Completeness
real 0	**30**		**100.00%**
real 1	**16**	**37**	**69.81%**
Correctness	**65.22%**	**100.00%**	
Accuracy=	**80.72%**		
X-sqr=	37.7892		
p <=	0.0000		

Table 7.10. Evaluation of the two-group predictive model for hypothesis 2.

Tested 83, errors 13 (15.7%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**25**	**5**		**83.33%**
real 2	**5**	**35**		**87.50%**
real 3	**1**	**2**	**10**	**76.92%**
Correctness	**80.65%**	**83.33%**	**100.00%**	
Accuracy=	**84.34%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	34.5465	35.1215	31.4685	37.0495
p <=	0.0000	0.0000	0.0000	0.0000

Table 7.11. Evaluation of the three-group predictive model for hypothesis 2.

Tested 83, errors 8 (9.6%)			
	predicted 0	predicted 1	Completeness
real 0	**26**	**4**	**86.67%**
real 1	**4**	**49**	**92.45%**
Correctness	**86.67%**	**92.45%**	
Accuracy=	**90.36%**		
X-sqr=	51.9571		
p <=	0.0000		

Table 7.12. Evaluation of the two-group predictive model for hypothesis 3.

Tested 83, errors 13 (15.7%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**23**	**7**		**76.67%**
real 2	**2**	**37**	**1**	**92.50%**
real 3		**3**	**10**	**76.92%**
Correctness	**92.00%**	**78.72%**	**90.91%**	
Accuracy=	**84.34%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	34.0541	37.5596	33.00	31.6026
p <=	0.0000	0.0000	0.0000	0.0000

Table 7.13. Evaluation of the three-group predictive model for hypothesis 3.

Tested 83, errors 2 (2.4%)			
	predicted 0	predicted 1	Completeness
real 0	**28**	**2**	**93.33%**
real 1		**53**	**100.00%**
Correctness	**100.00%**	**96.36%**	
Accuracy=	**97.59%**		
X-sqr=	74.6497		
p <=	0.0000		

Table 7.14. Evaluation of the two-group multivariate predictive model.

Tested 83, errors 5 (6.0%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**28**	**2**		**93.33%**
real 2	**2**	**37**	**1**	**92.50%**
real 3			**13**	**100.00%**
Correctness	**93.33%**	**94.87%**	**92.86%**	
Accuracy=	**93.98%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	46.9313	53.683	41.00	46.1109
p <=	0.0000	0.0000	0.0000	0.0000

Table 7.15. Evaluation of the three-group multivariate predictive model.

There are two types of misclassification errors in two-group models. If we take a closer look at the Table 7.8, we can see that six non-faulty components have been predicted as faulty, while three faulty components have been classified as non-faulty. Taking the project management's point of view into account, the later type of error is much more severe than the former. It does no harm[20] to test a component as if it was faulty while it was not. But, it would be definitely a mistake on behalf of the project manager, if she decides to deliver the product to the market without testing the components that are actually faulty, but have been classified as healthy.

One possible remedy for this situation is to test all the components that are on the boundary as if they were part of the group with a higher risk. This discussion can be generalized to accommodate N X N predictive models. The final quality of the product, as well as the reduction of the cost, and the time to market, will still be highly significant, comparatively with the software development processes that are "ignorant" regarding the fault-proneness.

In the remainder of this section, as well as the Appendix A, we compare the quality of our predictive models with the quality of those models built by other professionals within the software engineering community.

[20] Adding a few components more for rigorous testing and verification will not significantly affect the time to market, or the final cost, of a product.

Predictive Models				
Who	Almeida et al. [4]	Basili et al. [9]	Lounis et al. [39]	Our work
Technique	MLA (NewID, CN2, C4.5, Foil), Log.R.	C4.5	C4.5	C4.5
Independent var.	complexity metrics	complexity metrics	OO Design metrics	OO Design metrics
Dependent var.	Correction Costs	Correction Costs	Fault-Proneness	Fault-Proneness
Model Type	2 x 2	2 x 2	2 x 2	2 x 2, 3 x 3
Accuracy	[52%, 54%, 68%, 71%], 61%	66.37%	78.82%, 85.88%, 69.41%	97.59%, 93.98% avg. w/o multivar. mod. 85.74%
Correctness	[(53%, 50%), (56%, 53%), (74%, 64%), (80%, 65%)], (61%, 39%)	(62.5%, 69%)	(81%, 74%), (86%, 85%), (72%, 60%)	(100%, 96.36%), (93.33%, 94.87%, 92.86%)
Completeness	[(55%, 48%), (58%, 51%), (59%, 77%), (61%, 82%)], (69%, 48%)	(60%, 71%)	(87%, 64%), (92%,74%), (83%, 45%)	(93.33%, 100%), (93.33%, 92.50%, 100%)
Goodness-of-fit Chi-square (p-value)	[0.19 (p<= 0.66), 1.13 (p<= 0.29), 21.91 (p<= 0.0000), 30.39 (p<= 0.0000)], 7.70 (p<= 0.005)	11.2690 (p<= 0.0008)	24.1487 (p<= 0.0000), 40.5280 (p<= 0.0000), 8.1018 (p<= 0.0044)	74.6497 (p<= 0.0000), 46.9313 (p<= 0.0000)
Validation	V-fold cross-validation	V-fold cross-validation	V-fold cross-validation	V-fold cross-validation

Table 7.16. Comparison of our work with other research studies.

It is not easy to compare the quality of predictive models built in different environments (from both technical and human aspects). However, we conclude that our predictive models are superior than those described in the literature. We believe that this finding heavily depends on the experience of the LALO development team members. The effect is even magnified if we compare our results with those related to statistical analysis techniques (refer to the Appendix A).

7.4 Usefulness Degree of a Metric

We need to define a new measure in order to quantify the contribution of each metric in a predictive model based on the C4.5 machine learning algorithm. This new measure will, somehow, represent a ratio of the differentiation common in the independent and the dependent variables, to the overall differentiation of those variables. This ratio is usually called[21] a ratio of explained[22] variation to total variation. For example, the ratio of the part of the overall differentiation of the fault-proneness scores that can be accounted for by *NMI* measure to the overall differentiation of the fault-proneness scores, in the two-group model for hypothesis 1 is 38.10% (Table 7.17).

In order to define the new measure(s) consistently and in an unambiguous manner, we propose the following formalism.

Let M denotes the suite of metrics (the dependent variables) whose assumed relationships with the fault-proneness (the independent variable) we would like to investigate. Let PM represents the predictive model built. Let:

- $PM = \{M_S, R, M_S \mapsto R\}$, where $M_S \subseteq M$ is the set of metrics selected by the model, R is the set of rules in the model, and $M_S \mapsto R$ denotes the mapping from the set of metrics selected (by the model) to the set of rules. $Acc(PM)$ denotes the overall accuracy of the predictive model in question.
- $R = \{M_S, O\}$, where $O = \{<, \leq, =, \geq, >, and\}$ is a set of operands that are allowed in the C4.5 machine learning algorithm. Let r_i be the i-th rule of the model, such that $r_i \in R : i = 1, ..., |R|$.

[21] There are many measures of the magnitude of relationships among variables developed by statisticians. However, we believe that the mixture of statistical analysis and MLA, regarding the predictive modeling, is not the happiest choice.

[22] The term "explained variation" does not necessarily implies that we "conceptually understand" it. It only denotes the common variation in the variables in question—the part of variation in one variable that is explained by the specific values of the other variable.

- $M_{NS} \subseteq M$ be the set of metrics not selected by the predictive model, such that $M = M_S \cup M_{NS}$, and $M_S \cap M_{NS} = \varnothing$.
- $M_i \subseteq M_S$ represents the set of metrics—elements of the rule r_i, such that, $m \in M_i$.

We define:

1. Usefulness degree of the rules in the model *RUD(PM)*

$$RUD(PM) = \frac{1}{|R|} = const.$$

2. Usefulness degree of a metric *MUD(m)*

$$MUD(m) = RUD(PM) \cdot \sum_{i=1}^{|R|} \frac{j}{|M_i|},$$

where

$$j = \begin{cases} 1, & if\ m \in M_i \\ 0, & else. \end{cases}$$

3. Usefulness degree of a metric $MUD'(m)$

$$MUD'(m) = Acc(PM) \cdot MUD(m).$$

With the newly defined measures $MUD(m)$, and $MUD'(m)$, we can analyze the impact that each of the selected metrics (by the model) has on the fault-proneness in much easier, and more appropriate manner, than just by backtracking the model induced rules. Clearly, those metrics that have not been selected by the model ($m \in M_{NS}$; $MUD(m)$= 0) can be and should be avoided in the same type of future experiments. However, we cannot recommend further reduction of the metric set on the bases of a low usefulness degree.

Namely, in MLA even the lack of information is an information. It means that even a small contribution towards the better predictive quality serves building the best model(s).

	Hypothesis 1: Models					
	2 x 2		3 x 3		Average	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'
DIT	7.14%	6.37%	12.96%	11.09%	10.05%	8.78%
AID	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
CLD	4.76%	4.25%	9.26%	7.92%	7.01%	6.12%
NOC	11.90%	10.61%	14.81%	12.67%	13.36%	11.67%
NOP	0.00%	0.00%	3.70%	3.17%	1.85%	1.62%
NOD	7.14%	6.37%	5.56%	4.75%	6.35%	5.55%
NOA	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NMO	9.52%	8.49%	14.81%	12.67%	12.17%	10.63%
NMI	38.10%	33.97%	18.52%	15.84%	28.31%	24.73%
NMA	11.90%	10.61%	12.96%	11.09%	12.43%	10.86%
SIX	9.52%	8.49%	7.41%	6.34%	8.47%	7.39%
total:	100.00%	89.16%	100.00%	85.54%	100.00%	87.35%
RUD	14.29%		11.11%		12.70%	
Acc	89.16%		85.54%		87.35%	

Table 7.17. Hypothesis 1 models and the contribution of each metric.

	Hypothesis 2: Models					
	2 x 2		3 x 3		Average	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'
LCOM1	25.00%	22.29%	8.33%	7.13%	16.67%	13.60%
LCOM2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LCOM3	12.50%	11.15%	13.89%	11.88%	13.19%	10.90%
LCOM4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LCOM5	0.00%	0.00%	5.56%	4.75%	2.78%	2.34%
Coh	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Co	25.00%	22.29%	30.56%	26.14%	27.78%	22.98%
LCC	37.50%	33.44%	27.78%	23.76%	32.64%	26.85%
TCC	0.00%	0.00%	8.33%	7.13%	4.17%	3.51%
ICH	0.00%	0.00%	5.56%	4.75%	2.78%	2.34%
total:	100.00%	89.16%	100.00%	85.54%	100.00%	82.53%
RUD	14.29%		11.11%		20.83%	
Acc	89.16%		85.54%		82.53%	

Table 7.18. Hypothesis 2 models and the contribution of each metric.

Note that the usefulness degree measures do not make attempt to point the directions of the relationships among the independent variables and the dependent variable. Whether the direction is positive[23] or negative has lesser importance than the mere fact that the impact is high or low.

	Hypothesis 3: Models					
	2 x 2		3 x 3		Average	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'
CBO	21.43%	19.36%	10.71%	9.04%	16.07%	14.20%
CBO'	0.00%	0.00%	21.43%	18.07%	10.71%	9.04%
RFC_1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RFC_oo	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
MPC	4.76%	4.30%	4.76%	4.02%	4.76%	4.16%
ICP	7.14%	6.45%	14.29%	12.05%	10.71%	9.25%
IH-ICP	21.43%	19.36%	7.14%	6.02%	14.29%	12.69%
NIH-ICP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DAC	4.76%	4.30%	0.00%	0.00%	2.38%	2.15%
DAC'	4.76%	4.30%	0.00%	0.00%	2.38%	2.15%
IFCAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ACAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAIC	11.90%	10.76%	8.33%	7.03%	10.12%	8.89%
FCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
IFCMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ACMIC	0.00%	0.00%	4.76%	4.02%	2.38%	2.01%
OCMIC	0.00%	0.00%	4.76%	4.02%	2.38%	2.01%
FCMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DCMEC	0.00%	0.00%	3.57%	3.01%	1.79%	1.51%
OCMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
IFMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
AMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OMMIC	0.00%	0.00%	7.14%	6.02%	3.57%	3.01%
FMMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DMMEC	14.29%	12.91%	0.00%	0.00%	7.14%	6.45%
OMMEC	9.52%	8.61%	13.10%	11.04%	11.31%	9.83%
total:	100.00%	90.36%	100.00%	84.34%	100.00%	87.35%
RUD	14.29%		14.29%		14.29%	
Acc	90.36%		84.34%		87.35%	

Table 7.19. Hypothesis 3 models and the contribution of each metric.

[23] The direction of a relationship between two variables is positive if an increase/decrease of the value of the first variable provokes an increase/decrease of the value of the second variable. The negative direction is defined as the opposite.

We will not comment the obtained results in atomic details. However, we feel free to conclude that they further confirm the validity of the hypotheses 1, 2, and 3. We treat all of the selected metrics $m \in M_S : MUD(m) \neq 0$ as being empirically validated and useful predictors of fault-proneness. Moreover, we would like to emphasize that a model might reject a metric under the following two circumstances:

- *Truly not useful metric*: $MUD(m) = 0$, and there is no variance among the values of the metric, the most trivial case being—all of the metric values are equal to 0.
- *Labeled as not useful metric*: $MUD(m) = 0$, although the metric has enough variance. This may easily happen if two or more metrics have the same values throughout the sample population. It may also happen, if all metrics have distinctive values, but they cannot further improve the (predictive) quality of the model (some of the measures are outperformed by the others). The C4.5 MLA, logically, tries to avoid redundancy by cutting out excessive metric data.

For example, two measures in the hypothesis 1 models (Table 7.17)—*AID* and *NOA*—have been rejected because of redundancy. A multiple inheritance has not been really used in *LALO*. Therefore, *AID* and *NOA* are equal to *DIT* (with an average $MUD(DIT) = 10.05\%$). It would be erroneous to conclude, a priori, that these two measures would not be valid in other systems/environments.

Probably, the most important property of the usefulness degree measures is that they provide deeper insight in the design habits of the software product development team, as well as the structure of the final product. The high values for *NMI*, *NMA*, *NMO*, and *SIX*, witness the complexity of the system, as well as the regular use of inheritance. Together with *NOC*, *DIT*, *CLD*, and *NOD*, and the rules analysis they justify the hypothesis 1.

In the hypothesis 2 models (Table 7.18) the most important measures are *LCC*, *Co*, $LCOM_1(c)$, and $LCOM_3(c)$. $LCOM_2(c)$, and $LCOM_4(c)$ have not been selected by the models. However, both measures have been validated in the models where the dependent variable is a defect density (Appendix B).

	Multivariate Models					
	2 x 2		3 x 3		Average	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'
DIT	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
AID	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
CLD	6.94%	6.78%	0.00%	0.00%	3.47%	3.33%
NOC	2.78%	2.71%	11.46%	10.77%	7.12%	6.82%
NOP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NOD	2.78%	2.71%	0.00%	0.00%	1.39%	1.33%
NOA	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NMO	0.00%	0.00%	4.17%	3.92%	2.08%	2.00%
NMI	6.94%	6.78%	0.00%	0.00%	3.47%	3.33%
NMA	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
SIX	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LCOM1	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LCOM2	2.78%	2.71%	0.00%	0.00%	1.39%	1.33%
LCOM3	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LCOM4	2.78%	2.71%	0.00%	0.00%	1.39%	1.33%
LCOM5	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Co	8.33%	8.13%	16.67%	15.66%	12.50%	11.97%
LCC	6.94%	6.78%	13.54%	12.73%	10.24%	9.81%
TCC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ICH	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
CBO	0.00%	0.00%	4.17%	3.92%	2.08%	2.00%
CBO'	8.33%	8.13%	15.63%	14.68%	11.98%	11.47%
RFC_1	8.33%	8.13%	4.17%	3.92%	6.25%	5.99%
RFC_oo	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
MPC	2.78%	2.71%	0.00%	0.00%	1.39%	1.33%
ICP	4.17%	4.07%	6.25%	5.87%	5.21%	4.99%
IH-ICP	8.33%	8.13%	0.00%	0.00%	4.17%	3.99%
NIH-ICP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DAC	6.94%	6.78%	0.00%	0.00%	3.47%	3.33%
DAC'	2.78%	2.71%	3.13%	2.94%	2.95%	2.83%
ACAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAIC	2.78%	2.71%	9.38%	8.81%	6.08%	5.82%
DCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ACMIC	5.56%	5.42%	4.17%	3.92%	4.86%	4.66%
OCMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DCMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCMEC	0.00%	0.00%	7.29%	6.85%	3.65%	3.49%
AMMIC	2.78%	2.71%	0.00%	0.00%	1.39%	1.33%
OMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DMMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OMMEC	6.94%	6.78%	0.00%	0.00%	3.47%	3.33%
	100.00%	97.59%	100.00%	93.98%	100.00%	95.79%
RUD	8.33%		12.50%		10.42%	
Acc	97.59%		93.98%		95.79%	

Table 7.20. Multivariate models and the contribution of each metric.

The most important measures in the hypothesis 3 models (Table 7.19) are *CBO*, *IH-ICP*, *OMMEC*, *CBO'*, *ICP*, and *OCAIC*. Due to the design characteristics[24] of the investigated system six measures are not applicable (all of the friendship-based metrics). *ACAIC* and *DCAEC* have not been validated by any of the models.

Finally, we present the results for the multivariate models (Table 7.20). Most of the measures with high values for the *MUD* in the multivariate models are those with high values in the models around the specific design properties. The most important (average) measures regarding fault-proneness in our multivariate models are *Co*, *CBO'*, *LCC*, *NOC*, RFC_I, *OCAIC*, *ICP*, *ACMIC*, *IH-ICP*, *OCMEC*, *CLD*, *NMI*, *DAC*, and *OMMEC*.

[24] *LALO* developers have not used a friendship mechanism in C++, on purpose.

Chapter 8

Epilogue, or Lessons Learned

"Everything should be made as simple as possible, but not simpler".

Albert Einstein

In this chapter we present the conclusions with respect to our thesis research, based on the lessons learned, as well as our previous project management experience. We also, point out to the future works that, we expect will further improve the overall understanding that the practitioners in the computer-related business have about software development processes and project management.

8.1 Conclusions

Have we answered the questions from Chapter 3 "Specific Problem Statement" that triggered our thesis research? Have the problem solving approach—knotted throughout this writing and proposed in the section 3.1 "Solution Strategy"—delivered satisfactory result? Can we claim research success?

The short answer on all the questions from the last paragraph is—yes. The longer follows.

- *Which OO design product measures are good predictors of fault-proneness as a software product's quality indicator?*

i. Steps 1 and 2 of the "Solution Strategy" represented a good starting point. The proposed hypotheses were investigated using the measures extracted by the M-System automated tool (supports the collection of 49 plus OO design metrics). However, there is no direct and precise answer to the above aforementioned question. Many OO metrics have already been empirically validated (in various projects) in the software engineering community. Many more have been defined and are still waiting for an appropriate validation. Which of them should be selected as potential software's quality predictors, depends on the application domain. Common sense—regarding project management—suggests that only those measures that have already been proven as useful in some applications, and whose collection is supported by automated tools, should be selected. A project manager will not be interested in the investigation of the fault-proneness if it does not reduce the time to market and/or the final cost, and/or does not raise the quality of the product. It is important to note, that an initial investment in a software development process improvement, always turns out to be a huge profit at the bottom line (refer to Raytheon's experience, section 2.1.1).

ii. Whether certain metrics will be more or less useful depends not only on the application, but also on the overall environment housing the project. Why is the development team so important? Because, it is the team that will make all the design decisions, implement the code, and introduce the faults in the product. The more experienced the team, the better the design decisions, the better the implementation, the higher the quality of the product.

iii. Even in the same environment, within the same application domain, some of the metrics will be more or less important regarding the choice of the dependent variable. It is the case with, i.e., $LCOM_2(c)$, and $LCOM_4(c)$ as predictors, and the fault-proneness and the defect density (refer to the Appendix B), as dependent variables.

- *What is a suitable technique for predictive modeling and empirical validation of the selected OO design metrics?*

i. We proposed the C4.5 machine learning algorithm as a predictive modeling technique (step 3, section 3.1). Why? Because, somehow, it produces predictive models with superior quality than models based on statistical analysis. It is fairly easy to understand it, and then use it. But, perhaps the biggest advantage of machine learning algorithms—as a modeling technique—over the statistical analysis (SA) lies in the interpretation of the results. Instead of extracting *principle components* and then searching for patterns in numbers that represent their meaning (SA), the interpretation of the production rules is straightforward, and much more intelligible to human beings (MLA).

ii. Another advantage of MLA over SA is that the latter tries to fit the data to the model, while the former fits the model to the data. For example, MLA build models by inducing rules that will accommodate all cases in the sample population. Conversely, in SA the model designer will deliberately cut off the cases (called *outliers*) that do not get along with the rest of the cases from the sample population, grouped around the "fitting line". The real life models are non-linear by nature. Sometimes, they can be fairly approximated with mathematical counterparts (linear or non-linear), but sometimes that is not feasible. It is exactly the case of incomplete, inexact, and imprecise knowledge, where we believe the MLA provide much better solutions than the SA, with respect to the predictive modeling.

- *What are the objective criteria against which the models built should be evaluated with respect to their predictive quality?*

i. We proposed correctness, completeness, accuracy, and goodness-of-fit of data as an objective set of standards for model evaluation (step 4, section 3.1). The models built have high quality, and we would like to credit most of this finding to the quality of the LALO development team.

ii. The models have been validated via 10-way cross-validation. It would have been better if we could have validated them on different project data from the same environment, but during our research study such data set was not available.

iii. Our measures *MUD*, *MUD'*, and *RUD*, provide a deeper insight into the models built, and the design structure of the product. They recognize the specific contribution of each metric in the model and confirm/reject its potential as a predictor of the dependent variable. The use of these measures can be generalized to serve other rule-based MLA, and with slight modifications it is applicable to Fuzzy Logic.

We do not recommend verbatim reuse of our predictive models in other environments, since they also depend on other factors that are beyond the scope of this study. However, the OO design metric suite that we have used should be involved in other research studies, or business/industry related projects. We also recommend the replication of our approach, which we believe is more important than the final quality of the predictive models, or which metrics have been found as useful predictors.

Automated tools are necessary for data collection. They can significantly reduce the time required for data acquisition, and in the case of large systems, they provide the only way that makes that process feasible. The tools should be stand alone to the fullest extent possible. M-System runs only if put on top of Gen++. Gen++ recognizes AT&T C++ compiler only. On the other hand, the system that we have investigated has been implemented in Visual C++. It means, that the source code firstly had to be translated into AT&T C++ source code, then analyzed by Gen++, and finally, the metrics needed have been computed by M-System.

A very strong conclusion—based on our project management experience, contacts with the industry, research conducted in the area of software process engineering, and university activities—is that a synergy between the academia and the industry has to be stimulated and lovingly cultivated. Unfortunately for the computer-related business, most of the software development organizations still produce "guerilla" type of software, and have a little understanding of the five process maturity levels defined as Capability

Maturity Model[25] (CMM) at the SEI[26] [41]. The academic research can help project managers understand, and then learn, why the combination of a structured and rigorous approach to software development and their intuition, is better than the exclusively intuitive approach. Software development processes need to be made visible, then repeatable, and then measurable—in order to be improved. And, a process improvement is continual! Of course, the intuition remains one of the most important parts of the business acumen a project manager might have when decisions have to be made within the environment of uncertainty. On the other hand, the industry can guide the scientific research at the academia towards the solutions that are not only theoretical, but also have practical values. Everybody wins with this type of synergetic endeavors—the business, the educational system, and the society. The institutions like SEL[27], Carnegie Mellon University, McGill University, CRIM, IESE, etc., lead the way.

8.2 Future Works

There are many studies in the software engineering community devoted to the predictive modeling. Yet, studies regarding the experience of the practical use of the models built, practically, do not exist. We intend to engage such a type of study with the team responsible for future releases of LALO.

We also intend to perform a comparative study that will point to the pros and cons among MLA (C4.5, etc.) and various techniques based on statistical analysis. We will use the same data set.

We intend to provide theoretical proof of the usefulness degree measures that we have defined in this work, as well as to generalize their applicability to any rule-based system.

---

[25] The five process maturity levels are (1) initial, (2) repeatable, (3) defined, (4) managed, and (5) optimizing.

[26] The Software Engineering Institute, Carnegie Mellon University, Pitsburgh.

[27] The Software Engineering Laboratory—NASA Goddard Space Flight Centre, University of Maryland, and Computer Sciences Corporation.

The natural extent of our research will be a study that will relate the cost of the error correction with the design properties of the OO system in question.

We plan to combine MLA, Fuzzy Logic [48], and neural networks and/or genetic algorithms, in order to produce even more robust, more reliable, and more comprehensible predictive modeling techniques.

8.3 Closing Word

In this thesis research we have investigated the assumed relationships among OO design measures and the fault-proneness as software's quality indicator. The predictive models built represent empirical evidence that such relationships exist.

Firstly, we have described the broader context of our problem, and the motivation behind it. Then, we led the research towards the problem definition, providing a deeper understanding about it by locating its place on a gross software engineering scale. This way, we enforced a *system thinking* approach to problem solving—a solution for the part of the system is not supposed to provoke problems in the other parts of the system. We insisted on a management point of view, because the knowledge regarding which components in the system are likely to be faulty, does not make any sense whatsoever to the project manager if it does not help to solve the basic project management problem "delivery of a product with targeted quality, within the budget and on schedule".

So, finally, have we solved the project management problem with our thesis research?

The answer is—no. The problem is rather complex, and heavily depends on many other factors that are beyond the scope of this study. However, we described a method that gets closer to the solution by improving the software development process used. The quest for the universal solution (if possible) is still open.

We will be very satisfied if the software engineering community and the computer-business professionals recognize our study as a small contribution towards that goal.

Bibliography

[1] J. R. Abounader, and D. A. Lamb. *A Data Model for Object-Oriented Design.* Technical Report ISSN-0836-0227-1997-409, Department of Computing and Information Science, Queen's University, Kingston, Canada 1997.

[2] F. B. Abreu, and W. Melo. *Evaluating the Impact of Object-Oriented Design on Software Quality.* In Proceedings of IEEE Metrics'96, Berlin, Germany, March 1996.

[3] W. W. Agresti, and W. M. Evanco. *Projecting Software Defects from Analyzing ADA Designs.* IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992.

[4] M. A. D. Almeida, H. Lounis, and W. L. Melo. *An Investigation on the Use of Machine Learned Models for Estimating Software Correction Costs.* In 20th IEEE Interna7tional Conference on Software Engineering, 1998.

[5] J. Bansiya, and C. Davis. *Automated Metrics and Object-Oriented Development.* Dr. Dobb's Journal, December 1997.

[6] J. Bansiya, and C. Davis. *Using Automated Metrics to Track Object-Oriented Development.*

[7] V. Basili, L. Briand, S. Condon, Y. M. Kim, W. L. Melo, and J. D. Valett. *Understanding and Predicting the Process of Software Maintenance Releases.* In 18th IEEE International Conference on Software Engineering, Berlin, Germany, 1996.

[8] V. R. Basili, L. C. Briand, and W. L. Melo. *A Validation of Object-Oriented Design Metrics as Quality Indicators*. IEEE Transactions on Software Engineering, Vol. 22, No. 10, October 1996.

[9] V. Basili, S. Condon, K. E. Emam, and W. L. Melo. *Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components*. In 19th IEEE International Conference on Software Engineering, Boston, Massachusetts, May1997.

[10] S. Benlarbi. *Object-Oriented Design Metrics for Early Quality Prediction*. In Proceeding of ACM SIGPLAN OOPSLA'97 Workshop on Object-Oriented Design Quality, Atlanta, Georgia (USA), October 5-9, 1997.

[11] S. Benlarbi, and W. Melo. *Polymorphism Measures for Design Quality Prediction*. Submitted to ISSRE'98.

[12] J. M. Bieman and B. -K. Kang. *Cohesion and Reuse in an Object-Oriented System*. In Proceedings of ACM Symposium on Software Reusability (SSR '94), pp.259-262, 1995.

[13] A. B. Binkley, and S. R. Schach. *Metrics for Predicting Maintenance Effort in Object Oriented Software: A Java Case Study*. Technical Report TR 97-06, Computer Science Department, Vanderbilt University, 1997.

[14] G. Booch. *Object-Oriented design with Applications*. Benjamin/Cummings Publishing Company Inc., Santa Clara, California, 1994.

[15] L. C. Briand, V. R. Basili, and C. J. Hetmanski. *Developing Interpretable Models with Optimized Set reduction for Identifying High-Risk Software Components*. IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993.

[16] L. C. Briand, V. R. Basili, Y. M. Kim, and D. R. Squier. *A Change Analysis Process to Characterize Software Maintenance Projects*. In Proceedings of the International Conference on Software Maintenance 1994.

[17] L. C. Briand, P. Devanbu, and W. L. Melo. *An Investigation into Coupling Measures for C++*. In 19th IEEE International Conference on Software Engineering, Boston, Massachusetts, May1997.

[18] L. Briand, J. Daly and J. Wüst. *A Unified Framework for Coupling Measurement in Object-Oriented Systems*. Technical report ISERN 96-14, Fraunhofer Institute for Experimental Software Engineering, Germany, 1996.

[19] L. Briand, J. Daly and J. Wüst. *A Unified Framework for Cohesion Measurement in Object-Oriented Systems*. Technical report ISERN 97-05, Fraunhofer Institute for Experimental Software Engineering, Germany, 1997.

[20] L. Briand, S. Morasca, and V. R. Basili. *Defining and Validating High-Level design Metrics*. Technical Report CS-TR-3301-1, Computer Science Department, University of Maryland.

[21] L. Briand, S. Morasca, and V. R. Basili. *Goal Driven Definition of Product Metrics Based on Properties*. Technical Report CS-TR-3346-1, Computer Science Department, University of Maryland.

[22] L. C. Briand, W. M. Thomas, and C. J. Hetmanski. *Modeling and Managing Risk Early in Software Development*. In IEEE International Conference on Software Engineering, 1993.

[23] B. Cestnik, I. Bratko, and I. Konenko. *ASSISTANT 86: A Knowledge Elicitation Tool for Sophisticated Users*. Sigma Press, 1987.

[24] S. Chidamber, and C. Kemerer. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, Vol. 20, No. 6, June 1994.

[25] N. I. Churcher, and M. J. Shepperd. *Comments on "A Metrics Suite for Object-Oriented Design"*. IEEE Transactions on Software Engineering, Vol. 21, No. 3, March 1995.

[26] P. B. Crosby. *Quality Without Tears*. McGraw-Hill, New York. 1984.

[27] P. Devanbu and L. E. Eves. *How to Write a Gen++ Specification*. AT&T, June 1994.

[28] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. *Analytical and Empirical Evaluation of Software Reuse Metrics*. In 18th IEEE International Conference on Software Engineering, Berlin, Germany, 1996.

[29] R. Dion. *Process Improvement and the Corporate Balance Sheet*. IEEE Software, Vol. 10, No. 4, July 1993.

[30] B. Henderson-Sellers. *Software Metrics*. Prentice Hall, Hemel Hempstaed, UK, 1996.

[31] M. Hitz and B. Montazeri. *Measuring Coupling and Cohesion in Object-Oriented Systems*. In Proceedings of International Symposium on Applied Corporate Computing, Monterrey, Mexico, October 1995.

[32] M. Hitz, and B. Montazeri. *Chidamber and Kemerer's Metric Suite: A Measurement Theory Perspective*. IEEE Transactions on Software Engineering, Vol. 22, No. 4, April 1996.

[33] T. M. Khoshgoftaar, and J. C. Munson. *Predicting Software Development Errors Using Software Complexity Metrics*. IEEE Journal on Selected Areas in Comunications, Vol. 8, No.2, February 1990.

[34] T. M. Khoshgoftaar, J. C. Munson, B. B. Bhattacharya, and G. D. Richardson. *Predictive Modeling Techniques of Software Quality from Software Measures*. IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992.

[35] Y. -S. Lee, B. -S. Liang, S. -F. Wu, and F. -J. Wang. *Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow*. In Proceedings of International Conference on Software Quality, Maribor, Slovenia, 1995.

[36] W. Li, and S. Henry. *Object Oriented Metrics that Predict Maintainability*. Journal of Systems and Software, Vol. 23, No. 2, 1993.

[37] H. Lounis, W. L. Melo, and H. Sahraoui. *Identifying and Measuring Coupling in Modular Systems*. Centre de recherche informatique de Montréal, Montréal, Canada, 1997.

[38] N. H. Madhavji. *The Process Cycle*. IEE/BCS Software Engineering Journal, 6(5):234-242, September 1991.

[39] J. C. Munson, and T. M. Khoshgoftaar. *The Detection of Fault-Prone Programs*. IEEE Transactions on Software Engineering, Vol. 18, No. 5, May 1992.

[40] M. A. Ochs. *M System: Calculating Software Metrics from C++ Source Code*. IESE Fraunhofer Report No, 005.98/E, February 1998.

[41] M.C. Paulk, B. Curtis, M. B. Chrissis,and C. V. Weber. *Capability Maturity Model, Version 1.1*. IEEE Software, July 1993.

[42] J. R. Quinlan. *Induction of Decision Trees*. Machine Learning Journal, Vol. 1, No. 1, pp. 81-106, 1986.

[43] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman Publishers, San Mateo, California, 1993.

[44] H. D. Rombach. *Design Measurement; Some Lessons Learned*. IEEE Software, March 1990.

[45] J. Rubin. *Handbook of Usability Testing*. John Wiley and sons, 1994.

[46] R. M. Szabo, and T. M. Khoshgoftaar. *An Assessment of Software Quality in a C++ Environment*. In 6th International Symposium on Software Reliability Engineering, Toulouse, France, October 1995.

[47] R. Thayer (editor). *Software Engineering Project Management*. IEEE Computer Society Press, IEEE 1997.

[48] L. A. Zadeh. *Fuzzy Logic*. IEEE Computer, Vol. 22, No. 4, April 1988.

Appendix A

Predictive Models					
Who	**Technique**	**Independent var.**	**Dependent var.**	**Model Type**	**Accuracy**
Khoshgoftaar and Munson [33]	Lin. R.	complexity metrics	Fault-Proneness	linear function	Rsq= [0.69 - 0.81]
Li and Henry [36]	Lin. R.	OO Design metrics	Maintenance Effort	linear function	Rsq= [0.90, 0.87]
Basili et al. [7]	Lin. R.	SLOC (A,C,D)	Productivity(Effort)	linear function	Rsq= 0.75
Abreu and Melo [2]	Lin. R.	MOOD metrics	Defect Density, Failure Density, Normalized Rework	linear functions	
Agresti and Evanco [3]	Log. R.	some design metrics	Error Density	logistic function	Rsq= 0.74
Briand et al. [15]	OSR, Log. R.	compexity, system architecture	Fault-Proneness	2 X 2	
Briand et al. [20]	Log. R.	some design metrics	Fault-Proneness	logistic function	
Briand et al. [22]	OSR, Log. R.	compexity, system architecture	High Risk components (isolation cost, and completion cost)	2 X 2	
Basili et al. [8]	Log. R.	C&K metrics	Fault-Proneness	2 X 2	76.60%
Devanbu et al. [31]	Log. R.	reuse metrics	Productivity(Effort), Fault density	logistic function	Rsq= [0.51, 0.71], [0.49]
Almeida et al. [4]	MLA (NewID, CN2, C4.5, Foil), Log.R.	complexity metrics	Correction Costs	2 X 2	[52%, 54%, 68%, 71%], 61%
Basili et al. [9]	C4.5	complexity metrics	Correction Costs	2 X 2	66%
Lounis et al. [37]	C4.5	OO Design metrics	Fault-Proneness	2 X 2	78.8%
Munson and Khoshgoftaar [39]	Discriminant Analysis	complexity metrics	Fault-Proneness	2 X 2	75%, 62%
Szabo and Khoshgoftaar [46]	Discriminant Analysis	procedural, and OO metrics	Fault-Proneness	3 X 3	64.7%, 69.1%

Table A.1. A background and related work.

Appendix B

In this appendix we present the experimental results regarding the OO design properties and the defect-density as dependent variable. We used the definitions mentioned in the section 6.4 "Dependent and Independent Variables". We selected the best two-group, and three-group models per design property, as well as the best two-group and three-group multivariate models. Three components of the investigated OO system have not been affected by any change during the (so far) life cycle of the product. Therefore, they were not involved in our analysis.

Bin	*Freq.*
0.001285	1
0.00512	20
0.008955	16
0.01279	19
0.016624	10
0.020459	6
0.024294	3
0.028129	4
More	1
# of cases	80

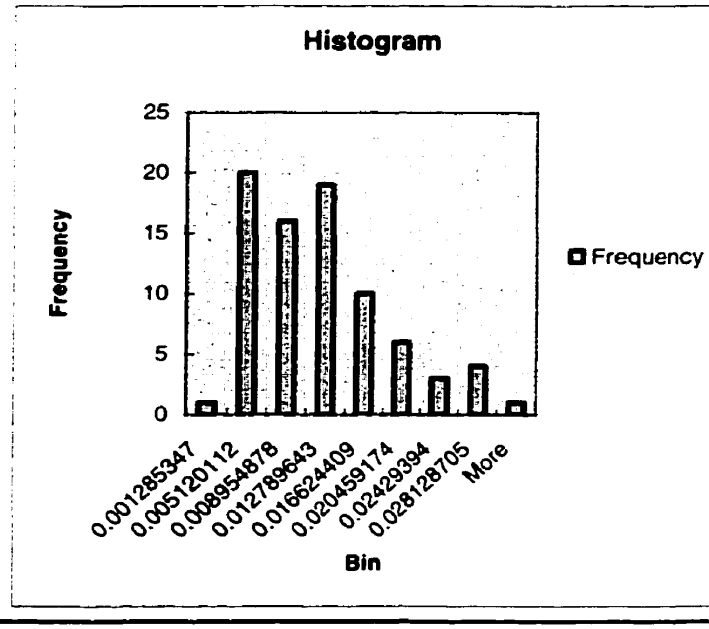
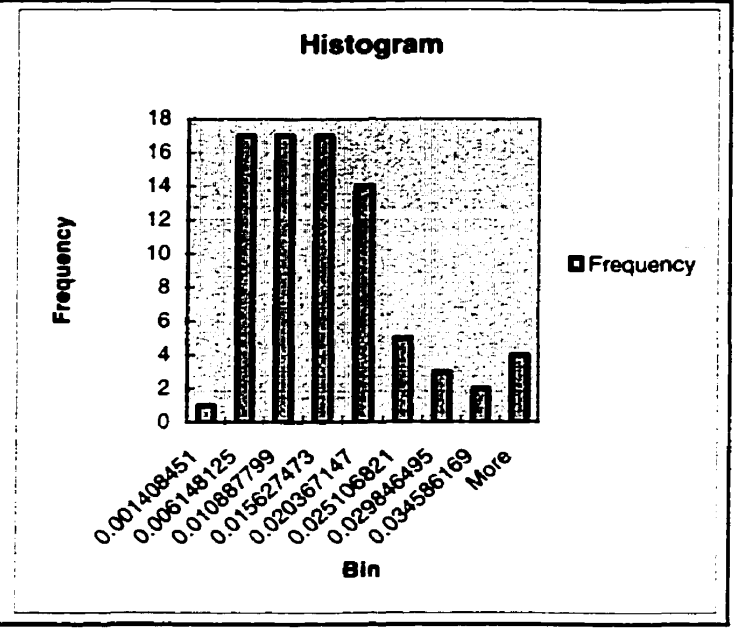


Figure B.1. Defining defect-density A.

Bin	Freq.
0.001408	1
0.006148	17
0.010888	17
0.015627	17
0.020367	14
0.025107	5
0.029846	3
0.034586	2
More	4
# of cases	80



FigureB.2. Defining defect-density B.

Bin	Freq.
0.001653	1
0.009459	19
0.017265	15
0.025072	17
0.032878	16
0.040684	3
0.04849	2
0.056296	3
More	4
# of cases	80

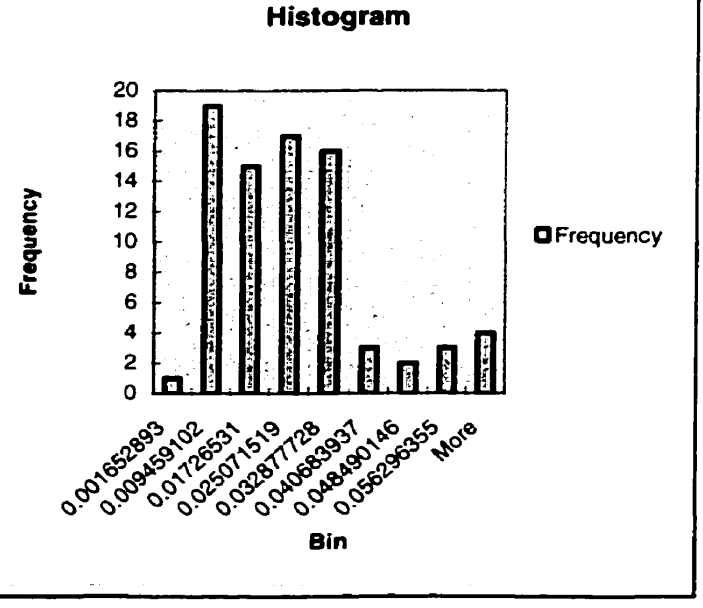


Figure B.3. Defining defect-density C.

Rule 1: NMI > 22 -> class 2 [75.8%]	Rule 2: DIT <= 1 CLD > 0 NOC <= 2 NMO <= 2 -> class 2 [70.7%]	Rule 3: NOP > 0 NMA <= 3 -> class 2 [50.0%]	Rule 4: NOD > 8 -> class 2 [50.0%]
Rule 5: NOD <= 8 NMI <= 22 -> class 1 [72.4%]	Default class: 1		

Figure B.4. Two-group hypothesis 1, defect-density A model.

Rule 1: NMA <= 3 -> class 3 [31.4%]	Rule 2: NMI > 6 NMI <= 22 -> class 1 [82.2%]	Rule 3: DIT > 1 NMA <= 12 -> class 1 [82.2%]	Rule 4: NOP <= 0 NMA > 13 NMA <= 18 -> class 1 [70.0%]
Rule 5: DIT <= 1 NOC <= 3 NMI > 4 NMI <= 6 -> class 2 [70.7%]	Rule 6: NOC <= 3 NOP <= 0 NMA <= 13 -> class 2 [67.3%]	Rule 7: NOC <= 2 NMA > 18 -> class 2 [66.2%]	Rule 8: DIT <= 1 CLD <= 0 SIX > 0.222222 -> class 2 [57.9%]
Rule 9: NMI > 22 -> class 2 [54.6%]	Default class: 1		

Figure B.5. Two-group hypothesis 1, defect-density B model.

Rule 1: NOD > 8 -> class 3 [50.0%]	Rule 2: NMA <= 3 -> class 3 [31.4%]	Rule 3: CLD <= 0 NMI > 6 NMI <= 28 -> class 1 [79.4%]	Rule 4: DIT > 1 CLD <= 0 NMA <= 11 -> class 1 [79.4%]
Rule 5: NOP <= 0 NMA > 13 NMA <= 18 -> class 1 [70.0%]	Rule 6: NOC > 2 NMA > 14 -> class 1 [63.0%]	Rule 7: NOD > 3 NOD <= 8 -> class 1 [50.0%]	Rule 8: NOD <= 3 NMI > 4 NMI <= 6 -> class 2 [75.8%]
Rule 9: NOP <= 0 NOD <= 3 NMA <= 13 -> class 2 [65.5%]	Rule 10: DIT <= 1 CLD <= 0 SIX > 0.222222 -> class 2 [57.9%]	Rule 11: CLD <= 0 NMA > 17 -> class 2 [54.6%]	Rule 12: CLD > 0 NOC <= 1 -> class 2 [50.0%]
Rule 13: NMI > 28 -> class 2 [45.3%]	Default class: 1		

Figure B.6. Three-group hypothesis 1, defect-density B model.

Rule 1: LCOM4 > 18 LCOM4 <= 24 -> class 2 [70.0%]	Rule 2: LCOM5 <= 0.25 -> class 2 [56.6%]	Rule 3: Coh <= 0.05351 -> class 2 [56.6%]	Rule 4: LCOM1 > 17 LCOM4 <= 18 -> class 1 [90.6%]
Rule 5: LCOM5 > 0.125 LCC > 0.6 -> class 1 [86.1%]	Rule 6: Co > 0.12121 LCC <= 0.54131 -> class 1 [79.4%]	Rule 7: LCOM3 <= 10 LCOM4 > 24 ICH <= 1 -> class 1 [74.1%]	Default class: 2

Figure B.7. Two-group hypothesis 2, defect-density C model.

Rule 1: LCOM1 <= 40 Co > -0.1666 LCC <= 0.16666 -> class 3 [70.7%]	Rule 2: LCOM1 > 58 LCC <= 0.16666 -> class 3 [50.0%]	Rule 3: TCC > 0.86666 -> class 3 [50.0%]	Rule 4: LCC <= 0.31818 ICH > 1 -> class 3 [50.0%]
Rule 5: Coh <= 0.54545 Co > 0.23214 -> class 1 [85.7%]	Rule 6: LCOM1 <= 45 LCOM3 > 4 LCC > 0.16666 -> class 1 [77.7%]	Rule 7: LCOM3 > 4 Co > 0.12121 -> class 1 [73.1%]	Rule 8: LCOM1 > 6 TCC > 0.77777 -> class 1 [70.7%]
Rule 9: Coh > 0.05351 LCC <= 0.15151 -> class 1 [70.7%]	Rule 10: LCOM3 <= 4 Coh > 0.54545 Co <= 0.35714 -> class 2 [85.7%]	Rule 11: LCOM3 <= 4 Co <= 0.23214 LCC > 0.16666 -> class 2 [83.3%]	Rule 12: TCC > 0.22222 TCC <= 0.25591 -> class 2 [79.4%]
Rule 13: LCC > 0.16666 TCC <= 0.175 -> class 2 [61.2%]	Rule 14: LCOM3 > 5 LCOM3 <= 7 ICH > 0 -> class 2 [61.2%]	Default class: 1	

Figure B.8. Three-group hypothesis 2, defect-density C model.

Rule 1: RFC_1 <= 18 DAC' <= 1 OCAEC <= 0 -> class 2 [82.0%]	Rule 2: IH-ICP > 18 ACMIC <= 3 -> class 2 [79.4%]	Rule 3: ACMIC <= 0 OCMEC > 7 -> class 2 [75.8%]	Rule 4: CBO <= 2 OMMEC <= 2 -> class 2 [75.8%]
Rule 5: IH-ICP <= 18 OCAEC > 0 OCMEC <= 7 OMMEC > 1 -> class 1 [93.0%]	Rule 6: RFC_1 > 15 IH-ICP <= 18 ACMIC <= 0 OCMEC <= 7 -> class 1 [89.8%]	Rule 7: ACMIC > 0 -> class 1 [79.3%]	Default class: 1

Figure B.9. Two-group hypothesis 3, defect-density A model.

Rule 1: IH-ICP > 16 ACMIC <= 2 -> class 2 [84.1%]	Rule 2: CBO <= 1 -> class 2 [70.7%]	Rule 3: RFC_1 <= 15 OCMEC > 3 -> class 2 [70.7%]	Rule 4: RFC_1 <= 15 OCAEC <= 0 -> class 2 [70.0%]
Rule 5: RFC_1 > 37 ACMIC <= 2 OCMEC > 2 -> class 2 [61.2%]	Rule 6: MPC <= 6 ACMIC > 2 -> class 2 [45.3%]	Rule 7: RFC_oo > 16 IH-ICP <= 16 -> class 1 [78.9%]	Rule 8: ACMIC > 3 -> class 1 [75.8%] Default class: 1

Figure B.10. Two-group hypothesis 3, defect-density B model.

Rule 1: OCMIC <= 0 OCMEC > 49 -> class 3 [50.0%]	Rule 2: CBO <= 1 -> class 3 [45.3%]	Rule 3: RFC_1 <= 15 OCMIC > 0 -> class 2 [80.9%]	Rule 4: IH-ICP > 16 -> class 2 [75.6%]
Rule 5: DCMEC <= 2 OCMEC > 7 -> class 2 [72.2%]	Rule 6: RFC_1 > 15 IH-ICP <= 16 OCMEC <= 7 -> class 1 [77.7%]	Default class: 1	

Figure B.11. Three-group hypothesis 3, defect-density A model.

Rule 1: NIH-ICP <= 5 DCMEC > 0 -> class 3 [70.7%]	Rule 2: CBO <= 1 -> class 3 [45.3%]	Rule 3: CBO > 5 ACMIC <= 0 OCMIC <= 7 DCMEC <= 0 -> class 2 [82.0%]	Rule 4: RFC_1 <= 15 OCMIC > 0 DCMEC <= 0 -> class 2 [79.4%]
Rule 5: IH-ICP > 2 IH-ICP <= 5 OCAEC <= 3 DCMEC <= 0 -> class 2 [79.4%]	Rule 6: DAC > 5 -> class 2 [63.0%]	Rule 7: OCAEC > 3 -> class 2 [54.6%]	Rule 8: RFC_1 > 44 IH-ICP <= 20 DAC <= 5 -> class 1 [84.3%]
Rule 9: RFC_1 > 15 IH-ICP <= 2 -> class 1 [73.5%]	Default class: 2		

Figure B.12. Three-group hypothesis 3, defect-density C model.

Rule 1: LCOM4 > 3 LCOM5 > 0.45454 Coh > 0.33333 Co > 0.06593 -> class 1 [82.0%]	Rule 2: NMI > 22 -> class 1 [75.8%]	Rule 3: CLD > 0 RFC_1 <= 15 -> class 1 [75.8%]	Rule 4: CBO <= 1 -> class 1 [70.7%]
Rule 5: LCOM5 <= 0.5147 IH-ICP <= 18 OCMIC > 0 -> class 0 [93.6%]	Rule 6: NMO <= 4 RFC_1 > 16 IH-ICP <= 18 DCMEC <= 0 -> class 0 [92.5%]	Rule 7: NMO > 8 -> class 0 [85.7%]	Rule 8: NMI <= 22 TCC <= 0.15151 OCMIC > 0 -> class 0 [85.7%]
Default class: 1			

Figure B.13. Two-group multivariate, defect-density A model.

Rule 1: NMA > 13 LCOM4 <= 10 -> class 1 [87.1%]	Rule 2: NMA > 13 DAC <= 1 -> class 1 [85.7%]	Rule 3: NMI <= 22 NMA <= 13 RFC_1 > 44 -> class 1 [84.1%]	Rule 4: ICP <= 9 DCMEC > 0 -> class 3 [70.7%]
Rule 5: NMI > 28 LCOM3 > 5 -> class 3 [50.0%]	Rule 6: CBO <= 1 -> class 3 [45.3%]	Rule 7: NMO <= 9 NMI <= 8 NMA <= 16 SIX <= 0.470588 LCOM3 <= 10 Co <= 0.35714 LCC > 0.183 MPC <= 23 OCAEC <= 3 DCMEC <= 0 -> class 2 [85.1%]	Rule 8: NMI > 22 Co > 0 -> class 2 [63.0%]
Rule 9: OCAEC > 3 -> class 2 [54.6%]	Default class: 1		

Figure B.14. Three-group multivariate, defect-density C model.

Tested 80, errors 11 (13.8%)			
	predicted 0	predicted 1	Completeness
real 0	**56**		**100.00%**
real 1	**11**	**13**	**54.17%**
Correctness	**83.58%**	**100.00%**	
Accuracy=	**86.25%**		
X-sqr=	36.2189		
p <=	0.0000		

Table B.1. Evaluation of the two-group hypothesis 1, defect-density A model.

Tested 80, errors 12 (15.0%)			
	predicted 0	predicted 1	Completeness
real 0	**52**		**100.00%**
real 1	**12**	**16**	**57.14%**
Correctness	**81.25%**	**100.00%**	
Accuracy=	**85.00%**		
X-sqr=	37.1429		
p <=	0.0000		

Table B.2. Evaluation of the two-group hypothesis 1, defect-density B model.

Tested 80, errors 12 (15.0%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**31**	**3**	**1**	**88.57%**
real 2	**3**	**33**		**91.67%**
real 3	**1**	**4**	**4**	**44.44%**
Correctness	**88.57%**	**82.50%**	**80.00%**	
Accuracy=	**85.00%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	29.3969	48.0409	21.8661	18.2838
p <=	0.0000	0.0000	0.0000	0.0000

Table B.3. Evaluation of the three-group hypothesis 1, defect-density B model.

Tested 80, errors 8 (10.0%)			
	predicted 0	predicted 1	Completeness
real 0	**49**	**3**	**94.23%**
real 1	**5**	**23**	**82.14%**
Correctness	**90.74%**	**88.46%**	
Accuracy=	**90.00%**		
X-sqr=	48.3917		
p <=	0.0000		

Table B.4. Evaluation of the two-group hypothesis 2, defect-density C model.

Tested 80, errors 6 (7.5%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**34**	**1**		**97.14%**
real 2	**3**	**30**		**90.91%**
real 3	**1**	**1**	**10**	**83.33%**
Correctness	**89.47%**	**93.75%**	**100.00%**	
Accuracy=	**92.50%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	42.9666	53.0891	39.7403	36.0704
p <=	0.0000	0.0000	0.0000	0.0000

Table B.5. Evaluation of the three-group hypothesis 2, defect-density C model.

Tested 80, errors 4 (5.0%)			
	predicted 0	predicted 1	Completeness
real 0	**56**	**0**	**100.00%**
real 1	**4**	**20**	**83.33%**
Correctness	**93.33%**	**100.00%**	
Accuracy=	**95.00%**		
X-sqr=	62.2222		
p <=	0.0000		

Table B.6. Evaluation of the two-group hypothesis 3, defect-density A model.

Tested 80, errors 11 (13.8%)			
	predicted 0	predicted 1	Completeness
real 0	**49**	**3**	**94.23%**
real 1	**1**	**27**	**96.43%**
Correctness	**98.00%**	**90.00%**	
Accuracy=	**95.00%**		
X-sqr=	63.8242		
p <=	0.0000		

Table B.7. Evaluation of the two-group hypothesis 3, defect-density B model.

Tested 80, errors 11 (13.8%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**32**	**3**		**91.43%**
real 2	**2**	**30**	**1**	**90.91%**
real 3	**2**	**3**	**7**	**58.33%**
Correctness	**88.89%**	**83.33%**	**87.50%**	
Accuracy=	**86.25%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	33.3347	48.5245	30.0131	21.4664
p <=	0.0000	0.0000	0.0000	0.0000

Table B.8. Evaluation of the three-group hypothesis 3, defect-density C model.

Tested 80, errors 1 (1.2%)			
	predicted 0	predicted 1	Completeness
real 0	**55**	**1**	**98.21%**
real 1		**24**	**100.00%**
Correctness	**100.00%**	**96.00%**	
Accuracy=	**98.75%**		
X-sqr=	75.4286		
p <=	0.0000		

Table B.9. Evaluation of the two-group multivariate, defect-density A model.

Tested 80, errors 9 (11.2%)				
	predicted 1	predicted 2	predicted 3	Completeness
real 1	**33**	**2**		**94.29%**
real 2	**3**	**29**	**1**	**87.88%**
real 3	**2**	**1**	**9**	**75.00%**
Correctness	**86.84%**	**90.63%**	**90.00%**	
Accuracy=	**88.75%**			
Model	Average	1 <-> 2	1 <-> 3	2 <-> 3
X-sqr=	37.4888	48.479	33.9429	30.0444
p <=	0.0000	0.0000	0.0000	0.0000

Table B.10. Evaluation of the three-group multivariate, defect-density C model.

	Hypothesis 1: Models					
	A: 2 x 2		B: 2 x 2		B: 3 x 3	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'
DIT	5.00%	4.31%	9.26%	7.87%	5.13%	4.36%
AID	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
CLD	5.00%	4.31%	0.00%	0.00%	16.67%	14.17%
NOC	5.00%	4.31%	5.56%	4.72%	7.69%	6.54%
NOP	10.00%	8.63%	0.00%	0.00%	6.41%	5.45%
NOD	30.00%	25.88%	13.89%	11.81%	21.79%	18.53%
NOA	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NMO	5.00%	4.31%	26.85%	22.82%	0.00%	0.00%
NMI	30.00%	25.88%	15.74%	13.38%	15.38%	13.08%
NMA	10.00%	8.63%	25.00%	21.25%	24.36%	20.71%
SIX	0.00%	0.00%	3.70%	3.15%	2.56%	2.18%
total:	100.00%	86.25%	100.00%	85.00%	100.00%	85.00%
RUD	20.00%		11.11%		7.69%	
Acc	86.25%		85.00%		85.00%	

Table B.11. Hypothesis 1 defect-density models and the contribution of each metric.

	Hypothesis 2: Models			
	C: 2 x 2		C: 3 x 3	
Measure	MUD	MUD'	MUD	MUD'
LCOM1	7.14%	6.43%	11.90%	11.01%
LCOM2	0.00%	0.00%	0.00%	0.00%
LCOM3	4.76%	4.29%	14.29%	13.21%
LCOM4	26.19%	23.57%	0.00%	0.00%
LCOM5	21.43%	19.29%	0.00%	0.00%
Coh	14.29%	12.86%	9.52%	8.81%
Co	7.14%	6.43%	14.29%	13.21%
LCC	14.29%	12.86%	21.43%	19.82%
TCC	0.00%	0.00%	14.29%	13.21%
ICH	4.76%	4.29%	14.29%	13.21%
total:	100.00%	90.00%	100.00%	92.50%
RUD	14.29%		7.14%	
Acc	90.00%		92.50%	

Table B.12. Hypothesis 2 defect-density models and the contribution of each metric.

	Hypothesis 3: Models							
	A: 2 x 2		B: 2 x 2		A: 3 x 3		C: 3 x 3	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'	MUD	MUD'
CBO	7.14%	6.79%	12.50%	11.88%	16.67%	14.38%	13.89%	11.98%
CBO'	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RFC_1	8.33%	7.92%	16.67%	15.83%	13.89%	11.98%	12.96%	11.18%
RFC_oo	0.00%	0.00%	6.25%	5.94%	0.00%	0.00%	0.00%	0.00%
MPC	0.00%	0.00%	6.25%	5.94%	0.00%	0.00%	0.00%	0.00%
ICP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
IH-ICP	14.29%	13.57%	12.50%	11.88%	22.22%	19.17%	12.96%	11.18%
NIH-ICP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	5.56%	4.79%
DAC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	14.81%	12.78%
DAC'	4.76%	4.52%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
IFCAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ACAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
FCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAEC	8.33%	7.92%	6.25%	5.94%	0.00%	0.00%	14.81%	12.78%
IFCMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ACMIC	32.14%	30.54%	29.17%	27.71%	0.00%	0.00%	2.78%	2.40%
OCMIC	0.00%	0.00%	0.00%	0.00%	16.67%	14.38%	6.48%	5.59%
FCMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DCMEC	0.00%	0.00%	0.00%	0.00%	8.33%	7.19%	15.74%	13.58%
OCMEC	14.29%	13.57%	10.42%	9.90%	22.22%	19.17%	0.00%	0.00%
IFMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
AMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
FMMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DMMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OMMEC	10.71%	10.18%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
total:	100.00%	95.00%	100.00%	95.00%	100.00%	86.25%	100.00%	86.25%
RUD	14.29%		12.50%		16.67%		11.11%	
Acc	95.00%		95.00%		86.25%		86.25%	

Table B.13. Hypothesis 3 defect-density models and the contribution of each metric.

	Multivariate Models					
	2 x 2		3 x 3		Average	
Measure	MUD	MUD'	MUD	MUD'	MUD	MUD'
DIT	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
AID	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
CLD	6.25%	6.17%	0.00%	0.00%	1.56%	1.43%
NOC	0.00%	0.00%	11.46%	10.77%	0.00%	0.00%
NOP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NOD	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NOA	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
NMO	15.63%	15.43%	4.17%	3.92%	13.69%	12.49%
NMI	16.67%	16.46%	0.00%	0.00%	5.43%	4.96%
NMA	0.00%	0.00%	0.00%	0.00%	2.65%	2.42%
SIX	0.00%	0.00%	0.00%	0.00%	0.19%	0.17%
LCOM1	0.00%	0.00%	0.00%	0.00%	1.67%	1.52%
LCOM2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LCOM3	0.00%	0.00%	0.00%	0.00%	1.11%	1.01%
LCOM4	3.13%	3.09%	0.00%	0.00%	7.38%	6.74%
LCOM5	7.29%	7.20%	0.00%	0.00%	1.22%	1.11%
Coh	3.13%	3.09%	0.00%	0.00%	0.52%	0.48%
Co	3.13%	3.09%	16.67%	15.66%	2.67%	2.44%
LCC	0.00%	0.00%	13.54%	12.73%	1.11%	1.01%
TCC	4.17%	4.11%	0.00%	0.00%	0.69%	0.63%
ICH	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
CBO	12.50%	12.34%	4.17%	3.92%	9.72%	8.87%
CBO'	0.00%	0.00%	15.63%	14.68%	0.00%	0.00%
RFC_1	9.38%	9.26%	4.17%	3.92%	11.09%	10.12%
RFC_oo	0.00%	0.00%	0.00%	0.00%	1.67%	1.52%
MPC	0.00%	0.00%	0.00%	0.00%	0.19%	0.17%
ICP	0.00%	0.00%	6.25%	5.87%	0.93%	0.84%
IH-ICP	7.29%	7.20%	0.00%	0.00%	11.39%	10.39%
NIH-ICP	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DAC	0.00%	0.00%	0.00%	0.00%	0.93%	0.84%
DAC'	0.00%	0.00%	3.13%	2.94%	0.00%	0.00%
ACAIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAIC	0.00%	0.00%	9.38%	8.81%	0.00%	0.00%
DCAEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OCAEC	0.00%	0.00%	0.00%	0.00%	2.04%	1.86%
ACMIC	0.00%	0.00%	4.17%	3.92%	1.11%	1.01%
OCMIC	8.33%	8.23%	0.00%	0.00%	5.24%	4.78%
DCMEC	3.13%	3.09%	0.00%	0.00%	6.77%	6.18%
OCMEC	0.00%	0.00%	7.29%	6.85%	6.25%	5.70%
AMMIC	0.00%	0.00%	0.00%	0.00%	2.78%	2.53%
OMMIC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
DMMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
OMMEC	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	100%	98.75%	100%	93.98%	100%	91.25%
RUD	12.50%		12.50%		15.46%	
Acc	98.75%		93.98%		91.25%	

Table B.14. Multivariate defect-density models and the contribution of each metric.