2M11.3103.7

Université de Montréal

Determination of Software Quality Through a Generic Model

par

Nouha Mehio

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Juin 2003

QA
76
U54
2004
V.001

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Determination of Software Quality Through a Generic Model

présenté par:

Nouha Mehio

A été évalué par un jury composé des personnes suivantes :

Philippe LANGLAIS
président-rapporteur

Houari SAHRAOUI
directeur de recherche

Hakim LOUNIS
codirecteur

Jian-Yun NIE
membre du jury

Mémoire accepté le : 2 décembre 2003

Résumé

Ce mémoire introduit une approche pour l'évaluation de la qualité du logiciel : **B**oîte à **O**util d'**A**nalyse de **P**rogrammes (BOAP). BOAP a été développée dans le cadre d'un projet pour le **C**entre du **T**est de **L**ogiciel (CTL). Le but de l'approche BOAP est d'évaluer l'architecture du logiciel indépendamment du langage de programmation en prenant en compte des caractéristiques architecturales comme l'évolvabilité et la stabilité. Cet objectif est atteint grâce à un module de mapping qui transforme les programmes en une représentation indépendante du langage de programmation soit un modèle générique. Le modèle généré est persistant; il permet donc son utilisation pour des analyses futures. Pour ce projet, le module de mapping a été développé pour supporter la transformation des programmes Java vers le modèle générique.

Dans le cadre de ce projet, la stabilité des interfaces des classes dans les systèmes orientés objet est évaluée durant leurs évolutions. Notre étude empirique a été réalisée à partir de données tirées des applications Java Open Source à plusieurs versions. Le module de mapping a été utilisé pour transformer ces applications en modèle générique. Un module métrique a été développé afin d'extraire, à partir de la représentation générée, des métriques de conception qui évalueront des propriétés orientées objet qui ont un effet sur la stabilité des interfaces des classes comme le couplage, l'encapsulation et l'héritage.

Un modèle de classification a été généré pour prédire les classes qui auront un impact sur la stabilité du système. Nous avons utilisé l'algorithme d'apprentissage C4.5 pour produire nos modèles prédictifs.

L'utilisation du modèle générique a l'avantage d'avoir les analyses et les modèles générés applicables à d'autres systèmes, indépendamment de leur langage de programmation.

Mots-clés: outils d'évaluation de qualité, représentation générique, modèle de qualité, stabilité, métrique de conception, algorithme d'apprentissage

Abstract

This thesis introduces an approach for system assessment: **BO**x for **A**nalyzing **P**rograms (BOAP) developed for the **C**entre du **T**est de **L**ogiciel (CTL). The driving goal of the BOAP approach is the assessment of architectural quality of any program with regard to important characteristics like evolvability and stability independently of its programming language. It is characterized by a mapping module that will transform any program into a language-independent representation i.e. a generic model. The generated model is persistent, and can therefore be retrieved and manipulated by different systems for quality evaluation and various analyses. Until now, BOAP supports only C/C++ language, therefore for this project we enhanced BOAP by supporting Java language.

In this project we investigated interface stability for object-oriented class of systems throughout their evolution. An empirical study was conducted on various versions of Open Source Java applications. These applications were mapped into the generic model using BOAP. A metric module was developed to read from the representation and to extract design metrics that evaluate object-oriented properties affecting interface stability like coupling, encapsulation and inheritance.

A classification model was built to predict which class is likely to have an unstable impact on the system. C4.5 machine learning language was used to build the predictive model. Since the predictive model was done out of the generic model, it has the advantage to be applicable on other systems independently of their programming language.

Keywords: quality assessment tool, generic representation, quality model, stability, design metric, machine learning,

Contents

List of Tables

List of Figures

Abbreviations

ASG:	**A**bstract **S**emantic **G**raphs
AST:	**A**bstract **S**yntax **T**ree
BOAP:	**B**oîte à **O**util d'**A**nalyse de **P**rogrammes
CRIM:	**C**entre de **R**echerche **I**nformatique de **M**ontréal
CTL:	**C**entre du **T**est de **L**ogiciel
GLIC :	**G**énie **L**ogiciel et **I**ngénierie de la **C**onnaissance
ISO:	**I**nternational **O**rganization for **S**tandardization
OO:	**O**bject **O**riented

Acknowledgement

Throughout my master, I have been fortunate to have the support and help of many people.

First I want to express my gratitude and my appreciation to my supervisor Dr. Houari Sahraoui for introducing me to the field of software quality. In my professional career this master project helped me tremendously in having a criticized judgment on how to design quality software. And I also appreciate the support, the guidance and the encouragement he provided throughout the development of this work.

Also, I want to thank my co-supervisor Dr. Hakim Lounis that gave me the opportunity to do my master research at CRIM (Centre de Recherche Informatique de Montreal) with the GLIC (Genie Logiciel et Ingénierie de la Connaissance) group where I learned a lot in the field of software quality and other software engineering related fields. At CRIM, I had the privilege to work with ElHachemi Alikacem; I want to thank him for his great support.

Thanks, Ali and Hala for your love, friendship and for simply being part of my life.

Finally, thank you Haitham, my partner of life, for being always here for me.

I dedicate this master thesis to my parents, Ziad and Amira. I have been blessed with their guidance, their advice, their dedication and their continuous support throughout my education and my life in general.

Chapter 1

1. Introduction

Software evolves with time due to changes in technologies, in customer requirements, hardware, etc. The lack of stability in evolving software can lead to poor product quality, increased project cost, and lengthened project schedule. Software instability is due mainly to poor architecture. Badly designed software will have an unstable architecture that will be difficult to adapt to changes. With the introduction of new changes and functionalities, the architecture tends to drift from its original architecture and becomes more complex and instable, which reduces system maintainability. Unless actions are taken, software maintenance will become increasingly difficult and risky. Therefore there is a need in evaluating and predicting the stability of existing software, and having defined criteria for developing stable software. Industries are looking for tools capable of evaluating system evolution for controlling its quality.

1.1 Quality Model

Predicting whether a component has an instable impact on the system or not is achieved through a quality model. Most of the existing evaluation tools are based on a defined quality model. Such models list major attributes that a high-quality software product should possess (e.g., reliability and maintainability). Many quality models and metrics have been suggested by the literature. One of the earliest software quality models was suggested by McCall [36]. McCall's quality model defines software-product qualities as a hierarchy of factors, criteria, and metrics and was the first of several models. International efforts have also led to the development of a standard for software-product quality measurement, ISO9126[25]. It proposed six characteristics that can define software quality: functionality, reliability, efficiency, usability, maintainability and portability. ISO9126 quality model classified stability as a sub-characteristic of maintainability. These characteristics, called quality factors, are high-level external factors and cannot be measured directly from the system being assessed. These key factors are further decomposed into lower-level internal factors called quality criteria. They include coupling, cohesion, encapsulation and modularity. These internal quality factors are decomposed into a set of low-level, directly measurable attributes called quality metrics. Primitive metrics (direct measures) are combined to form composite metrics (indirect measures). The distinction between external and internal factors is important, as the satisfaction of the internal qualities should ensure the satisfaction of external qualities. These relationships are represented in the figure below.

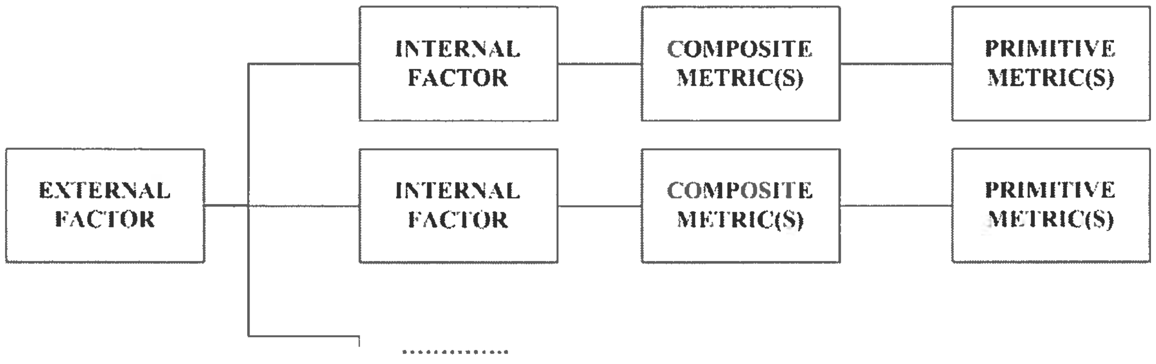


Figure 1: Quality Model Relationships

A quality model is developed using a statistical or machine learning modeling technique, or a combination of techniques. This is done using historical data. Once constructed, such a model takes as input the metrics extracted for a particular component and produces a prediction of the stability (stable or instable) for that component as shown in Figure 2.

Extracted Metrics
Quality Model
Predicted Stability

Figure 2: Overview of Quality Model

1.2 Motivation and Aim

Commercial and research-based tools for automatic data collection and analysis, called quality measurement and evaluation tools, have been developed to help in the management of software quality. Most of those tools extract metrics from the source code. They can be specific for a programming language or they can treat a variety of languages. Usually, depending on the organization's goals, assessment tools specialize for a set of metrics and assess specific characteristics. They can also be customizable by allowing the user to add new metrics and specify quality model.

Evolving software require a tool capable of evaluating change impact on software stability. Thus assessing the initial design of the system, estimating the changes in design between versions and predicting its stability in future releases using quality rules and criteria. This project takes into consideration the need for such a tool. Therefore, we adopted an approach for the architecture of a tool for system assessment: **BO**x for **A**nalyzing **P**rograms (BOAP). It is developed by a group of researchers at Centre de Recherche de Montréal (CRIM) as part of a project for the software-testing center, Centre de Tests du Logiciel (CTL). This approach is based on transforming the source code of the system into a language-independent representation i.e., a generic model[1]. This representation will be used to extract design metrics for software stability assessment.

The aim of this thesis is twofold. First, we want to enhance BOAP by supporting Java application. Second, we want to generate quality rules that will evaluate and predict class interface stability in object oriented systems. The use of the generic model generated by BOAP will allow us to extract metric from a language-independent representation thus having the ability of having generic rules applicable on other systems. The rules will be generated using machine-learning technique.

1.3 Project Strategy

My master thesis will take the following steps:

1. Adding Java language support to the mapping module.

   Until now the mapping module is done for the C/C++ language. The choice relied on Java because of its increasing popularity as the language of choice of the industries, especially in the development of distributed systems and web based applications. This is due to Java's portability and platform independence.

2. Building the BOAP repository with java applications with multiple versions.

   We chose open-source applications available on the web. Many open-source projects adopt CVS as version control platform; so we can have access to any version of source code programs. We will be able to study the changes done on the software throughout its evolution.

3. Defining internal factors that affect software stability.

   In our study, we focus on interface stability of classes for Object Oriented (OO) systems. Metrics capturing important structural characteristics like coupling, encapsulation, and inheritance have been chosen from the literature. A module for metric extraction has been developed. It reads and extracts measures from the generic representation.

4. Classifying interface stability of classes in OO applications.

   We classified an interface to be instable if its change will impact the structure of the system. The severity of this impact will depend on the visibility of the interface in the system. A module for stability classification has been developed. It compares versions and looks for changes in the interface and classifies it depending on its visibility.

5. Generating predictive models out of the extracted metrics by using C4.5 a machine-learning technique.

   A predictive model is built using the set of metrics extracted from the generic model. The predictive model will assist software developers to identify instable components.

The advantage of this predictive model is that it can be applied to any system independently of its language of implementation.

1.4 Thesis Organization

This thesis is organized as follow: chapter II will define software stability and will expose some related research. It will then introduce some of the architecture of some existing tools to measure software quality and it will survey some interesting object-oriented design metrics proposed by the literature. Chapter III introduces the generic metamodel and the architecture of the system BOAP followed by a description of the metrics that are extracted in terms of the metamodel. Chapter IV will detail the mapping of systems in Java into the metamodel. And a case study on the stability of software during their evolution will follow in chapter V. And finally, chapter VI attempts to draw some conclusion.

Chapter 2

2. State of the Art

Pressman estimated that 60% of the total effort of the software development is devoted to its maintenance, from which 80% is due to software evolution [41]. The effort of the maintenance is inversely proportional to the stability of the software. Consequently, it is important to evaluate the stability of the software during its evolution as an indicator of effort and cost.

In this respect, we will first define software stability and expose some of the research done in this field. And second, we will introduce some tools and design metrics that are used to assess quality characteristics and finally we will suggest a suitable approach to assess systems during their evolution.

2.1 Software Stability

The classic definition of the word stability is: "Not easily moved, changed, destroyed, or altered"[1].

Unlike hardware, software should not deteriorate with time. Unfortunately, it does and this is due mainly to changes in software. In fact, changes to software result in the degradation of the software structure. Parnas D. states "the software structure is deteriorated as the software is modified to the point where nobody understands the modified product" [40].

[1] The Concise Oxford Dictionary, 6th edition.

Long-lived software must adapt to several environmental changes including increased user expectations, new user requirements, software modifications, new hardware platforms, new operating systems, new compiler technology, new networking technologies, and new development languages and techniques to name a few. The software engineering communities have identified this problem. However as stated by Fayad M., "the software engineering techniques outlined by many software-engineering authors have not achieved an adequate amount of stability in software projects" [18].

Note that not all software requires the same level of stability. It really depends on the size of the application itself: the larger the system the higher the importance of stability.

So the key question is how designers and architects build stable software products?

Martin R. suggests that software developers should apply the Stable Dependencies Principle and the Stable Abstractions Principle to achieve more changeable architecture[32]. The author defines stability as the likelihood that a given module will change. Since it is certain that some modules must change to meet changing requirements Martin recommends that the key to module design is that "the dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is." The Stable Abstractions principle is defined as: "Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstraction of a package should be in proportion to its stability."

Recently, M. Fayad has introduced a new software engineering paradigm: the Software Stability Model[19]. The main objective of this model is the provision of a method to produce *stable* software products avoiding the reengineering of the system due to changes or extensions in the initial requirements.

This model introduces the concepts of Enduring Business Themes (EBTs) and Business Objects (BOs) as a solution to the reengineering problem. The idea is to identify those aspects of the environment in which the software will operate that will not change, and to cater the software to those areas. This yields a stable core design and, thus, a stable software product. The changes that will be introduced to the software project will then be in the periphery. The objects that do not remain constant over time are considered

Industrial Objects (IOs). Therefore, IOs should be moved to the periphery so that changes in them do not have ripple effect through the rest of the system. The table below lists seven criteria for distinguishing between EBTs, BOs and IOs.

Criteria	*Enduring Business Themes*	*Business Objects*	*Industrial Objects*
Stability Over Time	Stable over time	Externally stable	Unstable
Adaptability	Adaptable without change	Adaptable through internal change	Not necessarily adaptable
Essentiality	Essential	Essential	Replaceable
Explicitness	Implicit	Implicit or explicit	Explicit
Commonality to the Domain	Core	Core	Peripheral
Tangibility	Conceptual	Semi-tangible	Tangible

Table 1: A summary of the identification criteria for EBTs, OBs, and IOs.

These concepts provide a basis for the measurement and comparison of different architectures. Like other quality characteristics, stability can be evaluated through design metrics. Only recently the stability characteristic have been evaluated on object-oriented applications and more specifically on framework architecture. Object-oriented paradigm is relatively recent and the application of OOD languages to build systems is as recent. The stability of software will only appear with time, when changes start to be introduced to the system. Therefore there are only few existing studies that deal with the evaluation of stability of software.

Bansiya J. introduces a method to evaluate framework architecture characteristics and stability, which is based on a quantitative assessment of the changes in framework versions using object-oriented design metrics[7]. The author uses metrics to compute the structural *extent-of-change* in the framework architecture between releases. This extent-of-change is used as a measure of stability in framework architecture. The value of the measure indicates the relative stability of the architecture structure. Higher values of the measure are reflective of higher instability in the structure; values closer to zero indicate greater stability.

As frameworks evolve, the static and interaction (dynamic) structure of frameworks changes due to additions, deletions, or the modification of relationships between framework classes, the addition of new classes, or the removal of existing classes. Bansiya identifies two categories of evolution characteristics, architectural and individual class characteristics listed below.

Metric Name	*Description*
DSC: Design Size in Classes	Count of the total number of classes in the system.
NOH: Number of Hierarchies	Count of the number of class hierarchies in the system.
NSI: Number of Single Inheritances	Count of the number of classes (sub classes) that use inheritance in the system.
NMI: Number of Multiple Inheritances	Count of the number of instances of multiple inheritance in the system.
ADI: Average Depth of Inheritance	Average depth of inheritance of classes in the system. Computed by dividing the sum of maximum path lengths to all classes by the number of classes. Path length to a class is the number of edges from the root to the class in an inheritance hierarchy.
AWI: Average Width of Inheritance	Average number of children per class in the system. Computed by dividing the sum of the number of children over all classes by the number of classes in the system.

Table 2: System design metrics for static architectural structure assessment

Metric Name	***Description***
ACIS: Average number of Services	Count of the number of public methods in a class
ANA: Average number of Parents	Count of the number of distinct (parent) classes from which a class inherits. This metric is different from the depth of inheritance metric because it takes into account instances of multiple inheritances.
ADCC: Average Direct Class Coupling	Count of the different number of classes that a class is directly related to. Includes classes that are directly related by attribute declarations and parameter declarations in methods.

Table 3: Class metrics that influence interaction (dynamic) structural characteristics

Bansiya J. applied the assessment of stability on available frameworks. The result showed that the extent-of-change measure values increase in the early versions of the frameworks and drops close to zero in the later versions. The early versions, which have extent-of-change values significantly higher than zero, represent versions that are evolving with changing architecture whereas releases with extent-of-change's close to zero, represent versions with stable architecture.

Mattson M. uses the same approach introduced by Bansiya to assess a commercial framework[6][34]. Out of the metric results obtained, Mattson M. formulated the following hypotheses:

Hypothesis 1: *a stable framework tends to have narrow and deeply inherited class hierarchy structures, characterized by high values for the average depth of inheritance of classes and low values for the average width of inheritance hierarchies.*

Hypothesis 2: *a sable framework has an NSI/DSC ratio just above 0.8 if multiple* inheritance *is seldom used in the framework, i.e. the number of subclasses in a stable framework is just above 80%.*

Hypothesis 3: *a stable framework has the normalized ADCC metrics going towards 1.0.*

This indicates that the number of relationships is not increasing with newer versions.

Hypothesis 4: *a stable framework has a low value of Extent-of-change below 0.4.*

Hypothesis 5: *it takes the development and release of three to five versions to produce a stable framework.*

2.1.1 Conclusion

We exposed some of the research done on software stability. Some of the factors that cause software instability were identified. Proposed models and measures that can help designers to assess software stability were discussed. It can be concluded that a system is stable if its structure does not change during its evolution. The main factor that causes the system to be instable is mainly the level of class dependencies. In a bad design, any change in a class can have a ripple effect on the system, thus making it instable. Depending on the extent-of-change level, the system can be reengineered to support the new change.

Several object-oriented measures are used to assess the quality of system's architecture, like: coupling, encapsulation, cohesion, inheritance, modularity and others. From those measures we select the ones that are related to measuring the structure of the system: interconnection and interaction between classes to assess the interface stability of the system.

Coupling, encapsulation and inheritance are selected as measures of stability. Those measures are already used to assess the stability of frameworks, and hypotheses were formulated out of the metrics extracted. We want to assess interface stability in OO applications. A change in class interface will impact the structure of the system thus the stability if it is accessible by at least an application module. The severity of this impact will depend on the visibility of the interface in the system.

In object-oriented systems there are 3 main levels of visibility: *Public, Protected*, and *Private*. A class declared *public* can be accessed by the application, *protected* is limited to the module and *private* to the class itself. Therefore the visibility of the class plays an important role in determining the level of interaction between classes.

The overall architecture of the system can be pictured as a set of classes interacting with one another. The nature of the link between two interacting class depends on the type of relationship established between them. The different class interactions are illustrated in the figure below as defined by Martin C. [32].

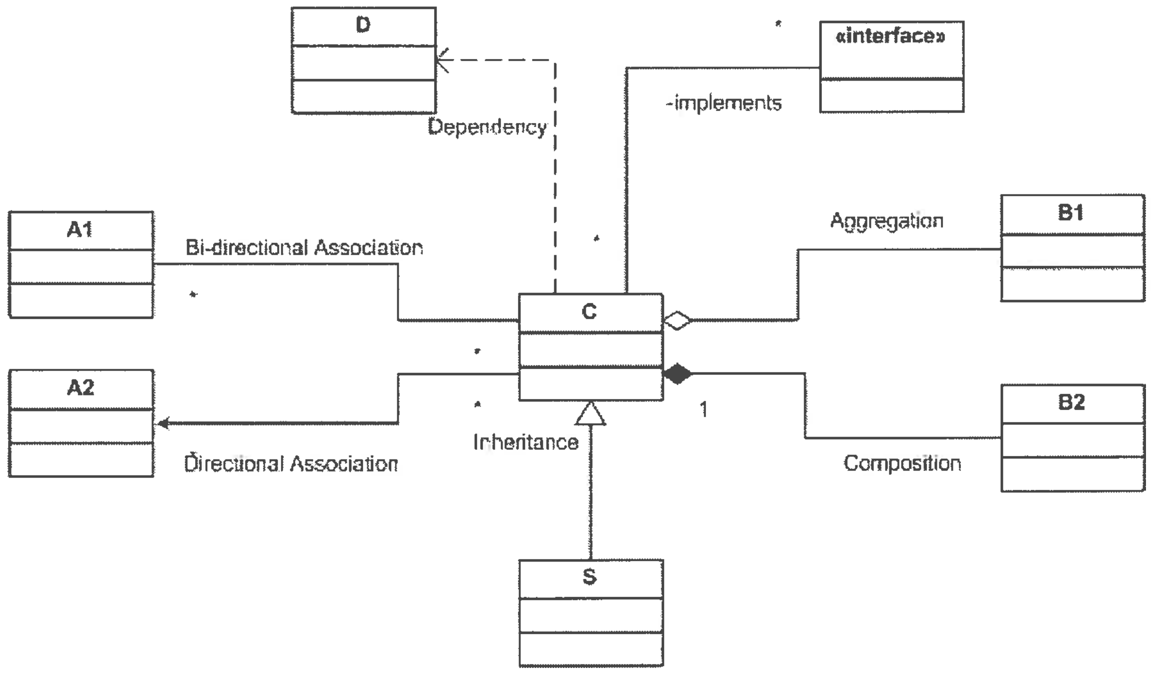


Figure 3: Class Interactions

In the following section we will expose some of the design metrics that we consider being associated to interface stability.

2.2 Object-Oriented Design Metrics

Design metrics assess the internal/external structure, relationships, and functionality of software components. Object-oriented development can be analyzed at both system and class levels. At the system level, the external structural characteristics of the system are analyzed. At this level, class hierarchies, and the relationships between classes are evaluated. At the class level, internal characteristics, components used in the assessment include methods, signatures of the methods, the number and types of attribute declarations in the class. The research community has proposed many different metrics for object-oriented systems. The specification of metrics was done according to the different research and literature exposed in the previous section.

Some metrics have been specified to assess the stability characteristic:

- Number of parameters referenced
- Number of global variables
- Number of parameters changed
- Number of called relationships

In this section, we will focus on some important design metrics that evaluate major OO concepts: methods, classes, encapsulation, coupling, and inheritance.

The metrics used for this project are defined by Chidamber and Kemerer[13], Briand and al.[11], and Bansiya J.[8].

2.2.1 Encapsulation

Encapsulation is defined as the enclosing data and behavior within a single construct. In object-oriented designs, encapsulation refers to designing classes that prevent access to attribute declaration by defining them private, thus protecting the internal representation of the object. Encapsulation contributes in "information hiding", and it offers two kinds of security:

1. Protects object's internal state from being changed from outside users.
2. Changes can be done to the behavior implementation without affecting other objects.

When users of an object, example methods, depend upon how the object is represented, or implemented they introduce a dependency that requires the user of the object also to be changed when the object changes. By hiding the internal representation (encapsulation), and thereby forcing the user of the object to use the object interface, objects have the flexibility to change their internal representation without cascading the changes to user of the object[22]. Encapsulation reduces the dependency between classes, thus making the structure stable to changes. Thus, encapsulation promotes flexibility, and reusability.

Bansiya J defines the following metric:

DAM (Data Access Metric)

DAM is the ratio of the number of private and protected attributes to the total number of attributes declared in the class. A high value for DAM is desired.

$$DAM(c) = \frac{\sum attribute(c)_{private} + \sum attribute(c)_{protected}}{\sum attribute(c)}$$

2.2.2 Coupling

Coupling is the degree of interdependence between two or more components. The fewer the connection, the less chance there is for the ripple effect. Ideally, changing one component should not require a change in another. The coupling between two classes is a measure of how they depend on each other. The general goal is to minimize the amount of coupling between classes.

Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to systems, which cannot change easily without changing many other classes. Such a system becomes hard to adapt, and maintain. On the other hand loose coupling increases the probability that a class can be reused by itself and a system that can be adapted, modified, and extended more easily. Weak coupling lets a designer vary the components of a design without affecting its clients.

Briand and al. proposed a suite of coupling metrics for object-oriented design[11]. They define three different facets of coupling between classes in OO systems: locus, type and relationship. Coupling between classes can be due to any combination of these facets.

Relationship refers to the type of relationship: friendship (C++ specific), inheritance, or other.

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).

Type refers to the type of interactions between classes. The three types of interaction defined are: Class-Attribute, Class-Method, and Method-Method.

Class-Attribute (CA) interaction: there is a class-attribute interaction between classes A and B, if class A has an attribute of type B.

Class-Method (CM) interaction: there is a class-method interaction between classes A and B, if the signature of a method of Class A has a reference of type of Class B.

Method-Method (MM) interaction: Let m_i and m_j belong to Class A and Class B respectively. There is a MM interaction if m_i calls method m_j.

OCAEC

The OCAEC metric is defined as:

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

$$\text{Where: } Others(c) = C - \left(Ancestors(c) \cup Descendents(c) \cup \{c\}\right)$$

This metric counts CA-interactions to class c from classes that are not descendants or ancestors of class c.

OCMEC

The OCMEC metric is defined as:

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

This metric counts CM-interactions from class c to classes that are not descendants of class c.

OCAIC

The OCAIC metric is defined as:

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

This metric counts CA-interactions from class c to classes that are not ancestors of class c.

OCMIC

The OCMIC metric is defined as:

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

This metric counts CM-interactions from class c to classes that are not ancestors of class c.

Chidamber and Kemerer defined the following coupling metrics

CBO

Coupling between object classes

Let $c \in C$

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

The CBO for a class is a count of the non-inheritance related couples with other classes. Two classes are coupled when the methods of one class use methods or instance variables of another class.

RFC

Response For Class is a measure of all the local methods, and all the methods called by local methods. The larger the response set for a class, the more complex the class, and the more difficult to maintain.

2.2.3 Inheritance

Inheritance is the mechanism where a derived class inherits characteristics from its base class. The programmer designs an inheritance hierarchy that defines how classes are derived from other class. If the inheritance is well defined it will allow code reuse and good modularity.

As more and more classes are added to design hierarchies, they become difficult to manage and hierarchies no more have a defined structure. This often leads to the need to reorganize the tree structure.

Inheritance exposes a subclass to details of its parent's implementation, thus "inheritance breaks encapsulation"[8]. The implementation of a subclass becomes bound up with the implementation of its parent class. In this case any change in the parent's implementation will force the subclass to change. This dependency limits flexibility. One cure for this is to inherit only from pure abstract classes or interfaces, which provide no implementation.

DIT and NOC have been taken from the Chidamber and Kemerer suite of OO design metrics.

Depth of inheritance tree (DIT)

The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree. This metric counts the number of ancestors of a class. The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.

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

Number of children (NOC)

The number of children is the number of immediate subclasses of a class in the hierarchy. It is an indicator of the potential influence a class can have on other classes in the design. If a class has a large NOC value, it may require more testing of the methods in that class.

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

The usefulness of the selected OO design metrics, with respect to their ability to predict interface stability is discussed in Chapter 5 where we present the experimental results of our study.

Several measurement tools have been developed to extract quality metrics. In the following section we will expose 4 measurement tools, we will compare their architecture and we will attempt to find the best suitable tool for our assessments.

2.3 Quality Measurement and Evaluation Tools

Several software metric tools have been developed for the estimation of software quality and have been used in real-world software development.

The tools for software measurement that are currently available can be divided into three categories[15]:

1. Tool-integrated facilities: A few software development tools have integrated some facilities to the software measurement. For example the Objectworks/Smalltalk 4.1 compiler has a class "measure", it calculates the number of instance variables, instance methods, class variables. Line of code is also measured by most of the compilers.
2. Tool-extended facilities: The user of the software development tool can extend the measurement facilities of the tool. The programming language Smalltalk/V can be extended to calculate system, class and method measures.
3. Stand-alone measurement tools: These are software tools specifically developed for performing metric-data gathering, analysis, and visualization.

In general, metric tools support the following functionalities:

- Collection of measurement data
- Inspection of source code
- The display of measurement results
- Application of the quality model
- Comparison of measurement results

2.3.1 TAC++

TAC++ (Tool for Analyzing C++ code) is a research prototype suitable for producing assessment of C++ applications[20]. It has been developed at the department of System and Informatics of the University of Florence and is capable of estimating the values of several of direct and indirect metrics for the evaluation of effort, maintenance and reusability costs.

The tool is comprised of three main components: evaluating low-level and high-level metrics, defining and showing metric histograms and profiles, and statistically analyzing system for metric validation.

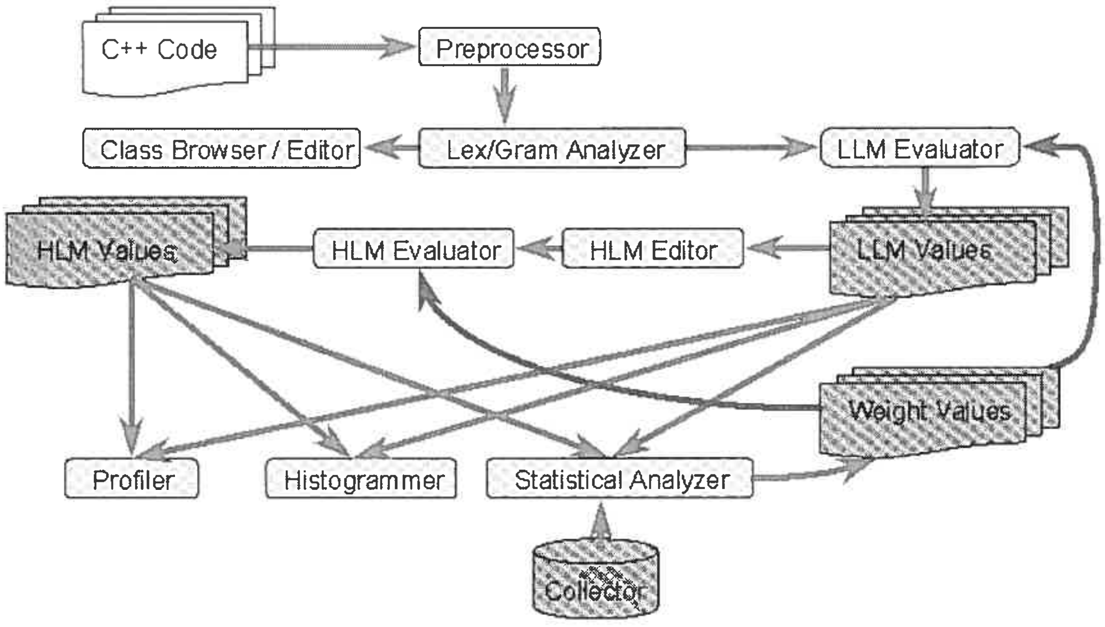


Figure 4: TAC++ Organization

The following is a general description of the TAC++ organization. The C++ code is preprocessed for inclusion of macro definitions, header files, conditional compilation and/or constant declarations. Once the code is preprocessed, it is analyzed by a Lexical Grammatical analyzer that provides LLM (Low Level Metric). These are direct metrics –

such as LOC, Mc, Ha, Number of Method, etc. The LLM Evaluator saves the relationships between classes. The obtained LLM can be managed to define high level, user-defined metrics. The High Level Metric Editor allows the user to define custom metrics according to the specified formula. Histograms and profilers (Kiviat diagram, tables, etc.) can be generated to analyze system, class, or method-level metrics. The metrics can be validated by the means of the Statistical Analyzer that correlates the real data collected with metric values. The Statistical Analyzer is mainly based on multilinear regression techniques.

The metrics extracted by TAC++ are proposed by known literature; some of them are listed in the following table.

Metric	*Comment*	*Reference*
NOC	Number Of Children	S. R. Chidamber and C. F. Kemerer
DIT	Depth Inheritance Tree	
WMC	Weighted Methods for Class	
Size2	Number of Class attributes and methods	W. Li and S. Henry
Ha	Halstead Metric	H. M. Halstead
Mc	McCabe Cyclomatic Complexity	T. J. McCabe

Table 4: TAC++ metrics

2.3.2 QMOOD

QMOOD (Quality Metrics for Object-Oriented Development represented in C++) is an automated tool developed by Jagdish Bansiya; it supports the assessment and analysis of over 30 object-oriented design metrics [5][8].

The tool has a repository in which the metric data of analyzed projects can be stored and retrieved later for comparisons.

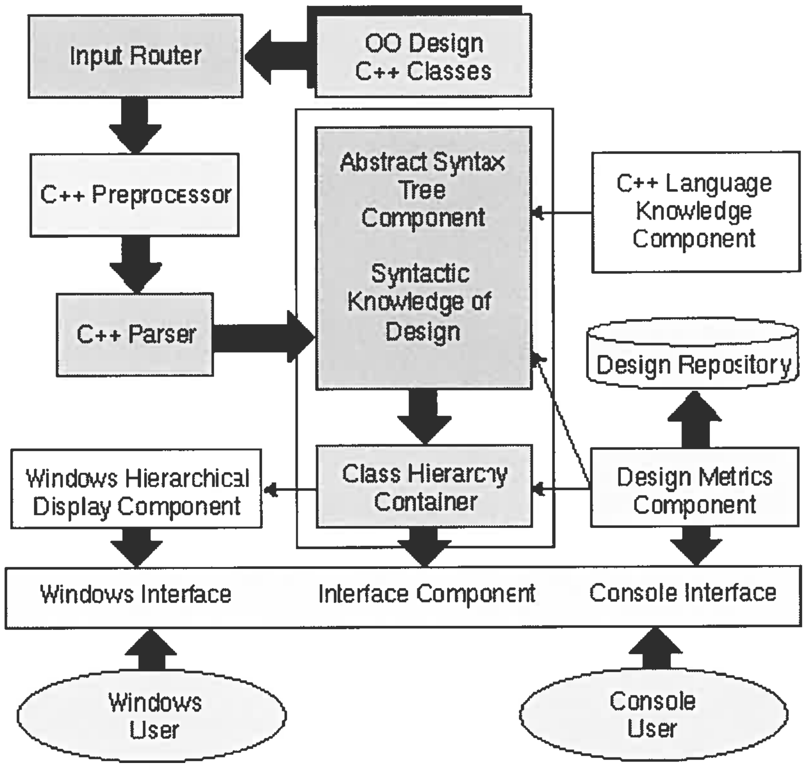


Figure 5: QMOOD Organization

The figure above shows the key components of the tool architecture. QMOOD handles inputs from several sources at a time. After preprocessing the files, a C++ Parser does a syntactic analysis of classes to build an Abstract Syntax Tree (AST). The AST constitutes the knowledge of the input system being analyzed. The C++ language Knowledge Component represents information necessary for interpretation of a group of nodes representing C++ constructs in the AST. This component generates structures that required for the processing of nodes of the AST. The AST is then traversed to collect metric information. Metric measures are provided for class level and system level. The tool supports object-oriented design metrics and they are implemented in the Design Metrics Component. The data for computing these metrics is collected using a set of 15 metric classes in the tool. The tool classifies the metrics based on their scope (Class or System) and information (Simple or Derived).

Information\ Classification	*Class*	*System*
Simple	Method Metric Class	System Metric Class
	Data Metric Class	Relational Metric Class
	Access Metric Class	Object Size Metric Class
Derived	Parameter Metric Class	Virtual Method Metric Class
	Cohesion Metric Class	Ancestor Metric Class
	Client Access Metric Class	Association Metric Class
	Class Complexity Metric Class	Abstraction Metric Class
		Modularity Metric Class

Table 5: QMOOD Classification of Metric Classes Based on Scope and Information

Results of a design assessment can be placed in Design Repository for future access and for comparison with other designs. QMOOD allows for calculated metric values to automatically be compared to other versions or systems that have been previously analyzed.

The QMOOD++ tool is easily adaptable to incorporate new metrics and changes to the relationships between design properties and design metrics and the weights assigned to the relationships. New metrics can be easily added by programming the tool to process the AST representation of the design for the desired information.

2.3.3 MOODKIT

The team of Fernando Brito e Abreu from the Faculty of Sciences and Technology of the Lisbon New University and INESC worked on the development of MOODKIT [2][3]. It includes a set of tools for extracting the MOOD metrics from several Object-Oriented languages; these tools are based on the GOODLY (Generic Object Oriented Design Language? Yes!) Language. GOODLY was conceived with the main purpose of facilitating the extraction of MOOD design metrics. It allows expressing the most relevant design information such as the inheritance relations, associations and message exchanges.

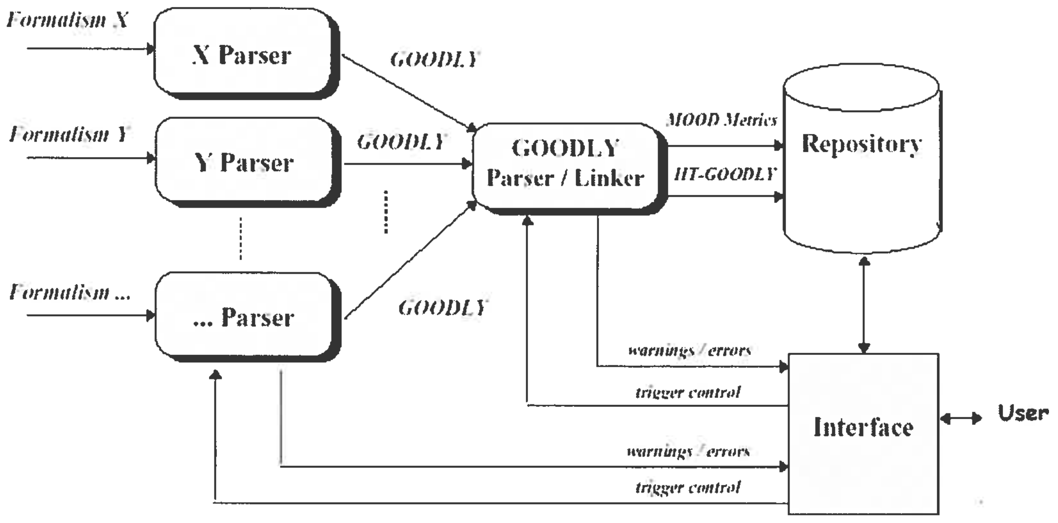


Figure 6: MOODKIT Organization

The first part of the MOODKIT system performs a formalism conversion: converts the code into GOODLY code in ASCII format, and the second part calculates the MOOD (Metrics for Object Oriented Design) metrics. The two parts are independent. This new architecture therefore allows proceeding independently in two directions: On one hand, enlarging MOODKIT applicability by adding new formalism converters. On the other hand, facilitating the expansion of the MOOD set of metrics in an unrestricted way. The MOOD set includes the following metrics:

MOOD Metrics	*Encapsulation*	*Inheritance*	*Polymorphism*	*Message Passing*
Method Hiding Factor (MHF)	X			
Attribute Hiding Factor (AHF)	X			
Method Inheritance Factor (MIF)		X		
Attribute Inheritance Factor (AIF)		X		
Polymorphism Factor (PF)			X	
Coupling Factor (CF)				X

Table 6: MOOD Metrics vs. Quality Attributes

This tool in its current version, allows parsing Eiffel, Smalltalk, C++ and Java code, as well as object models expressed in OMT and UML to produce GOODLY code.

2.3.4 DATRIX

Bell Canada in conjunction with University research has developed a measurement tool Datrix that focuses on a static source code evaluation that is mostly concerned with the maintainability attributes [14]. Datrix covers many of the ISO 9126 Maintainability sub-characteristics: Analyzability, Changeability, Stability, and Testability.

The tool provides source code metrics that describe properties of three types of components: the routines, the classes, and the files.

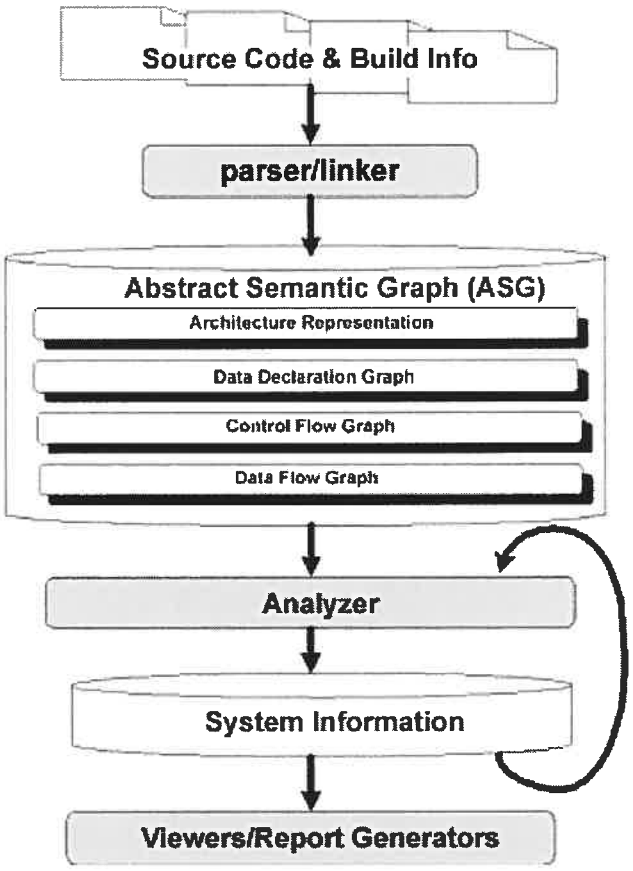


Figure 7: DATRIX Organization

Parsers/linkers are first used to extract information about the source code to analyze. The parsers/linkers generate an Abstract Semantic Graphs (ASG). The ASG is essentially an AST with embedded semantic information. Datrix ASG model pursues two main objectives: 1) Completeness: this means that any kind of Reverse Engineering analyses should be doable on the ASG, without even having to return to the source code. 2) Language Independence: the model should be the same for all common concepts of C++, Java, and other languages. The Analyzers module is a metric tool, it process and refines information provided by the ASG in order to generate specific analysis results like architectural analysis, inheritance graphs, etc. More than fifty metrics are calculated, they characterize files, classes and functions.

The ASG is a semantically enhanced AST. It is composed of nodes and edges. It has also a number of analyzers that compute metrics at routine, class, and file-level. These tools work either on the source code or on the ASG.

Datrix measures 53 characteristics of routines, which cover 5 domains[29][30][35]. The *Programming Rules* is the first domain, mostly evaluating whether the structured programming rule (each block has one point of entry and one point of exit) are applied.

The *Coupling* domain includes metrics that determine to what extent a given routine is coupled to the rest of the system, regarding the number of global identifiers it uses (global variables references, and call to other routines).

Dimension and Complexity forms the third domain. Various measurements are made on the size of the routines, as well as the complexity of the statements and of the control flow graph (CFG). The last 3 metrics in this domain also provide information on the size and complexity of the routine, but in the dimension of the data instead of the control flow. Metrics that influence the number of test cases required for unit testing, as well as the effort required to build each test case are classified in the *Testability* domain.

The fifth and last domain is made up of all metrics related to the internal documentation.

2.3.5 MOOSE

MOOSE is a metric tool developed as part of the FAMOOS project [16]. The main purpose of this project is to help the management and the evolution of object-oriented systems. Moose is a language independent reengineering tool. It consists of a repository to store models of software systems, and provides facilities to analyze, query and navigate them. It is based on the FAMIX metamodel, which provides a language representation of object-oriented sources and contains the required information for system reengineering.

The figure below represents the architecture of MOOSE.

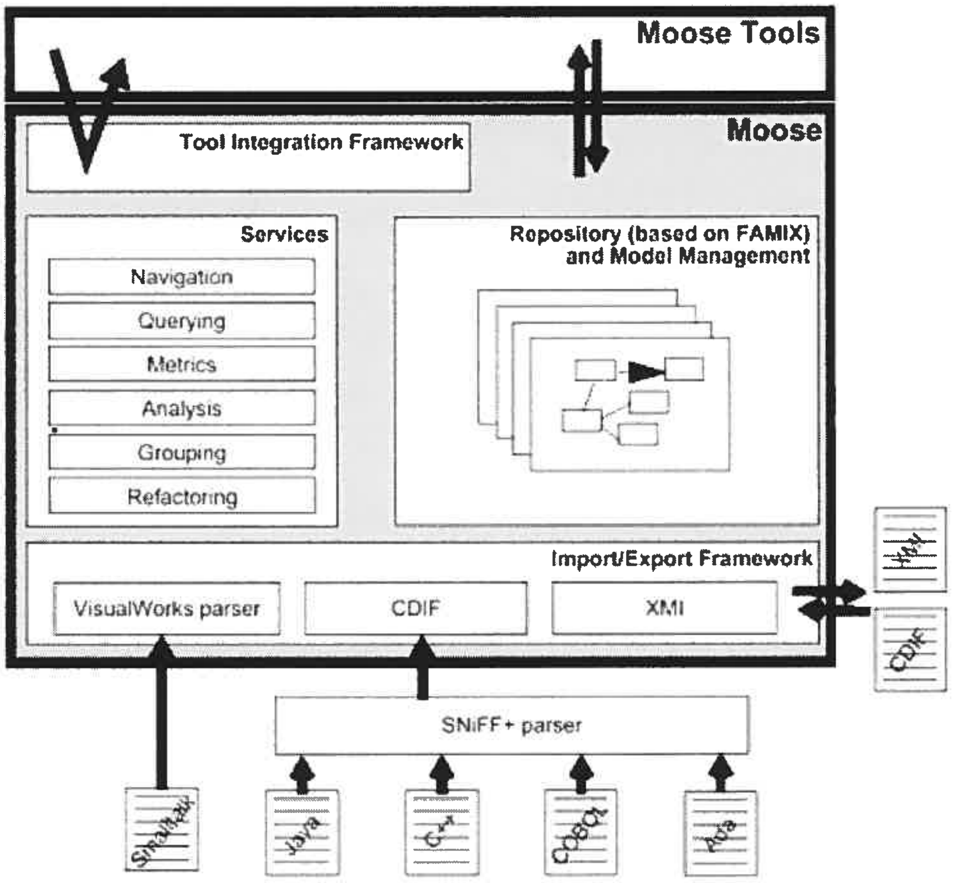


Figure 8: Architecture of MOOSE

Its architecture allows flexibility and extensibility. It supports reengineering of applications developed in different object-oriented languages, as its core model is language independent. Also new entities like measurements can be added to the environment.

The architecture of MOOSE is composed of several modules. The Import/Export Framework provides support to import information into and export information from the MOOSE repository. CDFI and XMI, two standard exchange formats are used as interface to the metamodel parser. The Services component includes metrics and other analysis supports. The metric module supports language-independent design metrics as well as language-specific metrics. The language-independent metrics are computed based on the FAMIX representation. FAMIX does not support metrics that need information about method bodies. The services provided by MOOSE are limited to the class interface level therefore only static information can be obtained. Detailed abstract syntax tree information is not covered. But still in comparison to the previously exposed tools MOOSE has an advantageous architecture because of its flexibility and extensibility. In addition MOOSE has been used for reverse engineering and refactoring purposes.

The table below is a summary of the different tools discussed.

TOOL	Languages supported	Metrics Calculated	Intermediate Representation	Flexibility	Limitations
TAC++	C++ only	Direct Metrics defined by the literature.	None	The tool allows the user to customize derived metrics from the directly calculated metrics generated.	The tool is limited to C++ applications.
QMOOD	C++ only	Object-Oriented Design Metrics proposed by the literature as well as defined by the tool designer.	AST	The tool allows the user to incorporate new metrics and changes to the relationships between design properties and design metrics and the weights assigned to the relationships.	The tool is limited to C++ applications.
MOODKIT	Object-Oriented languages	MOOD Metrics	GOODLY	The tool generates an intermediate language representation, GOODLY, the analysis is done on the representation, and therefore it is language independent.	The tool is limited to MOOD metrics.
DATRIX	C/C++ and Java	Datrix Design Metrics	ASG	The tool generates an intermediate language representation, ASG; the analysis is done on the representation.	The ASG is syntax specific, therefore the results obtained from the analysis of the ASG is language specific.
MOOSE	Object-Oriented languages	Design Metrics	FAMIX	The tool supports reengineering of applications developed in different object-oriented languages, as its core model FAMIX is language independent.	The FAMIX representation is limited to the class interface. Therefore dynamic information cannot be obtained.

2.3.6 Conclusion

This section exposed the architecture of several tools used for metric extraction and quality assessment. In general measurement tools are primarily intended for the analysis and exploration of specific language application, therefore the metrics are tailored to measure properties of software components written in this specific language. However, there are few tools that support several languages.

The Datrix and MOOSE tools showed the most flexibility in terms of their architecture. Unlike TAC++, where the metric extraction module is directly from the source code, Datrix and MOOSE extraction module is independent of the source code. This is due to the use of an intermediate representation of the code: ASG and FAMIX respectively. This approach will allow independent development between the Analyzer and the Parser module. Plus, the extraction of metrics will be on the intermediate representation only. In addition the ASG and FAMIX treat the semantic of the language unlike the AST representation used by QMOOD. This means that any kind of reverse engineering analyses should be doable, without even having to return to the source code. MOODKIT uses the same concept of an intermediate language, the GOODLY language but it was conceived specifically to extract MOOD metrics. Consequently, the use of an intermediate representation is a key factor for the efficiency of the metric tool to support several programming languages.

The measurement tools will assess and predict specific quality characteristics out of the metrics extracted. Statistical analysis is done on the data obtained, and predictive models are generated. In the tools exposed, the analysis and predictive models generated are language-specific, since the intermediate representation like the ASG keeps into account the syntax of the language. Therefore we cannot apply a predictive model specific to C++ on a Java application. The predictive model must be general enough to suit different languages.

In this respect, we decided to go one step further similar to FAMIX by designing a generic model that is programming language independent representation, meaning that it abstracts away from language specific syntax, so analysis can be applied onto different languages. The ASG will be used as the interchange format input. BOAP that consists of a set of tools will generate the generic model. BOAP approach and the generic model will be discussed in detail in the subsequent sections.

Chapter 3

3. BOAP Approach

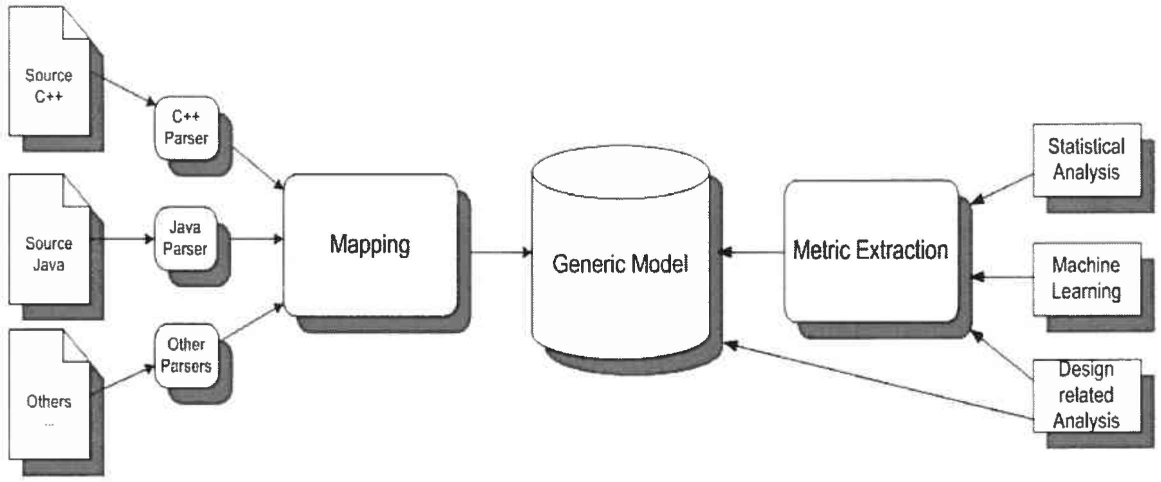


Figure 9: BOAP Architecture

The BOAP architecture has two key components. The first is the mapping module; the second is the metric extraction module. In this architecture, the Generic Model representation plays a central role that is stored in an object-oriented database for future analysis.

One of the overriding principles in this design is the fact that all language-specific inferences take place before getting to this Generic Model [1]. In consequence, this representation can accommodate different programming languages: both object-oriented and non-object-oriented languages.

This architecture allows proceeding independently in two directions. On one hand, enlarging BOAP applicability by enriching the database with new system representations and treatment other programming languages. On the other hand, facilitating the extraction and expansion of the set of metrics used later for design related analysis. The following section gives a general overview of BOAP.

3.1 Parsing Module

First the Datrix parser parses the application. This produces an abstract syntax tree (AST) that is transformed to an abstract semantic graph (ASG).

"An ASG is essentially an AST with embedded semantic information. Specifically, in an AST, a reference to an entity is represented by an edge pointing to a simple leaf node that holds the name of the entity. In an ASG, a reference is represented by an edge pointing to the root of the (shared) subgraph in the ASG that represents the declaration of the entity "

The programming languages supported by Datrix are Java and C++. The output of Datrix parser, ASG, is in plain ASCII text. The ASG defined is made of nodes and edges. These nodes and edges are typed. Edges are not simply pointers from one node to another, but entities with their own properties. The following is an example of a node and an edge Datrix representation in a TA-like format:

```
$INSTANCE 1 cAliasType
1{
beg = 1.13
end = 1.15
name = "foo"
visb = pub
}


(cArcSon 17 19)
{
order = 3
}
```

Each node begins with the letters $INSTANCE. The type of the node is cAliasType and the node id is 1.

We have an edge if the line begins with "(". The type is cArcSon. The id of the startNode edge is 17 and the id of the endNode is 19. The order of an edge is represented with the respect to the startNode of the edge. The information extracted from the node, the edge and their attributes will be the input for the mapping module.

3.2 Mapping Module

The mapping module gathers language-specific semantic and transforms the ASG into a language-independent representation. Language specific inferences have to take place to convert Datrix output into the generic model.

The approach consists of two steps:

1. Reconstruction of the ASG as the input phase for the generation of the representation.
2. Performance of language-specific inferences to generate the Generic Model.

The first step seems redundant, but it is preferable to generate our own representation as we parse in Datrix's input so possible future changes in the Datrix format will not affect the generation phase.

The second step will first generate the skeleton of the program. Then the subsequent step will consist of filling up the rest of the model using language specific inferences.

The module implements the representation in Java. The Java objects representing the generic model are stored in an object-oriented database. The implementation of the database is based on ObjectStore PSE (Personal Storage Editions) PRO that provides persistent storage for Java objects. ObjectStore was chosen because it allows you to store, and query complex data.

This object base database maintains all the entities that are part of the system, all information about the history and status of the project. Analysis and metric extraction will be done directly from the database.

The advantages of our program representation are:

1. It is a language-independent representation. In general parsers are specific to a programming language. Therefore, the representation generated is closely related to the programming language. On the other hand, our model is used to represent various languages.
2. It produces generic code analysis. Analysis modules are related to the architectural representation of the code. The use of the generic model allows us to do the analysis only once.
3. It is adaptable to our needs. The model representations generated by the existing parsers may not be adaptable to our needs. Aspects or information needed may be missing in those models. From our knowledge, there are no open models that can be customized to our needs. By defining a model of representation we define an architecture that is adaptable to our need. Different aspects and information can be represented that are essential to our analysis.

In the following section we will explore the generic model where we will describe the components that defines it.

3.3 Generic Metamodel

There are a number of existing metamodels for representing software. Several of those are aimed at object-oriented analysis and design, Unified Modeling Language (UML)[38] is an example. However, these metamodels represent software at the design level. Reengineering requires information about software at the source code level. The starting point is to have a precise mapping of the software to a model rather than a design model that might have been implemented in lots of different ways.

In the reengineering research community several metamodels have been developed. They are aimed at procedural languages, object-oriented languages or object-oriented/procedural hybrid languages. Most metamodels support multiple languages.

This thesis answers this question by specifying a language-independent metamodel for object-oriented software, the generic metamodel. The figure below shows a part of the metamodel that has been developed until now for the scope of this project.

The metamodel is defined using an object structure written in Java. The object structure consists of approximately 40 classes that model specific components of the abstract representation.

The metamodel is persistent. This allows the generated model to be stored to secondary storage and retrieved again, possibly by a different system. Making the model persistent greatly simplified the design of the supporting tools each tool can retrieve an existing system and extend or modify it and store it again. Having the metamodel persistent also provided a simple solution to the problem of parsing very large class libraries.

A partial object diagram of the metamodel classes is shown in figure 10.

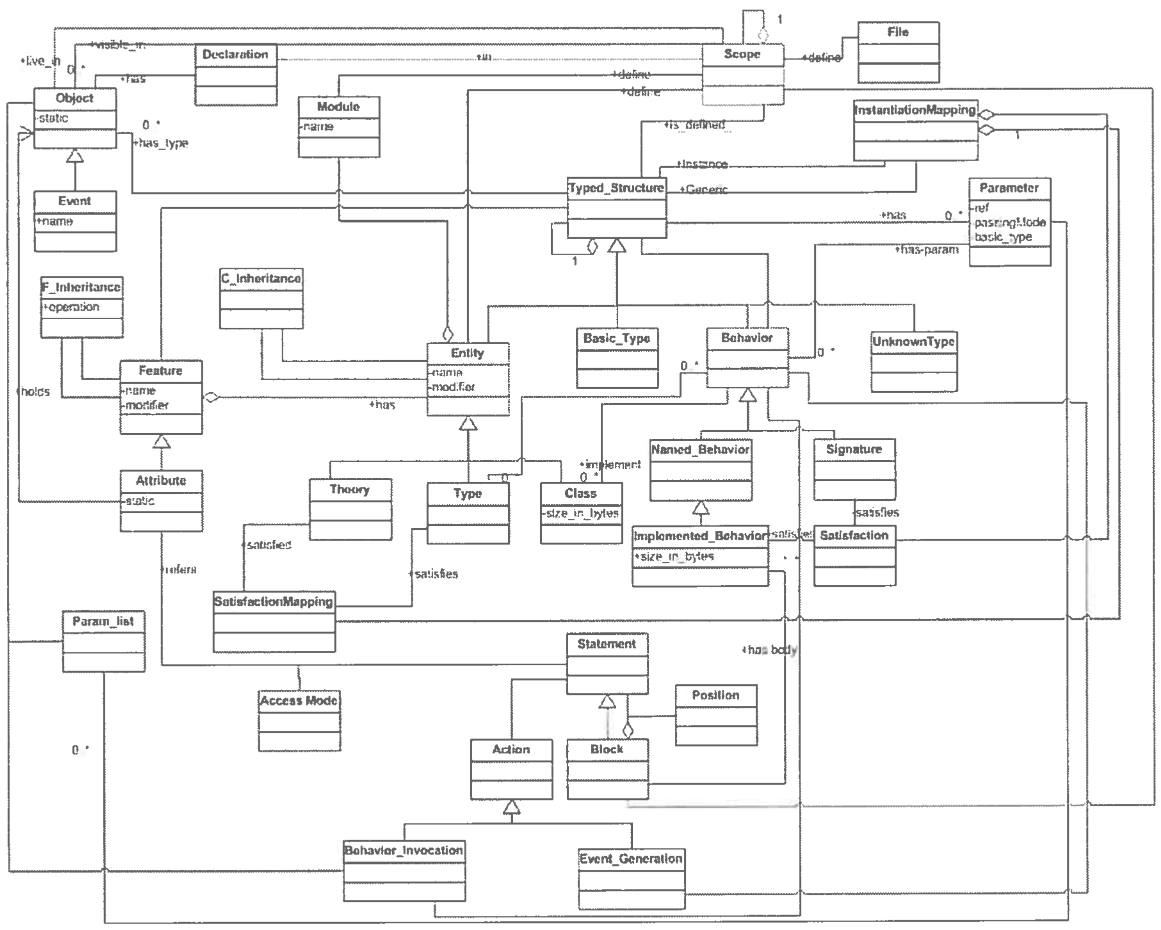


Figure 10: Generic metamodel

One of the main principles in this design is the fact that all language-specific inferences have taken place before getting to this representation.

Both object-oriented and non-object can be represented in this generic model. In addition, design level and implementation level analysis can be done out of the metamodel.

The main aspects treated in this representation are: Scopes, types and compositions, methods and attributes, functions and parameters, and finally instructions. The following section will describe all the different aspects mentioned above.

3.3.1 Scopes

The *scope* of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name (provided it is visible). A declaration is said to be *in scope* at a particular point in a program if and only if the declaration's scope includes that point.

In object-oriented languages there is the concept of visibility and there are 4 levels of visibility: private, protected, public and unknown (for other). The visibility defines the access control and it can be specified in an *Entity* or a *Feature* to control when access to a member is allowed. *Private* limits the access to the *Entity* itself, *Protected* gives access within the *Module* and finally *Public* it is accessible by the entire application.

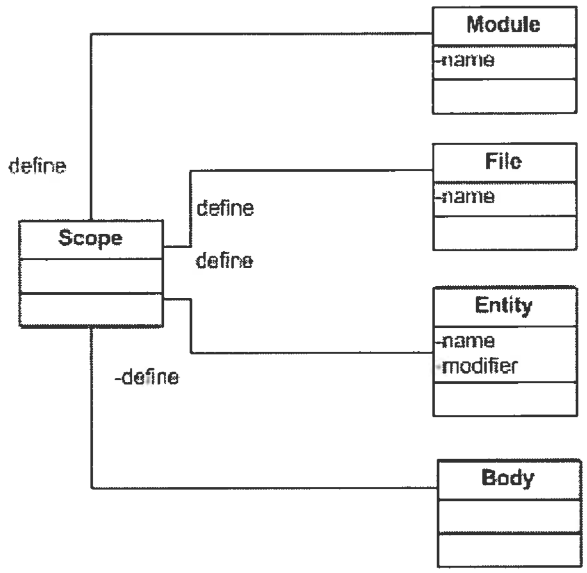


Figure 11: Scope Diagram

Global Scope

Identifiers with global scope are visible to all the application, which include all the modules.

Module Scope

Identifiers with module scope are visible only within the module. A module is defined as a subset of an application. It composed of one or several files. In Java, this scope corresponds to a package.

File Scope

Identifiers with file scope are visible only within the file. A file includes is composed of one or several entities.

Block Scope

The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer. As a general rule, a variable that is declared in a block is only visible within the block.

3.3.2 Types

In general, the types are divided into two categories: primitive types and reference types.

Primitive Types

The primitive types are the Boolean type and the numeric types. The numeric types are the integral types byte, short, int, long, and char, and the floating-point types float and double.

Reference Types

The reference types are defined by the user. They are class types, interface types, structure types, array types, etc.

In our model, the different types have been represented by specific objects. Every element belonging to a component (attribute, method, parameter…) refers to the object representing his type. In this section we will describe the different classes representing the types in the model.

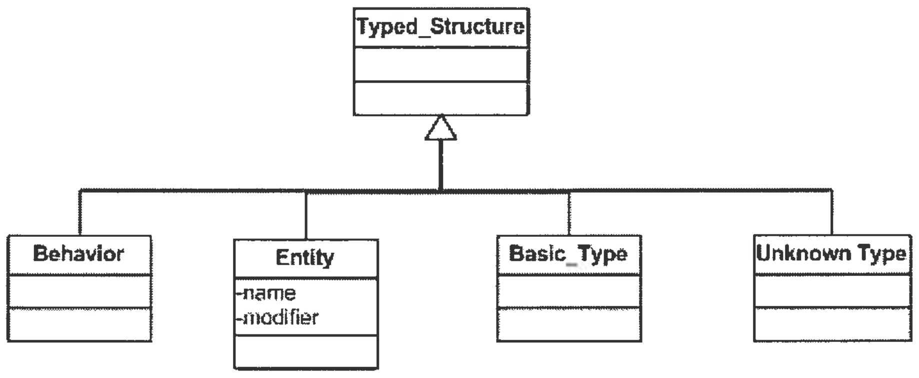


Figure 12: Type Diagram

The class *Typed_Structure* is one of the major roots of this model. This class includes all sort of set of types.

- *Behaviors* consist of method signatures. A signature consists of an identifier, a return type and a list of parameters' type.
- *Basic_Type* represents the primitive types. Until now, the primitive types considered in our model are: Boolean, char, float, double, short, byte, long, int and void.
- *Entity* includes *class*, *type*, *interface* and *theory*. An *Entity* declaration has a *name* and may include *modifier*: public, protected, private, package, static. Entity can have one or more features. *C_Inheritance* represents inheritance relationships between entities. Thus, there are two associations between *C_Inheritance* and Entity that correspond to the two roles of an entity within an inheritance relationship.
- *Unknow_Type* represents unresolved types or types that are not defined in the application. Types that are not defined in the application are imported types from external libraries.

3.3.3 Methods and Attributes

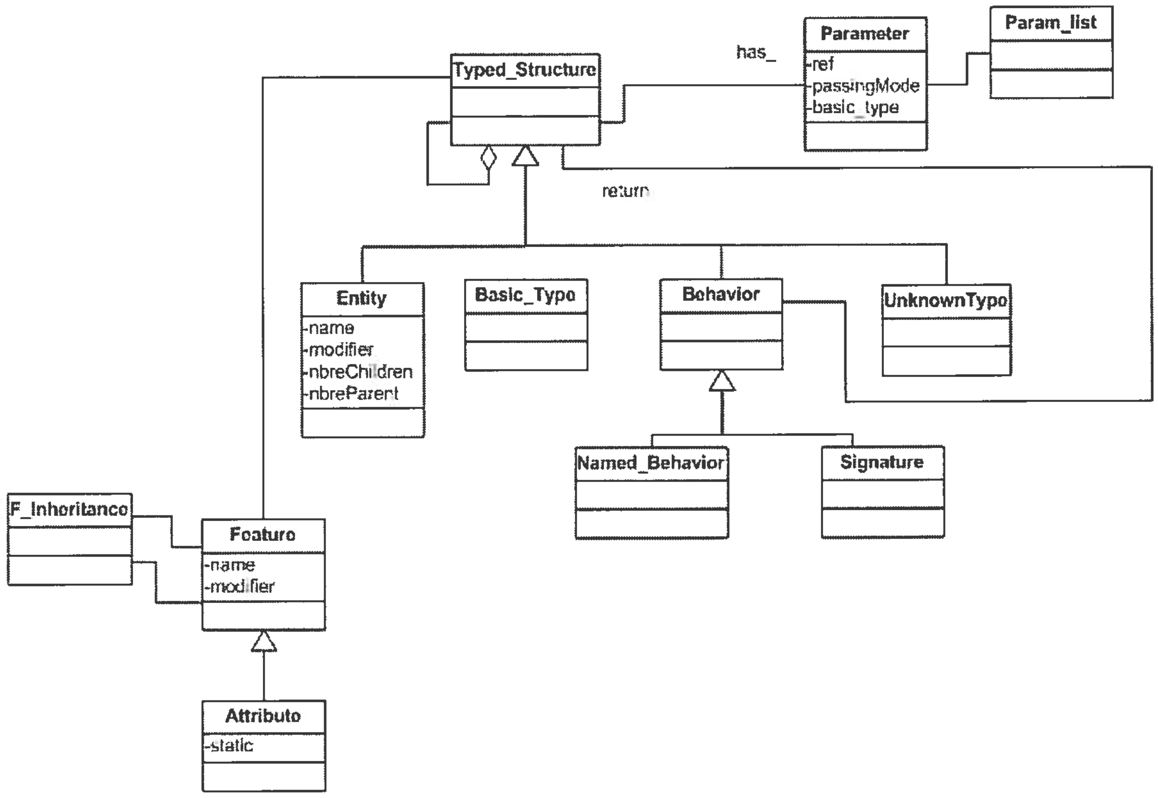


Figure 13: Feature Diagram

Methods and attributes are defined respectively in the model by the class *Feature* and *Attribute*. *Feature* has a *name* (attribute name or function name) and modifiers, which correspond to visibility categories (private, protected, public, package, etc). *F_Inheritance* represents inheritance relationships between *Features*. *F_Inheritance* allows treating inheritance during the mapping by taking into account the semantic of each programming language. Once the mapping is done, with *F_Inheritance* we can have abstraction of how the features are inherited.

As discussed above, each *Entity* can have one or several *Feature*. In addition, the Class *Feature* has a link with the class *Typed_Structure* representing the return type of the method and a link with *Parameter* representing the parameters of the method and a link to *Block* representing the body of the method. The class *Attribute* is a sub-class of *Feature*.

3.3.4 Functions and parameters

In hybrid languages such as C++, functions can be defined and they are presented in the model by the class *Named_Behavior*. This class has a link to the class *Typed_Structure* to represent the returning type of the function, a link to class *Parameters* representing the parameters of the function, a link to class *Block* representing the body of the function, a link to class *File* in which this function is defined and a link to class *Scope* defining its visibility within the system.

3.4 Summary of the Metamodels

BOAP's metamodel has advantages over the already exiting ones. Not only it is language independent but also it has the capability to represent an application at its design level as well as the implementation level. The following table gives a general comparison of the different metamodels exposed.

Metamodel	Language independent	Metrics Supported
AST-QMOOD	No	Design Metrics
ASG-DATRIX	Partially: language independent core with multiple language extensions.	Design Metrics
GOODLY-MOODKIT	Partially: language independent core with multiple language extensions.	Design Metrics
FAMIX-MOOSE	Partially: language independent core with multiple language extensions.	Design Metrics
GENERIC-BOAP	Yes	Design Metrics and Metrics of Implementation

Table 7: Summary of Metamodels

As well as storing the architectural information of system components, the meta-model classes can be queried to generate system information. The information gained from these queries can be used to generate further information such as metrics. The next step of this project is to generate a module that extracts metrics from the BOAP's generic metamodel. Next section outlines the application of this information for metrics collection.

3.5 Metric Extraction Module

This module will traverse the representation stored in the database starting from the system level to the module level to the entity level and finally to the block level. The mapping is not done to represent inside the block. For the purpose of our study, this information will not be needed. Detailed information that is commonly available within the definition of a block, for example method invocations were not taken into account. Since this information is not available at the design level, we excluded metrics related to cohesion and complexity. It is at the design level that the interface of classes and the class dependencies are defined. We can extract information about the inheritance relationships among classes. Within the class we can extract the attributes, the methods and their parameters, their type and their visibility. The metrics that can be collected during design stage of a project like inheritance, coupling and inheritance metrics were calculated. We defined and extracted relevant metrics to our study at the system and class level and they are listed below:

System level metric

Metric	Description	Summary
TBC	Total base classes of system	This metric is a count of the number of classes that don't have parent classes.
CLS	Number of classes in a system	This metric is a count of the total number of classes in the system.
NLC	Number of Leaf Classes	This metric counts the number of leaf classes in the hierarchies of the design.
ADI	Average Depth of Inheritance	This metric is the average depth of inheritance of classes in the design. It is computed by dividing the summation of maximum path lengths to all classes by the number of classes. The path length to a class is the number of edges from the root to the class in an inheritance tree representation.
AWI	Average Width of Inheritance	This metric is the average number of children per class in the design. The metric is computed by dividing the summation of the number of children over all classes by the number of classes in the design.

Class Level metric

Metric	Description	Summary
DIT	Depth of Inheritance	This metric is the length of the inheritance path from the root to a class.
NOC	Number of Children	This metric is a count of the number of immediate children (sub classes) of a class.
NPA	Number of Public Attributes	This metric counts the number of attributes that are declared as public in a class.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class.
NOM	Number of Methods	This metric is a count of all the methods defined in a class.
NOD	Number of Attributes	This metric is a count of the number of attributes in a class.
CSM	Class Size Metric	This metric is an ordinal number that is the sum of the number of methods and attributes in a class.
OAM	Operation Access Metric	This metric is the ratio of the number of public methods to the total number of methods declared in the class. A high value for OAM is desired.
DAM	Data Access Metric	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired.
DCC	Direct Class Coupling	This metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
DAC	Direct Attribute Based Coupling	This metric is a direct count of the number of different class types that are declared as attribute references inside a class.
DPC	Direct Parameter Based Coupling	This metric is a count of the number of class object types that are required directly for message passing (parameters) to methods in a class.
OCMIC	Other Class-Method Import Coupling	O: Relation type other then *Hereditary* or *Friend* type CM: Class-Method Interaction IC: The count of method having as a return type or parameter type a class type.
OCMEC	Other Class-Method Export Coupling	O: Relation type other then *Hereditary* or *Friend* type CM: Class-Method Interaction EC: The count of parameter of a method's class or method's retuning type with a class C type.
OCAIC	Other Class-Attribute Import Coupling	O: Relation type other then *Hereditary* or *Friend* type CA: Class-Attribute Interaction IC: The count of Attribute of type class object
OCAEC	Others Class-Attribute Export Coupling	O: Relation type other then *Hereditary* or *Friend* type CA: Class-Attribute Interaction EC: The count of Attribute having a class C type.

3.5.1 Metric Calculation

As stated before, the metrics are calculated using the representation according to our metamodel. In the remaining of this section we will show how the system and class metrics can be derived from the representation.

A system is defined as a collection of OO classes. Let us assume a function called *Classes* which when applied to a system *S*, gives the distinct classes of *S* represented by the set of instances of the class "*Class*", refer to Figure 10: Generic metamodel.

CLS: Number of classes in a system

CLS is defined as the cardinality of the set Classes defined as follow:

$$CLS = |Classes(S)|$$

TBC: Total base classes of system

Let *Base* be the set of base classes of a system such that

$Base(S) = \{c_i \mid c_i \in Class \wedge \sup(c_i) = \varnothing\}$ where *sup(c$_i$)* is the set of parent classes of c$_i$ that can be obtained using the relation with the class "C_Inheritance".

Therefore, TBC can be defined as the cardinality of the set Base defined as follow:

$$TBC = |Base(S)|$$

NLC: Number of Leaf Classes

Let *Leaf* be the set of classes without children in a system such that

$Leaf(S) = \{c_i \mid c_i \in Class \wedge sub(c_i) = \varnothing\}$ where *sub(c$_i$)* is the set of children classes of c$_i$ that can be obtained using the relation with the class "C_Inheritance".

$$NLC = |Leaf(S)|$$

NOC: Number of Children

Let *Children* be the set of subclasses of a class c_i, where c_j is a subclass of c_i.

c_i and c_j are instances of "Class" and their relation is obtained by the class "C_Inheritance": $Children(c) = \{c_j \mid c_j \in c_i.sub\}$

Then we can define NOC of a class as:

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

In the system level we can also define the average width, which is the NOC of each class of the system over the total number of class.

AWI: Average Width of Inheritance

$$AWI = \frac{\sum_{i=0}^{n} NOC(c_i)}{CLS}$$

where n is the total number of classes in a system.

DIT: Depth of Inheritance

The Depth of Inheritance of a class c_i can be defined as follow

$$DIT(c_i) = \begin{cases} 0, if \sup(c) = \varnothing \\ 1 + \max\{DIT(c') \mid c' \in sup(c)\}, else. \end{cases}$$

In the system level we can define the average depth of inheritance as follow:

ADI: Average Depth of Inheritance

$$ADI = \frac{\sum_{i=1}^{n} DIT(c_i)}{CLS}$$

where *n* is the total number of classes in a system.

Now that we've exposed the inheritance related metric in terms of the generic representation, the rest of the metrics are related to coupling and encapsulation.

CSM: Class Size Metric

Method and attribute of a class are considered as "Features". Let Feature be the set of method and attributes of a class c then

$$CSM = |\,Feature(c)\,|$$

DAM: Data Access Metric

DAM measures the ratio of non-public attributes to the total number of attribute in a class.

$$DAM = \frac{NOD - NPA}{NOD}$$

This metric can be equated by getting the total number of instances of the class "Attribute" from which we identify the attribute "modifier" that are equal to public.

Let *pubAttribute* be the public attribute of a class, therefore there is an instance of "Attribute" that has its attribute "modifier" = public.

$$\forall x pubAttribute(x) = \{x \mid x \in Attribute \wedge x \in Attribute.modifier = public\}$$

Therefore for a class c:

$$NOD = |Attribute(c)|$$

$$NPA = |pubAttribute(c)|$$

OAM: Operation Access Metric

$$OAM = \frac{CIS}{NOM}$$
$$where$$
$$NOM = |Methods(c)|$$

CIS is the number of public method in a class. Let m_i be a method of a class. In terms of our representation it is an instance of the class "Feature" but not an instance of the subclass "Attribute". Feature has an attribute "modifier" for assigning the visibility of the method. Therefore the total of public method is obtained by summing the instances of the class "Feature" excluding "Attribute" in a class having the attribute "modifier" set to "public".

Let *PubMethods* be the set of methods in the system.

$$\forall x PubMethod(x) = \{x \mid x \in Feature \wedge x \notin Attribute \wedge modifier = public\}$$

Therefore *CIS* is the cardinality of public Methods in a class

$$CIS = |PubMethod(c)|$$

DAC: Direct Attribute Based Coupling

Let *A* be the set of attributes in a class that are of a class type.

Let *C* be the set of classes in the system.

$$\forall x DAC(x) = \{x \mid x \in A_i \ni (x.has\_type \in C)\}$$

DPC: Direct Parameter Based Coupling

We should note that *Behavior* is used to represent both functions and methods associated with classes.

NamedBehavior is a method signature, with parameter types, a return type and a *name*.

A *Behavior* has a set of parameters through the association *has-parameter*.

A Parameter is associated with *TypedStructure*, which can be an *entity*.

Set $Parameter_i$ *of* $method_j = NamedBehavior_j.has\_parameter$

Then

Let P be the set of parameter of a $method_j$

$P = \{Parameter_1, Parameter_2, ...Parameter_n\}$where n is the total number of method's parameter.

$$\forall x DPC(x) = \{x \mid x \in P \ni (x.has\_type \in C)\}$$

DCC: Direct Class Coupling

$$DCC(c) = |DPC(c)| + |DAC(c)|$$

OCAEC: Other Class-Attribute Export Coupling

Class-Attribute (CA) interaction: there is a class-attribute interaction between classes A and B, if class A has an attribute of type B.

Let D be the set of attributes of class A.

$$\forall x OCAEC(x) = \{x \mid x \in D \wedge x.has\_type \in B\}$$

OCMEC: Others Class-Method Export Coupling

CM-interactions: there is a class-method interaction between classes A and B, if the signature of a method of Class A has a reference of type of Class B.

Let M be the set of behaviors of class A.

$$\forall x OCMEC(x) = \{x \mid x \in M \wedge \{x.return\_type \in B \vee x.has\_param.has\_type \in B\}\}$$

The metrics chosen measure characteristics related to coupling, inheritance and encapsulation. The combination of the resulting metrics will be used to estimate the quality design, in our case software stability.

This chapter gave an overview of the BOAP architecture. A fundamental role is played in this architecture by a language-independent representation: the generic metamodel. It provides great flexibility by allowing the mapping and the analysis to be independent from each other. In addition we discussed the metric module that was developed to extract design metrics from the metamodel. The next chapter will describe the mapping of Java applications.

Chapter 4

4. The Extraction of Java Programs into the Generic Representation

4.1 General approach

Language specific inferences have to take place to convert Datrix output, the ASG, into the generic model.

The approach consists of two steps:

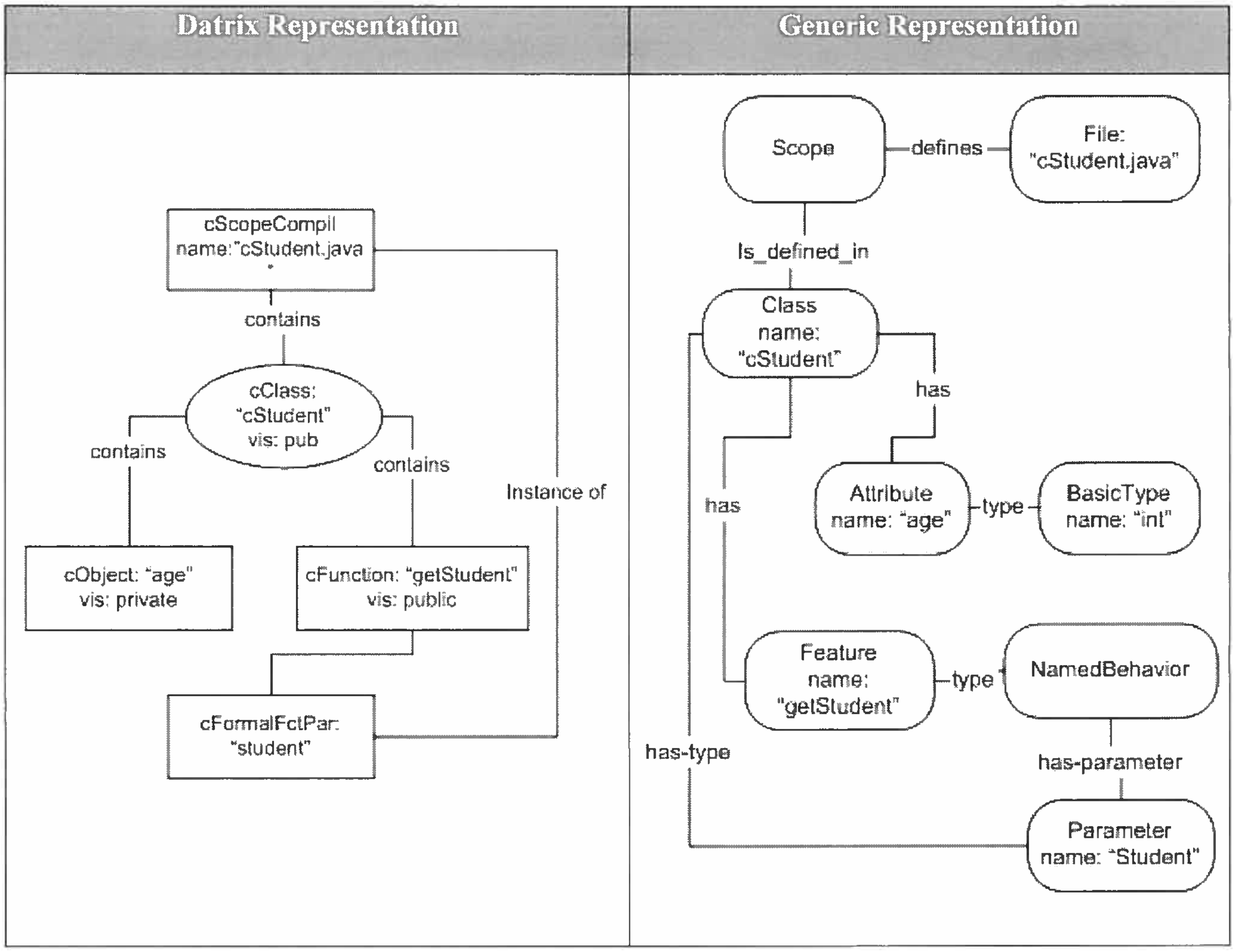
1. Reconstruction of the ASG as the input phase for the generation of the representation.
2. Performance of language-specific (Java) inferences to generate the Generic Model.

The first step seems redundant, but it is preferable to generate our own representation as we parse in Datrix's input so possible future changes in the Datrix format will not affect the generation phase.

The second step will first generate the skeleton of the program. Then the subsequent step will consist of filling up the rest of the model using language specific inferences. Consider the following source code:

```
public class cStudent{
private int age
public void getStudent(cStudent  student);}
```

The figure below illustrates the two steps mentioned above. The left hand-side graph shows a simplified Datrix model while the right hand side it is our defined representation.



For this particular example, it is worth noting that the generic model can be generated from a single traversal of the Datrix ASG. However, there are cases when the generic representation cannot be generated in a single traversal and that includes language-specific inferences, for example in case of inheritance.

```
public class cPerson{
public String name;}
```

```
public class cStudent extends cPerson{
private int age;
public void getStudent(cStudent  student);}
```

Datrix's ASG would represent the attribute "name" in the above example only once, as part of the definition of the class cPerson. The class cStudent has access to the attribute "name" through the inheritance relation: cStudent extends cPerson. In our model we treat the language specific inferences, therefore we repeat the attribute "name" in the subclass cStudent. For this purpose we need an operation to copy it down which is done in the mapping module.

4.1.1 Reconstructing Datrix's ASG

The output file of Datrix contains nodes and edges that define the graph. It is in a text file format that makes it convenient for data processing. The nodes are identified by an id together with the name of the particular type of node. The general strategy is to make a class for each type of node. So when we distinguish a node type, we make an object of the corresponding type, read the attribute values (order, name, type, and number of attributes depending on the node type) and use the values to set the corresponding fields in the newly created object. The name of the node type in the Datrix output file becomes the name of the class. For instance the Datrix's node type cScopeGlb becomes the class cScopeGlb.java. Similarly, we make an object of the correspondent type for each type of edge we find. We set the startNode and endNode with the corresponding id of the start and end node. We store each node in a hashmap, and we use the id of the object as the index to the hashmap. For the graph edge, we get its StartNode, and add the edge to the list of children for the node, thus constructing the abstract semantic graph in memory. Once we have the abstract semantic graph, we can perform a language-specific traversal in order to generate our representation.

4.1.2 Mapping of Java code

The main objects created to represent Java code are:

- The object dbApplication refers to all the entities defined in the application. The object dbScopeGlb refers to all the entities where their scope is the application. The two objects refer to each other.
- The object dbModule refers to all the entities defined in the package. The object dbScopeModule refers only to entities where their scope is limited to the package level. The two objects refer to each other.
- The object dbFile refers to all the entities defined in the file. The object dbScopeFile refers only to entities where their scope is limited to the file level. The two objects refer to each other.
- The object dbBasicType refers to primitive type. The primitive types have a global scope; therefore the object dbScopeGlb refers to dbBasicType objects.
- The object dbClass refers to the class defined in the application. The identifier of the dbClass object is its fully qualified name for example packageName.ObjectName. The object dbClass refers to the dbFile object corresponding to the file where the class is defined. It also refers to the object Scope to represent the scope of the class.
- The object dbFeature refers to the method defined in the application. This object dbFeature is referred by the object dbClass or dbInterface where it is defined.
- The object dbAttribute refers to the attribute defined in the application. It is a subclass of dbFeature.

We used the tool Objectstore to have persistent objects. As a consequence, the representation generated will be saved in an OO database.

The Datrix parser is limited to treat one file at a time. On the other hand, our mapping module is developed in such a way that we are able to map several Datrix graphs at once. This way, we can represent the entire application in one database. This is made possible for the following reasons:

- We removed the redundancy in the Datrix graph. We created once the object representing the same entity for dbApplication and dbScopeGlb objects. These objects represent the application. The objects representing primitive types were also created once, as well as for the objects dbModule and dbScopeModule representing the packages.
- We treated the naming resolution.

Given the following example:

```
public class A {
   public B ref;
}
public class B {
   ...
}
```

Class A and *class B* are declared in two distinct files. In *class A*, the attribute ref is of type *class B*. The Datrix parser will treat each class individually and will generate two separate files. During the mapping, the dbAttribute object that represents *ref* will refer to the dbClass object representing *class B*. This reference is made possible only after all the classes of the application have been treated.

The dbClass objects are identified by their fully qualified name. Usually in a declaration we use the simple name of the class. In an application we can have the same simple name for two different classes, but we can only have one fully qualified name.

In Java, the naming resolution consists in getting the qualified name from the simple name of the class. We will expose the rules of Java for naming resolution. Those rules were also applied in our mapping module.

If name of the class is a simple name, then there are three possible scenarios:

1. The class is defined in the same file or it is referenced. In the case where it is referenced then, its qualified name is composed of the package name that the class belongs to followed by its simple name. For example:

```
import boap.mapping.cTest;
....
public  cTest ref;
```

The qualified name of the class cTest will be boap.mapping.cTest.

2. The class is a member of the same package as the file from which it is referenced.

3. The class is imported from another package. For example:

```
import boap.mapping.*;
import boap.metric.*;
...
public cTest ref;
```

We look for the class cTest in the imported packages by the file. If the class is found in one of the packages, the fully qualified name will constitute with the package name and the class simple name: package_name.class_simple_name, for example: boap.mapping.cTest.

Below we will illustrate by an example the mapping of a small program in Java to the generic model. This example illustrates some of the Java specific notions like Package, Interface, and Inheritance and shows how they are represented by the generic metamodel.

Student.java

```
package University;
import java.lang.*;

public class Student extends Person implements Department
{
  private String Name;
  public String getName();
  public void setName(String aName) {};
}
```

The system level illustrates the external property of the class in the application.

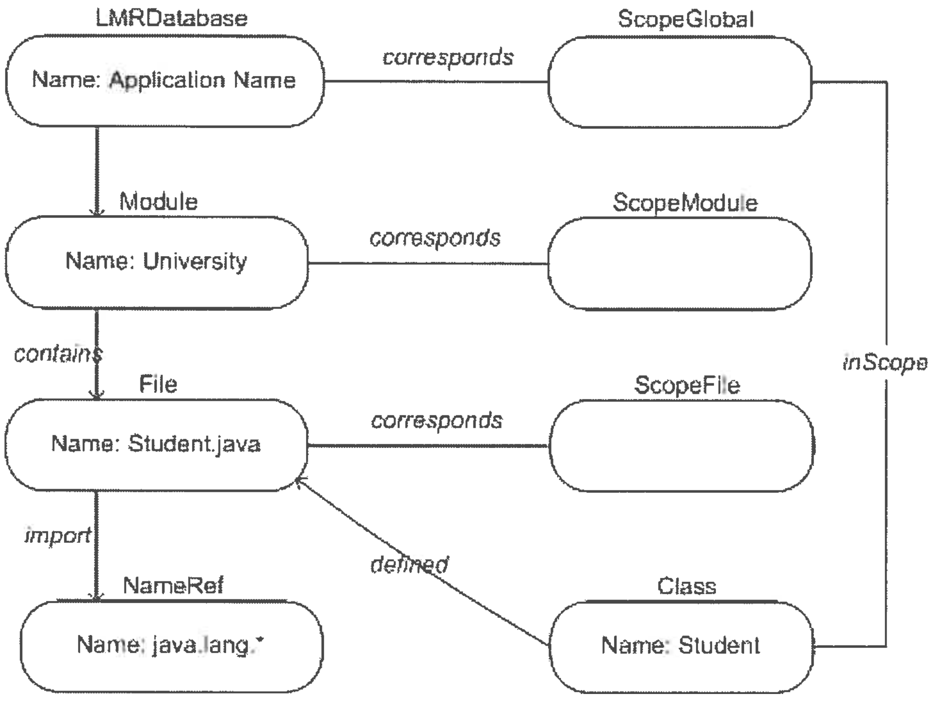


Figure 14: Mapping example at the system level

The class level illustrates the internal structure of the class.

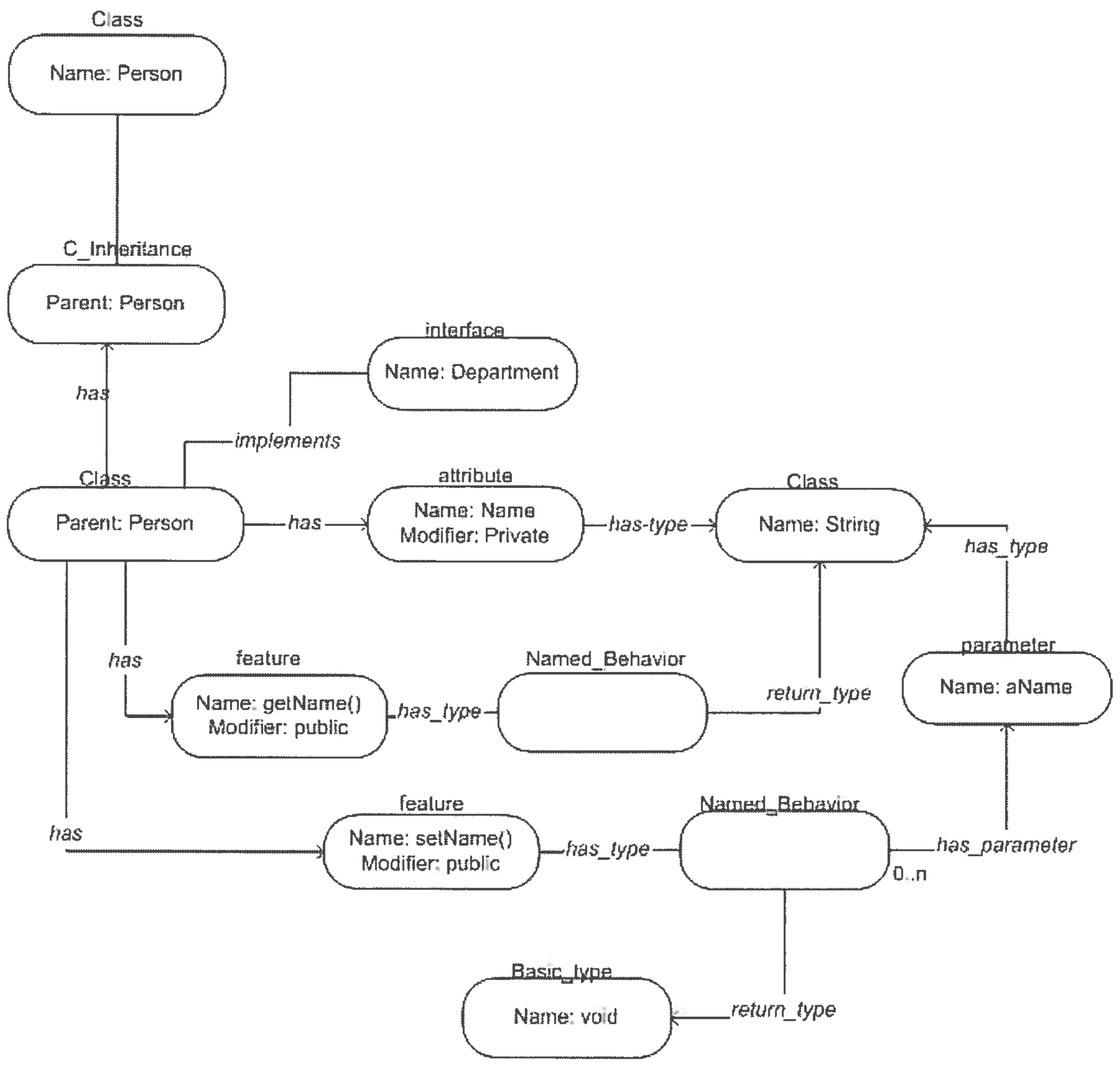


Figure 15: Mapping example at the class level

4.1.3 Conclusion

This chapter gave a general overview of the mapping of the Java code into the generic representation. Not all the aspects related to the mapping of Java were covered. We covered only the aspects that were developed for this project. Java is a language that is in constant evolution, where new features are added or replaced with new Java releases. As a consequence, the mapping of Java is a work in progress.

Now that we've defined our generic model, the next step is to use it to generate predictive model to predict interface stability of classes in OO systems. The next chapter will expose a case study on the stability of Java applications.

Chapter 5

5. Evaluation

5.1 Experimental Framework

5.1.1 Hypothesis

Hypotheses have been formulated on the stability of evolving frameworks [6][34] and on interface stability of evolving class libraries [42].

In order to define the hypothesis we need to define first what makes a system stable. In Chapter 2 we exposed some of the research done on software stability. The main factor that causes the system to be stable is mainly the lack of interdependencies [32]. Yet, the system does have some dependencies that do not interfere with stability because they are unlikely to change. We defined the hypothesis as follow:

Hypothesis 1: *The degree of interdependence between the class and the environment within which it is defined has an effect on its stability.*

In order to measure the interdependencies we selected metrics related to coupling inheritance, and encapsulation as indicator of stability.

5.1.1.1 Class Interdependencies vs. Stability

The first measure of interdependence is coupling. It refers to the degree of interdependence between parts of the design[13]. High quality software design should obey the principal of low coupling. The stronger the coupling between modules, i.e., the more inter-related they are, the more difficult there modules are to change and correct, and thus changing a module can cause a negative ripple effect, affecting the stability of the system.

In order to improve modularity, inter-object class couples should be kept to a minimum [13]. 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 instable. The higher the export coupling of class ***c***, the greater the impact of a change to ***c*** will be on other classes [11], and thus this class will affect the stability of the system. We propose the following hypothesis:

Another measure of class interdependency is encapsulation. It is defined as the enclosing of data and behavior within a single construct. In object-oriented designs the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects. By hiding the internal representation (encapsulation), classes have the flexibility to change their internal representation without cascading the changes to other classes. This significantly reduces the dependency between classes, thus promoting stability.

Finally, system component can be interdependent through inheritance. Inheritance is defined as a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (*single inheritance* and multiple *inheritance respectively*). As we evolve the inheritance hierarchy, the structure and behavior that are the same for different classes will tend to migrate to common superclasses. Superclasses are generalized abstraction, and subclasses represent specializations in which fields and methods from superclass are added, modified, or hidden.

The lower a class is in the hierarchy, the more it will inherit from superclasses and therefore has more chances to be instable. On the other hand, the more children a class has, the more classes it may potentially affect because of inheritance. Any change to the accessible class interface can affect its subclasses. Thus making the class instable. We propose the following hypothesis.

Consequently, through the following class interdependencies: Inheritance, Coupling and Encapsulation we should be able to measure and predict the level of stability. But first we need to identify the level of class stability affected by change.

5.1.2 Identifying changes in application interface

Changes can concern data, a method, a class or a class library. Two categories of changes at the class level were identified by Sahraoui H. et al.[42]. Let C_i be the interface of a class C in version i of the application and C_{i+1} be the interface of C in the version $i+1$. The two categories of changes for C are:

A. The interface Ci is no longer valid in version $i+1$. This happens in four cases:

1. C is removed
2. $C_{i+1} = C_i$ – some public members
3. $C_{i+1} = C_i$ – some protected members
4. $C_{i+1} = C_i$ – some private members

B. The interface C_i is still valid in version $i+1$. This happens in two cases

5. $C_{i+1} = Ci$
6. $C_i \subset C_{i+1}$

Types of changes (1 to 6) are ranged from worst to best according to the degree of impact of each type.

- The deletion of a class has a more serious impact than the deletion of a subset of its protected method.
- The deletion of a public method has a more serious impact than the deletion of protected method.
- The deletion of a subset of a private method has no serious impact on the system since its visibility is limited to the file itself.

Types are exclusive: The change of class is classified into type *k* only if it cannot be classified into the *k-1* previous types.

For class C some public methods are deleted and some other public methods are added, C belongs to type 2 and not type 6.

If a class is renamed then this is considered as a deletion of class (type 1) and the creation of a new class.

In the same way, a change in a method signature is considered as a method deletion. A scope change that narrows the visibility of a method (from public to protected or private and from protected to private) is considered also a method deletion.

A stability module was developed that will classify a class as instable (Type1) or stable (Type2) following the above criteria.

Type 1: the public/protected interface of a class has changed or has been deleted

Type 2: the public/protected class interface didn't change.

Let F_i be the set of public/protected features in a system of version i

Let F_{i+1} be the set of public/protected features in a system of version i+1

$$\forall x Type1(x) = \{x \mid x \in F_i \wedge x \notin F_{i+1}\}$$

$$\forall x Type2(x) = \{x \mid x \in F_i \wedge x \in F_{i+1}\}$$

Now that we defined the two levels of stability, in the next section we present the metrics related to class interdependency.

5.1.3 Defining the Metrics

We have discussed the design metrics whose extraction is supported by the metrics module. Initially, several design metrics have been extracted. Some of them were redundant or their values didn't show significant relation with the change of class interface. Out of the metrics extracted, the following metrics: DAM, OCAEC, OCMEC, DIT, NOC have been chosen to evaluate interface stability in evolving applications. They evaluate the following quality characteristic: Encapsulation, Coupling and Inheritance, which are criteria of software stability.

Symbol	Definition	OO Properties	Comment of implementation
OCAEC	Others class-attribute export coupling	Export Coupling	This metric is a count of the number of attributes class types in a class.
OCMEC	Others class-method export coupling	Export Coupling	This metric is a count of the number of different parameter types used in the methods of a class and the number of return types of the methods of a class.
OAM	Operation Access Metric	Coupling	This metric is the ratio of the number of public methods to the total number of methods declared in the class. A high value for OAM is desired.
DAM	Data Access Metric	Encapsulation	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.
DIT	Depth of Inheritance	Inheritance	This metric is the length of the inheritance path from the root to a class.
NOC	Number of Children	Inheritance	This metric is a count of the number of immediate children (sub classes) of a class.

Table 8: Metrics implemented

Now that the framework for our experiment is defined, we need to build our database from which we will extract metrics and define the stability. For this, we would need to collect data from several systems. Next section lists the different systems used for our experimental data collection.

5.2 Data Collection

In this case study we want to demonstrate the relation between the internal factors and the interface stability in evolving applications.

For our study, we chose open-source applications available in the web. Many open-source projects adopt CVS as version control platform; therefore we can have access to any version of source code programs. We will be able to study the changes done on the software throughout its evolution. We chose the following systems: Jigsaw, JEdit, Jext, and GNU.Free. The applications were compiled and the classes generated parsed with the Datrix parser, and then mapped into the generic representation. The different versions of an application were gathered in a common ObjectStore database, and thus we generated 4 different database for each application. Gathering the versions in one database facilitate the comparison and the evaluation of the stability of the system components. From the generic representation of each application, we extracted the metric chosen to evaluate encapsulation, inheritance and coupling. We also evaluated the stability of classes for each system by comparing their components between their consecutive versions. As defined in the previous section, classes of version i+1 are classified as stable if its public interface (method and attribute) is the same as version i, otherwise the class is considered instable.

We found that with the software evolution, there is an increase of stable classes compared to a decrease of instable classes.

The following table lists the application that will be used for our study:

Applications	No. Version	No. Major Version	No. Package	No. Files
Free	5	0	23	182
Jigsaw	3	2	211	2086
Jext	11	2	83	1059
Jedit	9	2	78	1664

Table 9 Summary Java Applications

Major Releases

A major software version is a standalone release containing major functional enhancements, new features, or significant changes to the software. These versions are indicated by the number to the left of the first decimal point, and are labeled as 2.0, 3.0, etc.

Minor Releases

A minor software version is a standalone release containing minor functional enhancements and/or bug fixes. These versions are indicated by the number to the right of the first decimal point, and are labeled as 2.1, 2.2, etc.

Maintenance Releases (Patches)

A maintenance version is a partial release containing only bug fixes or minor feature enhancements. Maintenance versions are indicated by the number to the right of the second decimal point, and are labeled as 2.0.1, 2.0.2, etc.

For this project, we did not classify the stability between maintenance releases because the design change is irrelevant between those versions.

5.2.1 Jigsaw

Jigsaw [28] is W3C's Web server platform, providing a sample HTTP 1.1 implementation and a variety of other features on top of an advanced architecture implemented in Java. Jigsaw is a W3C Open Source Project; started May 1996.

Jigsaw 2.0 beta 1 was the first stable and complete version of Jigsaw 2.0, offering improved extensibility, with the ability to serve resources using multiple protocols. This was motivated by the desire to support both the HTTP-NG and HTTP/1.1 specifications.

Releases of the Jigsaw server that were considered for our study are: v2.1.2, v2.0.5 and v.1.0 b2.

The graph below shows the evolution of Jigsaw application from version 1.0.2 to 2.1.2.

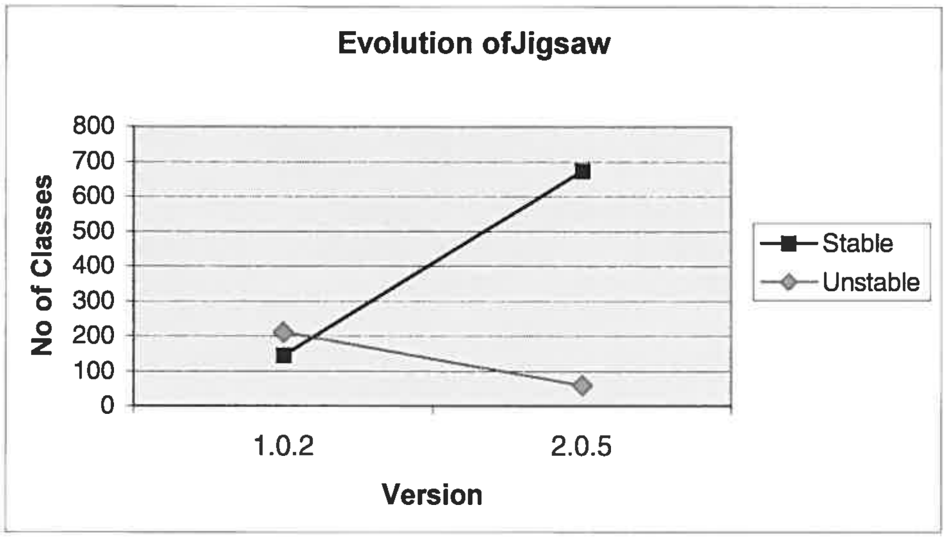


Figure 16: Evolution of Jigsaw

Version 1.0.2 is an early version of Jigsaw with around 300 classes, and it consists of more instable classes then stable ones when comparing it with version 2.0.5. This can be due mainly to the change of design, absence of specific requirements. Version 2.0.5 compared with version 2.1.2, shows more stability. We notice that the number of classes in the system have doubled, and the ratio of instable classes has significantly decreased. We can deduce that in the later version the design was for the major part intact, and that only new requirement have been added.

5.2.2 JEdit

JEdit [26] is a programmer's text editor written in Java, being developed by Slava Pestov and others. This text editor is Open Source software, released under the GNU General Public License. jEdit supports syntax highlighting for more than 60 file types.

Releases of jEdit text editbor that were considered for our study are: v1.2, v1.3, v1.4, v2.0, v2.1, v2.2, v2.3, v2.4, v2.5, and v2.6.

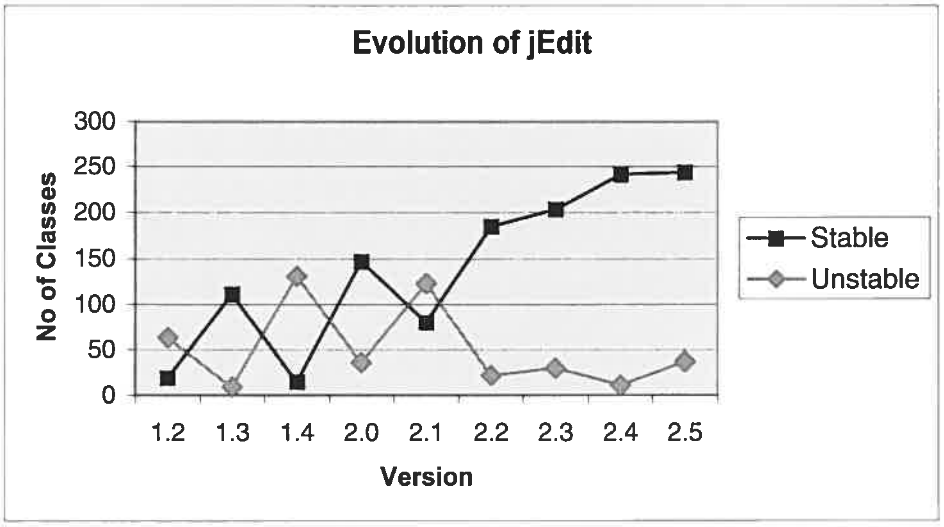


Figure 17 Evolution of jEdit

Later versions of JEdit are more stable then the earlier ones. The biggest ratio of instable classes can be found from version 1.4 to 2.0. First of all, this is a migration to a major version, and it constitutes in a design change, which explain the instability of the system. After version 2.1, we can notice the diversion between the stable and instable line, which is an indication of a stable design.

5.2.3 Jext

Jext [27]is a Java programmer's text editor developed by Romain Guy and others. This text editor is Open Source software, released under the GNU General Public License.

Releases of Jext text editor that were considered for our study are: v1.2, v1.4, v2.0.2, v2.2.7, v2.3, v2.4.8, v2.5, v2.6.1, v2.7, v2.8, and v2.9.

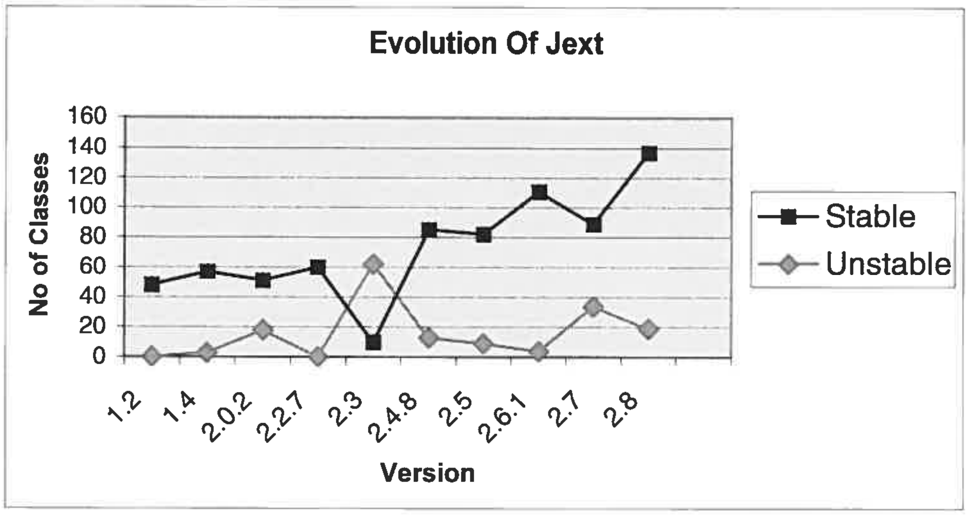


Figure 18: Evolution of Jext

The migration of Jext from version 2.3 to 2.4.8 involved the redesigning of the system. Later versions involved addition of requirements without major design change.

5.2.4 GNU.Free

GNU.FREE [21] is an Internet Voting system and is written in Java. This voting system software suite is an official package of the Free Software Foundation's GNU project and is supported by FreeDeveloper.net.

Releases of Free that were considered for our study are: v1.0, v1.1, v1.2, v1.3, and v1.4. Each of these versions includes new features to the editor as well as bug fixing.

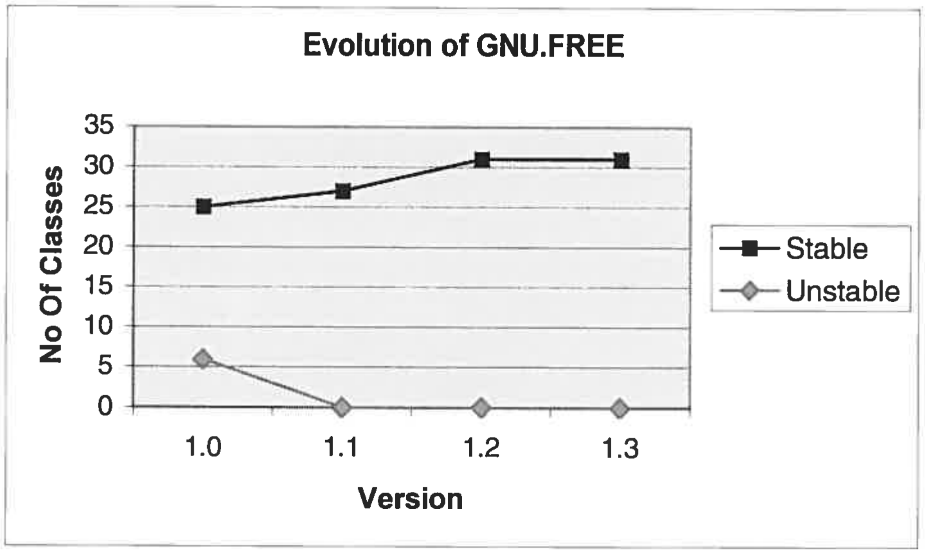


Figure 19: Evolution of GNU.FREE

GNU.FREE is a relatively small application, around 30 classes. Version 1.2 and 1.3 didn't encounter a design change.

We can conclude that the smaller the application, the less it undergoes design change. Also the latest versions of the application are stable application. There is less design change in terms of deleting or modifying the accessible interface of a class and thus affecting the interdependency between the classes of the system.

5.3 Building models of evaluation and detection

In order to verify the interface stability hypotheses proposed above, we need to build some characterization models. We use these models to emphasize the relationship between interface stability and specific properties of OO components. These models can be used to easily assess interface stability based on their level of inheritance, coupling, or encapsulation. The model building technique that we used is a machine-learning algorithm called C4.5.

C4.5 introduced by Quinlan is a program for inducing classification rules in the form of decision trees from a set of given examples. C4.5 was used in past works to generate estimation models in software engineering. It was used to build predictive models for the detection of faulty and reusable components in OO systems [24][33], and detection of interface stability in class libraries [42].

All trees produced, both pre- and post-simplification, are evaluated on the training data. If required, they can also be evaluated on unseen data in the file filestem.test.

C4.5 consists of two types of variables: a set of variables *independent* variable and only one *dependent* variable. Independent variables are those that are manipulated, whereas dependent variables are only measured. The terms dependent and independent variable apply mostly to experimental research where some variables are manipulated, and in this sense they are independent from the initial reaction patterns, features, intentions, etc. of the subjects. Some other variables are expected to be dependent on the manipulation or experimental conditions. That is to say, they dependent on what the subject will do in response. In this research, the dependent variable is the classification of stability.

The C4.5 machine learning algorithm partitions continuous attributes, in our case the design metrics, independent variable, finding the best threshold among the set of training cases in order to classify them on the dependent variable, which is in our case *stable/instable* classes. Independent variables are those that are manipulated, whereas dependent variables are only measured or registered.

The following table presents a sample of the input data needed for the process of predictive model construction

Jext Version 2.3 Class	**Design Metrics**						**Stability Type**
	DIT	NOC	DAM	OAM	OCAEC	OCMEC	
com.chez.powerteam.jext.Jext	0	0	0.79	0.97	2	3	1
com.chez.powerteam.jext.misc.AutoSave	0	0	1.00	1.00	1	0	1
com.chez.powerteam.jext.FindReplace	0	0	1.00	0.33	0	0	2
com.chez.powerteam.jext.JextTextArea	0	0	1.00	0.94	2	1	2
com.chez.powerteam.jext.MenuAction	0	58	0.00	1.00	0	0	2
....	...	...	...	...	...	...	....

Table 10: Dependent and Independent variables

In order to be meaningful to the C4.5 system, this information has to be converted into two files: *names* and *data.*

Names file: defines class, attribute and attribute value names. The first entry in the names file gives class name, separated by commas. In our case it will be 1 and 2 (for stability type). 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. Four specifications are possible:

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

```
1, 2. |Stability Type
DIT:    continuous. | Depth of Inheritance
NOC:    continuous. | Number of Children
DAM:    continuous. | Data Access Metric
OAM:    continuous. | Operation Access Metric
OCAEC:  continuous. | Others class-method export coupling
OCMEC:  continuous. | Others class-attribute export coupling
```

Figure 20: A names file (classes, attribute, and attribute values)

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 attributes (design metrics) and then class of the case (stability type), separated by commas. 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.

```
3,0,1.00,0.90,0,0,1
1,0,0.75,0.70,0,0,1
0,0,1.00,1.00,0,0,1
0,0,1.00,1.00,0,0,1
5,2,1.00,0.93,0,0,1
1,0,1.00,0.92,0,0,1
0,0,0.00,1.00,0,0,1
0,0,0.00,1.00,2,13,1
0,0,0.00,1.00,0,0,1
..............
```

Figure 21: A portion of *data* file

5.3.1 Results

First we studied the stability of system class between application's major and minor versions. Major software versions include mainly new features, minor versions relate mainly to bug fixing. Therefore we expect that the migration to a major version can cause more design change and thus more instable classes.

The number of instable classes between Major Versions doubles the stable classes, a ratio of 1:2 for stable to instable classes. As for Minor Versions, the number of stable classes is 3 times more then the instable classes, a 3:1 ratio for stable to instable classes. This is an indication that the migration to a Minor Version involves mostly changes within the scope of the class itself for example changes in the private interface or within a method, which is related to bug fixing. Migration to a Major Version showed that it is the visible interface of a class that is affected, which has an impact on the interdependency between classes.

We split the data into 2 partitions: 75% are training data to build the model and 25% are test data to verify how well it actually works. The test file is formatted exactly like the training data file, however it is used to validate the decision tree generated from the training data file.

Case1: Between Major Versions

Major Version				
Systems	**Version**	**No of Classes**	**#Stable**	**#Instable**
Jedit	1.2 (compared with v 2.6)	82	2	80
Jext	1.2 (compared with v 2.9)	48	3	45
Jigsaw	1.0.2 (compared with v.2.1.2)	356	145	211
Total		486	150	336

	Percentage	**No Of Classes**	**Actual No of Classes**	**#Stable**	**#Instable**
Machine Learning	75% of 486	340.2	336	100	236
Testing	25% of 486	145.8	150	50	100

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

Rule 1:
`DIT = 0`
`DAM > 0.67`
`OAM <= 0.94`
`OCAEC = 0`
`OCMEC = 0`
`-> class 2`
`[86.0%]`
	Rule 2:
	`DIT <= 1`
	`OCAEC <= 6`
	`OCMEC > 2`
	`OCMEC <= 8`
	`-> class 2`
	`[84.1%]`
		Rule 3:
		`OAM <= 0.63`
		`-> class 2`
		`[75.1%]`

Rule 4:
`NOC > 2`
`OCMEC <= 2`
`-> class 2`
`[66.2%]`
	Rule 5:
	`OAM > 0.63`
	`OCMEC > 8`
	`-> class 1`
	`[77.7%]`
		Rule 6:
		`DIT > 1`
		`-> class 1`
		`[76.5%]`

Rule 7:
`DIT <= 1`
`OAM > 0.72`
`OCAEC > 0`
`OCMEC <= 2`
`-> class 1`
`[71.8%]`		

Figure 22: Predictive model for hypothesis 1 in rule based model between major version s

Coupling vs. Stability

Rule 5 presents the impact of coupling on stability. It reads: "If Operation Access Metric is greater then 0.63 and Class-Method Export Coupling is greater then 8, then that component is likely to be stable (with confidence factor of 77.7%)". A class with high coupling is less susceptible to change, because the interdependency is high and a change can cause a negative ripple effect. Therefore, it is unlikely to change a class that can cause a design change.

Encapsulation vs. Stability

Rule 1 presents the impact of encapsulation on stability. It reads: "If Data Access Metric is greater than 0.67, then that component is likely to be stable (with confidence factor of 86.0%) ". By hiding the internal representation, this enhances the flexibility to change the internal representation without cascading the changes to other classes. Encapsulation reduces the dependency between classes, thus making a design stable.

Inheritance vs. Stability

Rule 4 presents the impact of inheritance on stability. It reads: “If the number of children is greater than 2, then that component is likely to be stable (with confidence factor of 66.2%) ”. If there are 2 classes or more that inherit from a class, this class should be stable. A change in a super class will have a ripple effect on the sub classes. Therefore it is unlikely that this class will change, and thus stable.

Rule 6 presents the impact of the depth of inheritance on stability. It reads: “If the depth of inheritance is greater than 1, then that component is likely to be instable (with confidence factor of 76.5%)”. A class with DIT value greater than one is not in the top portion of its hierarchy. The deeper a class is within the hierarchy, the more specific it is and susceptible to change and thus instable.

Case2: Between Minor Versions

Minor Version			
Systems	**No of Classes**	**#Stable**	**#Unstable**
Total	3809	2910	899

	Percentage	**No Of Classes**	**Actual No of Classes**	**#Stable**	**#Unstable**
Machine Learning	75% of 3809	2666	2600	2100	500
Testing	25% of 3809	1143	1209	810	399

Rule 1:
```
DIT = 0
DAM <= 0.88
OAM <= 0.55
OCAEC = 0
          ->  class 1
[73.1%]
```

Rule 2:
```
DAM > 0
DAM <= 0.88
OAM > 0.82
OCAEC > 0
          ->  class 1
[67.5%]
```

Rule 3:
```
DIT = 0
DAM <= 0.15
OAM > 0.55
OAM <= 0.82
          ->  class 2
[93.0%]
```

Rule 4:
```
NOC > 0
OCAEC = 0
          ->  class 2
[82.7%]
```

Rule 5:
```
DIT = 0
DAM > 0.07
OCAEC <= 1
          ->  class 2
[80.2%]
```

Rule 6:
```
DAM > 0.88
          ->  class 2
[80.1%]
```

Rule 7:
```
DIT = 0
OCAEC > 0
          ->  class 2
[78.5%]
```

Figure 23: Predictive model for hypothesis 1 in rule based model between minor version s

Coupling vs. Stability

Less accurate rules were found related to coupling and stability. As discussed previously, usually the change between minor versions is mainly due to bug fixing and less is due to design change.

Some conclusions were found related to encapsulation and inheritance.

Encapsulation vs. Stability

Rule 6 states that if Data Access Metric is greater than 0.88, then that component is likely to be stable (with confidence factor of 80.1%).

Inheritance vs. Stability

Rule 4 states that if the Number of Children is greater than 0, then that component is likely to be stable (with confidence factor of 82.7%).

Rule 7 states that if the Depth of Inheritance of the class is 0, then that component is likely to be stable (with confidence factor of 78.5%).

5.3.2 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[5][24]. The high accuracy of the predictive model means that the selected OO design measures have been useful for identification of class stability. 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., stable, instable) and actually did belong to that classification group. If correctness is low, it means that the model is identifying more classes as being instable when they are actually stable or vice-versa.[9]

Completeness: is defined as the percentage of those components that belonged to certain classification group (i.e., stable, instable) and were identified by the model. If completeness is low, then more components that likely to be instable or stable will not be identified.

Predicted Stability

		Instable	Stable	Completeness
Real Stability	Instable	n_{11}	n_{12}	$\frac{n_{11}}{\sum_{j=1..2} n_{1j}}$
	Stable	n_{21}	n_{22}	$\frac{n_{22}}{\sum_{j=1..2} n_{2j}}$
	Correctness	$\frac{n_{11}}{\sum_{i=1..2} n_{i1}}$	$\frac{n_{22}}{\sum_{i=1..2} n_{i2}}$	

Finally the model accuracy measure how correct is the model. It is given by the following formula:

$$Accuracy = \frac{\sum_{i=1..2} n_{ii}}{\sum_{i,j=1..2} n_{ij}}$$

The following tables present the empirical evidence of the quality of the models built.

Major Version: Testing Data

Tested 100, errors 40 (40%)			
	Predicted Instable	Predicted Stable	Completeness
Actual Instable	**48**	**3**	**94.12%**
Actual Stable	**37**	**12**	**24.49%**
Correctness	**56.47%**	**80.00%**	
Accuracy = **60.00%**			

Minor Version: Testing Data

Tested 1060, errors 384 (36.2%)			
	Predicted Instable	Predicted Stable	Completeness
Actual Instable	**45**	**272**	**14.20%**
Actual Stable	**112**	**631**	**84.93%**
Correctness	**28.66%**	**69.88%**	
Accuracy = **63.77%**			

We can notice that the accuracy percentage is low. This is mainly due to the unbalance number between stable and instable classes. In the major version case there are more instable classes then stable ones, therefore it is difficult to predict the stability of classes. In the minor version case there are more stable classes then instable ones, making it difficult to predict the instability of classes. In consequence we've decided to apply Youden's J-index in order to resolve the unbalance factor.

5.3.2.1 J-index

Software quality prediction data are often unbalanced, that is, software components tend to have one label with a much higher probability than other labels. For example, in our experiments we had much more stable than unstable classes between minor versions and vice-versa between major classes. On an unbalanced data set, low training error can be achieved by the constant classifier function that assigns the majority label to every input vector. By using the training error, we found that C4.5 tended to "neglect" stable classes for major versions and "neglect" instable classes for minor versions. To give more weight to data points with minority labels, we decided to use Youden's J-index[43][10] defined as

$$J = \frac{1}{k}\sum_{i=1}^{k}\frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}$$

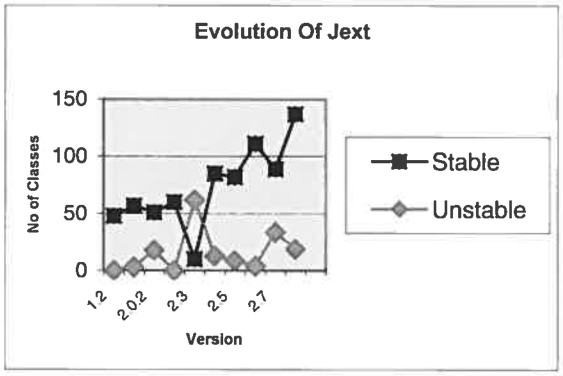
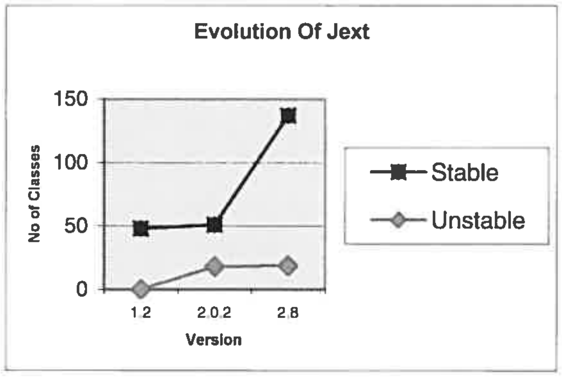
Where J is the average correctness per label. If we have the same number of points for each label, then $J = Accuracy$. However, if the data set is unbalanced, J gives higher relative weight to data points with rare labels. In statistical terms, J measures the correctness assuming that the a-priori probability of each is the same. A constant classifier would have a J-index close to 0.5, while a perfect classifier would have $J(f) = 1$.

		J-Index *J(f)*
Training	Major Version	0.71
	Minor Version	0.56
Test	Major Version	0.60
	Minor Version	0.50

Table 11: J-Index Value

The J-Index values indicate that stability prediction between minor versions is less accurate then between major versions. In fact, migration to a minor version involves mainly bug fixing and minor changes, therefore it is difficult to predict the changes affecting the stability of the software.

In consequence if we look at the evolution of software and compare between Minor version and the Major version graph representation we can see clearly that the stability is better defined when comparing classes between major versions.

Minor Version Graph	**Major Version Graph**
	

As a result, we will consider only the values obtained between Major versions to validate our hypothesis.

The measures that were taken into account are: DIT, OCMEC, OAM and NOC with the following rules.

Rule 5:	**Rule 6:**	**Rule 3:**

`OAM > 0.63`
`OCMEC > 8`
`->`
`class 1`
`[77.7%]`
	`DIT > 1`
	`->`
	`class 1`
	`[76.5%]`
		`OAM <= 0.63`
		`->`
		`class 2`
		`[75.1%]`

On the other hand, for the following metrics: DAM and OCAEC, the results showed that they were less significant in predicting the level of stability of a class. This indicates that the change in a class attribute does not affect the interdependency between classes, thus its stability. Usually, class attributes are private within the class, and an attribute value is usually accessed by a get function. Also for NOC we didn't get significant results.

From our result, we can conclude that the degree of interdependence between the class and the environment within which it is defined has an effect on its stability and DIT, OAM, and OCMEC are indicators of the level of stability.

Chapter 6

6. Conclusion

A new measurement tool, BOAP, for assessment of design and implementation quality attributes in object-oriented designs has been introduced. The key component of this tool is its language-independent metamodel.

The metamodel has proven to have advantages over existing ones. First of all, the supporting modules, i.e. metrics extraction module and analysis module, support one representation. Therefore, it allows having a common analysis set tool applied to an application independently of its programming language. This is ideal when the purpose is to develop generic predictive rules that can be applicable on any system.

In this project research, the BOAP approach was used to analyze the class interface's stability during the evolution of its system. By using the generic metamodel as our system representation, this allowed us to generate generic predictive rules on stability that can be applicable on other systems.

The object oriented implementation of the metamodel, and especially its persistence, has provided a key advantage when utilizing the model in other applications.

The major contributions of this work are:

- The definition of the BOAP architecture.
- The addition of the following functionalities to the BOAP architecture:
  - The mapping of Java programs into the generic metamodel
  - The development of a metric module that extracts its measures from the Generic Model database.

    - The development of a stability module that uses the database and the metrics extracted to classify the stability of a class.

- The stability hypotheses and their metrics suites
- The building of predictive models

As for future work, first of all, we are still refining the generic metamodel. This includes the addition of language features currently not supported, detailed information below the method body level and the support for more languages beyond the two C++ and Java that are currently supported.

Presently, the ASG representation is the input for the mapping module; consequently the mapping module is limited to the programming languages supported by the Datrix parser. Therefore, the next step is to make the mapping module independent of the Datrix parser. A solution would be to customize our own parsers.

Apart from refining the metamodel itself we are looking at explicit metamodel support. The goal is to be able to generate generic tools. We have developed a metric module that extracts metrics from the generic representation, and a stability classification module that classifies the stability of the components of the system. Once the implementation of the generic metamodel is complete, further analysis can be done on the implementation level.

In addition, a parallel experiment can be done on other systems of different programming languages, and compare the results of the predictive model obtained for the accuracy of the generic quality of our model.

7. References

[1] Abety C., Jia S.H., Lounis H., Mili H., Rizand J.F., Sahraoui H., 'RAME-Rétro-ingénierie d'éléments Architecturaux de programmes pour en Mesurer la facilite d'Evolution et la résilience au changement' Livrable1, Centre de recherché informatique de Montréal, August 1999.

[2] Abreu F. B., Ochoa L., Goulão M., 'The GOODLY Design Language for MOOD Metrics Collection', ISEG/INESC

[3] Abreu F. B., Carapuça R., 'Object-Oriented Software Engineering: Measuring and Controlling the Development Process', Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994.

[4] Alikacem E. H., Lounis H., Sahraoui H., Mehio N., Cantave R., 'Livrable Rédigé pour le CTL: Boîte à outils d'Analyse de Développement de Programmes: BOAP', June 2000.

[5] Almeida M. A., Lounis H., Melo W., 'An Investigation on the Use of Machine Learned Models for Estimating Software Correction Costs', In 20th IEEE International Conference on Software Engineering, 1998.

[6] Bansiya J., 'Evaluating Framework Architecture Structural Stability', ACM Computing Surveys (CSUR) Vo 32, March 2000

[7] Bansiya J., Davis C., 'Automated Metrics and Object-Oriented, Dr. Dobb's Journal, Vol. 22, No12, pp.42, December 1997.

[8] Bansiya J., 'A hierarchical Model for Quality Assessment of Object-Oriented Design', PhD Dissertation, University of Alabama, 1997.

[9] Basili V., Condon S., Emam K.E., and Melo W.L., 'Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components', In 19th IEEE International Conference on Software Engineering, May 1997.

[10] Bouktif S., Kégl B., Sahraoui H. 'Combining Software Quality Predictive Models: An Evolutionary Approach', ICSM 2000: 385-392.

[11] Briand L., Devambu P., and Melo W.L., ' An Investigation into Coupling Measures for C++', In 19th IEEE International Conference on Software Engineering, Boston, Massachusetts, May 1997.

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

[13] Chidamber S. R., Kemerer C. F., 'A Metrics Suite for Object-Oriented Design', IEEE Transactions on Software Engineering, Vol. 20, No6, pp. 476-493, June 1994.

[14] Datrix, Abstract Semantic Graph, Reference Manual, Version 1.3, January 2000. http://www.iro.umontreal.ca/labs/gelo/datrix.

[15] Dumke, R. R., & Kuhrau, I., 'Tool-Based Management in Object-Oriented Software Development,' *IEEE* 1994.

[16] Ducasse, S., Lanza M., Tichelaar S., 'Moose: an extensible language- independent environment for reengineering object-oriented systems, in: Proc. 2nd Int'l Syrup. Constructing Software Engineering Tools,' CoSET 2000.'

[17] Ducasse, S., Demeyer S., 'The FAMOOS Object-Oriented Reengineering Handbook.' University of Berne, October 1999. See http://www.iam.unibe.ch/~famoos/handbook.

[18] Fayad M., 'Accomplishing Software Stability', Communications of the ACM, Vo 45, Jan 2002.

[19] Fayad M., Altman A., 'An Introduction to Software Stability', Communications of the ACM, Vo. 44, September 2001.

[20] Fioravanti, F., Nesi, P., Perlini, S., 'A Tool for Process and Product Assessment of C++ Applications', *Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering,* CSMR98, IEEE Press, Florence, Italy, pp.89-95, 8-11 March 1998.

[21] Free: http://www.free-project.org/

[22] Gamma, E., Helm, R., Johnson, R., & Vlissides, J., 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley, 1994.

[23] Gosling, J., Joy, B., Steele G., 'The JavaTM Specification (2nd edition)', Addison Wesley Longman, Reading, MA, United Stated 1996. http://java.sun.com/docs.

[24] Ikonomovski S. 'Detection of faulty components in OO systems using design metrics and a machine learning algorithm', Master thesis, McGill University, 1998.

[25] ISO 9126 Information technology –Software Product Evaluation- Quality Characteristics and Gidelines for Their Use. International Organisation for Standardization. Geneva, 1992.

[26] jEdit: http://jedit.sourceforge.net/

[27] Jext: http://www.jext.org/

[28] Jigsaw - W3C's Server: http://www.w3.org/Jigsaw/

[29] Laguë B., April A., 'Mapping of Datrix software metrics set to ISO 9126 Maintainability subcharacteristics', in Software Engineering Standards Workshop, Oct. 96, Montreal, Canada.

[30] Laguë B., Leduc C., Le Bon A., Merlo E., Dagenais M., 'An analysis framework for understanding layered software architectures'. In Proceedings IWPC'98, 1998.

[31] Lehman M., 'Programming and Methodology', Springer, Verlag, pp. 42-62, 1978.

[32] Martin R. C., 'Stability', C++ Report, Feb 1997.

[33] Mao Y., Sahraoui H. A. and Lounis H. 'Reusability Hypothesis Verification Using Machine learning Techniques: A Case Study', Proc. of IEEE Automated Software Engineering Conference, 1998.

[34] Mattsson M., Bosch J., Characterizing Stability in Evolving Frameworks, In Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS EUROPE '99, Nancy, France, pp. 118-130, June 7-10, 1999

[35] Mayrand J., Laguë B., 'Object Oriented Architecture Assessment Using Metrics', OOPSLA96.

[36] McCall J.A., Richards P.K., and Walters G.F., 'Factors in Software Quality,' vol. 1, 2, and 3, AD/A-049-014/015/055, National Tech. Information Service, Springfield, Va., 1977.

[37] Michael M., Evolution Characteristics of an Industrial Application Framework, Workshop on Object-Oriented Architectural Evolution at the 13th European Conference on Object-Oriented Programming, ECOOP '99, Lisbon, Portugal.

[38] Object Management Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, June 1999.

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

[40] Parnas D. L., 'Software Aging' Plenary Talk, 1994.

[41] Pressman R. S., 'Software Engineering. A Practical Approach', fourth edition, McGraw-Hill, 1997.

[42] Sahraoui H. A., Boukadoum A. M., Lounis H., Ethève F., 'Predicting Class Libraries Interface Evolution: an investigation into machine learning approaches', Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC'00).

[43] Youden W. J., 'How to evaluate accuracy. Materials Research and Standards', ASTM, 1961.

Appendix A

C4.5 result for Stability predication between Major Versions

```
C4.5 [release 8] rule generator
-------------------------------

    Options:
          File stem
          Rulesets evaluated on unseen cases

Read 220 cases (6 attributes) from ApprentissageMajeur1

------------------
Processing tree 0

Final rules from tree 0:

Rule 3:
         DIT <= 0
         DAM > 0.67
         OAM <= 0.94
         OCAEC <= 0
         OCMEC <= 0
         ->  class 2  [86.0%]

Rule 11:
         DIT <= 1
         OCAEC <= 6
         OCMEC > 2
         OCMEC <= 8
         ->  class 2  [84.1%]

Rule 1:
         OAM <= 0.63
         ->  class 2  [75.1%]

Rule 5:
         NOC > 2
         OCMEC <= 2
         ->  class 2  [66.2%]

Rule 13:
         OAM > 0.63
         OCMEC > 8
         ->  class 1  [77.7%]

Rule 17:
         DIT > 1
         ->  class 1  [76.5%]

Rule 9:
         DIT <= 1
         OAM > 0.72
         OCAEC > 0
         OCMEC <= 2
         ->  class 1  [71.8%]

Rule 4:
         DIT <= 0
         NOC <= 2
         OAM > 0.94
         OCMEC <= 0
         ->  class 1  [58.2%]
```

```
Default class: 1


Evaluation on training data (220 items):

Rule  Size  Error  Used  Wrong                  Advantage
----  ----  -----  ----  -----                  ---------
   3     5  14.0%    18      1 (5.6%)      8 (8|0)        2
  11     4  15.9%     8      0 (0.0%)      7 (7|0)        2
   1     1  24.9%    23      5 (21.7%)    13 (18|5)       2
   5     2  33.8%     5      1 (20.0%)     3 (4|1)        2
  13     2  22.3%    11      1 (9.1%)      0 (0|0)        1
  17     1  23.5%    32      5 (15.6%)     0 (0|0)        1
   9     4  28.2%    25      5 (20.0%)     0 (0|0)        1
   4     4  41.8%    40     16 (40.0%)     0 (0|0)        1

Tested 220, errors 61 (27.7%)   <<


           (a)  (b)         <-classified as
          ---- ----
           112    7         (a): class 1
            54   47         (b): class 2


Evaluation on test data (100 items):

Rule  Size  Error  Used  Wrong                  Advantage
----  ----  -----  ----  -----                  ---------
   3     5  14.0%     6      3 (50.0%)    -1 (1|2)        2
  11     4  15.9%     1      0 (0.0%)      1 (1|0)        2
   1     1  24.9%     5      0 (0.0%)      5 (5|0)        2
   5     2  33.8%     3      0 (0.0%)      3 (3|0)        2
  17     1  23.5%    11     11 (100.0%)              0 (0|0)        1
   9     4  28.2%     5      5 (100.0%)              0 (0|0)        1
   4     4  41.8%    53      9 (17.0%)     0 (0|0)        1

Tested 100, errors 40 (40.0%)   <<


           (a)  (b)         <-classified as
          ---- ----
            48    3         (a): class 1
            37   12         (b): class 2
```

C4.5 result for Stability predication between Minor Versions

```
C4.5 [release 8] rule generator
-------------------------------

    Options:
         File stem
         Rulesets evaluated on unseen cases

Read 1682 cases (6 attributes) from ApprentissageMineur

------------------
Processing tree 0

Final rules from tree 0:

Rule 1:
         DIT <= 0
         DAM <= 0.88
         OAM <= 0.55
         OCAEC <= 0
         ->  class 1  [73.1%]

Rule 11:
         DAM > 0
         DAM <= 0.88
         OAM > 0.82
         OCAEC > 0
         ->  class 1  [67.5%]

Rule 4:
         DIT <= 0
         NOC <= 0
         DAM <= 0.17
         OAM > 0.82
         OCAEC <= 0
         ->  class 1  [49.7%]

Rule 2:
         DIT <= 0
         DAM <= 0.15
         OAM > 0.55
         OAM <= 0.82
         ->  class 2  [93.0%]

Rule 6:
         NOC > 0
         OCAEC <= 0
         ->  class 2  [82.7%]

Rule 12:
         DIT <= 0
         DAM > 0.07
         OCAEC <= 1
         ->  class 2  [80.2%]

Rule 24:
         DAM > 0.88
         ->  class 2  [80.1%]

Rule 7:
         DIT <= 0
         OCAEC > 0
         ->  class 2  [78.5%]

Default class: 2
```

```
Evaluation on training data (1682 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
   1     4  26.9%     9      1 (11.1%)      7 (8|1)        1
  11     4  32.5%    11      2 (18.2%)      7 (9|2)        1
   4     5  50.3%   130     61 (46.9%)      8 (69|61)      1
   2     4   7.0%    19      0 (0.0%)       0 (0|0)        2
   6     2  17.3%    78     11 (14.1%)      0 (0|0)        2
  12     3  19.8%   259     44 (17.0%)      0 (0|0)        2
  24     1  19.9%   193     48 (24.9%)      0 (0|0)        2
   7     2  21.5%    53     10 (18.9%)      0 (0|0)        2

Tested 1682, errors 477 (28.4%)   <<

            (a)  (b)         <-classified as
           ---- ----
             86  413         (a): class 1
             64 1119         (b): class 2


Evaluation on test data (1060 items):

Rule  Size  Error  Used  Wrong                 Advantage
----  ----  -----  ----  -----                 ---------
   1     4  26.9%    32     23 (71.9%)    -14 (9|23)       1
  11     4  32.5%    10      6 (60.0%)     -2 (4|6)        1
   4     5  50.3%   115     83 (72.2%)    -51 (32|83)      1
   2     4   7.0%     2      1 (50.0%)      0 (0|0)        2
   6     2  17.3%    37     14 (37.8%)      0 (0|0)        2
  12     3  19.8%   275     98 (35.6%)      0 (0|0)        2
  24     1  19.9%   161     88 (54.7%)      0 (0|0)        2
   7     2  21.5%    24      9 (37.5%)      0 (0|0)        2

Tested 1060, errors 384 (36.2%)   <<

            (a)  (b)         <-classified as
           ---- ----
             45  272         (a): class 1
            112  631         (b): class 2
```