L-UNIVERSITÀ TA' MALTA UNIVERSITY OF MALTA

An Automated Software Quality Measurement Tool

Student: Mark Micallef

Course: B.Sc. (Hons) Information Technology (Computer Science)

Supervisor: Dr. Ernest Cachia

Observer: Mr. Adrian Francalanza

To my parents and sisters - for being there no matter what

In memory of Uncle Colin

Acknowledgements

I would like to thank my supervisor, Dr. Ernest Cachia for the help, patience, and insightful advice that he provided throughout the course of this project.

A special thanks goes out to Dr. Jason Robbins, Curt Arnold, Marko Boger, Toby Baeir and the rest of the ArgoUML development community (www.argouml.org) for helping me get acquainted with the inner workings of ArgoUML, the UML editor which this project extends.

Last but not least, I want to thank my closest friends Clyde, James, Daniel, Melissa, Pat and Andrew for helping to make these four years a truly memorable experience.

Table of Contents

Table of Figures

Abstract

Software quality remains a very subjective and at times vague, notion. However it is a fact that most serious developers and the majority of software users require some form of qualitative measure for the software systems they are concerned with. Software quality assurance can be carried out at various stages of the software development process but this project deals with the measurement of the quality of object-oriented designs. The implemented system will allow the user to design any system using the Unified Modeling Language (UML) and then provide quality readings from different perspectives of the system. It is worth noting that it is virtually impossible to come up with a quality scale to gauge the quality of systems. i.e. The project will not produce a result such as “The system is considered to be 85% good quality.” This is mainly due to the fact that when asking for a quality system, conflicts and contradictions tend to occur (e.g. trying to develop a highly reusable system that is very efficient). In the light of this fact, the system will not gauge the quality of the system on a predefined scale but rather provide the user metric readings that provide different views of the quality characteristics of the system in question. The user will use the tool to identify potential problem areas and fix them before the project goes into implementation stage.

1. Theoretical Background

1.1 Introduction

This section will give the reader an insight to the theoretical principles involved in the conception, design and implementation of this project. The content here however is not meant to describe what decisions where taken and why. Such information will be presented in future chapters.

1.2 Quality in General

Quality – arguably one of the most ambiguous terms in popular and professional vocabularies. Is a Porsche a higher quality car than say a Volkswagen is? Is a state-of-the-art hi-fi of higher quality than the cheap stereo system I have at home? Similarly, is the latest 3D-graphics generation software costing thousands of dollars better than a shareware 2D paint package? From a popular viewpoint the answer to these questions may very well be *yes*. However, if one was to approach the interpretation of the term *quality* from the engineering or management perspective, we would see that things are not as straightforward as they seem.

1.2.1 Ambiguity of the term 'Quality'

Quality is by nature a multi-faceted concept that means different things to different people. The concept of quality depends highly on the entity of interest, the viewpoint on that entity, and the quality attributes of that entity. A good quality car for a family would probably be one that has enough room for all the members of the family, has an economical engine and is safe in the event of a collision with another car. On the other hand, a good quality car to a racecar driver would be one that is lightweight, has high-acceleration, good brakes etc.

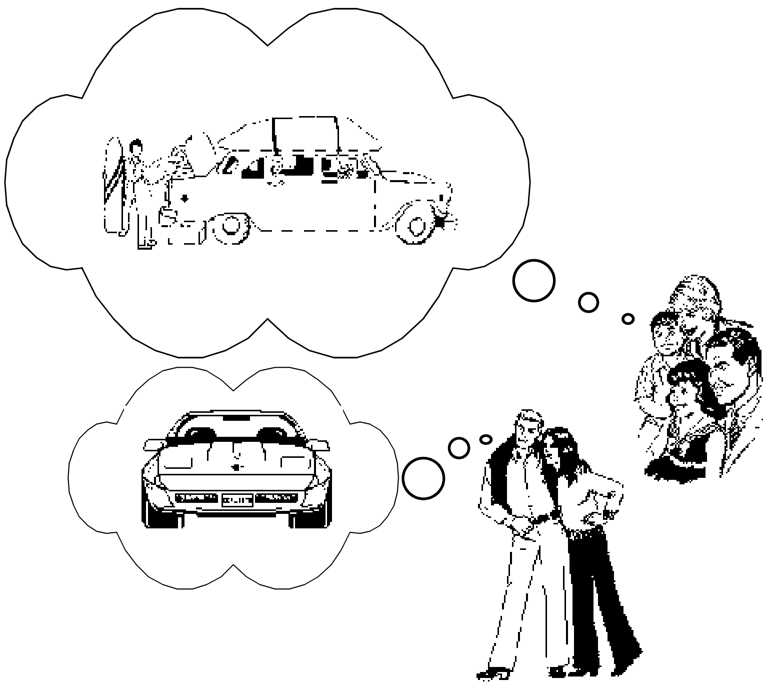


Figure 1 - Different concepts of a quality car

Also, the term *quality* tends to be used at different levels of abstraction. A manager might instruct a designer to design a good-quality mobile phone and he would be referring to quality in the broadest sense of the word. On the other hand, the designer will probably have a more specific concept of a good-quality mobile phone including attributes such as reliable communications and an attractive exterior. These attributes can also probably be re-explained at a lower level of abstraction.

Lastly, the term *quality* has been made a part of our daily language and it is used very liberally to describe a seemingly endless number of entities (cars, movies, software, etc). Consequently widespread views of the term may be very different from its use in professions where it is approached from the engineering or management perspective. This also contributes to the ambiguity of the term especially in cases where say software developers interact with their would-be customers.

1.2.2 A Definition of Quality

Crosby [CRO79] defines quality as the level to which a product conforms to its requirements. This implies that the requirements must be clearly and unambiguously stated in such a way that they cannot be misunderstood. Measurements can then be taken during the production of a product to determine the level of conformance to those requirements. Then by this definition, if a Porsche conforms to all the requirements of a Porsche, then it is a quality car. Similarly if a Volkswagen conforms to all the requirements of a Volkswagen then it too is a quality car.

1.3 Software Quality

We now switch our attention to a more specific topic – the quality of software. This section will discuss why the development high quality of software is becoming increasingly important, the nature of quality in software as well as different approaches to measuring and improving software quality.

1.3.1 Why is Software Quality important?

Four Marines were killed when their Osprey crashed on December 11th 2000 on approach to the Marine Corps Air Station New River, North Carolina. An enquiry concluded that the crash was caused by the failure of a hydraulic system component compounded by an anomaly in the vehicle's computer software. [CNN01a]

Between 1985 and 1987, seven people died while receiving radiation therapy from a medical linear accelerator at a Texas hospital. Investigations revealed that software controlling the apparatus caused the accidents. If the operator entered an unusual but nonetheless possible sequence of commands, the computer controls would put the machine’s internals into an erroneous and very hazardous state, subjecting patients to a massive overdose. [JAC90]

In June 1996, the Ariane 5 satellite launcher malfunction was caused by a faulty software exception routine resulting from a bad 64-bit floating point to 16-bit integer conversion. [ARI96]

These stories are not everyday occurrences but they nonetheless illustrate the high degree to which we now let computer software influence our lives. Businesses now conduct core transactions amongst themselves via software. Modern cars have embedded computer systems controlling various aspects of their functions. People routinely entrust their money to an ATM whilst making deposits. Airplanes are now capable of taking off and landing with little or minor contributions from pilots.

It seems that people and businesses are naturally resolving to solve and problems they have by building computer systems to deal with them. This is not in itself a harmful trend but as a consequence, software developers are under incredible pressure to deliver increasingly large systems in proportionately less time. Thus insuring the delivery of high-quality software is becoming an increasingly important goal in the life-cycle of software developed by serious companies.

1.4 Measuring Software Quality

The question remains: How can Software Quality be measured? The software development lifecycle produces a number of artifacts from specification documents through to the finished implementation and accompanying documentation. If we take the definition of software quality to be the degree to which the finished product conforms to its specifications, one possible way of measuring quality could be to ensure every one of these artifacts is still inline with specifications as it is produced. If for example, the design is inconsistent with specifications then the software development cycle is not allowed to continue until the design conforms to specifications.

This approach seems to make sense. However, it is at too high a level of abstraction to be of much use. If user-requirements consisted solely of functions to be offered by the system, it would just be a matter of checking that each required function has been implemented. This scenario rarely materializes. Users usually add ‘magic’ phrases like “the system should be efficient” or “the system should be maintainable”. How exactly does one verify that a system is efficient or that a system is maintainable? Before attempting to answer that question, we will look at quality attributes.

1.4.1 Software Quality Attributes

Software quality attributes are a high-level a set of attributes of a software product by which its quality is described and evaluated. A software quality attribute may be refined into multiple levels of sub-attributes. There is also the concept of *high-level quality attributes* and *low-level quality attributes*. High-level quality attributes are at a high level of abstraction or generalization that can usually be broken down into sub-attributes. For example, the attribute *reliability* can be broken down into the sub-attributes *Maturity, Fault-Tolerance,* and *Recoverability*. Achieving these more specific sub-attributes will mean achieving the overall attribute of reliability. The sub-attributes can occasionally be again divided into sub-sub-attributes. This forms a tree of attributes starting from the most general (and high-level) attribute at the root and the most specific (and low-level) attributes at the bottom.

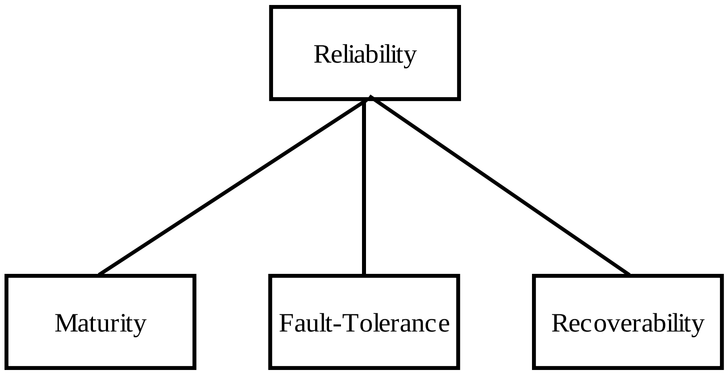


Figure 2 - The *Reliability* high-level attribute and it's low-level sub-attributes

1.4.2 Software Quality Attributes and the ISO-9126 Standard

ISO-9126 is a *Software Product Evaluation Standard* published by the International Standards Organization (ISO) in 1991. In it, the ISO claims that software quality can be defined using the following attributes:

Functionality

A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy the stated or implied needs of the client.

Reliability

A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

Usability

A set of attributes that bear on the effort needed for use, and on the individual assessment of such use by a stated or implied set of users.

Efficiency

A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

Maintainability

A set of attributes that bear on the effort needed to make modifications to the finished system.

<u>Portability</u>

A set of attributes that bear on the ability of software to be transferred from one environment to another.

With the ISO-9126 being an international standard, I have to be critical of the fact that *reusability* was omitted from this list. Given the obvious benefits reuse is known to give (chapter 1.6.1), it should follow that making a software product (or parts of it) reusable will make it a better quality product. Also, this quality will be transferred to new systems where the current product is reused.

It is worth noting that the attributes defined above can be broken down into further sub-attributes as shown below.

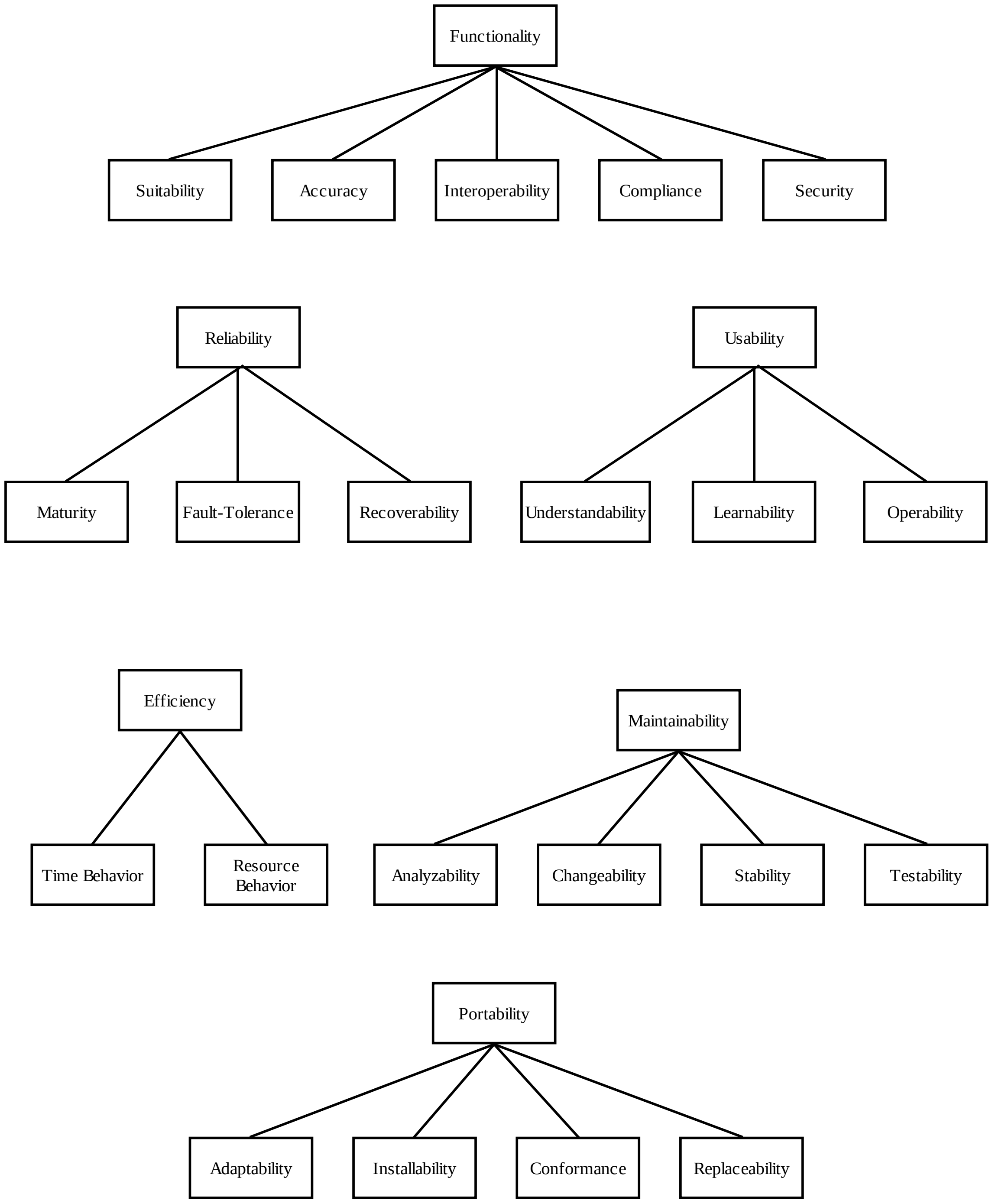


Figure 3 - The sub-attributes of the attributes defined by ISO-9126

1.4.3 Objective vs. Subjective Quality Assessment

It is fair to say that the software engineering community's understanding of software quality is a long way from the required level of refinement and standardization.

Consequently, we are at a stage where there are many arguments for and against different measurement methodologies. There two main schools of thought in this area: there are those who believe that quality assessment is to be an *objective* process whilst others believe that quality is inherently a *subjective* concept and thus the process used to measure it should follow course.

Supporters of objective quality assessment argue that metrics can provide an unbiased assessment of the different facets of quality such as reusability, flexibility, understandability, functionality, extendibility or effectiveness.

On the other hand, those who believe that quality measurement should be subjective emphasize the necessity of design guidelines and a development culture that encourages simplicity, intuitiveness and understandability of software designs to humans. They also argue that quality depends on its context and that metrics cannot provide accurate numbers for the many viewpoints one can have on an application domain. Quality, they say, can only be measured relative to a particular viewpoint.

My personal viewpoint on this (objective vs. subjective measurement) is that I believe in the objective powers of metrics but also appreciate the fact that quality is a multi-faceted concept that means different things to different people. I propose that objective metrics be used as the primary tool for assessing quality. The individual users (companies) of metrics can assign different weights to the metrics they use in order to measure quality from their viewpoint. For example, my company's quality policies may define a quality product as being highly reusable whiles another company might define a quality product as being highly portable. It is simply a case of the first company assigning more weighting to metrics that measure reusability and the latter assigning more weighting to metrics that measure portability. This is of course an over-simplified example but it serves its purpose in illustrating my viewpoint on this topic.

1.4.4 An Introduction to Software Quality Metrics

The word '*metrics*' strikes fear into the heart of many people involved in software engineering. The reason for this being that many people still consider software engineering to be an undisciplined craft rather than a profession where rigorous mathematical tools may (and need to) be utilized. Even people who agree with metric usage disagree amongst themselves with regards to what metrics should be used and how they should be interpreted. Therefore, the first step to making use of metrics should involve assessing a number of available metrics and choosing a suite for use according to a company's quality objectives. This exercise has been carried out for this project and is described in chapter 2.

With that being said, one important question crops up: "*What do we measure?*" There are essentially three measurable entities: *products*, *processes*, and *people*. The term '*products'* refers to the artifacts produced during the software development life cycle. One possible approach to quality assessment could be to verify that each of these artifacts is of a high-level of quality. A different approach could entail the study and refinement of *processes* involved in the life cycle of a product. For example, one can study the process by which requirements are gathered and transformed into specifications. Good and bad trends can be identified enabling management to impose rules on the process thus increasing the likelihood of having higher-quality specifications. Finally, one must not forget that *people* are an integral part of the software development process. One could also take the approach of actually measuring the performance of individuals or groups in a company. However, this could bring about various ethical considerations and should be implemented with caution.

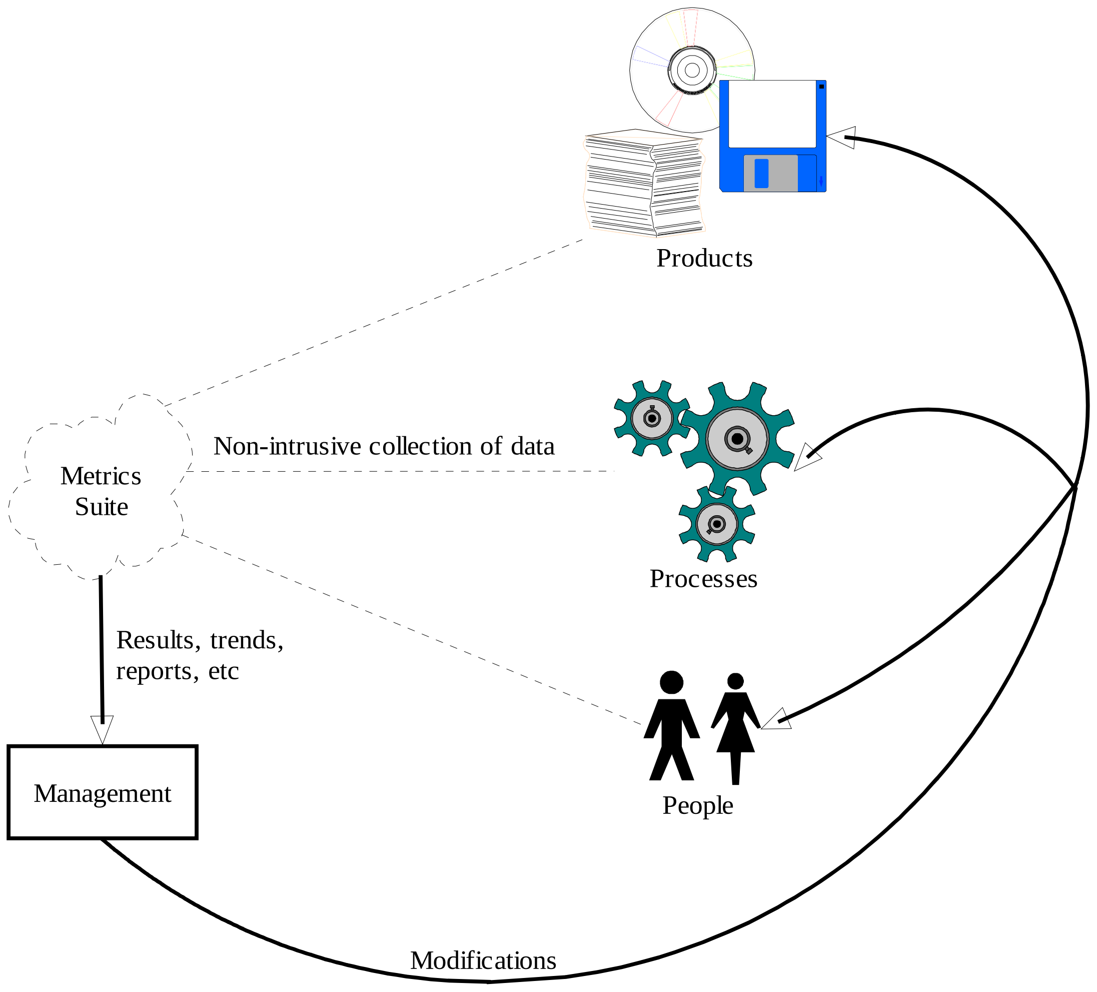

Products
Non-intrusive collection of data
Metrics Suite
Processes
Results, trends, reports, etc
People
Management
Modifications

Figure 4 - The use of metrics in an organization

As shown in figure 4 above, metric data-collection should be a non-intrusive process. Designers, developers and other people involved in the life cycle should be left to go about their business without having to consciously contribute to the data-collection process.

At the end of the day, if used properly, metrics allow us to:

- Quantitatively define success and failure, and/or the degree of success or failure for a product, process or person.

- Identify and quantify improvement, lack of improvement or degradation in the performance of a product, process or person.

- Make meaningful and useful managerial and technical decisions.

- Identify trends.

- Make quantified and meaningful estimates.

On the other hand, metrics can easily be misused. Firstly, those gathering metrics must be aware of items that may influence the metrics they are gathering. For example, one must be aware of the Heisenberg effect whereby it was observed that initial improvements in a process of production might only be happening because of the obtrusive observation of that process and not because of a change in working conditions. The same could be said for a software development environment. If programmers know that management is collecting metrics, their initial reaction will be to be more careful and to increase the quality of their work. Taking an exaggerated example, management may also be giving employees free coffee to make them more content. Care must be taken not to correlate the increase in quality (due to the Heisenberg effect) with the free coffee.

Another way in which metrics could be misused is by making comparisons between two products based only on their similarities. Bernard [BER00] observes that meaningful comparisons can only be made if both the similarities and dissimilarities of the *products*, *processes* or *people* being compared are taken into account.

Metrics must be correlated with reality before any meaningful decisions can be taken. It is important that one does not simply look at the numbers on paper and take a decision based on them. The numbers must first be confirmed as being realistic.

1.4.5 Metrics and the Object-Oriented Paradigm

While metrics for the traditional functional decomposition and data analysis design approach measure the design structure and/or data structure independently, object-oriented metrics must be able to focus on the combination of function and data as an integrated object [CHI94]. The object-oriented paradigm introduces new concepts and structures that traditional metrics will fail to measure. This is not to say that pre-OO metrics are now obsolete but rather that we need to utilize new object-oriented metrics along with selected traditional metrics. At this point, it is worth taking a look at the new features presented by the object-oriented paradigm and why traditional metrics do not measure them.

Localization is the process of placing items in close physical proximity to each other. This process was not introduced by the object-oriented paradigm but rather the object-oriented paradigm localizes information differently to other paradigms. More specifically, *functional decomposition* processes localize information around functions; *data-driven approaches* localize information around data whilst *object-oriented approaches* localize information around objects. In most conventional software, localization is based on functionality and therefore a great deal of metrics gathering has traditionally focused largely on functions and functionality. Also, units of software were functional in nature, thus metrics focusing on component interrelationships emphasized functional interrelationships such as module coupling. In object-oriented software however, localization is based on objects. This means that although we may speak of the functionality offered by an object, at least some of our metrics gathering effort must recognize the *object* as the basic unit of software.

Encapsulation is the binding together of a collection of items. In traditional paradigms, encapsulation consisted of records, arrays, procedures, functions, subroutines etc. Object-oriented languages offer higher level of encapsulation through classes (Java, C++), packages (Java, C++, Ada) and modules (Modula 3) to name a few. Objects encapsulate, *knowledge of state*¸ *advertised capabilities*, *corresponding algorithms to accomplish these capabilities*, *other objects*, *exceptions*, *constants*, and *concepts*.

Encapsulation has an impact on metrics in the sense that the basic unit will no longer be the subprogram but rather the object. Also, we will have to modify our thinking on characterizing and estimating systems.

Information hiding is the suppression (or hiding) of details. The main concept here is that we show only that information which is necessary to accomplish our immediate goals. Some may tend to conclude that encapsulation and information hiding are one and the same. This is not the case since for example an item may be encapsulated but may still me totally visible. Since this is mainly an object-oriented paradigm feature, new metrics are needed for measuring it.

Inheritance is a mechanism whereby on object acquires characteristics from one or more other objects. The amount of inheritance used in a hierarchy will affect high-level quality attributes such as efficiency and reusability. It is thus important that inheritance metrics be included in object-oriented projects.

Abstraction is a mechanism for focusing on the important details of a concept or item, while ignoring the inessential details. It is a relative concept in the sense that as we move to higher levels of abstraction we ignore more and more details thus providing a more general view of a concept or item. As we move to lower levels of abstraction we introduce details and provide a more specific view of a concept or item. There different types of abstraction, namely functional abstraction, data abstraction, process abstraction, and object abstraction. In object abstraction, objects are treated as high-level entities. We know what functionality they offer but we don’t usually care how they implement it (black box principle). Traditional metrics will not measure object abstraction properties and thus new object-oriented metrics are required.

1.5 Measuring the Quality of Object-Oriented Designs

This section aims to show sufficient motive for measuring the quality of object-oriented designs (as opposed to simply testing a finished product) and to introduce foundational concepts related to the topic.

1.5.1 Why Measure the Quality of Designs?

Why would a company want to measure the quality its designs? Why not simply wait for the implemented software package and then subsequently perform testing, fix bugs and release it to market? Let us assume for a moment that there are numerous companies that are using this method and that their software is being released to market fully functional with a minimal number of undiscovered/uncorrected defects. It is worth asking these companies a number of questions: What did it cost to develop that software? What proportion of man-hours was spent on fixing defects as opposed to the actual development? How many of the defects discovered during testing needed major sections of the software to be redesigned? Did these changes in design have a ripple effect on the rest of the project? Did the number of entries in the bug list seem to grow instead of shrink when you started to fix errors?

The consensus in the industry is that there is a direct correlation, which relates the cost of detecting and correcting a fault, with the timing of identifying the fault. Simply stated, the earlier that a fault is detected and removed, the cheaper it is to fix. Testing the product after implementation is complete is almost the worst (worst being not doing any testing at all) and most expensive way to find and fix defects in the product. Testing individual modules as they are developed would be a step forward but why stop there? Why not review and inspect the quality of a design before actual implementation commences? Using this reasoning, one could rightly argue that quality assurance should start from the requirements stage. Don Mills [MIL98] presents research showing that over 55% of errors of a project are introduced during the requirements stage. Figure 5 below shows the estimated cost of fixing a defect that occurred at requirements stage depending on the stage it is found and fixed.

Ideally, software quality assurance procedures should be carried out on each stage of the development life cycle but **the scope of this project lies solely in evaluating the quality of object-oriented designs.**

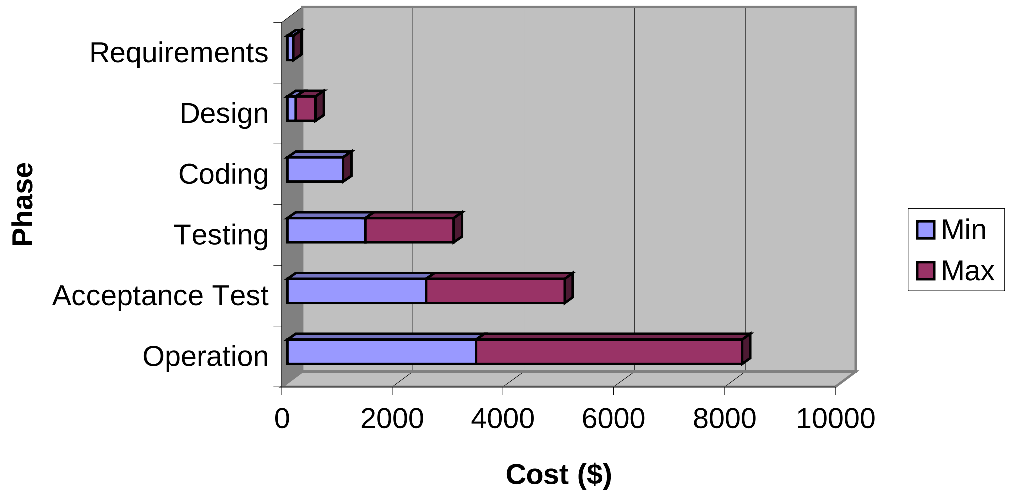


Figure 5 - Increase in cost of correcting a requirements defect by phase of discovery [MIL98]

1.5.2 Measurable Structures in Object-Oriented Designs

The metrics that will be used in this project should measure principle structures that, if improperly designed, will negatively affect the quality attributes of the design and subsequently the code. This aim of this section is to identify these structures and describe how they may affect the quality of the overall design. However, this section will not present the metrics that operate on these structures. The metrics are presented in section 2.4.

In object-oriented design, there are five key structures that should be measured: *Classes*, *Messages*, *Cohesion*, *Coupling*, and *Inheritance*.

A **class** is a template from which objects can be created. This set of objects shares a common structure and a common behavior manifested by the set of methods. Classes play a major role in the object-oriented paradigm. The way in which classes are designed will affect the overall understandability of a system making it easier or more difficult to maintain. The reusability of a system could also be affected by the way classes are designed. For example, classes with a large number of methods would tend to be more application-specific thus reducing the reuse value of the overall system.

A **message** is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. It is important to study the message flow between objects because this will affect the understandability, maintainability and testability of a system. The more complex the message flows between objects are, the less understandably and maintainable the system is. This will also make the system more difficult to test.

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Effective OO designs maximize cohesion because they promote encapsulation. The degree of cohesion in a system will affect the system's efficiency and reusability. A high degree of cohesion in a system indicates that most classes are self-contained thus increasing the efficiency of the system because fewer messages will be passed between objects. Self-contained classes can easily be plugged in for reuse in another system since they do not depend on other classes to function.

Coupling is a measure of the strength of association established by a connection from one entity to another. Coupling between classes occurs through the use of one class' methods and attributes by another class and through inheritance. Since good object-oriented design requires a balance between coupling and inheritance, inheritance couplings are usually not taken into account when evaluating the degree of coupling in a system. Coupling will affect the efficiency and reusability of a system. Strong coupling complicates a system, thus making it less efficient. A designer should strive to design a system with the weakest possible links between classes. This will also make individual classes within the system more reusable.

Inheritance is a mechanism whereby on object acquires characteristics from one or more other objects. Inheritance can reduce the complexity of a system by reducing the number of methods and attributes in child classes, but this abstraction of objects can make maintenance and design difficult. One can look at the inheritance characteristics of a system from two viewpoints: the *depth* of the inheritance tree and the *breadth* of the inheritance tree. The *deeper* a class is in the inheritance tree, the more methods it is likely to inherit thus making it more difficult to predict its behavior. In a general deeper trees constitute greater design complexity but a balance needs to be struck because the greater the use of inheritance, the greater the reuse of methods and attributes in higher-level classes. Analyzing the *breadth* of the inheritance tree of a project would involve looking at the number of immediate children of particular classes. This is an indicator of the potential influence a class can have on the design and on the system. The more children a class has, the higher the amount of reuse in that design but then again, a large number of children could indicate the improper abstraction of the parent class.

1.5.3 What makes a good object-oriented design?

The question "*what makes a good object-oriented design?*" does not have a clear usable answer. Creating a good object-oriented design involves combining the features of the object-oriented paradigm to create a fine balance between all the quality attributes of interest to the particular designer. Maximizing the value of all quality attributes that are important to a company is virtually impossible since some attributes invariably contradict (or conflict with) each other. For example, a company cannot expect its developers to create a system and make its classes highly reusable without compromising on the efficiency of these classes. This is due to the fact that in most cases, reusable classes are general and thus will not be optimized to specific applications. Management has to be made aware of the numerous **conflicts and contradictions** involved in asking for good object-oriented designs so as not to expect the impossible from their designers and developers.

This project will attempt to provide a tool that will allow project managers to keep track of the quality of their designs by the use of various metrics described in section 2.4.

1.6 Reusability of Object Oriented Designs

The British Computer Society's (BCS) Software Reuse Specialist Group defines the aim of Software Reuse as:

"The planned use of Software Artifacts in the solution to multiple problems."

Many people would think that software reuse involves the reuse of code libraries in new projects. However, code is not the only available artifact produced in the software development process. Requirement specifications, designs and test plans are all artifacts that could potentially be fully or partially reused in different projects. For the purposes of this project, we are only interested in the reuse of object-oriented designs and how this can be measured, however reusable designs often map directly to reusable classes of implemented code.

1.6.1 Benefits of Reusing Designs

There are a number of benefits to be gained from reusing designs:

- Productivity is increased because designers do not re-invent the wheel

- Performance can be increased by having an expert define key blocks
- Correctness is increased because portions of design have been proven in the past
- Predictability is increased because reused modules statistics are known.

1.6.2 What makes designs reusable?

A design is reusable in a new project if it has functional commonality with the new project. Also, a design is more reusable if the user (designer) can customize it.

1.6.3 What can make design reuse unsuccessful?

There are various reasons why design reuse may not be possible or economically feasible. For example, in order for a design to be reused, it must be designed in a flexible manner. In many cases this will result in **lost performance** with respect to both space and time complexities since the implementation of a flexible design will obviously consume more space than that of a design that solves only one particular problem. Similarly, the algorithm will not be optimized towards a particular problem thus increasing time complexity.

Also, designing for reuse takes up more time and effort. Usually, there is a good return on this investment but project **deadlines** and **budgets** sometimes make it impossible to dedicate enough time and resources for a project to be designed with reuse in mind.

1.6.4 Design Reuse and the Object-Oriented Paradigm

The object-oriented paradigm exhibits major differences in the way a system is designed when compared to its procedural equivalent. Do the methods and concepts facilitated by the object-oriented paradigm encourage reuse?

Lewis [LEW91] points out that the most prominent feature offered by object-oriented languages is **encapsulation**. Encapsulation capabilities create self-contained objects, which are easily incorporated into a new design. The data-based decomposition of objects resulting in class-hierarchies and inheritance, promotes reuse far more than the top-down approach. Greater abstraction is the key to a greater reusability, and object based languages provide abstraction far better than procedural languages.

Lewis and his colleagues go on to carry out a detailed empirical study and manage to come up with a few interesting conclusions. The first conclusion is as follows:

"*The object-oriented paradigm substantially improves productivity, although a significant part of this improvement is due to the effect of reuse.*"

This is very interesting. Following this conclusion, one may also state that reuse is an inherent property of the object-oriented paradigm. Hence it makes perfect sense for a company making use of this paradigm to invest in formal training of staff in reuse techniques. According to Lewis' conclusion, the increased productivity gained would make up a significant part of the improvement gained collectively due to the other features offered by the object-oriented paradigm.

The following conclusions were also reached by the same study:

1. Software-reuse improves productivity no matter which language paradigm is used.
2. Language differences are far more important when programmers reuse than when they do not.
3. The Object-Oriented paradigm has a particular affinity to the reuse process.

These conclusions seem to "scream out" that **reuse is good** and that **object-oriented methodologies** are a much better way to go if you want to reuse software artifacts.

1.6.5 Measuring Reuse in Object-Oriented Designs

Not much work has been done in the evaluation of reusability of designs. However, Price and Demurijan [PRI97] came up with an interesting method of measuring the degree of reusability of a design through the analysis of different types of coupling. This method is described in detail in section 1.7 below.

1.7 Measuring the Reusability of Object Oriented Designs

This section describes a method developed by Price and Demurijan [PRI97] for measuring the reusability of object-oriented designs. The method attempts to combine the subjective nature of software design with the objective nature of mathematical metrics.

1.7.1 Overall View

The method provides a set of metrics that work on any object-oriented design irrespective of the domain of the system being designed. There are three steps involved:

1. Allowing the designer to design a system
2. Collecting subjective data from the designer
3. Use objective metrics on subjective data in order to come up with analysis results.

As you can see, the process is not totally automated. It still requires some input (explained below) by the designer. However, in my opinion this is a good point since subjective and domain-specific reasoning is being taken into account. Actually, this feature is what allows this method to be a generic one over all designs in all domains.

1.7.2 Collecting data from the designer

It is safely assumed that the designer is equipped with domain-specific knowledge. Also, it would be very helpful if the designer had an idea of what systems were to be designed in future. There are two types of data that need to be collected from the system designer.

General and Specific Classes

Firstly, all the classes in the system must be categorized as ***general*** or ***specific***. *General* classes are classes that are expected to be reused in future projects. These can either be domain independent classes (such as a GUI component) or domain specific classes that can be reused in other applications in the same domain. An example of the latter might be *patient* class in a hospital system when the designer knows that another system in the medical domain (e.g. Dentist application) may be developed in future.

Specific classes are application-specific classes that are not intended tot be reused in other projects.

Related Class Hierarchies

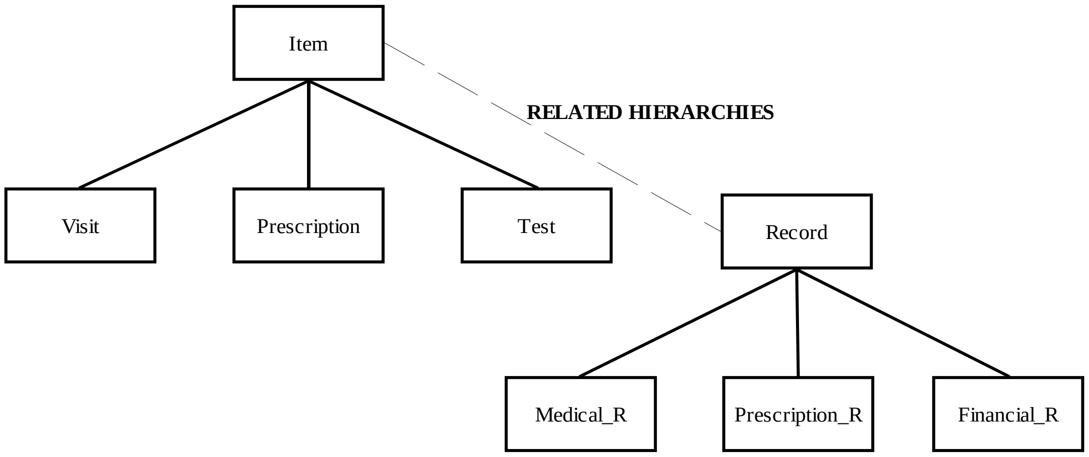
Secondly, the designer also needs to separate his/her classes into hierarchies and specify which hierarchies are related to each other.

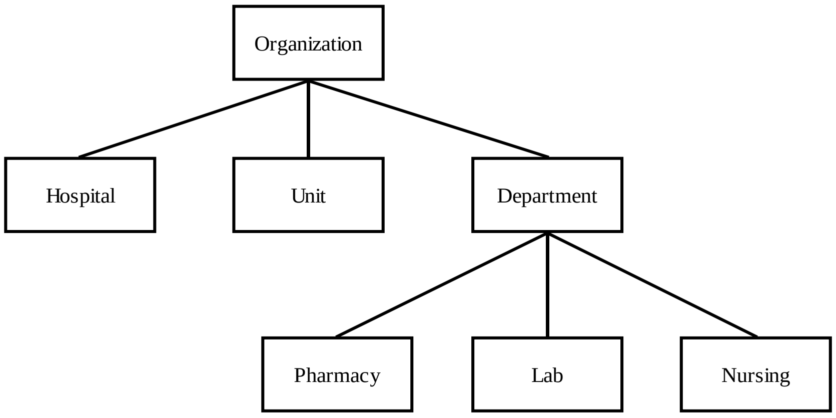
Definition: A class hierarchy is defined as being related to another if they are related in concept and are expected to be reused together in future systems.

As an example, take the three hierarchies presented in the diagram below. These hierarchies form part of a software system for the ministry of health. The designer calculates that there is a strong possibility that the hierarchy **Item** and the hierarchy **Record** will be reused together in future. Maybe his/her company is thinking of developing a system for a small private clinic. This would be a very different to the “ministry of health” system but the mentioned class-hierarchies may still be reused.

Moreover, they probably "have to" be reused together since the **Prescription** class may need to make use of the **Prescription_R** class.

It is worth noting that the hierarchy **Organization** is not marked as been related to any other hierarchy. Reasons for this may and will vary but at the end of the day, to the method, it doesn't make a difference why hierarchies are related or not. The designer is being trusted as the person with the right knowledge to make these decisions. Hence we are capturing subjective data.





1.7.3 Coupling Categories

This method's foundation lies in checking the couplings (dependencies) between different types of classes in different hierarchies. Not all dependencies between classes are bad for reuse. Dependencies between classes that are meant to be reused together are not a hindrance to reuse. In fact, they add more value to a design because a larger portion of the design is reused.

To this extent, it is worth listing the different types of coupling that could occur and define what (if any) effect they have on the reuse value of a design. The creators of the method also make suggestions of what can be done in the case of undesirable coupling. There are eight types of coupling between classes. In the explanations below the letter **G** is read as "General Class" whilst the letter **S** is read as "Specific Class". Also, the notation "S → G" is read "Specific class depending on a General Class".

<u>Type 1: G → G among related hierarchies</u>

A dependency from a general class to another general class in a related hierarchy is not a hindrance to reuse. Actually, increasing these couplings in a design yields a potential for more reuse.

<u>Type 2: G → G among unrelated hierarchies</u>

Although both classes in this coupling are reusable classes, they are not meant to be reused together in future systems. This is an undesirable situation because we cannot use one of the classes in a different system without having to import the other one as well.

It is recommended to move the dependency to their specific descendent classes that are most relevant. Create new classes if necessary.

<u>Type 3: G → S among related hierarchies</u>

In this case, the class that may be reused in future (the general class) depends on a specific class. The specific class is not meant to be reused in future so this is an undesirable form of coupling.

The designer should attempt to move the destination to an appropriate general ancestor of the specific class.

Type 4: G → S among unrelated hierarchies

This is very similar to a type 3 coupling only it is more undesirable because the dependency is between unrelated hierarchies.

The designer is advised to try to move the source of the coupling to an appropriate specific descendent class.

Type 5: S → G among related hierarchies

This form of coupling does not impede reuse because the class depending on another class is a specific one. However, we might be able to increase the value of reuse by moving the source to an appropriate general ancestor. This would convert the coupling into a type-1 coupling.

Type 6: S → G among unrelated hierarchies

A type-6 coupling has absolutely no effect on the value of reuse of a design. There are also no transformations that can be done in order to maybe increase the value of reuse of the design.

Type 7: S → S among related hierarchies

Although this form of coupling is no hindrance to reuse, the designer could both the source and destination of the coupling to general ancestors. If this is possible, it would convert the coupling into a type-1 coupling – the most desirable form of coupling.

Type 8: S → S among unrelated hierarchies

Not only is this situation not a hindrance to reuse but it is also the ideal form of coupling between unrelated classes. If there must be couplings between unrelated hierarchies, we should push the designer to create these couplings between specific classes.

Summary of Coupling Categories

The following table summarizes the information given above.

#	Effect on Reuse Value	Recommended Action / Comments
1	Neutral / Positive	Increasing these couplings in a design yields a potential for more reuse.
2	Negative	Attempt to move the source and destination of the dependency to more specific descendent classes that are most relevant. Create new classes if necessary.
3	Negative	Attempt to move the destination of the coupling to an appropriate ancestor class.
4	Negative	Attempt to move source of coupling to an appropriate specific descendant class.
5	Neutral	Attempting to move the source to an appropriate general ancestor will convert the coupling to a type-1 coupling.
6	Neutral	
7	Neutral	Attempting to move both the source and destination to General ancestors will convert the coupling into a type-1 coupling.
8	Neutral	Ideal situation for coupling between unrelated hierarchies.

1.7.4 Design Reusability Metrics

The creators of the method propose a set of metrics, which take the form of eight summations that correspond to the eight types of coupling given above. Coupling is defined as an inter-hierarchy dependency that results when a method of one hierarchy uses methods or instance variables of another hierarchy. We use the term Coupling Counts (CC) to represent these interactions between hierarchies. The metrics are defined as follows:

$$CC_1 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_jG_i \qquad CC_5 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_jG_i$$

$$CC_2 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_jG_i \qquad CC_6 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_jG_i$$

$$CC_3 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_jS_i \qquad CC_7 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_jS_i$$

$$CC_4 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_jS_i \qquad CC_8 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_jS_i$$

where

m: # of hierarchies which are related to this one
n: # of hierarchies which are not related to this one
x: # of General classes in this hierarchy
y: # of Specific classes in this hierarchy

G_jG_i: # of couplings from the j-th General class to all General classes in the ith hierarchy

G_jS_i: # of couplings from the j-th General class to all Specific classes in the ith hierarchy

S_jG_i: # of couplings from the j-th Specific class to all General classes in the ith hierarchy

S_jS_i: # of couplings from the j-th Specific class to all Specific classes in the ith hierarchy

1.8 What to expect in the next chapter…

This chapter has hopefully instilled in the reader the motivation for measuring software quality and has also introduced the main topics needed to equip him/her with enough theoretical background to understand the rest of this document. So far however, the information has been of a very general and abstract nature. In the next chapter, the content will have the narrower focus of applying the general information and theory presented in this chapter to the more specific task at hand: creating a software quality measurement system. Chapter 2 is a chapter where decisions are made with regards to quality attributes that are going to be evaluated, metrics that are going to be calculated, and how information for these metrics can be extracted from UML. In some cases, UML will be found not to provide enough information for the scope of this project and lightweight extensions will have to be defined. Also, the concept of function points is introduced and their use in this project explained.

2. Methodology

2.1 Introduction

The end-goal of this project is to design and implement a tool that analyses the quality of object-oriented designs. However, decisions need to be taken: *What are the limits of scope of the tool? How will designs be represented? Will the design-notation language need to be extended? What metrics are to be used to analyze the designs and what quality attributes do they measure? How will the information for these metrics be extracted from the chosen design notation? Will the tool make an emphasis on any particular quality attributes? How will the tool allow the user to compare the current project with previously completed projects?* The purpose of this section is to answer these questions and lay the foundations for the specification, design and implementation of the tool itself.

2.2 Limits of Scope of Tool

The final implemented tool will be able to analyze an object-oriented design by extracting the required data to calculate a number of object-oriented metrics. The tool will provide the user with information about each metric such as the quality attributes that it evaluates and how to interpret its readings. Graphical reports should also be generated so as to give the user a clear overall view of his/her project. The tool should also provide a way of comparing the current project with other projects, which have been previously analyzed by the same tool.

2.3 Representation of Object-Oriented Designs

The first decision that must be taken before going any further concerns the notation to be used to represent designs. It was decided that the designer is to communicate his/her design to the system using the Unified Modeling Language (UML). The Unified

Modeling Language is a language that unifies the industry’s best engineering practices for modeling systems [ALH98] and allows the designer to specify a system by making use of nine different types of diagrams. The tool will only analyze a subset of these diagrams depending on the metrics that are going to evaluate the system. A description of these diagrams will given in the methodology section after the metrics have been introduced.

2.3.1 Why UML?

There are various reasons why UML was chosen as the notation for design. Firstly, UML is inherently related to the object-oriented paradigm and thus fits in with the goals of this project. Secondly, UML is considered to be a notational approach that does not define how to organize development tasks. Therefore, it can be tailored to different development situations thus making the tool usable by any development team making use of the object-oriented paradigm.

Usage of UML is very widespread and is expected to keep growing. This means benefits this project in various ways. Firstly, there is a wealth of existing designs, which can be feed into the system for analysis consequently making it easy to build a repository of projects for comparison with future projects. Secondly, any users of the system who need to be trained in UML will probably see it as a plus due to the fact that they can make use of their newly acquired knowledge in other places of work.

UML is also considered to be able to completely describe a system’s structural view, behavioral view, implementation view, environment view and user view. Therefore it does not limit the designer from expressing the system’s design in any way. The tool will of course not be analyzing all the diagrams provided by UML but a select view as describe in the methodology section.

Finally, UML is an extensible language, a feature that comes in useful in this project since some of the reuse metrics being used will need certain parameters to be included in the design. The extension of UML in this project has been kept to a bare minimum so as to preserve the standard form of UML thus enabling the analysis of previously designed systems.

2.4 Choice of Metrics

The tool will be designed to make it easy for new metrics to be added to it if its user requires that different aspects of the system be measured. However, a base set of fourteen metrics has been chosen for initial implementation for the system and will be separated into two groups: *Structural Metrics* and *Reuse Metrics*.

The *Structural Metrics* group contains six metrics, which between them evaluate all the principle structures discussed in section 1.5.2 (Measurable Structures in Object Oriented Designs).

The *Reuse Metrics* are coupling-based metrics that are specifically designed to measure reuse. The reasoning behind them is explained in section 2.4.2 below.

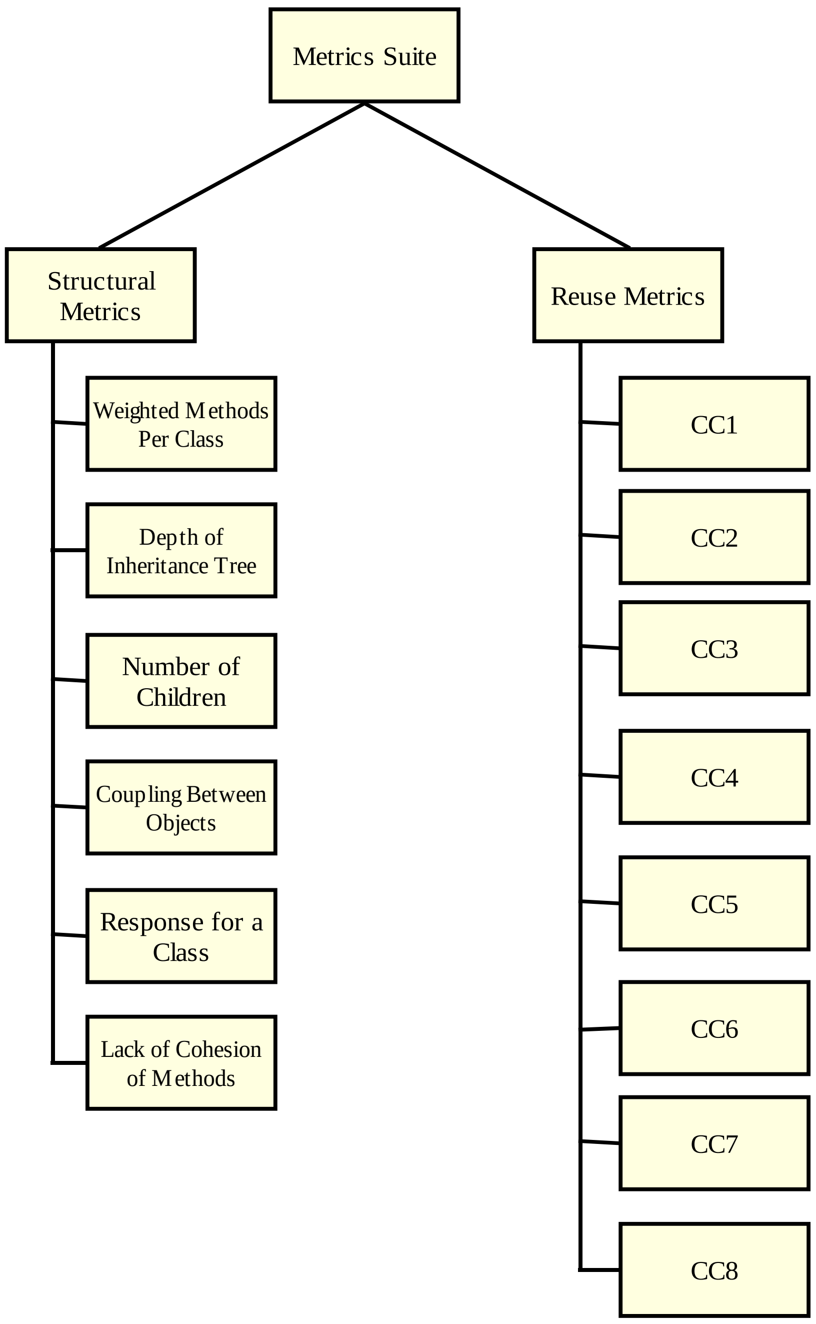


Figure 6 - The metrics to be evaluated by the system (descriptions below)

2.4.1 The *Structural Metrics* Group

The following metrics measure the principle structures offered by an object-oriented design, namely *class, message, coupling, cohesion,* and *inheritance*. Chidamber and Kemerer [CHI94] from MIT, proposed these metrics back in 1994.

Weighted Methods Per Class (WMC)

Consider a class $\boldsymbol{C_1}$, with methods $\boldsymbol{m_1, \dots m_n}$. Let $\boldsymbol{c_1, \dots c_n}$ be the static complexity of the methods. Then…

$$WMC = \sum_{i=1}^{n} c_i$$

Chidamber and Kemerer did not define how the static complexity of each method is to be calculated. For the purposes of this project, the static complexity (C_i) will be calculated using *McCabe's Cyclomatic Complexity* measure. This is simply a count of test cases that are needed to test the method comprehensively.

WMC analyzes the *class* structure and the result has a bearing on the *understandability*, *maintainability*, and *reusability* of the system as a whole. The number of methods and the complexity involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children, since children inherit all of the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Churcher and Shepperd [CHU95] point out that the definition of the method count is imprecise because it does not say whether or not to count inherited methods as belonging to a class or not. Different interpretations of this could change the measurement drastically. Chidamber and Kemerer clarified their position by saying that "*the methods* *that require additional design effort and are defined in the class should be counted, and those that do not, should not*".

Depth of Inheritance Tree (DIT)

The depth of inheritance of a class is its depth in the inheritance tree. If multiple inheritance is involved, then the depth of the class is the length of the maximum path from the node representing the class to the root of the tree. The root class has a DIT of 0.

DIT is essentially a measure of how many ancestor classes can possibly affect this class. It is worth noting that deeper trees constitute greater design complexity, since more methods and classes are involved. However, deeper trees also signify a greater level of internal reuse in the system so a balance between reuse and reduced complexity needs to be struck.

This metric primarily evaluates efficiency and reuse but also relates to understandability and testability.

Number of Children (NOC)

NOC simply counts the number of immediate sub-classes subordinate to a particular class in the class hierarchy. This gives an indication of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper parent abstraction, and it may be an indication of sub-classing misuse. Again, there has to be a compromise because a greater number of children indicate a larger degree of internal reuse of the particular class. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. NOC, therefore, primarily evaluates efficiency, reusability, and testability.

Coupling Between Objects (CBO)

CBO for a class is a count of the number of non-inheritance related couples with other classes. Excessive coupling between objects outside of the inheritance hierarchy is detrimental to modular design and prevents reuse since the more independent an object is, the easier it is to reuse in a different application. Also, the larger the number of couples a class has, the more sensitive it is to changes in other parts of the design thus making maintenance more difficult.

CBO has been criticized by Hitz and Montazeri [HIT96] because it assumes that all couples are of equal strength. They claim this does not make it a sensitive enough measure because (for example) an object using another object's attributes constitutes a stronger coupling than pure message passing between objects as does message passing with a wide parameter interface vs. one with a slim interface.

Response For a Class (RFC)

RFC = $|RS|$ where RS is the response set for the class.

$RS = \{M_i\} \cup \{R_i\}$

Where: $\{M_i\}$ is the set of methods in the class

And: $\{R_i\}$ is the set of methods called by methods in $\{M_i\}$

Basically, RFC is a count of the methods in a particular class and the methods in other classes that are called by the class. This gives an indication of a system's *understandability*, *maintainability*, and *testability*. If a large number of methods can be invoked in response to a message, the testing and debugging of the object becomes more complicated. Also, the larger the number of methods invoked by an object, the more complex it is thus decreasing understandability and testability. It is worth noting that worst-case values for possible responses will assist in the appropriate allocation of testing time.

Lack of Cohesion of Methods (LCOM)

Consider a class C_1 with methods M_1, M_2, … , M_n. Let { I_i } = set of instance variables used by method M_i. There are n such sets: { I_1 }, … { I_n }.

LCOM = The number of disjoint sets formed by the intersection of the n sets.

LCOM uses the notion of similarity of methods. The number of disjoint sets provides a measure for the disparate nature of methods in a class. Fewer disjoint sets imply greater similarity of methods whilst higher number of disjoint sets indicate that the methods in the class are not cohesive and that the class can probably be split into two or more subclasses. Cohesiveness of methods within a class is desirable since it promotes encapsulation of objects.

Henderson-Sellers [HEN96] evaluated the LCOM metric and criticized it due to the fact that while large values of LCOM suggest poor cohesion, a zero value does not necessarily indicate good cohesion.

LCOM evaluates efficiency and reusability.

2.4.2 The *Reuse Metrics* Group

The theory behind the reuse metrics has already been explained in section 1.7 but the metrics are reproduced here for the sake of completion:

$$CC_1 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_jG_i$$

$$CC_2 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_jG_i$$

$$CC_3 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_jS_i$$

$$CC_4 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_jS_i$$

$$CC_5 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_jG_i$$

$$CC_6 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_jG_i$$

$$CC_7 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_jS_i$$

$$CC_8 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_jS_i$$

where

m: # of hierarchies which are related to this one
n: # of hierarchies which are not related to this one
x: # of General classes in this hierarchy
y: # of Specific classes in this hierarchy

G_jG_i: # of couplings from the j-th General class to all General classes in the ith hierarchy

G_jS_i: # of couplings from the j-th General class to all Specific classes in the ith hierarchy

S_jG_i: # of couplings from the j-th Specific class to all General classes in the ith hierarchy

S_jS_i: # of couplings from the j-th Specific class to all Specific classes in the ith hierarchy

2.5 Extracting Structural Metric Information from UML Diagrams

This section goes through the *structural* metrics one by one and describes how the information needed to calculate each metric could be extracted from UML notation. It has been established that all the information required can be obtained from a combination of class diagrams, activity diagrams, collaboration diagrams and sequence diagrams. It is assumed that the reader has a basic knowledge of UML but specific details about diagrams will be discussed where appropriate so as to make the solution to certain issues as clear as possible.

2.5.1 Weighted Methods per Class (WMC)

Consider a class $\boldsymbol{C_1}$, with methods $\boldsymbol{m_1, \dots m_n}$. Let $\boldsymbol{c_1, \dots c_n}$ be the cyclomatic complexity of the methods. Then…

$$WMC = \sum_{i=1}^{n} c_i$$

Calculating WMC will require the information from two types UML diagrams:

1. **Class Diagram**

   The class diagram will be used for obtaining a list of methods for each class. By default, the cyclomatic complexity of each method will be one. However, if there are methods for which there exists an *Activity Diagram* describing changes in activity within the methods, the cyclomatic complexity for those methods should be calculated from their *Activity Diagrams*.

   Please note that inherited methods are not counted unless they are re-defined in the current class.

2. **Activity Diagram**

   Activity diagrams can be used to show the changes in activity within the methods. They are very similar to flowcharts. If a method has an activity diagram associated with it, its cyclomatic complexity is calculated as follows:

   Cyclomatic Complexity = no. of edges – no. of nodes + 2

   If a method does not have an activity diagram associated with it, then its cyclomatic complexity is taken to be **1**. This follows from the notion that in theory, object-oriented methods are so small and specific that their complexity can be taken to be 1.

2.5.2 Depth of Inheritance Tree (DIT)

The depth of inheritance of a class is its depth in the inheritance tree. If multiple inheritance is involved, then the depth of the class is the length of the maximum path from the node representing the class to the root of the tree. The root class has a DIT of 0.

As shown in the example below, the DIT metric is easily measured by looking at a particular class in a **class diagram**. The class **Animal** is the root class of the hierarchy shown in the example and therefore has a DIT of 0. The classes below it (**DomesticAnimal**, **FarmAnimal**, **WildAnimal**) have a DIT of 1 and their children in turn have a DIT of 2.

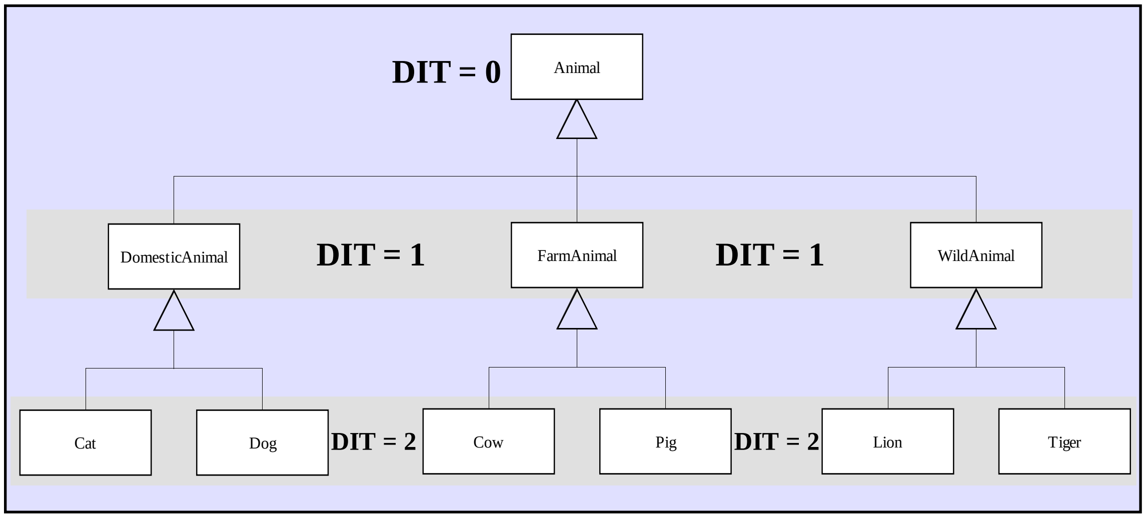


Figure 7 - Illustrating how DIT readings can be made from UML Class Diagrams

2.5.3 Number of Children

NOC simply counts the number of immediate sub-classes subordinate to a particular class in the class hierarchy. This information is easily obtainable from a **class diagram** as shown below.

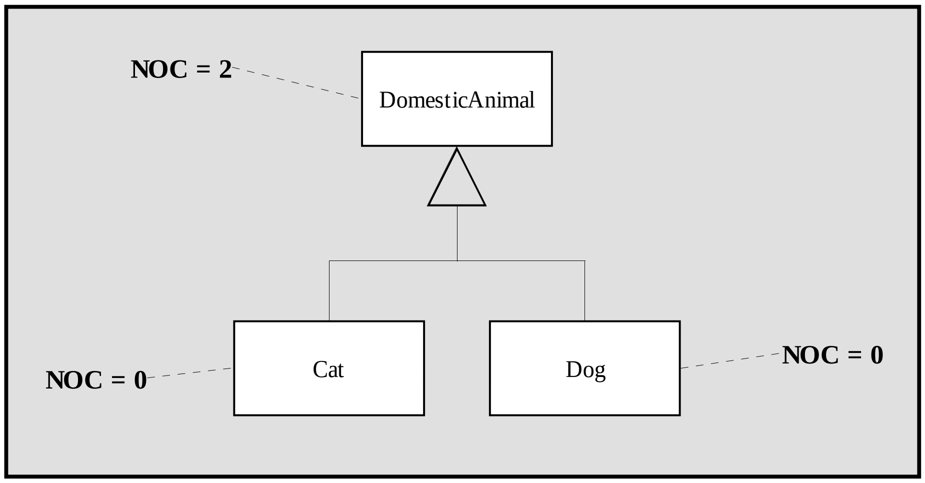


Figure 8 - Illustrating how NOC values can be obtained from UML Class Diagrams

2.5.4 Coupling between Objects (CBO)

CBO for a class is a count of the number of non-inheritance related couples with other classes.

UML **class diagrams** will be needed to obtain information for calculating CBO. Class diagrams can show the different couplings between objects. Before describing how the CBO metric will be calculated from a class diagram, it is worth looking at the different types of couples that can be illustrated within a UML class diagram. These are as follows:

Associations

Associations represent relationships between two or more classes. Associations can either be unidirectional or bi-directional.

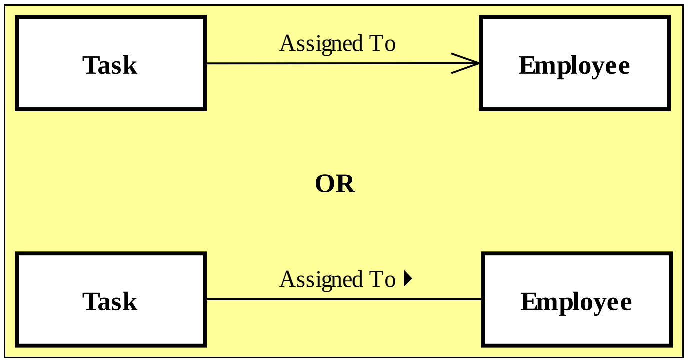


Figure 9- Unidirectional Associations

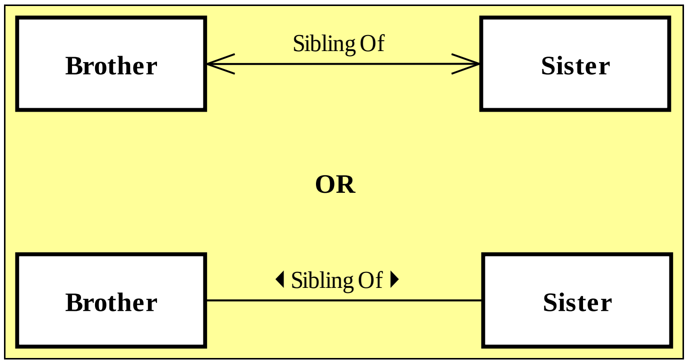


Figure 10- Bi-directional Associations

Compositions

Compositions are used to indicate situations where a class contains one or more other classes. In the example below, a Project contains 1 or more Activity objects each of which contain 1 or more Task objects. Please note that Activity objects cannot exist without being associated to a Project object. Therefore if a Project object is removed, the Activity objects associated with it will also have to be removed. If activities could exist without being associated with a project, or a project could be removed without having to remove its activities, the relationship will no longer be a composition – it will become an aggregation (see below).

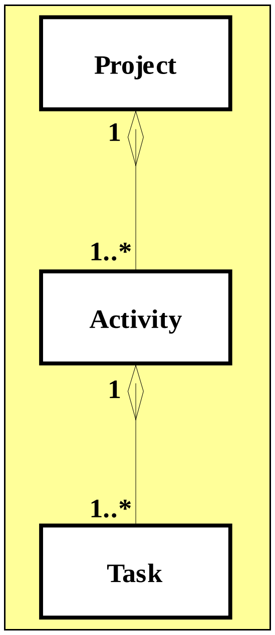


Figure 11- Illustration of compositions in use

Aggregations

Aggregations are very similar to compositions. They differ in the way that the classes being contained are also able of exist on their own. So in the example below, a research team can be made up of one or more researches, however because the relationship is an aggregation (hollow diamond), a researcher can exist without being part of a research team. Also, if a research team ceases to exist, its researchers can remain in existence on their own.

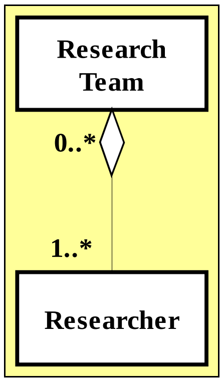


Figure 12- Aggregations - a researcher need NOT be part of a research team.

Generalizations

Generalizations are used to illustrate inheritance. They are not used in the calculation of CBO but they will be used in other metrics. The example below shows a hierarchy where the general class is *Animal* and there are two more specific classes which inherit the *Animal* class' attributes and methods called *Cat* and *Dog*.

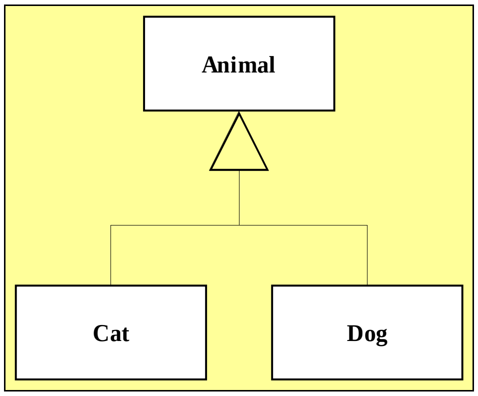


Figure 13- Generalizations

CBO for a class will be calculated by counting all the relationships that a class participates in except generalizations.

2.5.5 Response for a Class (RFC)

RFC = |RS| where RS is the response set for the class.

And $\mathbf{RS = \{ M_i \} \cup \{ R_i \}}$

Where $\mathbf{M_i}$ **= all methods in the calss**

And $\mathbf{\{ R_i \}}$ **= set of methods called by** $\mathbf{M_i}$

Basically, we need to extract $\mathbf{M_i}$ and $\mathbf{R_i}$ from the design. $\mathbf{M_i}$ is easily extracted for each class from a class diagram. However, extracting $\mathbf{R_i}$ will prove to be a slightly more complicated process. The only UML diagrams that show what methods a class calls (besides its own) are **sequence diagrams** and **collaboration diagrams**. $\mathbf{R_i}$ will be built by examining all the sequence and collaboration diagrams in a project and keeping track of what external methods are being called by a particular class. Since $\mathbf{R_i}$ is a set, encountering a method more than once will not affect our calculations.

2.5.6 Lack of Cohesion of Methods

Consider a class $\mathbf{C_1}$ with methods $\mathbf{M_1, M_2, \ldots, M_n}$. Let **$\{ I_i \}$ = set of instance variables used by method M_i**. There are **n** such sets: **$\{ I_1 \}, \ldots \{ I_n \}$**.

LCOM = The number of disjoint sets formed by the intersection of the n sets.

The method for extracting information for this metric involves collecting additional information from the designer. This will be done by asking the designer to indicate which instance variables are to be used by each method. From this information, $\{I_i\}$ can be built and the metric can be calculated.

2.6 Collecting Information for the Price-Demurijan Metrics

The reuse metrics devised by Price and Demurijan described in section 1.7 are being treated separately from the structural metrics when it comes to extracting information UML diagrams. This decision was taken because UML does not provide all the information need for these metrics and needs to be slightly extended.

2.6.1 Overview

The Price and Demurijian metrics were designed to evaluate the reusability of a design. The calculation of these metrics requires that the system knows the following:

1. Which classes are General and which are specific
2. Which class hierarchies are meant to be reused together in future
3. Couplings between classes

The first two items in the list are specific to this method and are therefore not catered for in UML notation. We will have to define a simple extension to UML in order to be able to capture the required information in a design.

2.6.2 General and Specific Classes

The concept of General and Specific classes is not currently supported in UML. Therefore, for the purposes of this project, we will extend the UML to cater for this concept.

As a default, all classes will be assumed to be specific. The designer should depict a general class by stereotyping it as **<<General>>**.

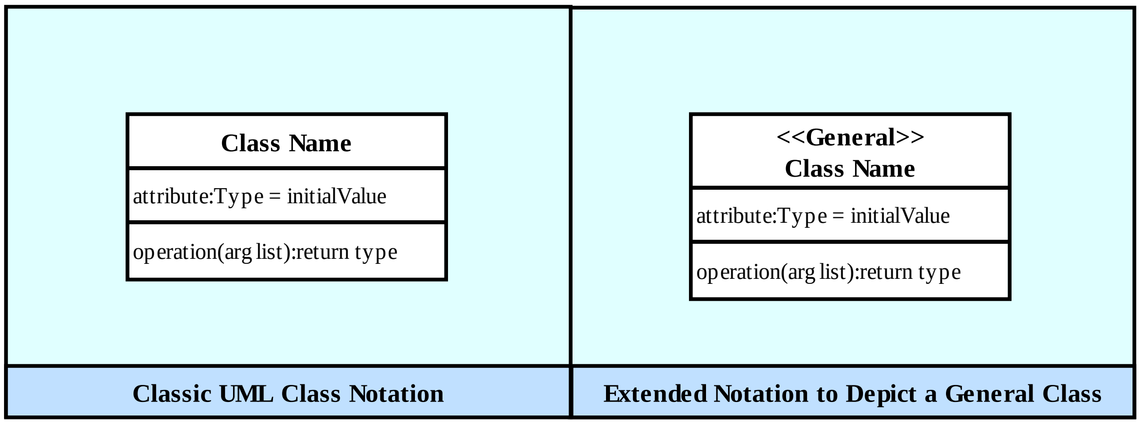

Class Name
attribute:Type = initialValue
operation(arg list):return type
<<General>>
Class Name
attribute:Type = initialValue
operation(arg list):return type
Classic UML Class Notation
Extended Notation to Depict a General Class

Figure 14 - Illustrating how to classify a class as being General

2.6.3 Related Class Hierarchies

A class hierarchy is defined as being related to another if they are related in concept and are expected to be reused together in future systems.

We will extend UML class-diagram notation to show related class-hierarchies by adding a new stereotype that can be used with an association. The new stereotype will be **<<Reuse-Related>>**. This does not indicate a dependency or a coupling between classes. It merely indicates that the classes are related in concept and that there is a good possibility of them being reused together in future. Therefore it follows that an association with this stereotype must be bi-directional and the classes on each end of the association must be **<<General>>** classes. Also, a class cannot be reused without its ancestors being reused with it. Therefore this type of association must only exist between root classes of hierarchies.

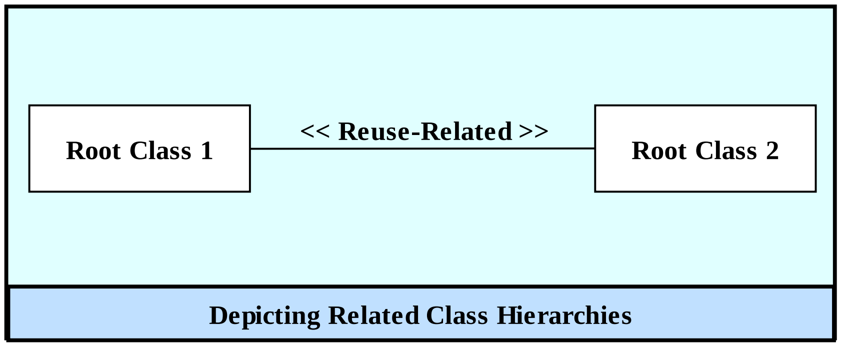


Figure 15 - Illustrating the use of the <<Reuse-Related>> Stereotype

The following is a slightly more elaborated and specific example extracted from the design of a hospital system. Please note that just because the root class of a hierarchy is general, it does not mean that its children need be general too. However, it does follow that if a class is specific, then all its descendants will be specific.

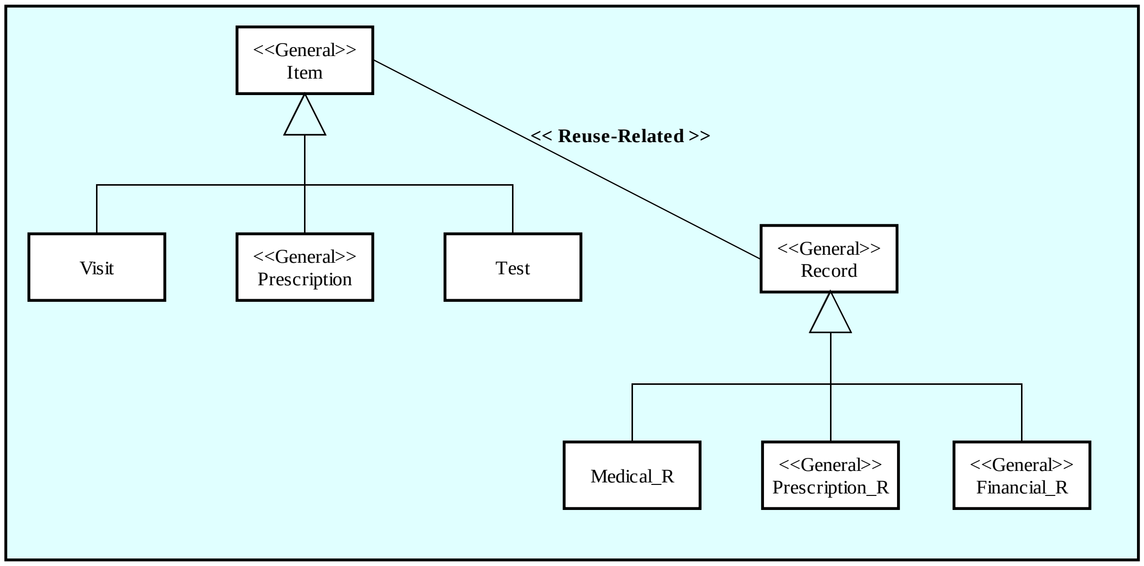


Figure 16 - An elaborated example illustrating the use of the UML extensions defined in this section.

2.6.4 Extracting Metric Information

If the designer makes use of the above extensions, it will be quite straightforward to extract the required information from a design for metric calculation. Remember that the metrics basically consist of counting the number of interactions (couplings) between classes of different hierarchies. We then analyze the couplings with regards to the types of classes they relate and decide on whether the coupling has a favorable, neutral or negative impact on the reusability of the design.

Couplings are already catered for in UML by aggregations, dependencies and compositions. We can use these relationships in class diagrams to determine the amount of couplings. We then make use of the extensions defined above to categorize these couplings.

2.7 Use of Function Points for Comparing Different Projects

Metrics are seldom useful in isolation. The final implemented tool will provide the functionality of comparing different projects together but there is a decision that needs to be taken first: *How will the user know which projects are of the same (or similar) size as* *the current project?* This is important because a metric reading has to be taken in the context of the size of the current project.

Traditionally, project size has been estimated using the lines-of-code (LOC) metric. However this metric is highly unreliable because the same application implemented in different languages or even by different programmers will give different LOC readings. Also, we are analyzing projects before they have even been implemented so the LOC metric does not make sense.

As an alternative, project size will be quantified using function points. Function points were proposed by Albrecht in 1979 and measure the size of a system depending on the functions it offers. The intended use of function points was for the extraction of productivity statistics. For example, the average number of man-hours per function point for developing a system or the cost per function point. This makes it easier for managers to estimate the size and cost of a project after the specification stage and thus be able to calculate a charge for the client based on statistics that are more reliable than LOC. In this project however, function points will be used to allow the user to compare different projects with similar function points. The tool will provide the functionality of asking the user questions about the functions to be provided by the system and using Albrecht's method to calculate the function points of the project.

2.8 What to expect in the next chapter…

Now that the theory has been introduced and the methodology defined, it is time to concentrate on the specification of the system that is to be implemented. The next chapter attempts to unambiguously specify the functionality that is to be provided by the system.

3. Specification

3.1 Overview

A tool is required for analyzing the quality of object-oriented designs. The user will be provide the tool with a system design in the format of UML diagrams and the tool will in turn derive metric-measurement from that design and present the user with the results. Information on each metric and the quality attributes it evaluates are to be presented to the user and graphical interpretation of results is to be implemented in order to facilitate easier metric interpretation. A metrics repository should also be implemented and the user should be given the facility to compare metric readings from different projects.

3.2 The ArgoUML Cognitive CASE Tool

It does not make any sense to re-invent the wheel and create a UML editor from scratch. Such a task would go beyond the scope of this project and will probably take up all the available time itself. Instead, it was decided that an existing UML editor – ArgoUML (www.argouml.org).

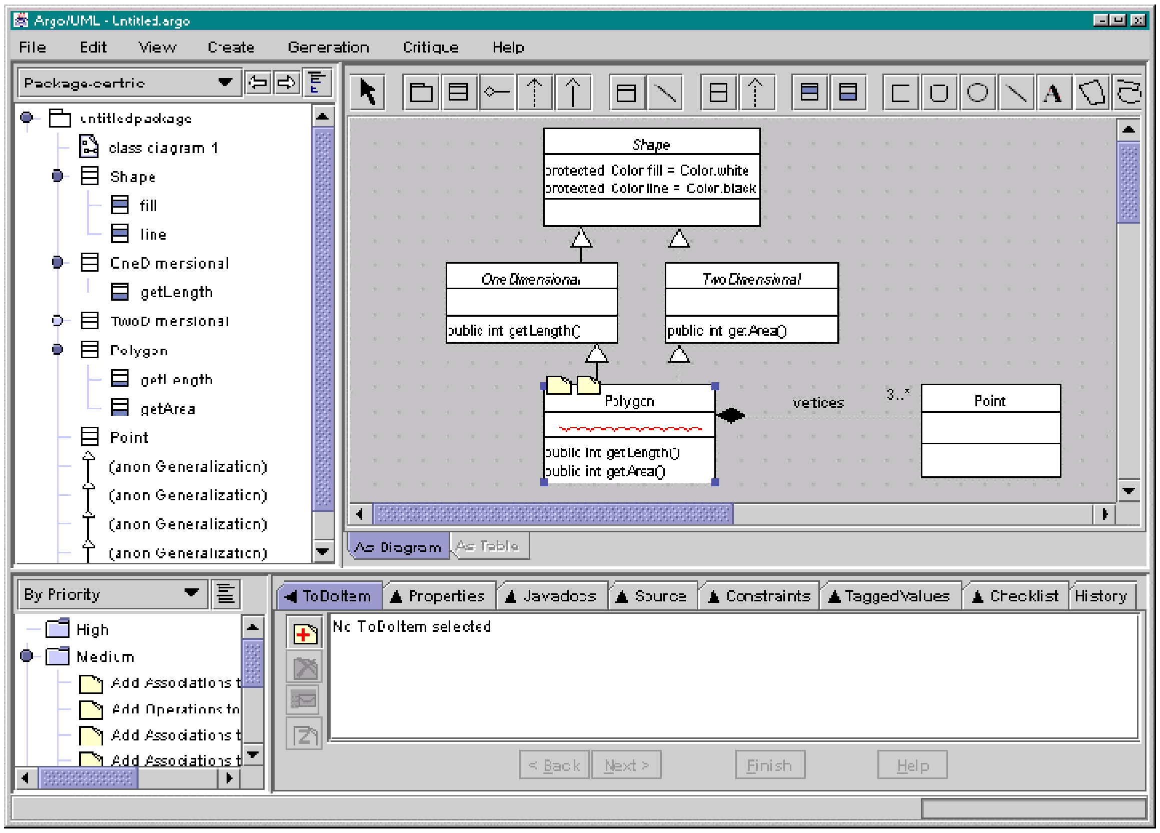
ArgoUML is the brainchild of Dr. Jason Robbins. He originally started working on the project as part of his Ph.D. on providing cognitive support to software designers. The project has since been released as an open-source project and at last count on June 2000, there were 38,000 downloads. I have participated in the developer's mailing list for more than nine months and the project is still very active with new features being discussed and added regularly.

The project makes use of a number of other open-source projects in order to achieve it's goals. The two most influential ones are:

- Graph Editing Framework (GEF) - a graph editing library that can be used to construct many, high quality graph editing applications. (http://gef.tigris.org)

- Novosoft UML API (NSUML) – a representation of the UML meta-model by java classes. (http://nsuml.sourceforge.net)

ArgoUML supports editing of all nine UML diagrams and thus provides this project with a very solid base to which quality-measurement capabilities can be added. Since the software is continuously under development, problems are expected to crop up but I am confident that other people participating in the project will provide all the necessary help.



No ToDoItem selected

Figure 17 - A screenshot of ArgoUML

3.3 Use-Case Analysis

The following use-case diagram gives and graphical overview of the functionality that is to be offered by the tool.

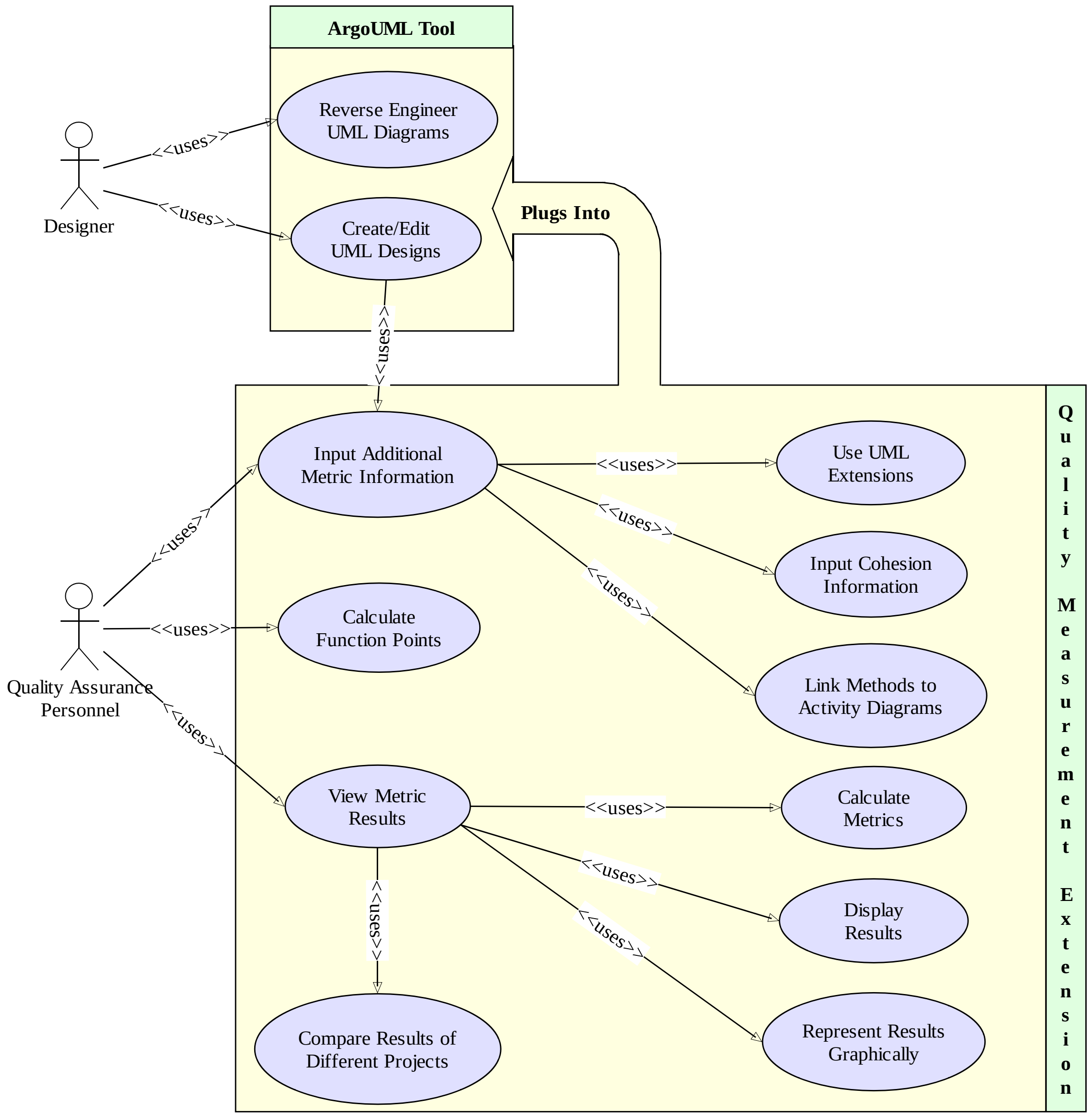

ArgoUML Tool
Designer
<<uses>>
Reverse Engineer UML Diagrams
<<uses>>
Create/Edit UML Designs
Plugs Into
<<uses>>
Quality Assurance Personnel
Input Additional Metric Information
Use UML Extensions
Input Cohesion Information
Link Methods to Activity Diagrams
Calculate Function Points
View Metric Results
Calculate Metrics
Display Results
Represent Results Graphically
Compare Results of Different Projects
Quality Measurement Extension

Figure 18 - A use-case analysis of the quality measurement tool

3.4 Overview of Required Extensions to ArgoUML

This section gives a high-level view of what extensions are required to be added to ArgoUML. Each of these extensions will be discussed in more detail below.

Firstly, since ArgoUML must be modified to collect information required for metric calculation but is not extractable from UML. More specifically ArgoUML needs to be extended to cater for:

1. UML extensions defined in section 2.6

2. Gathering information for linking methods to activity diagrams. This is needed for the calculation of the *Weighted Methods per Class (WMC)* metric.

3. Gathering information for instance-variable usage for each method in a class. This information will be utilized for evaluating the *Lack of Cohesion of Methods (LCOM)* metric.

4. Calculation of function points for each project. This feature will be used at the user's discretion in order to be able to compare different projects together based on their function point readings.

5. Metric calculation facilities.

6. User interface for presenting metric results in both text and graphical formats.

The following sections will describe each of the above in more detail.

3.5 Extension 1: UML extensions

ArgoUML should be extended to allow the user to make use of the UML extensions defined in section 2.6. These UML extensions consist solely of new stereo-types:

- **<General>** - Used to indicate that a class is a general class in the sense that it is expected to be reused in future systems.

- **<Reuse-Related>** - Used with associations to indicate that two general classes (and the hierarchies they define) are meant to be reuse together in future systems.

3.6 Extension 2: Gathering Information for WMC

By default the WMC method takes the complexity of each method in a class to be one. However, there will be cases where a method has had its behavior defined via one or more *Activity Diagrams*. There is no notation for linking an *Activity Diagram* to a method in UML and this information needs to be gathered from the user.

3.7 Extension 3: Gathering Information for LCOM

LCOM evaluates the level of cohesion of a class by determining the degree of similarity in the sets of instance variables used by the methods of the class. Again, UML does not provide a notation for showing which instance variables a method makes use of. Therefore, ArgoUML is to be extended to provide functionality for the user to input this information. This has to be done in as much a user-friendly way as possible since providing this information is compulsory if the designer wants LCOM readings. Keep in mind the expected amount of classes and methods in a system.

3.8 Extension 4: Function-Points Calculation

As described in section 2.7, the user will be given the functionality of calculating the function points of each project. This will help the user to filter out which projects are comparable with each other when it comes to metric interpretation. The system is to allow the user to list the functions to be offered by that system and fill in the tables as described in the original function points calculation method. The user should also be able to answer the fourteen *general system characteristics* question thus enabling the system to have enough information to calculate the function points for the particular project.

3.9 Extension 5: Metrics Calculation

This feature has to integrate seamlessly with ArgoUML in the sense that it will be able to read the internal representation of the current design project and extract enough information for the calculation of metrics. Some metrics (WMC and LCOM) need more information than is extractable from UML diagrams so metrics calculation will have to make use of extensions 2 (Gathering information for WMC) and 3 (Gathering Information for LCOM).

The system should provide a reusable framework for metric calculation in the sense that should new metrics be needed in future, it should be sufficiently easy to create new metrics and plug them into the quality measurement system.

The following information needs to be tracked by the system for each metric:

- Number of times calculated in this project
- Average value over this project
- The highest value over this project
- The lowest value over this project
- All results for all entities in this project that the metric was calculated on

Initially, the system is to have calculation of the following metrics implemented:

1. Weighted Methods per Class (WMC)
2. Depth of Inheritance Tree (DIT)
3. Number of Children (NOC)
4. Coupling between Objects (CBO)
5. Response for a Class (RFC)
6. Lack of Cohesion of Methods (LCOM)
7. Coupling-Count 1 (CC1)
8. Coupling-Count 2 (CC2)
9. Coupling-Count 3 (CC3)
10. Coupling-Count 4 (CC4)
11. Coupling-Count 5 (CC5)
12. Coupling-Count 6 (CC6)
13. Coupling-Count 7 (CC7)
14. Coupling-Count 8 (CC8)

The exact definitions and implications for these methods are discussed in section 2.4

3.10 Extension 6: User Interface for Representation of Metrics

At the end of the day, the ultimate goal of improving the quality of a particular design can only be achieved if the system can provide the user with an intuitive and friendly way of viewing and interpreting metric results. This section explains in detail what information is to be presented to the user and in what formats.

3.10.1 Information to be presented

The interface is to display the following information:

For each **metric**, the system is to display:

1. The name of the metric
2. A description of the metric
3. A list of quality attributes that the metric evaluates
4. The number of entities in the current project that the metric has evaluated
5. The average value of the metric over the current project
6. The highest value of the metric over the current project
7. The lowest value of the metric over the current project
8. A detailed list of entities that where evaluated by the metric, what diagrams they appear in and the result of the metric for each entity.
9. Graphical representations of the design

For each **quality attribute**, the system is to describe how different values of the metric indicate the degree of presence of that quality attribute in the design.

The system will also provide a way for comparing the values of each metric in the current project with metrics of other projects. To this extent, a *metrics repository* needs to be maintained. It should be organized by metric and each metric will have information about all the projects it was used in.

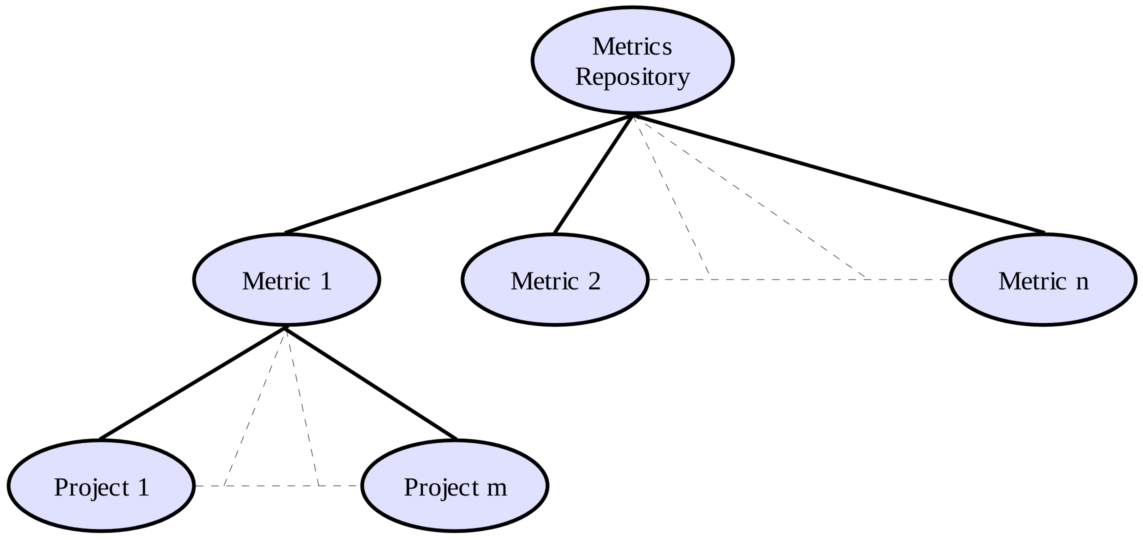


Figure 19 - The recommended structure of the metrics repository

As shown in figure 19, the *metrics repository* will hold information about two entities: *metrics* and *projects*. For each metric, the repository is to contain the following information:

- The name of the metric
- The average value of the metric over all projects in the repository
- The number of times the metric was used throughout the projects in the repository
- The minimum value of the metric over all the projects in the repository
- The maximum value of the metric over all the projects in the repository

Also, for each project in which the metric was used, the repository should hold:

- The name of the project
- The function points of the project
- A timestamp of the last time the repository was updated with this project
- The number of times the relevant metric was used in this project
- The average value of the metric over this project
- The minimum value of the metric in this project
- The maximum value of the metric in this project

3.10.2 Organization of Information

All the information described above is to be presented to the user in a hierarchical representation that can take one of two views:

1. **Metric-Central View** – Hierarchical representation is centered on metrics grouped into the two groups *Structural Metrics* and *Reuse Metrics*.

2. **Attribute-Central View –** The user may wish to approach the analysis of his/her design by starting from the quality attributes he/she wishes to evaluate and then

see which metrics affect each attribute. A view of the data is required to facilitate this approach.

3.10.3 Textual Representation

All the information specified above is to be presented in textual format as clearly and intuitively as possible. No recommendations are being made with regards to user-interface design as yet.

3.10.4 Graphical Reports

The main strength of the system is expected to be the graphical representation of the metric results. This is to take the form of interactive graphs plotted using the results of the metrics as data. The graphs are expected to be interactive in that sense that since in most case, single columns, points and lines on the graphs may represent multiple entities, the user should be able to click on a column, point or line and get information about the entities represented by that column. For example, a bar on a graph showing the complexities of classes may list 10% of the classes as having unacceptable complexity. The user should be given the facility of selecting that bar and be shown a list of the classes. Also, each graph is to have a knowledge base linked to it whereby the user is provided with information on how the graph should be used and interpreted.

A detailed specification of the required graphs is given in section 3.11 below.

3.11 Detailed Specification of Required Graphs

This section will describe in detail all the required graphical output of the system. Some background information and suggestions will also be given as to how the graphs could be interpreted by the user.

3.11.1 Weighted Methods per Class (WMC) Histogram

This histogram should plot complexity ranges against the number of classes whose WMC value lies in each range. This will give the user a good overall view to recognize the distribution of complexities over the whole project. As a rule, most classes should have a WMC that is under 20 and it is recommended that 40 not be exceeded however each project has its own circumstances and it is up to the quality assurance personnel to draw conclusions and recommend changes to the design. A user should be able to select a bar on the histogram and get a list of all the classes represented by that bar.

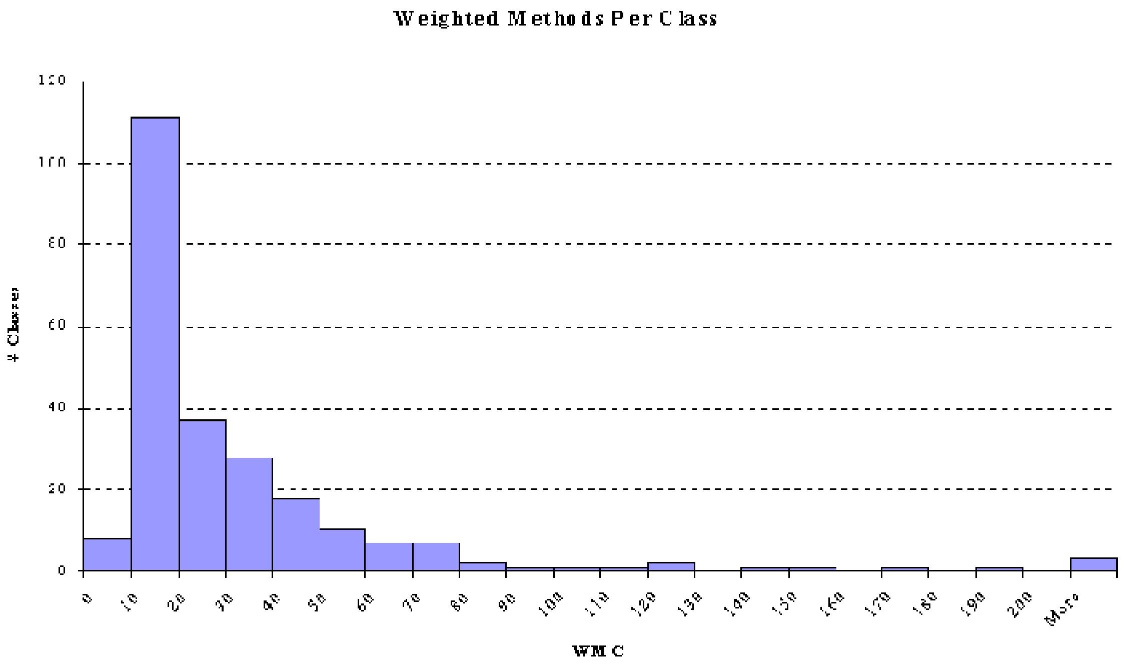


Figure 20 - An example of what the WMC Histogram might look like

3.11.2 Number of Children (NOC) Scatter Graph

This graph should plot the number of children of each class against its depth in the tree. Higher DIT values indicate a trade-off between increased complexity and increased reuse. Higher NOC values indicate increased internal reuse but also show the increased need for testing. One point on the graph may represent more than one class so the user is to be allowed to select any point and get a list of classes represented by that point. The graph is expected to take the following form:

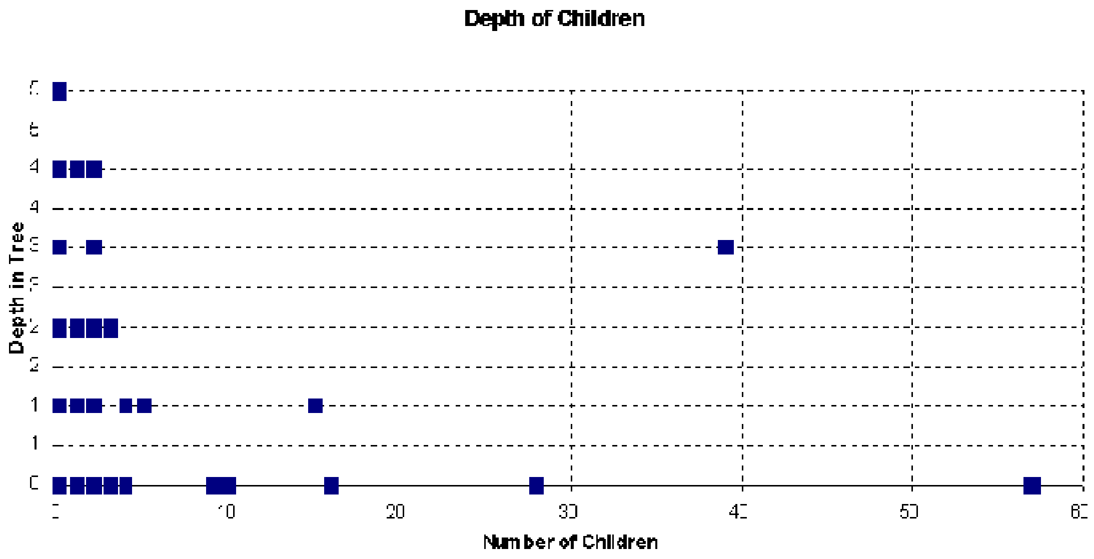


Figure 21 - An example of the graphical representation of NOC

3.11.3 Depth of Inheritance Tree (DIT) Histogram

The DIT histogram should plot DIT values against the percentage of classes in at each inheritance level. Higher percentages around levels 2 and 3 would show a high level of reuse but also indicate increase complexity. A trade-off should be reached according to the goals of each project. As with other histograms, if a user selects a particular bar, the classes being represented by that bar should be listed for the user.

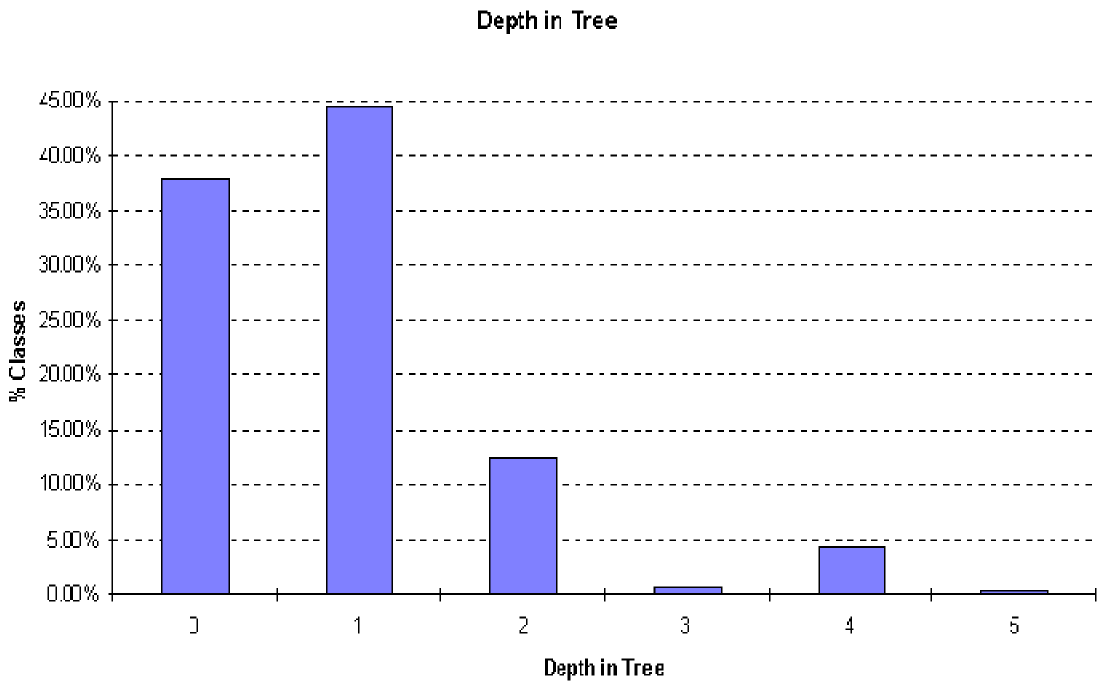


Figure 22 - An example of the DIT Histogram

3.11.4 Coupling Between Objects (CBO) Histogram

This graph should plot different levels of CBO against the number of classes that have that level of coupling. The user should use this graph to determine the distribution of CBO over the whole project. If there are more classes at higher levels of CBO, then the project is bound to be very difficult to understand and maintain. Also, the user can identify which classes have high levels of coupling and take steps to reduce CBO for those classes. As in other histograms, selecting a column will result in the user being presented with a list of classes represented by that histogram.

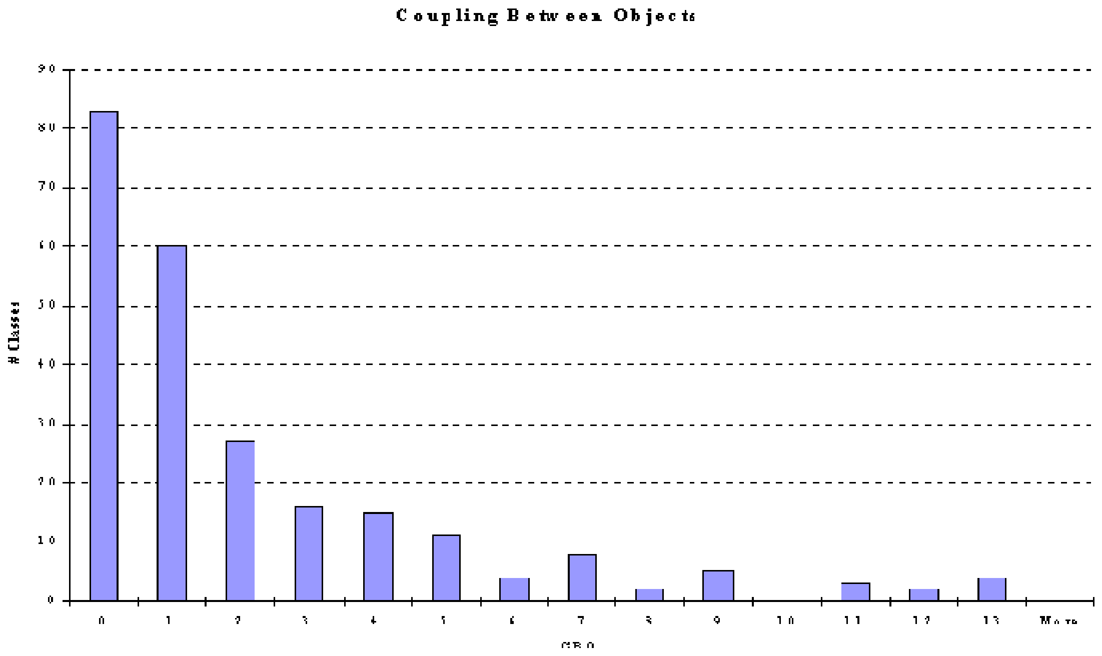


Figure 23 - An example of the CBO Histogram

3.11.5 Response for a Class (RFC) Histogram

This graph will map ranges of RFC against the number of classes in each range. Classes with large RFC values have a greater complexity and decreased understandability. Testing and debugging will also be more complicated. If the user selects a column on the histogram, the classes represented by that histogram are to be listed.

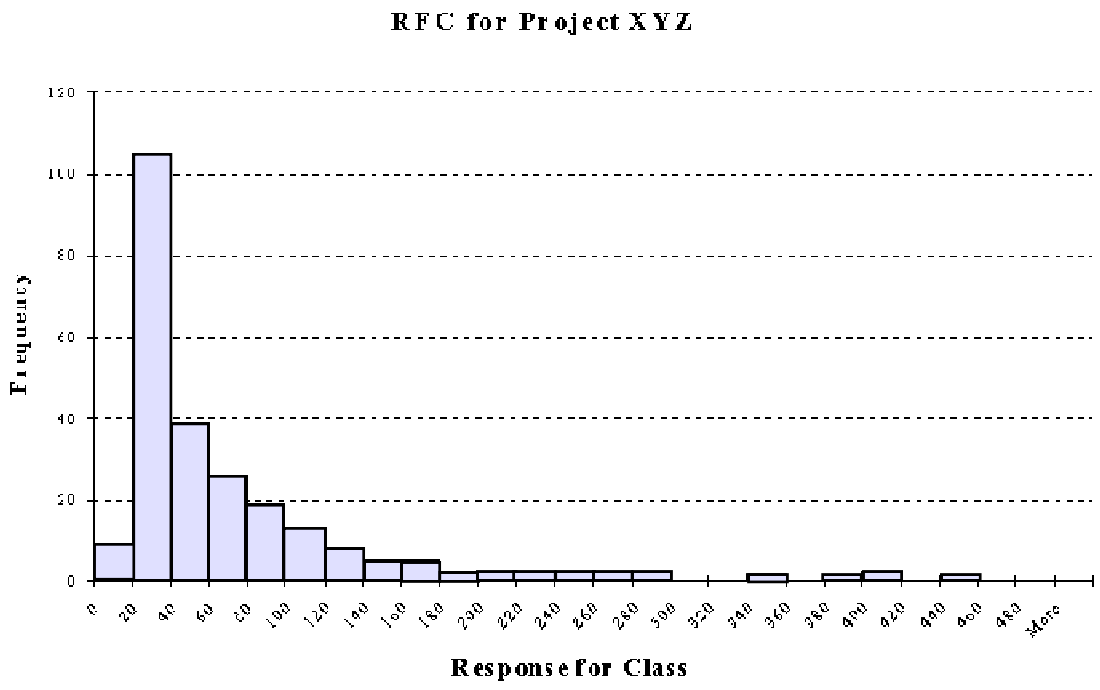


Figure 24 - An example RFC histogram

3.11.6 Lack of Cohesion of Methods (LCOM) Scatter Graph

The value of Lack of Cohesion of Methods (LCOM) depends on the number of methods, so there is a maximum value possible. This graph will plot the value of LCOM for each class against possible maximums. The closer an LCOM point is to its maximum, the less the classes represented by that point utilize the encapsulation feature of the object-oriented paradigm. This may also be an indication that that the class could be split into two or more subclasses. Again, click on a point will result in the user being shown a list of classes related to that point.

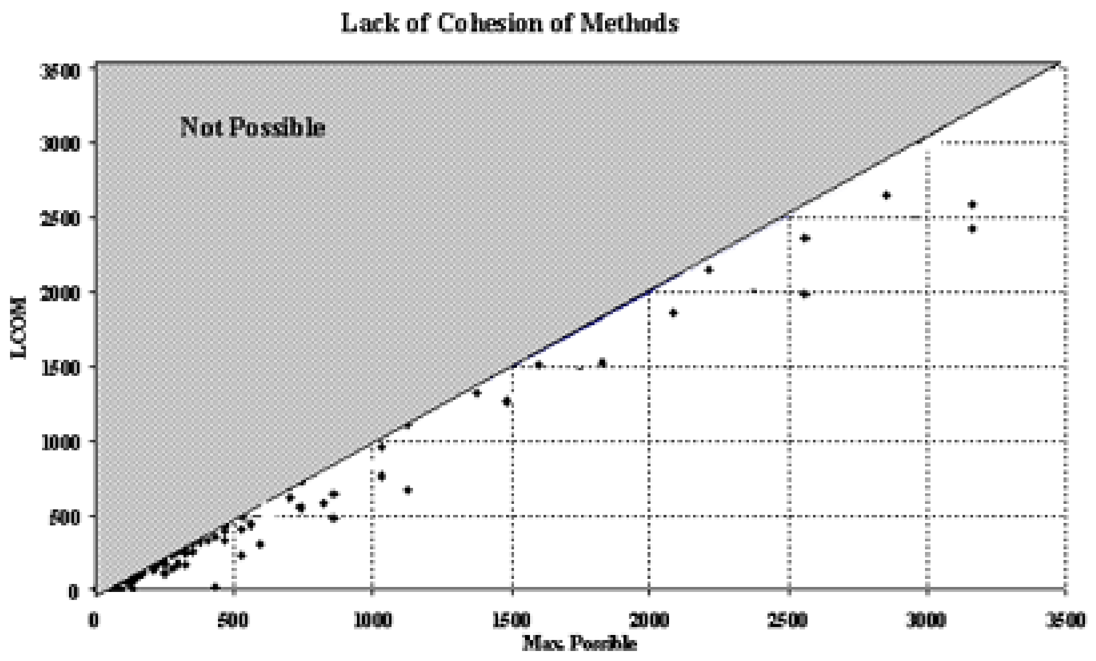


Figure 25 - An example of the LCOM Scatter Graph

3.11.6 Reuse Metrics Pie Chart

The reuse metrics are better looked at as a whole suite rather than as individual metrics. The pie chart is to represent the amount of couplings in the system and how they are distributed across the eight types of coupling defined for reuse metrics. Couplings that are beneficial for reuse are to be color-coded in shades of green, couplings that have a neutral effect on reusability are to be color-coded in shades of yellow whilst coupling which are detrimental to the reusability of the design are to be color-coded in shades of red. If the user selects a slice of the pie chart, a list of the couplings represented by that slice are to be shown. It will be the goal of the user to minimize the proportion of red slices and maximize the proportion of green slices in the pie chart.

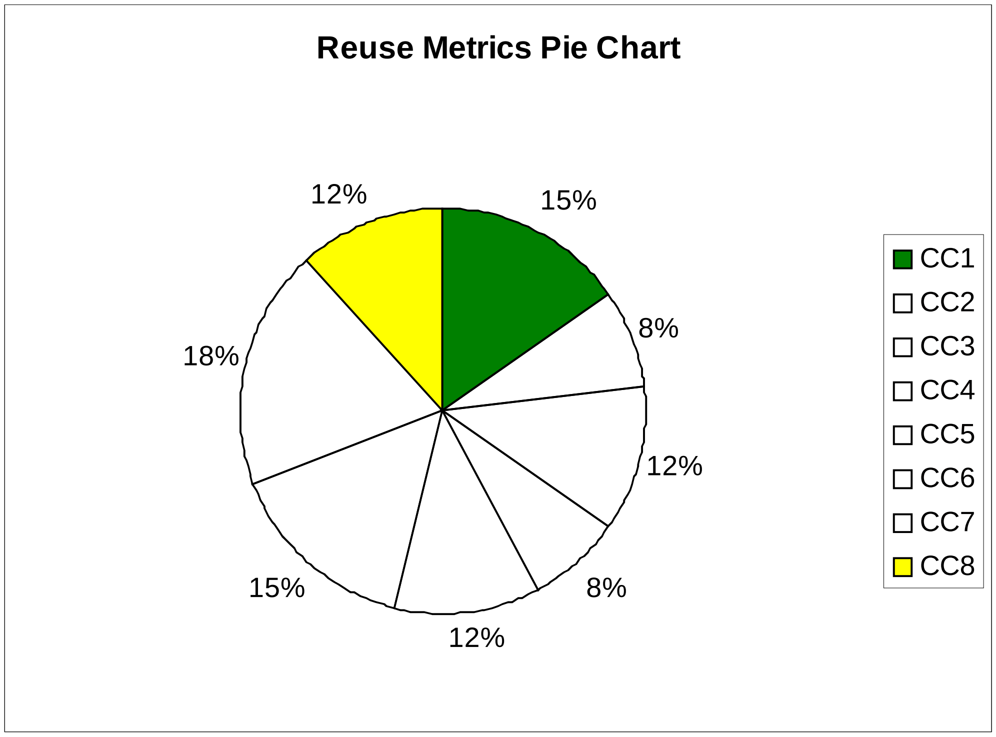


Figure 26 - An example Reuse Metrics Piechart

3.11.7 Averages Histogram sorted by Average

The user should be given the facility of viewing a histogram showing each project in the repository and the average value of a particular metric for that project. The current project should be placed on the graph and its bar should be a different color so as to make it stand out. In this diagram bars are to be displayed from left to right starting from the project with the lowest average. Clicking on a bar will give details of the project represented by that graph.

3.11.8 Averages Histogram sorted by Function Points

This graph will be identical to the on presented in section 3.11.7 only with the bars displayed from left to right starting with the project with the lowest function points. This will help aid the user to compare the current project against projects with similar function point readings.

3.12 What to expect in the next chapter…

With the system being specified according to the theory presented and chapter 1 and the decisions taken in chapter 2, the next logical step would be to present the design decisions taken with respect to the system that is to be implemented. The next chapter is a detailed explanation of the system's design. The structure of the system will be presented and major design decisions will be explained…

4. Design

4.1 Underlying Design Principles

This design was created using object-oriented principles and techniques. Wherever diagrams were needed, the Unified Modeling Language (UML) was used. It is assumed that the reader is familiar with UML diagrams and the principles behind the object-oriented paradigm.

4.2 Integration with ArgoUML

The specification requires that the tool be an extension to ArgoUML. To this extent it is worth taking a look at the structure of ArgoUML and define how the tool will fit into the picture. It is beyond the scope of this document to explain the exact inner workings of ArgoUML. Instead, an overview of the structure of the software shall be given and any areas of interest will be described in brief. This is being done because it will require a lot of space to explain exactly how the software represents the diagrams and the underlying UML metamodel.

4.2.1 ArgoUML Package Diagram

At the time of printing, standard ArgoUML consists of twelve top-level packages, which are depicted in the package diagram in figure 27. It was decided that a new package called **quality** will be added and this package contain classes related to the tool. There will be minimal tampering with existing code – only classes in the **ui** (user interface) package will be modified so as to provide a link between ArgoUML and the quality measurement tool. This is not to say that classes in other packages will not be used. They will just be used by classes in the **quality** package without any need to change existing code.

The contents of the **quality** package will be discussed in a future section.

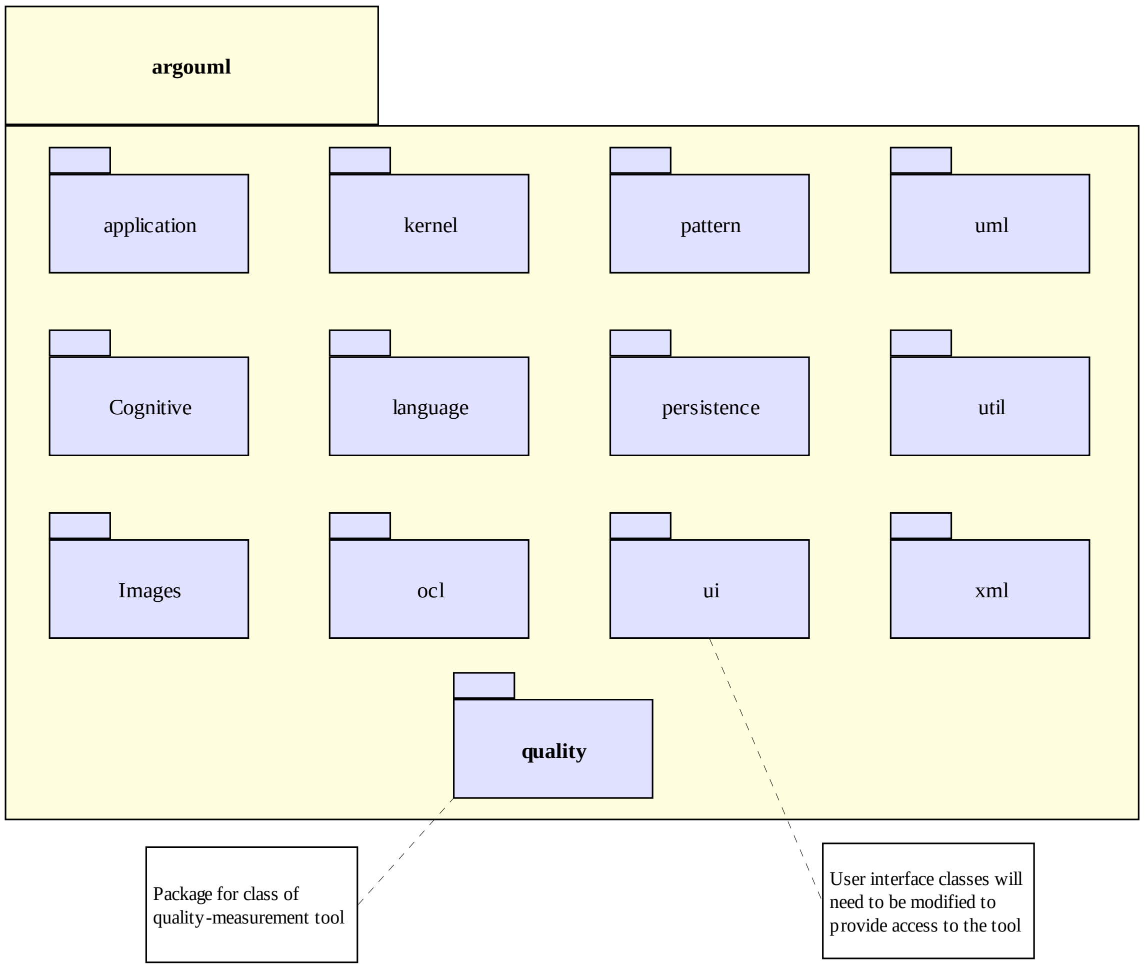

argouml
application
kernel
pattern
uml
Cognitive
language
persistence
util
Images
ocl
ui
xml
quality
Package for class of quality-measurement tool
User interface classes will need to be modified to provide access to the tool

Figure 27 - Package diagram for ArgoUML

4.2.3 The NSUML API

The entities (classes, packages, methods, etc) in the UML diagrams created by ArgoUML are represented internally using Novosoft's NSUML API. This API is a Java implementation of the UML 1.3 physical meta-model. Understanding this API is of extreme importance since it is used to represent the designs that tool will analyze. However explaining this here is beyond the scope of this document. If the reader is interested in the inner workings of the NSUML API, more details are available at http://nsuml.sourceforge.net/.

4.2.4 Key Classes in ArgoUML

The following is a list of classes in ArgoUML that will need to be modified in order to provide access to the quality measurement tool. Please note that design decisions regarding user interface, serialization of data, etc are discussed in a future section.

1. **org.argouml.ui.ProjectBrowser –** This class is responsible for setting up the main user window in ArgoUML. It needs to be modified to add new menus to provide access to the tool.

2. **org.argouml.kernel.Project –** This class encapsulates data and methods related to a project of UML diagrams. It will be modified to link the project file to a corresponding *quality file*.

3. **org.argouml.ui.Actions** – Implements classes that carry out actions related to the user interface. Modifications are needed to add new actions related to accessing the quality measurement tool.

4. **org.argouml.ui.NavigatorPane** – Implements the navigator pane in ArgoUML. A user will be able to right-click on a class and access quality-related functions from a popup menu. This functionality has to be taken care of in the NavigatorPane class.

4.3 Design Issues and Decisions

This section will describe the main design issues that needed to be solved and the decisions that were taken in their respect.

4.3.1 Separation of the System into Modules

The system will be separated into 3 main modules:

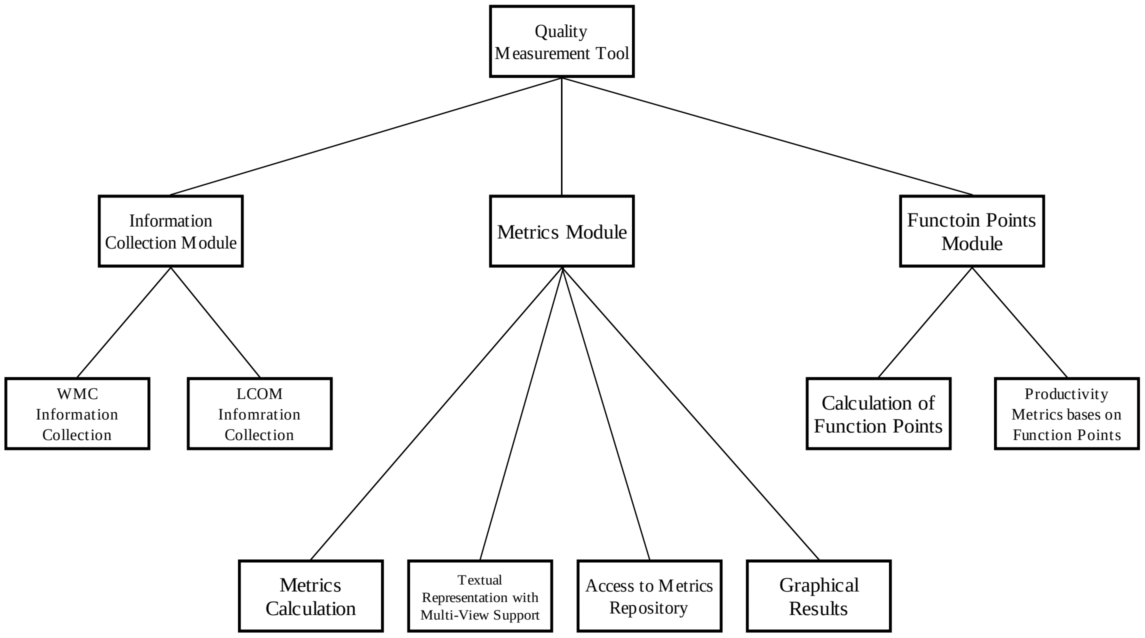

Quality Measurement Tool
Information Collection Module
Metrics Module
Functoin Points Module
WMC Information Collection
LCOM Infomration Collection
Calculation of Function Points
Productivity Metrics bases on Function Points
Metrics Calculation
Textual Representation with Multi-View Support
Access to Metrics Repository
Graphical Results

Figure 28 - A hierarchical diagram of the system modules

1. The **information collection module** will be responsible for collecting from the user information that cannot be obtained from UML diagrams. For more details, review the specifications of the tool.

2. The **metrics module** will provide the interface for metrics calculation, multi-view textual representation of results, comparison of the current project with projects in the metrics repository, as well as access to graphs provided by the metrics.

3. The **function points module** will be responsible for calculating the function points of a system and providing very basic productivity metrics based on the user’s cost and timing estimates.

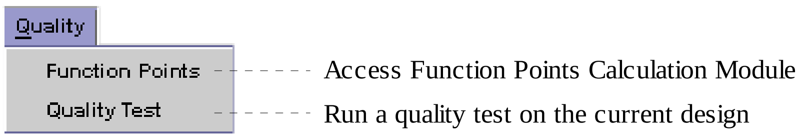
4.3.2 Accessing the Tool from ArgoUML

It was decided that the functions offered by the tool will be accessed from ArgoUML in the following ways…

The Quality Menu

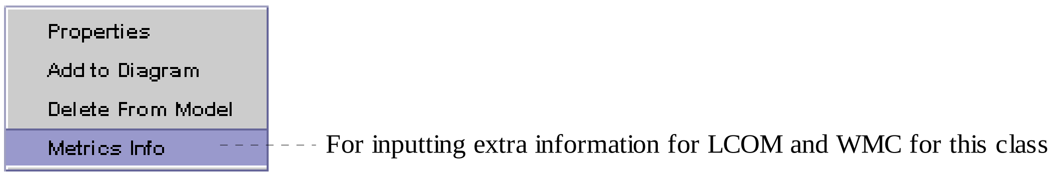
A "Quality" pull-down menu will be added to ArgoUML's menu bar. This will provide 2 options:

1. Accessing the function points calculation module and
2. Running a quality test on the current design. This will bring up the results and allow the user to compare with projects in the repository as well as view graphical representations of the data.


Quality
Function Points
Quality Test
Access Function Points Calculation Module
Run a quality test on the current design

The Class Popup Menu

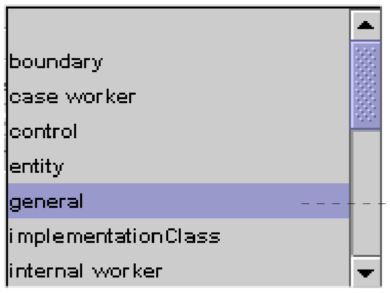
This menu is a feature of ArgoUML but a new option will be added to it for the convenience of the tool. A user can right-click on a class and the following popup menu will be displayed:


Properties
Add to Diagram
Delete From Model
Metrics Info
For inputting extra information for LCOM and WMC for this class

The first 3 options are standard ArgoUML options but the last one will be added in order to access a module that captures extra information needed for calculating the LCOM and WMC metrics on that class.

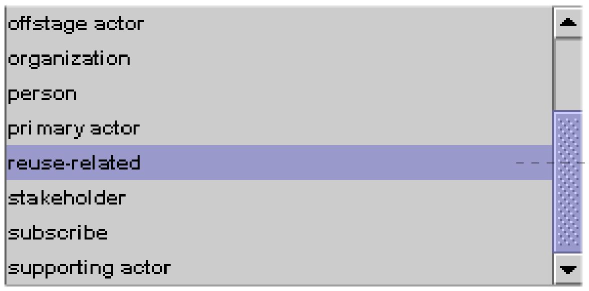
The <<General>> Stereotype for Classes

A new stereotype was defined for classes as part of the methodology for reusability metrics. The user basically uses it to indicate which classes are meant to be reused in other systems.



The <<Reuse-Related>> Stereotype for Associations

A new stereotype was defined for use with associations in class diagrams. This stereotype is used to show that two class hierarchies (defined by the two classes at the ends of the association as their roots) are meant to be reused together in future systems.



4.3.3 Saving of Quality-Related Data

It was decided that data files be saved using the eXstensible Markup Language (XML). XML is the universal format for structured documents and data on the Web. It basically provides a method for placing structured data into a text file.

Why XML? Storage of data in binary files has many pitfalls. Firstly, it is usually platform-dependant, a pitfall better avoided seeing the interoperability requirements of today's software. Secondly binary files are not extensible. With XML you can come up with a standard format for saving data, do the actual saving, extend the format of the XML file and the old data will still be readable. Also, keeping in mind that a metrics repository is going to be implemented, using XML is a convenient way of enabling multiple users of the tools to make their repositories available to each other since XML is easily transferable over the web and files can be easily merged. The fact that XML files are basically strings also results in them being easily compressed before data is transferred over the web for sharing.

It is up to the reader to familiarize him/herself with XML technology, as it is not feasible to go through it here.

4.3.4 What data needs to be saved?

There are two items of data that need to be saved:

1. The metrics repository and
2. The extra information collected from the user such as function points, cohesion information, WMC information etc.

4.3.5 XML Structure for the Metrics Repository

XML files are structured in a hierarchical system of nodes and each node has attributes associated with it. The metrics repository will be structured as follows:

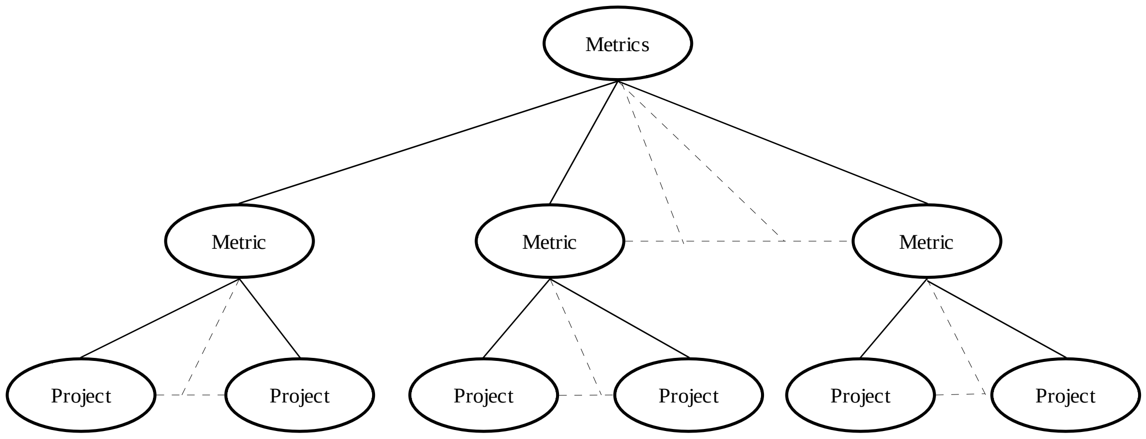


Figure 29 - XML Structure for the Metrics Repository

So there are basically 2 types of nodes (to root node is not really a node, it is just a tag):

1. Metric nodes, which encapsulated data representing a metric. Attributes are: *name, max (highest result obtained), min (lowest result obtained), results (number of entities on which the metric was calculated in total, average*.

2. Project nodes, which encapsulate summarized data about projects with respect to their parent metric node. Attributes are: *name, timestamp (time of last update in repository), FP (function points of project), calculations (number of times the metric (parent node) was calculated on this project), average)*

4.3.6 XML Structure for the Quality Data file

The tool will save a quality-information file along with the normal ArgoUML project files whenever the user saves his/her designs. This will be an XML file with the following structure:

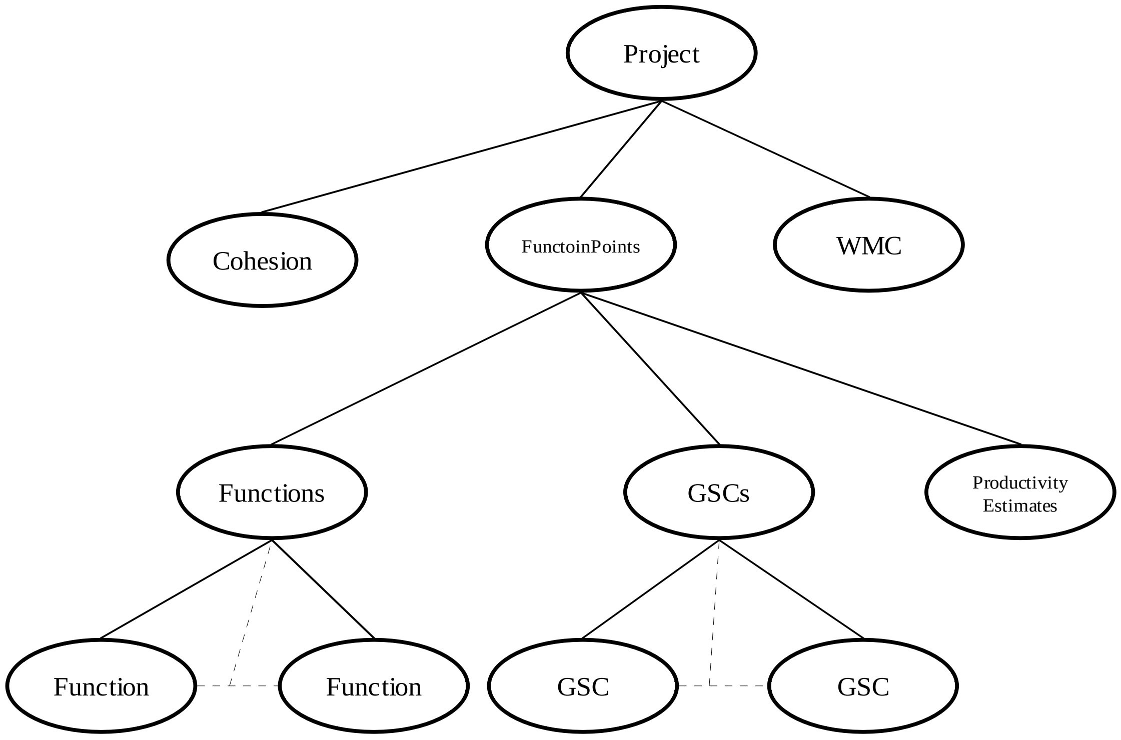


Figure 30 - Structure of XML Quality-Info File

8 types of nodes are used:

1. **Cohesion –** Has attributes that define information about methods and the instance variables they use

2. **WMC –** Has attributes that link activity diagrams in a project with methods in particular classes

3. **FunctionPoints** - Root Node for function points data

4. **Functions** – All the children of this node represent functions used in function points calculation.

5. **GSCs –** Mother node for general system characteristics nodes

6. **ProductiviteEstimates** – Holds information about the user's estimates on programmer productivity and cost per function point

7. **Function –** Holds data about 1 function

8. **GSC –** Holds data about 1 general system characteristic

4.4 Structure of Packages

This section describes how the classes of the tool will be laid out in different packages and how the individual classes will be interrelated. The following is an overall view of the packages involved in the system. There are more packages that are part of ArgoUML but are of no particular interest to the quality measurement tool so they have been omitted.

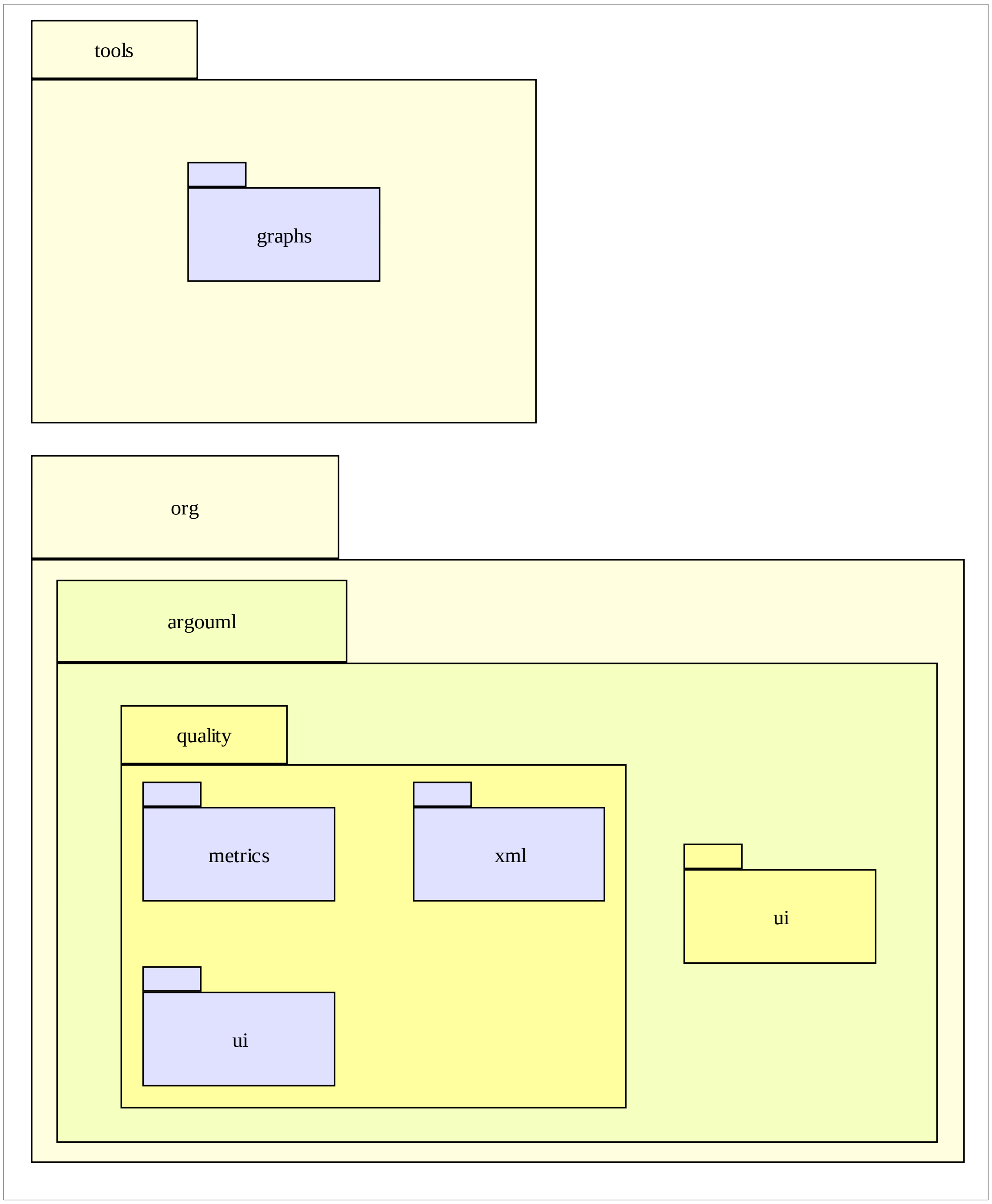


Figure 31 - An high-level view of the packages of the system

Here is a brief explanation of the packages in the above package diagram:

1. **tools** – This package contains ‘helper’ classes that are of importance to this project but are reusable in other systems. This package also contains the *graphs* package.

2. **tools.graphs** – Contains classes responsible for the graphs produced by the tool. These classes were designed with reuse in mind so they have been placed in the *tools* package.

3. **org** – A mother package that holds various packages. The package diagram only displays the *argouml* package because it is the only package of interest to us.

4. **org.argouml** – The package that contains all the classes for ArgoUML. The tool will be created in a child-package of the *argouml* package.

5. **org.argouml.ui** – Handles the user interface functions of ArgoUML. Some of the source code in this class will need to be modified so that ArgoUML can link to the tool.

6. **org.argouml.quality** – This is the package where classes that implement the quality measurement tool will be held.

7. **org.argouml.quality.metrics** – Contains the metrics classes and all related support classes.

8. **org.argouml.quality,xml** – Contains classes that handle parsing, serialization and abstraction of XML files used by the quality measurement tool.

9. **org.argouml.quality.ui** – Contains classes that handle user interface functionality for the quality measurement tool.

4.4.1 The *org.argouml.quality* Package

The *org.argouml.quality* package contains the classes and packages that implement the quality measurement tools. The following diagram takes a more detailed look at this package:

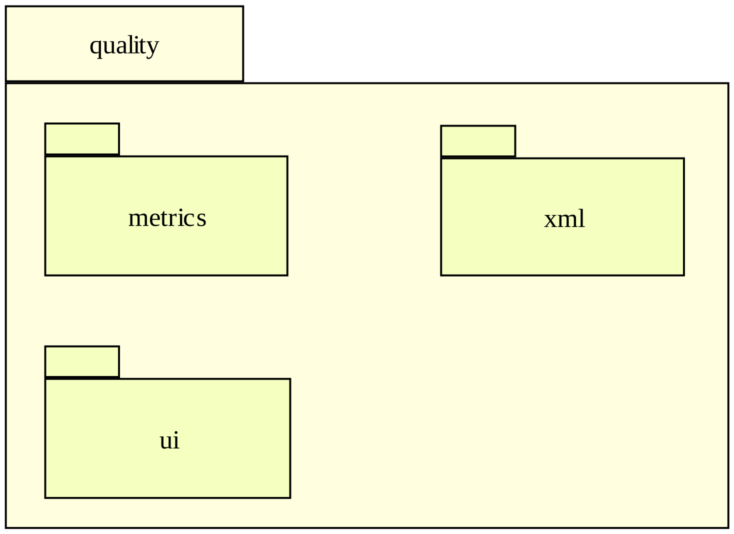


Figure 32 - The *org.argouml.quality* package

4.4.2 The *org.argouml.quality.metrics* Package

The *org.argouml.quality,metrics* package contains the classes that represent metrics as well as support classes for metric calculation. The following class diagram takes a more detailed look at this package:

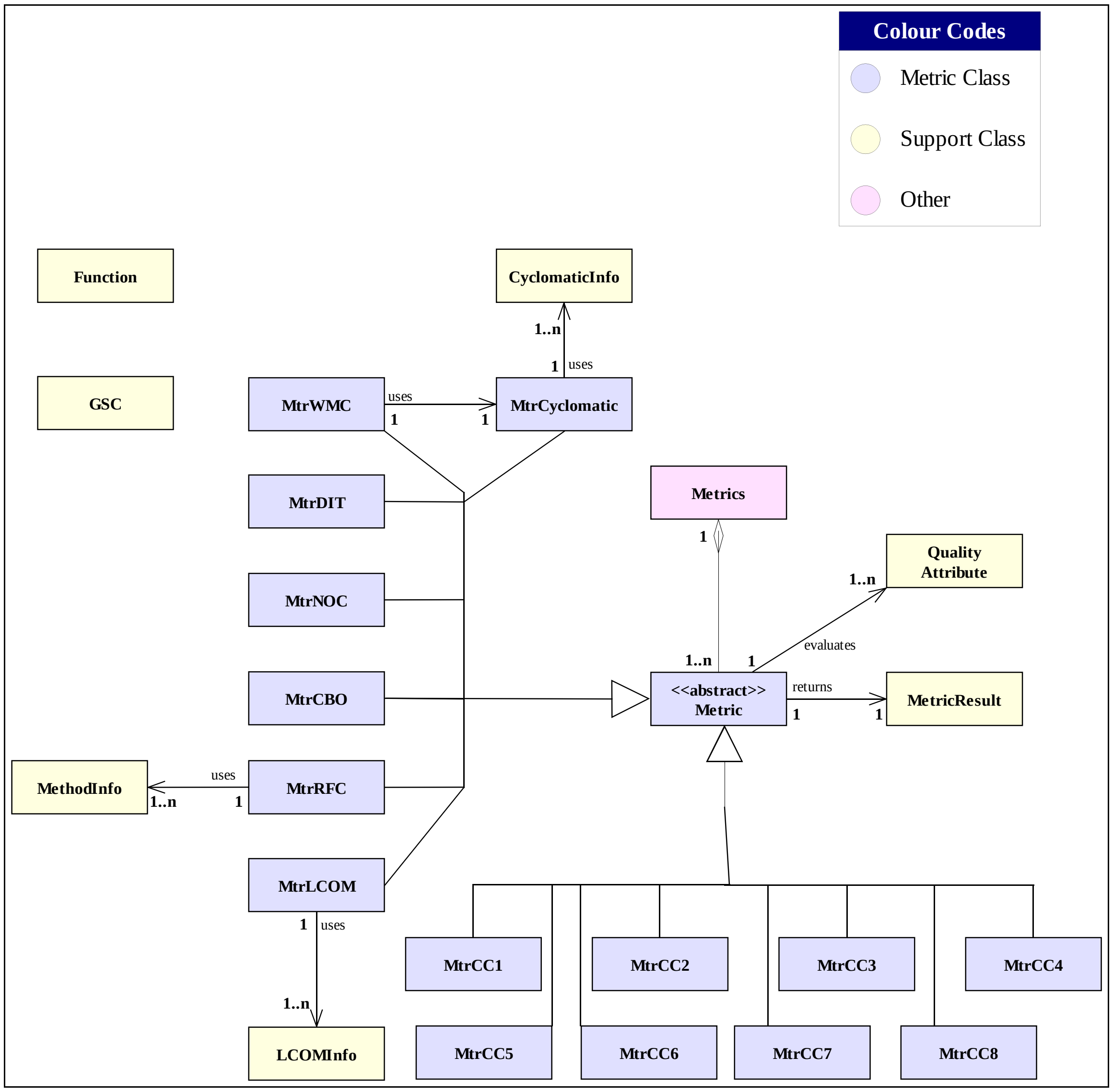


Figure 33 - Class Diagram for the *org.argouml.quality.metrics* package

Although important classes will be explained in more detail later, here is a brief description of the classes in the diagram:

1. **Metrics** – A class composed of all the metrics available to with the tool. It is responsible for separating metrics into suites, handling basic statistics collection and initiating procedures for updating the metrics repository with the statistics of the current project.

2. **Metric** – An abstract class that implements methods that are common to all metrics and defines abstract methods that need to be implemented by its subclasses.

3. **MtrWMC –** Implements the metric *Weighted Methods per Class (WMC)*

4. **MtrCyclomatic –** Implements the *Mc. Cabe's Cyclomatic Complexity* metric. This metric is not available to the user. It is a helper metric for *MtrWMC*.

5. **MtrDIT –** Implements the metric *Depth of Inheritance Tree (DIT)*

6. **MtrNOC –** Implements the metric *Number of Children (NOC)*

7. **MtrCBO –** Implements the metric *Coupling Between Objects (CBO)*

8. **MtrRFC –** Implements the metric *Response for a Class (RFC)*

9. **MtrLCOM –** Implements the metric *Lack of Cohesion of Methods (LCOM)*

10. **MtrCC1 –** Implements the metric *Coupling Count 1 (CC1)*

11. **MtrCC2 –** Implements the metric *Coupling Count 2 (CC2)*

12. **MtrCC3 –** Implements the metric *Coupling Count 3 (CC3)*

13. **MtrCC4 –** Implements the metric *Coupling Count 4 (CC4)*

14. **MtrCC5 –** Implements the metric *Coupling Count 5 (CC5)*

15. **MtrCC6 –** Implements the metric *Coupling Count 6 (CC6)*

16. **MtrCC7 –** Implements the metric *Coupling Count 7 (CC7)*

17. **MtrCC8 –** Implements the metric *Coupling Count 8 (CC8)*

18. **CyclomaticInfo –** A support class for *MtrCyclomatic* **–** Multiple instances of this class hold tuples of a method and the activity diagram that describes its behavior.

19. **MethodInfo –** A support class for *MtrRFC* **–** It basically hold information about a single method in a sequence or collaboration diagram. Remember that RFC counts the number of methods in a class as well as the methods it calls from other classes. The information about methods in other classes is only available through sequence and collaboration diagrams.

20. **LCOMInfo –** A support class for *MtrLCOM* **–** Multiple instances of this class hold tuples relating methods to the instance variables they use.

21. **QualityAttribute –** Encapsulates the data related to a quality attribute, namely the name of the attribute and information about why a particular metric evaluates it. If a quality attribute is evaluated by more than one metric, a different *QualityAttribute* instance needs to be created for each metric because the information about how the metric evaluates it will change.

22. **MetricResult –** Metrics return their results as an instance of this class.

23. **Function –** Encapsulates the data of a *Function* with respect to function points calculation. This class has no associations to any other classes. It is only used by *FunctionPointFrame* in the *org.argouml.quality.ui* package.

24. **GSC –** Encapsulates the data of a General System Characteristic with respect to function points calculation. This class has no associations to any other classes. It is only used by *FunctionPointFrame* in the *org.argouml.quality.ui* package.

4.4.3 The *org.argouml.quality.xml* Package

This package contains classes that handle the parsing, serialization and abstraction of xml files into higher-level objects for use by the tool. XML is user for 2 purposes:

1. To save extra quality-related data along with a project
2. As the basis of storing the Metrics Repository

The following diagram illustrates the classes in this package and there relationship to one another:

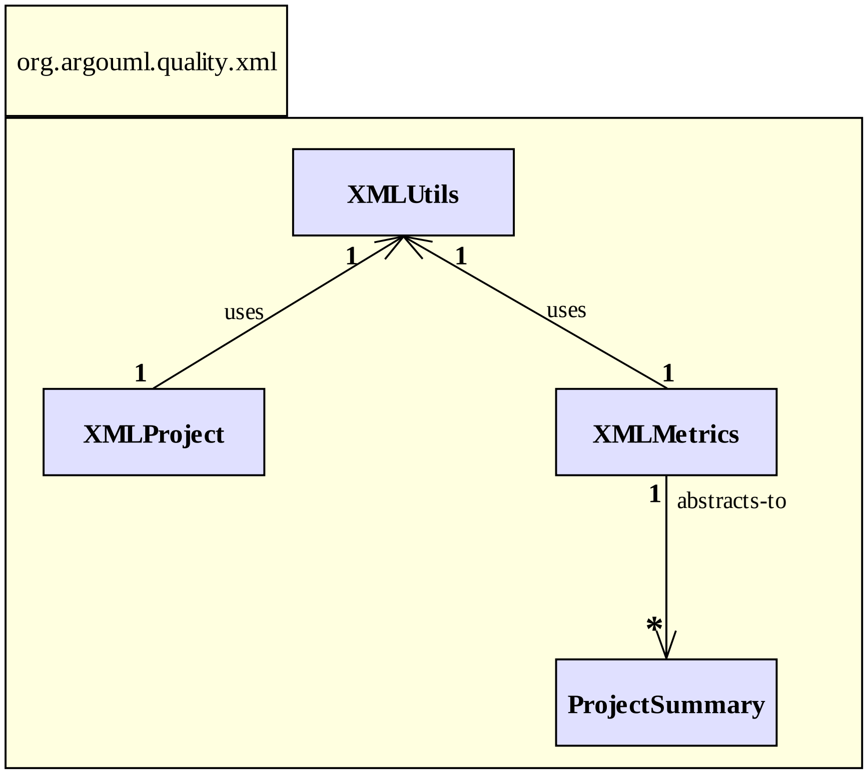


Figure 34 - The *org.argouml.quality.xml* package

The following is a brief explanation of the classes in the package:

1. **XMLUtils –** This class provides generic XML service such as loading XML files into memory or serializing them to secondary storage.

2. **XMLProject –** Provides high-level services for saving projects to and loading them from XML. The user of the this class need not know any details about XML but rather request that details about a project be loaded, saved, updated etc and the class will perform the low-level operations. This class makes use of the services provided by **XMLUtils**.

3. **XMLMetrics –** Provides high-level services for manipulating data in the XML metrics repository. Again, the user of the this class need not know any details about XML but rather request that details about a project/metric be loaded, saved, updated etc and the class will perform the low-level operations. This class makes use of the services provided by **XMLUtils**.

4. **ProjectSummary –** A class that encapsulates data about a project in the *metrics repository*. It is used as an abstraction of the low-level XML nodes that do the actual storing of the data.

4.4.4 The *org.argouml.quality.ui* Package

The *org.argouml.quality,ui* package contains the classes that handle the user interface functionality provided by the quality measurement tool. The follow is a class diagram of the classes in this package:

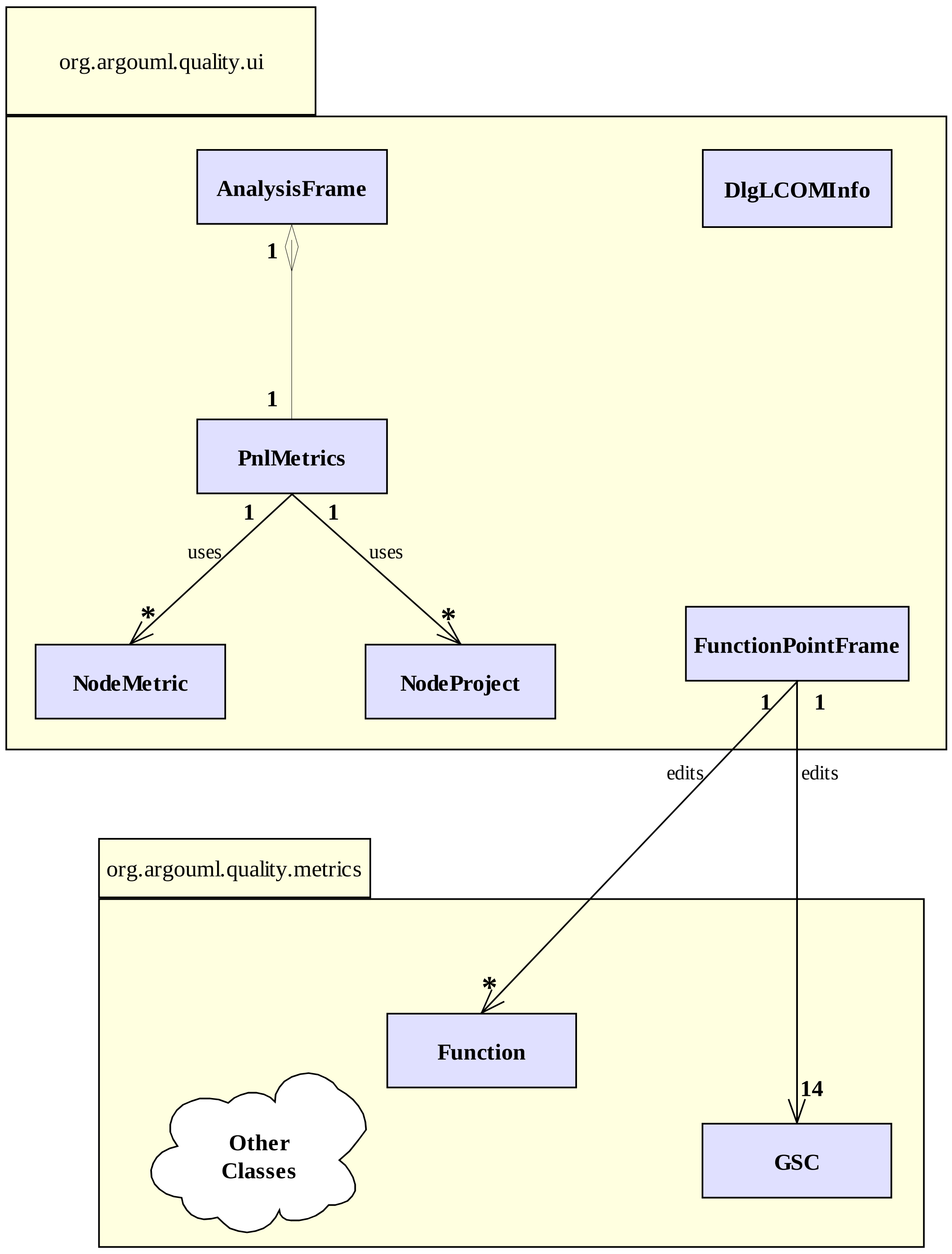


Figure 35 - A class diagram of the *org.argouml.quality.ui* package

The following is a brief explanation of the classes in the diagram:

1. **AnalysisFrame** – This class is responsible for displaying the *Metrics Module* as well as handling all the screen events related to metric analysis.

2. **PnlMetrics –** A panel that will take on part of the responsibility of the work done by AnalysisFrame.

3. **NodeMetric** – A node in a *JTree* (a class that implements a visual tree). The default node class could not be used because the node will need to encapsulate metric-specific data as well.

4. **NodeProject** - A node in a *JTree* (a class that implements a visual tree). The default node class could not be used because the node will need to encapsulate project-specific data as well.

5. **FunctionPointsFrame** – Responsible for displaying and handling the events of the *Function Points* module in the quality measurement tool.

6. **DlgLCOMInfo** – A dialog that collects the extra information needed for LCOM and WMC.

It is worth noting that the *FunctionPointFrame* class links to support classes in the *org.argouml.quality.metrics* package. An explanation of these classes can be found in the section describing the *org.argouml.quality.metrics* package.

4.4.5 The *tools* Package

The *tools* package contains the classes that were developed for this project but are reusable:

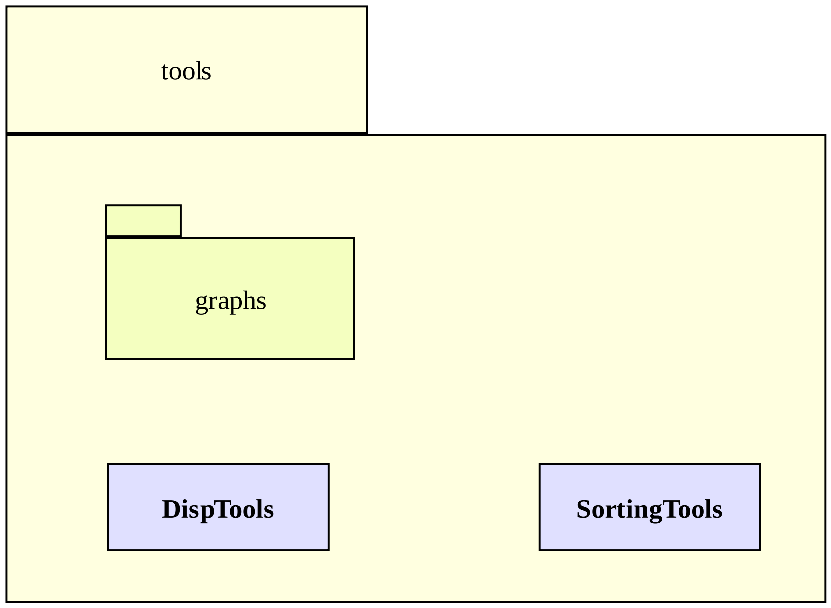


Figure 36 - The *tools* package

There are only two classes in this package:

1. **DispTools –** Contains static methods that perform display-related functions such as centering windows on the screen, etc.
2. **SortingTools –** Contains static methods that provide sorting functionality on a variety of objects.

4.4.6 The *tools.graphs* Package

This package is used extensively in this project and was developed to represent metric results graphically. However, extra effort was taken to make the graphs independently reusable in any other application. They are therefore placed in this package:

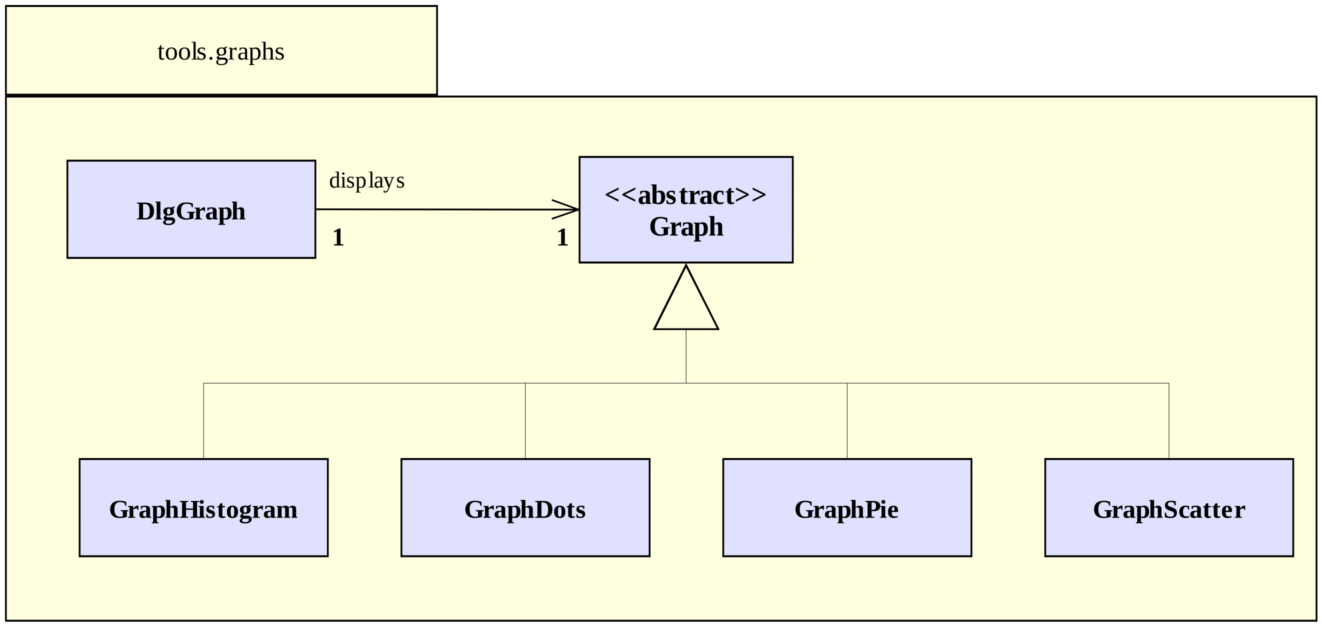


Figure 37 - The *tools.graphs* package

Here is a brief explanation of the graphs in this package:

1. **DlgGraph –** A dialog that interfaces with a graph object to display it and handle user events mainly regarding requesting help and information from the graph. Graphs can give information to the user about how they can be interpreted.

2. **Graph –** An abstract class that implements methods and encapsulates data that is common to all graphs. It also defines abstract methods that should be implemented by each individual subclass.

3. **GraphHistogram –** A general-purpose histogram.

4. **GraphDots –** Implements the dot-graph for representing the NOC metric. Other applications for this type of graph are probably limited but it is still reusable.

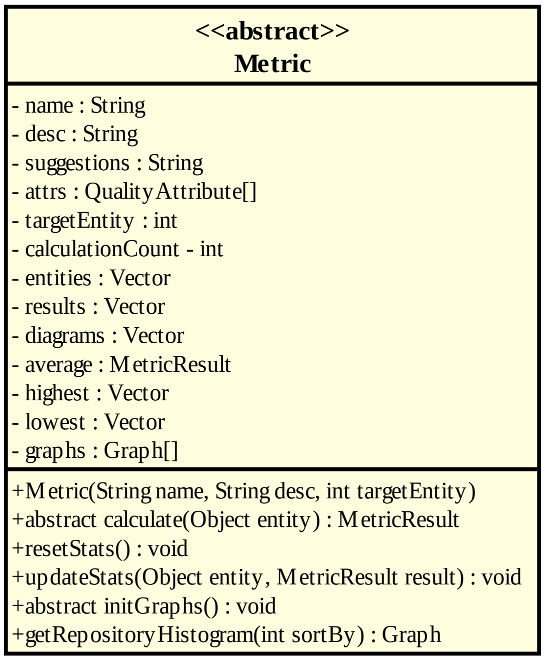
5. **GraphPie –** Implements a general-purpose pie chart.

6. **GraphScatter –** Implements a general-purpose scatter graph.

4.5 A closer look at key Classes

It is worth taking look at the key classes defined for the system. The chosen classes are either important super-classes that provide the framework for important parts of the system to be implemented by their sub-classes, or other classes that are responsible for important functions in the system. Please note that only the key methods and attributes are shown. Methods such as getter and setter methods are to be implemented as needed by the developer.

4.5.1 org.argouml.metrics.Metric



<<abstract>>
Metric

- name : String
- desc : String
- suggestions : String
- attrs : QualityAttribute[]
- targetEntity : int
- calculationCount - int
- entities : Vector
- results : Vector
- diagrams : Vector
- average : MetricResult
- highest : Vector
- lowest : Vector
- graphs : Graph[]

+Metric(String name, String desc, int targetEntity)
+abstract calculate(Object entity) : MetricResult
+resetStats() : void
+updateStats(Object entity, MetricResult result) : void
+abstract initGraphs() : void
+getRepositoryHistogram(int sortBy) : Graph

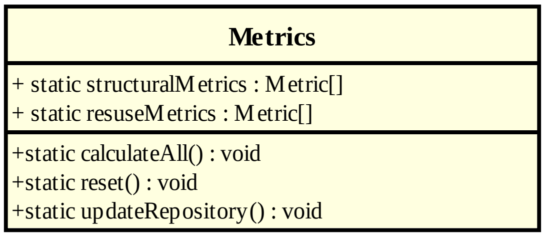
The *Metric* class is the parent class of all the metrics implemented by the system. It implements statistics-handling routines that are common to all metrics and defines abstract methods that need to be implemented by all its subclasses.

Attributes

Attribute Name	Type	Description
name	String	The name of the metric
desc	String	A description of the metric
suggestions	String	Suggestions to the user on how to use the metric, consequences of extreme values, etc.
attrs	QualityAttribute[]	An array of QualityAttribute objects representing the quality attributes that the metric evaluates.
targetEntity	int	Indicates what entity the metrics meauses (class, method, etc)
calculationCount	int	A statistics variable for keeping track of the number of times the metric was used.
entities	Vector	A vector of the entities (classes, methods, etc) that the metric was calculated on.
results	Vector	A vector of MetricResult objects corresponding to the *entities* vector.
diagrams	Vector	A list of diagrams representing the diagrams in which the entities in the *entities* vector exist.
average	MetricResult	The average value of the results of this metric.
highest	Vector	A vector of entities that obtained the highest value of the metric so far.
lowest	Vector	A vector of entities that obtained the highest value of the metric so far.
graphs	Graph[]	An array of graphs that can be used with this metric.

Methods	
Method Name	**Description**
Metric(String,String,int)	Constructor that initializes the name, description and targetEntity of the metric.
abstract calculate(Object)	An abstract method that is to be implemented by subclasses. The method calculates the metric of the given object (class, method, etc) and returns a *MetricResult* object.
resetStats()	Resets statistical counters and vectors.
updateStats(object, MetricResult)	Updates the statistics of this metric with the given object and corresponding result.
abstract initGraphs()	Creates graphs that interpret this metric and set the *graphs* attribute.
getRepositoyHistogram(int)	Creates a histogram comparing the average of this project with other projects in the metrics repository. The parameter defines how columns in the histogram should be sorted.

4.5.2 org.argouml.metrics.Metrics

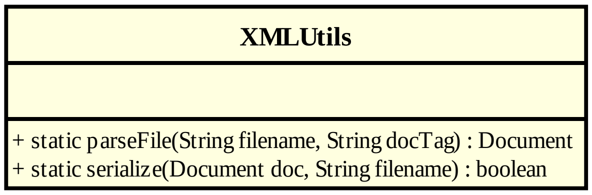

Metrics
+ static structuralMetrics : Metric[]
+ static resuseMetrics : Metric[]
+static calculateAll() : void
+static reset() : void
+static updateRepository() : void

The *Metrics* class is the class to which the tool refers to for information on the metrics available to it. It also acts as a starting point for a quality test. ArgoUML will call the calculateAll() method of this class to calculate all the available metrics on the current project. The class also does some aggregate statistics maintenance.

Attributes		
Attribute Name	**Type**	**Description**
structuralMetrics	Metric[]	An array consisting of a static instance of each of the structural metrics available to the system.
reuseMetrics	Metric[]	An array consisting of a static instance of each of the reuse metrics available to the system.

Methods	
Method Name	**Description**
calculateAll()	Loops through the metrics calculates each metric on the current project whilst updating statistics in the process.
reset()	Resets the statistics and counters of all metrics in the suites.
updateRepository()	Updates the metrics repository with the current project.

4.5.3 org.argouml.xml.XMLUtils

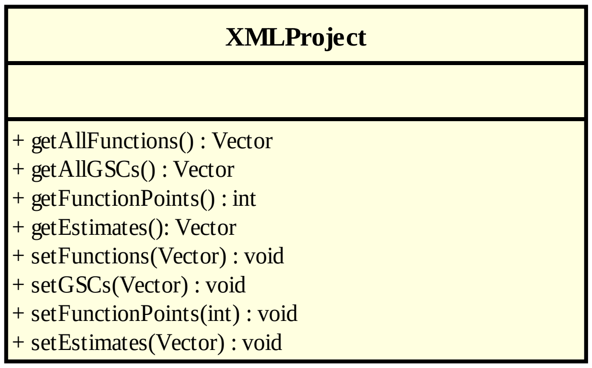

XMLUtils
+ static parseFile(String filename, String docTag) : Document
+ static serialize(Document doc, String filename) : boolean

This class basically offers two services:

1. Parsing of XML files and conversion into Document (org.w3c.dom) objects
2. Serialization of Document (org.w3c.dom) objects into XML files.

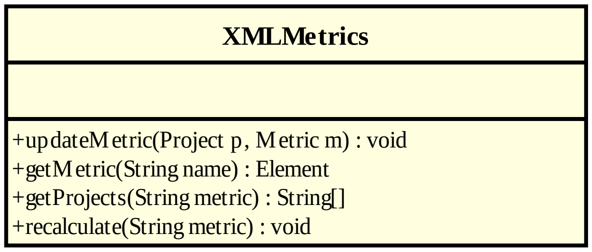
Methods	
Method Name	**Description**
parseFile(String filename, String doctag)	Parses and XML file and returns an *org.w3c.dom.Document* object representing the contents of the file.
serialize(Document doc, String filename)	Serializes an *org.w3c.dom.Document* into a file. It returns true if the operation is successful and false if not.

4.5.4 org.argouml.xml.XMLProject


XMLProject
+ getAllFunctions() : Vector
+ getAllGSCs() : Vector
+ getFunctionPoints() : int
+ getEstimates(): Vector
+ setFunctions(Vector) : void
+ setGSCs(Vector) : void
+ setFunctionPoints(int) : void
+ setEstimates(Vector) : void

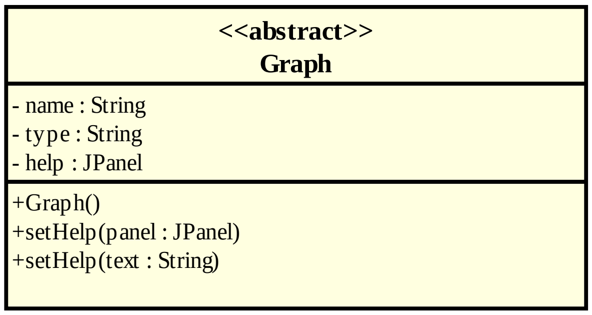
This class offers high-level manipulation of the XML model that saves the quality-information file. The methods look like simple getters and setters but they will be working at node and attribute level in XML.

4.5.5 org.argouml.xml.XMLMetrics


XMLMetrics
+updateMetric(Project p, Metric m) : void
+getMetric(String name) : Element
+getProjects(String metric) : String[]
+recalculate(String metric) : void

This class offers high-level manipulation of the XML model that saves the metrics repository. The user of this class should be kept as high as possible with regards to abstraction from core attributes and elements of XML. If needed, utilize helper classes that encapsulate element data.

4.5.6 tools.graphs.Graph



A simple class that acts as the mother-class of all graphs. It encapsulates the name of the graph (eg: "WMC Histogram"), the type of the graph (eg "Histogram") and a JPanel, which will contain help for the user regarding a particular graph. There should be getter and setter methods for all of the instance variables as well as the 3 methods shown in the diagram:

1. Graph() : Default constructor for all graphs
2. setHelp(JPanel) – Sets the help to the given JPanel
3. setHelp(String) – Takes the string, puts it inside the required visual components and sets the help attribute to a panel containing thos components.

4.6 Algorithms for Metric Calculations

It was decided that the algorithms for metric calculations be explained in an appendix since the majority of readers are not interested in the low-level implementation of the system. If you are interested in how the metrics are calculated, please refer to Appendix B.

4.7 What to expect in the next chapter…

The next chapter is a short one that reflects on the implementation of the system, difficulties encountered, its strong points and pitfalls, as well as the possibility of improvements.

5. Implementation

5.1 Proof of Completion

Proof of completion will be / has been given in the project presentation. Screenshots of the tool are given in the user documentation (end of this document) of the tool.

5.2 Implementation Difficulties

A number of difficulties were encountered along the way:

1. There was no clear documentation regarding how diagrams are saved and internally represented in ArgoUML so the process of getting to know this was like a long, agonizing debugging session where I experimented with different designs and then analyzed the contents of the variables in the program to see what was happening.

2. ArgoUML is an open source project and development is carried out by many individuals on a voluntary basis. This means that bugs in the system sometimes took a long time to be solved and in some cases this set my work back because I was relying on the bugs being fixed. There were problems with the Sequence and Collaboration Diagram editors, which were not functioning correctly. The Sequence Diagram editor was fixed in time but the Collaboration Diagram editor wasn’t, thus resulting in the RFC metrics not being completely implemented. Also, ArgoUML sometimes encounters problems with saving/loading projects. This was particularly frustrating during testing.

3. The XML API was particularly complicated to learn for a first-time XML user. It may seem like a simple dump of information to a file but the process involved

days of getting to grips with concepts, interfaces, classes and methods before the working version was finished.

5.3 Limitations of System

The system is by no means perfect. Nor was it meant to be so. In these projects, deadline and delayed research often make it impossible to implement optimized algorithms and/or user interfaces:

1. During testing, it was found that the collection of information for LCOM (defining the set of instance variables used by each method) will prove to be a lengthy process for medium-large projects and may not be feasible. It is recommended that a new metric with fewer overheads be found to measure cohesion.

2. There is plenty of space for optimization with regards to the time complexity of a total quality check. Quality tests for larger projects with a few hundred classes will take usually take a few minutes to complete.

3. There is no repository editor. Projects and metrics can only be added to the repository and not edited or removed.

4. There is no printing functionality. This would prove to be very useful if a tool of this kind were to be used in a real-life situation.

5.4 Possible Improvements

The system has plenty of space for future improvements and increases in features:

1. Intelligent project-specific advice where the tool will give advice on each particular project it analyzes, pointing out the extreme values and explaining the difficulties they might create.

2. The tool could be improved to integrate the design process with the preceding specifications stage by providing a framework for verification of design with specification documents.

3. A repository browser where the user can remove/edit projects and metrics from the metrics repository. Currently, the tool only supports the adding of new data to the repository.

4. Snapshot features where a user will be able to keep a snapshot of metric results at different stages of the project in order to analyze changes in metric results over time.

5. The system could be modified to allow the user to add upper and lower limits on each metric. This could then be used as a benchmark for classes passing or failing the quality test.

6. Printing of reports could be implemented as this would be a valuable feature if the tool were to be used in a real-world situation.

5.5 What to expect in the next chapter…

Chapter 6 will conclude this project by reflecting on what has been learnt from the experience of researching, specifying, designing and implementing a large project.

6. Conclusions

6.1 What have I learnt from this project?

6.1.1 Issues involved in Software Quality Assurance

This project has given me the fantastic opportunity to learn more about the ambiguous notion of *software quality*. I now feel I have a deeper understanding of the issues involved and the difficulties that exist in this area. I have a healthy respect for the sheer size of the software quality assurance problem and cannot pretend to really understand all facets of it. Software quality assurance should be carried out at each stage of development cycle as well as on the processes and people of a company. This is easier said than done with companies taking years to reach a quality standard they can be proud of.

6.1.2 Object Oriented Design

This project has also allowed me to reinforce my knowledge of the object-oriented paradigm especially in the area of design. Issues such as “what makes a good design” and “how this can be achieved” are far from being formally solved but this project has given me an insight into these issues and why there is no clear-cut way for solving them.

6.1.3 Learning from Mistakes

In a 10-credit project there is a lot of room for mistakes. This need not be a negative experience since any mistakes made during this project provided experience and equipped me with more mental knowledge about difficulties that may arise when working in the software engineering industry.

6.1.4 Value of Reuse

Working with Java and OO techniques gave me a feel of what reusability is all about. Java is packed full of modules built for reusability. For example, if I need to open a file dialog box, I just call the existing Java version without having to create one myself. Whereever possible, the classes in this system were designed with reuse and maintainability in mind. For example, the graphs are reusable in other projects with possibly different domains and the metrics framework makes it easy for new metrics to be added to the quality measurement tool. Of course this is an extremely small scale when compared with industry standards but over time, if you design with reuse in mind, you’ll end up with a substantial library of reusable components that will speed up future development.

6.1.5 Time Management

The experience of researching, designing, implementing and documenting a project of this size has proved to be an exercise in project and time management more than an exercise in academic skills. Although this project is dwarfed by large-scale industrial problems, I feel I am equipped with a better understanding of managing time as a limited resource.

Appendix A: Algorithms for Calculating Metrics

This section will deal with the algorithms for calculating each metric in the system. All the algorithms are to be implemented in the *calculate(Object)* method of the individual metric classes. Therefore, the algorithms will be expressed here as *UML Activity Diagrams* of those methods.

These algorithms are described in an appendix because most readers will not be interested in the low-level implementation details of the project.

Activity Diagram for Calculating WMC

The following is an activity diagram describing the algorithm for calculating WMC for a class. The responsibility for this is split over 2 classes (metrics). This is because WMC requires that the complexity of each method be computed.

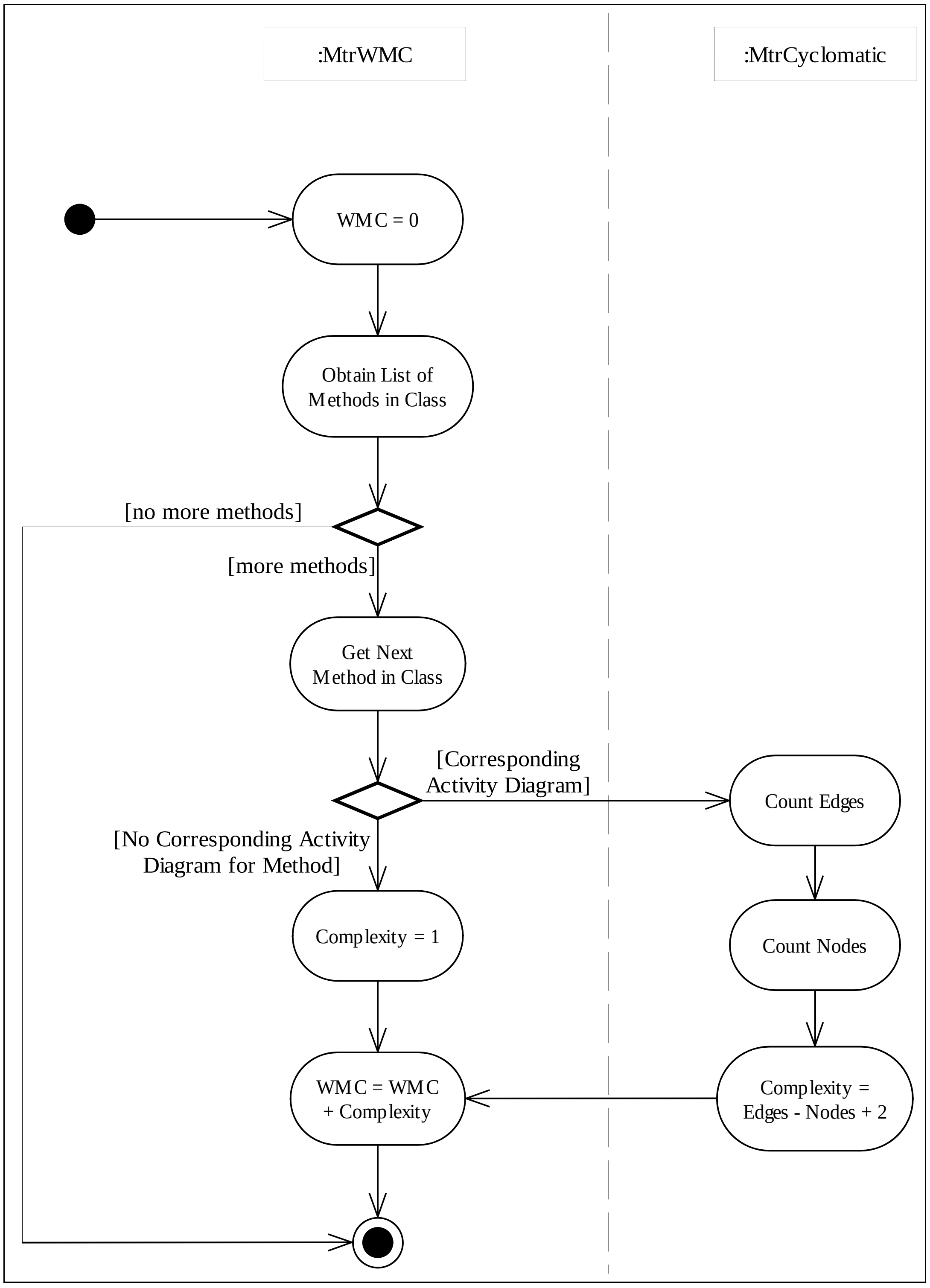


Figure 38 - Activity Diagram for calculating WMC

Activity Diagram for Calculating DIT

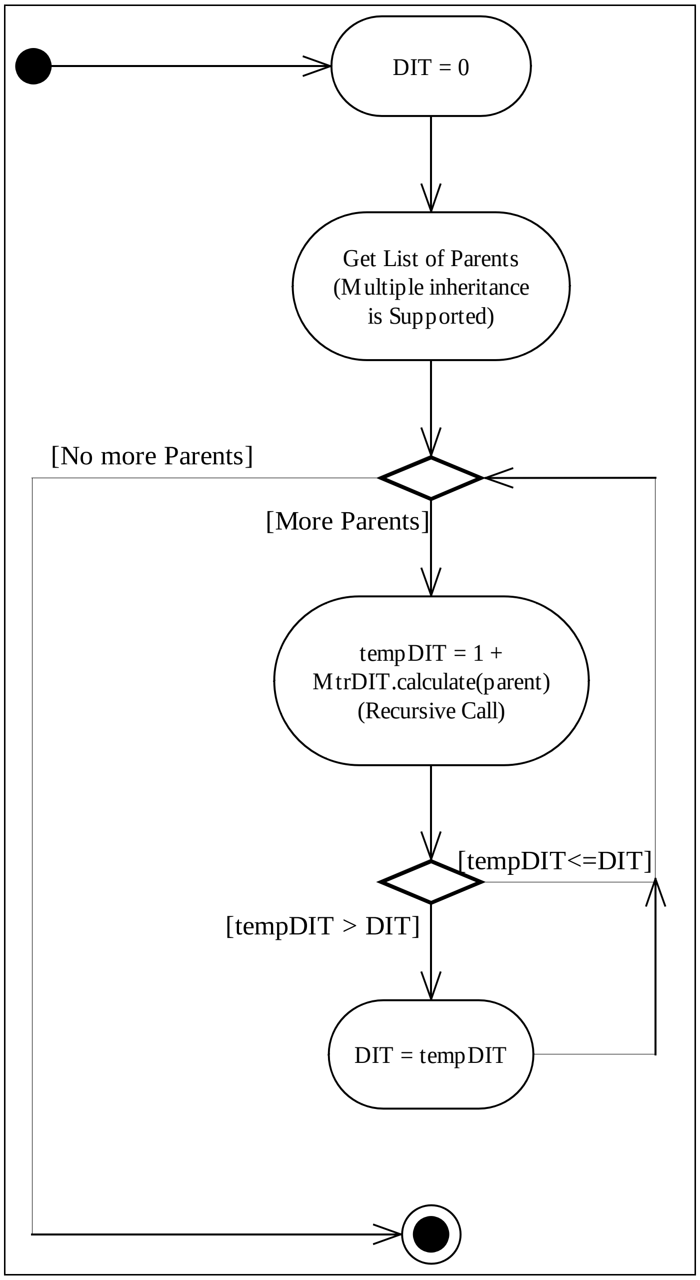

DIT = 0
Get List of Parents (Multiple inheritance is Supported)
[No more Parents]
[More Parents]
tempDIT = 1 + MtrDIT.calculate(parent) (Recursive Call)
[tempDIT<=DIT]
[tempDIT > DIT]
DIT = tempDIT

Figure 39 - Activity Diagram for calculating DIT

Activity Diagram for Calculating NOC

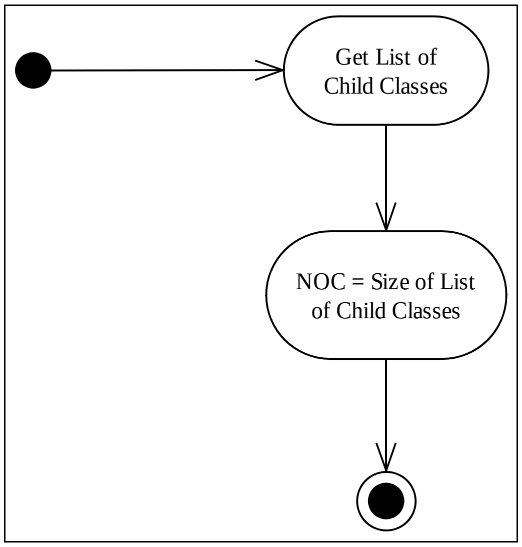

Get List of Child Classes
NOC = Size of List of Child Classes

Figure 40 - Activity Diagram for Calculating NOC

Activity Diagram for Calculating CBO

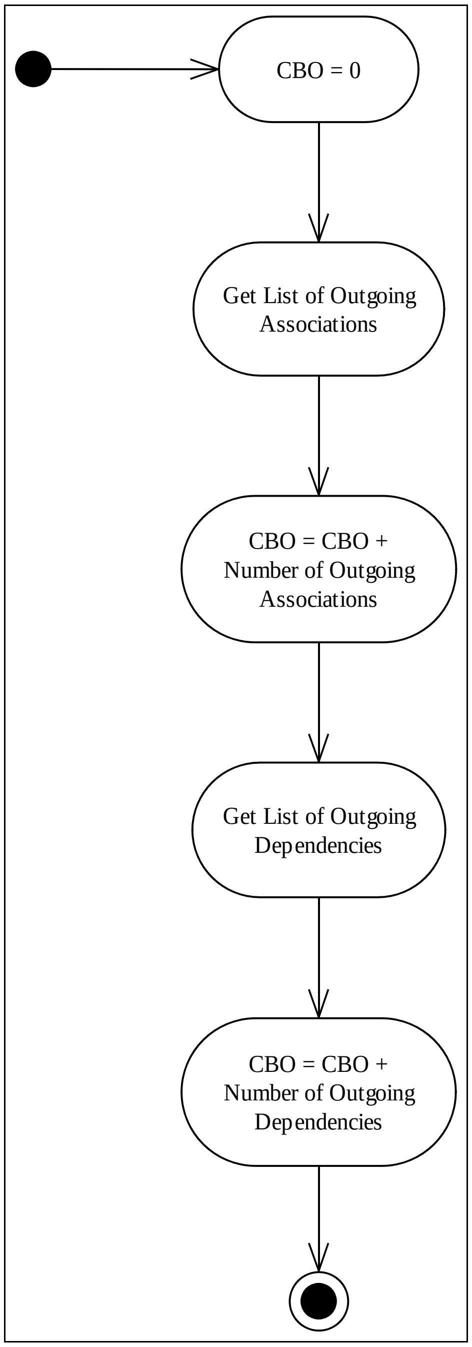

CBO = 0
Get List of Outgoing Associations
CBO = CBO + Number of Outgoing Associations
Get List of Outgoing Dependencies
CBO = CBO + Number of Outgoing Dependencies

Figure 41 - Activity Diagram for Calculating CBO

Activity Diagram for Calculating RFC

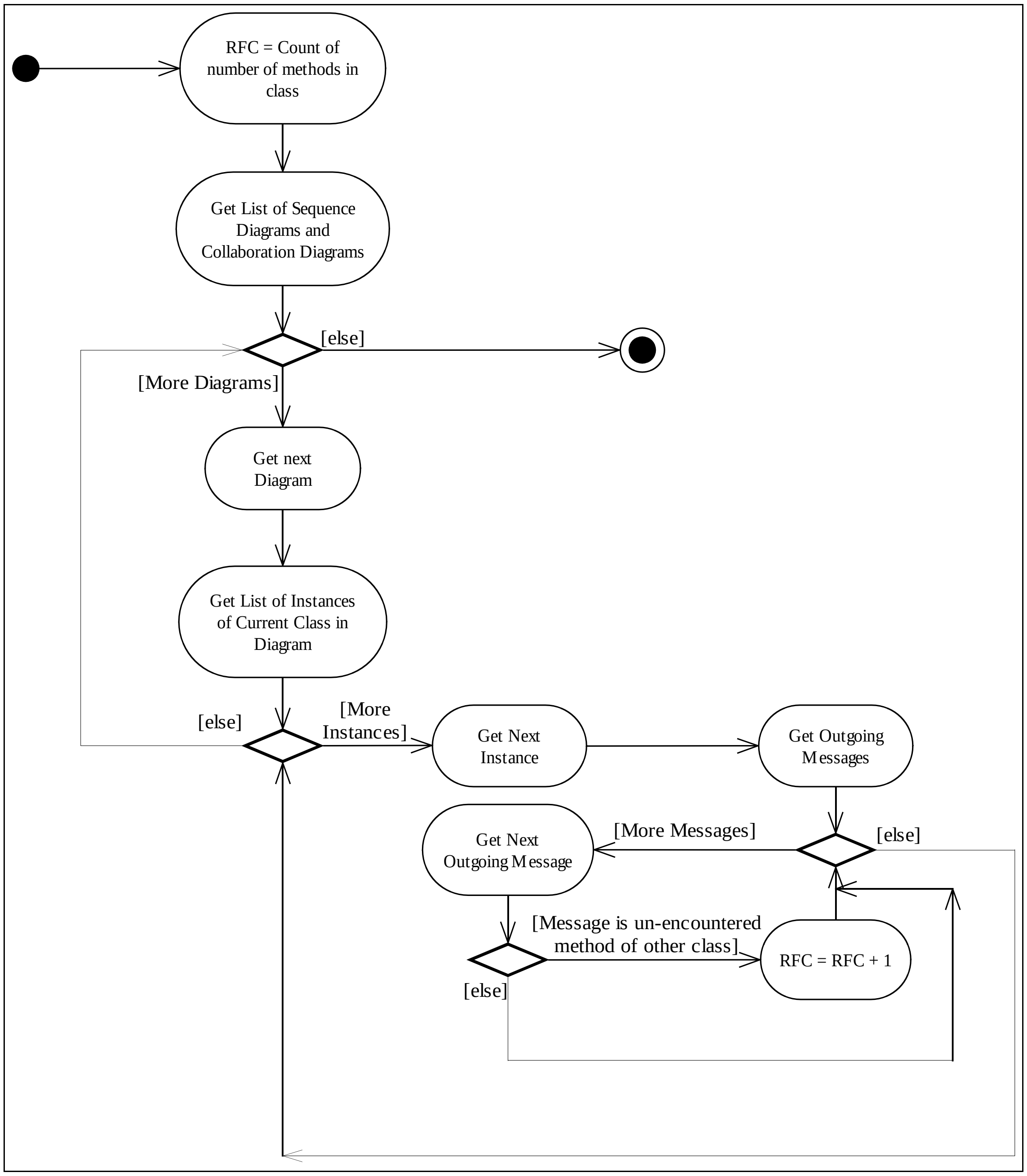

RFC = Count of number of methods in class
Get List of Sequence Diagrams and Collaboration Diagrams
[else]
[More Diagrams]
Get next Diagram
Get List of Instances of Current Class in Diagram
[else]
[More Instances]
Get Next Instance
Get Outgoing Messages
[More Messages]
[else]
Get Next Outgoing Message
[Message is un-encountered method of other class]
RFC = RFC + 1
[else]

Figure 42 - Activity Diagram for calculating CBO

Activity Diagram for Calculating LCOM

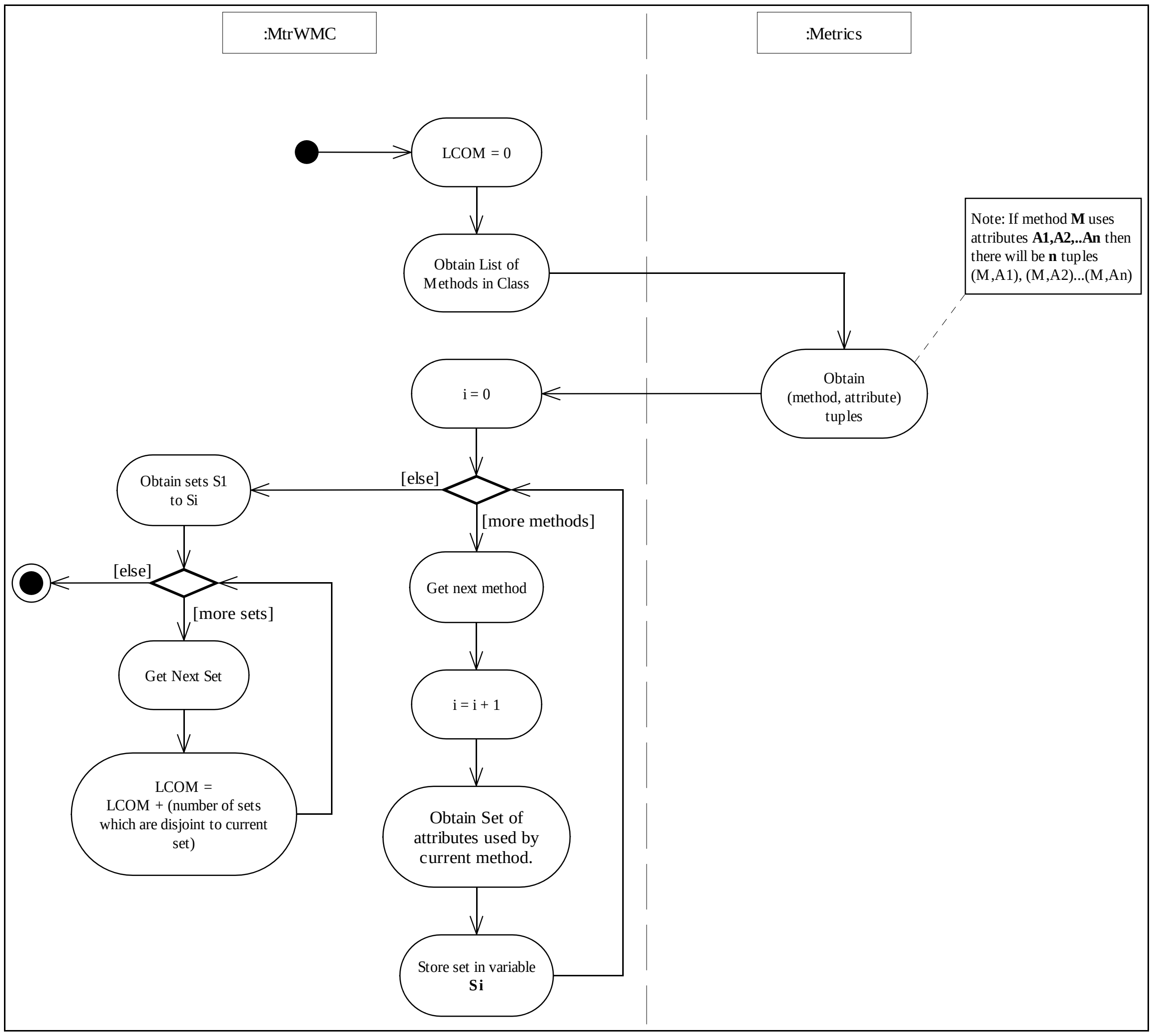

:MtrWMC
:Metrics
LCOM = 0
Obtain List of Methods in Class
Note: If method M uses attributes A1,A2,..An then there will be n tuples (M,A1), (M,A2)...(M,An)
Obtain (method, attribute) tuples
i = 0
[else]
[more methods]
Obtain sets S1 to Si
Get next method
[else]
[more sets]
Get Next Set
i = i + 1
LCOM = LCOM + (number of sets which are disjoint to current set)
Obtain Set of attributes used by current method.
Store set in variable Si

Figure 43 - Activity Diagram for calculating LCOM

Activity Diagram for Calculating CC1

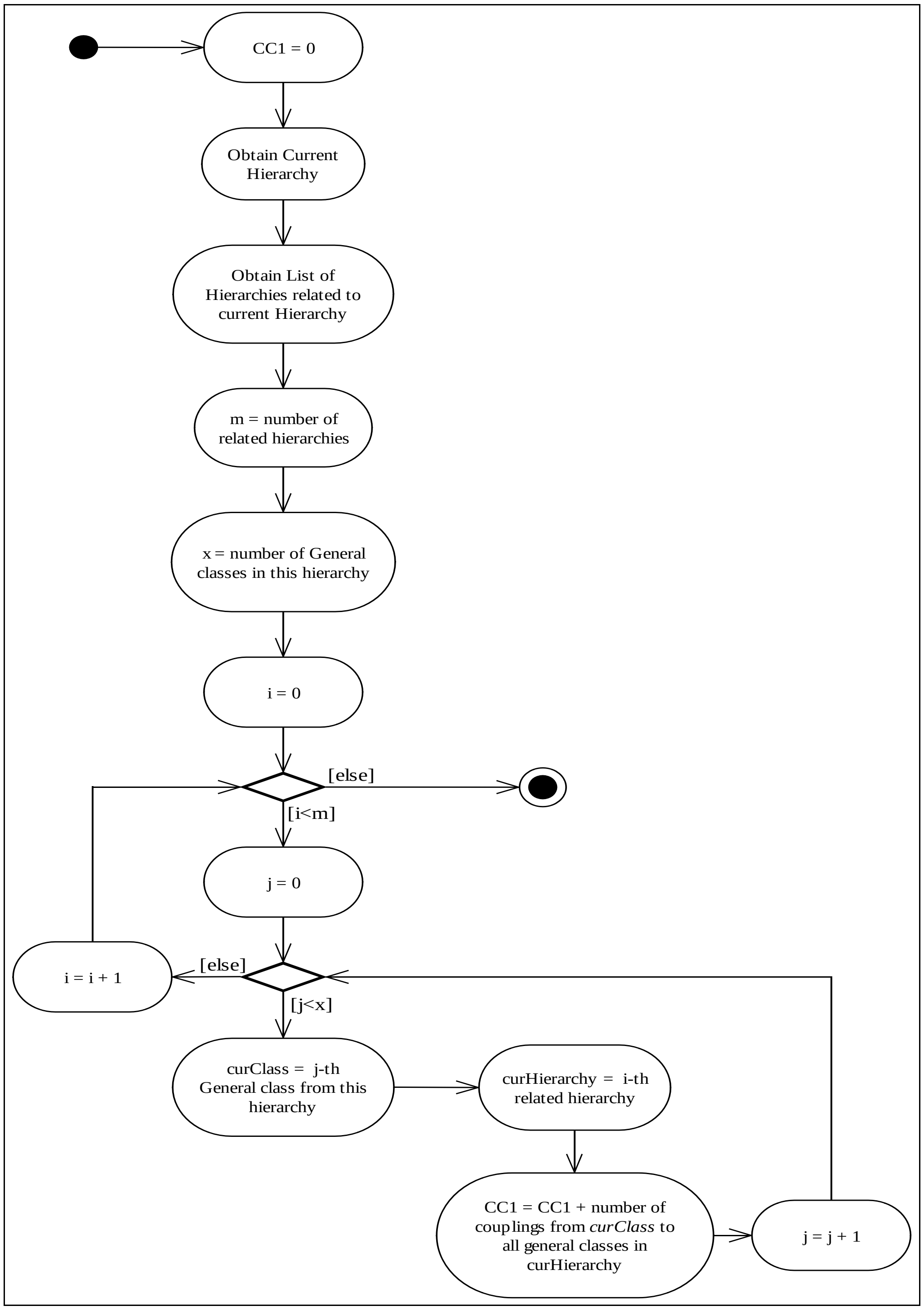

CC1 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies related to current Hierarchy
m = number of related hierarchies
x = number of General classes in this hierarchy
i = 0
[else]
[i<m]
j = 0
i = i + 1
[else]
[j<x]
curClass = j-th General class from this hierarchy
curHierarchy = i-th related hierarchy
CC1 = CC1 + number of couplings from *curClass* to all general classes in curHierarchy
j = j + 1

Figure 44 - Activity Diagram for Calculating CC1

Activity Diagram for Calculating CC2

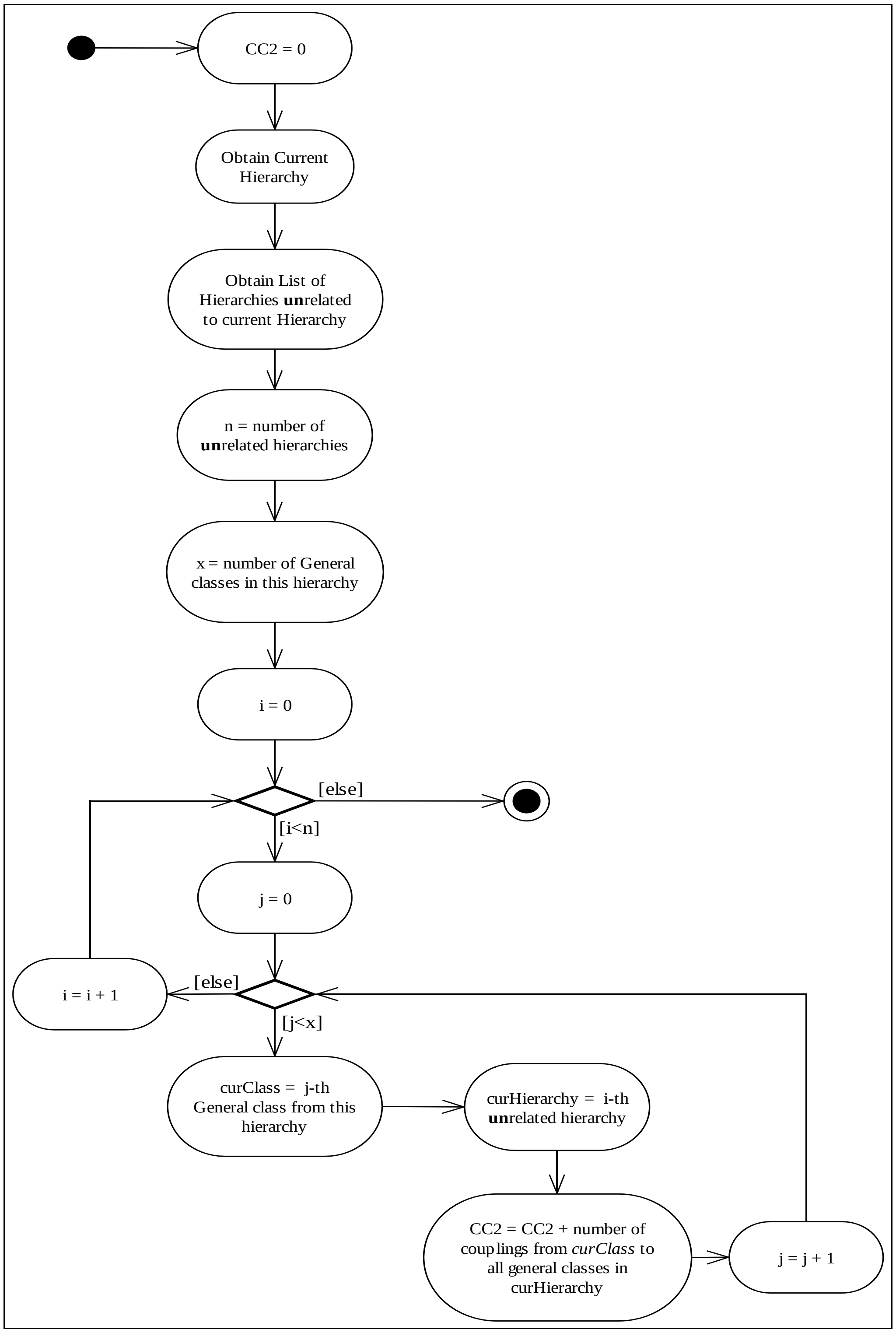

CC2 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies **un**related to current Hierarchy
n = number of **un**related hierarchies
x = number of General classes in this hierarchy
i = 0
[else]
[i<n]
j = 0
i = i + 1
[else]
[j<x]
curClass = j-th General class from this hierarchy
curHierarchy = i-th **un**related hierarchy
CC2 = CC2 + number of couplings from *curClass* to all general classes in curHierarchy
j = j + 1

Figure 45 - Activity Diagram for calculating CC2

Activity Diagram for Calculating CC3

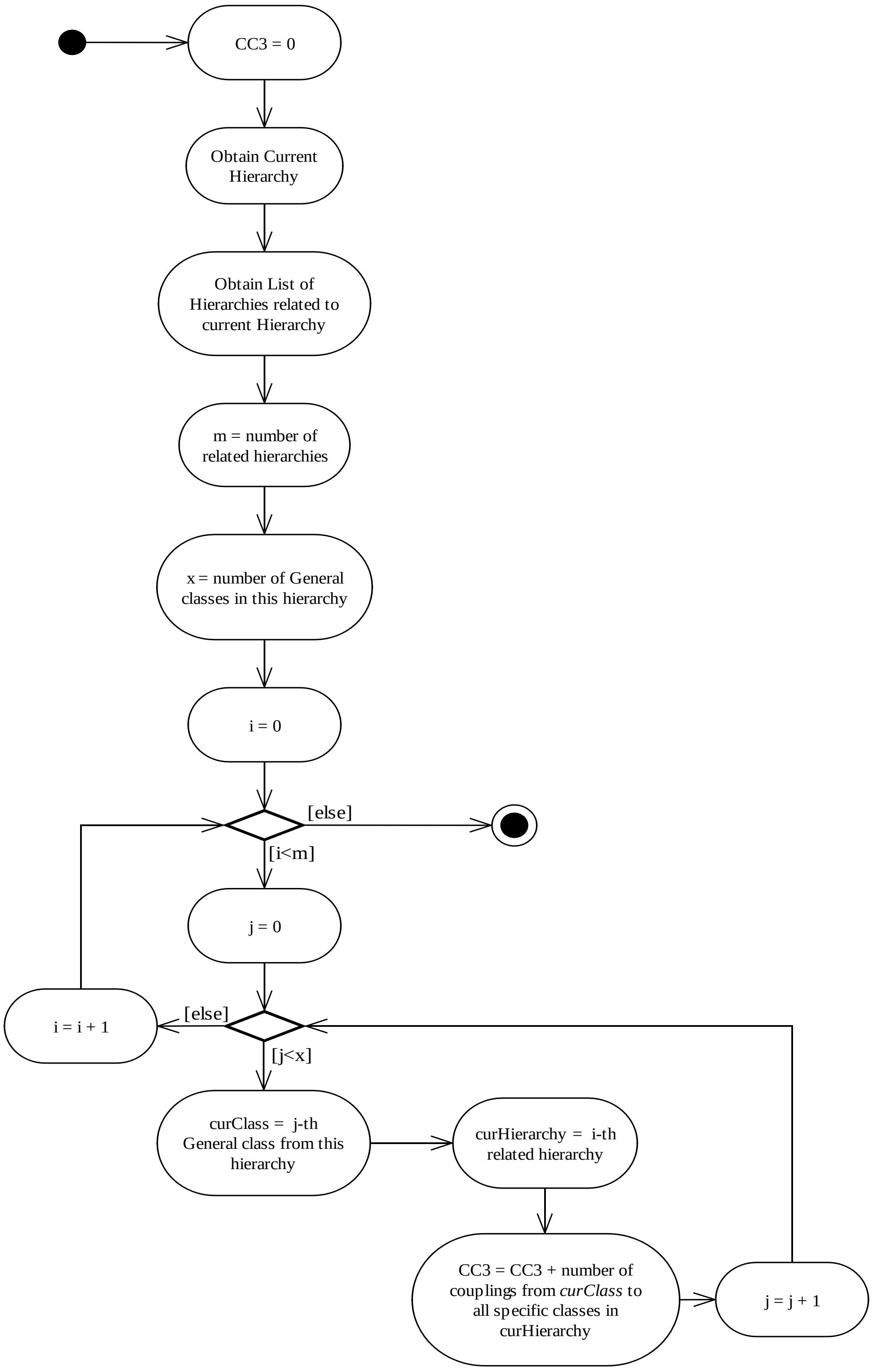

CC3 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies related to current Hierarchy
m = number of related hierarchies
x = number of General classes in this hierarchy
i = 0
[else]
[i<m]
j = 0
i = i + 1
[else]
[j<x]
curClass = j-th General class from this hierarchy
curHierarchy = i-th related hierarchy
CC3 = CC3 + number of couplings from *curClass* to all specific classes in curHierarchy
j = j + 1

Figure 46 - Activity Diagram for calculating CC3

Activity Diagram for Calculating CC4

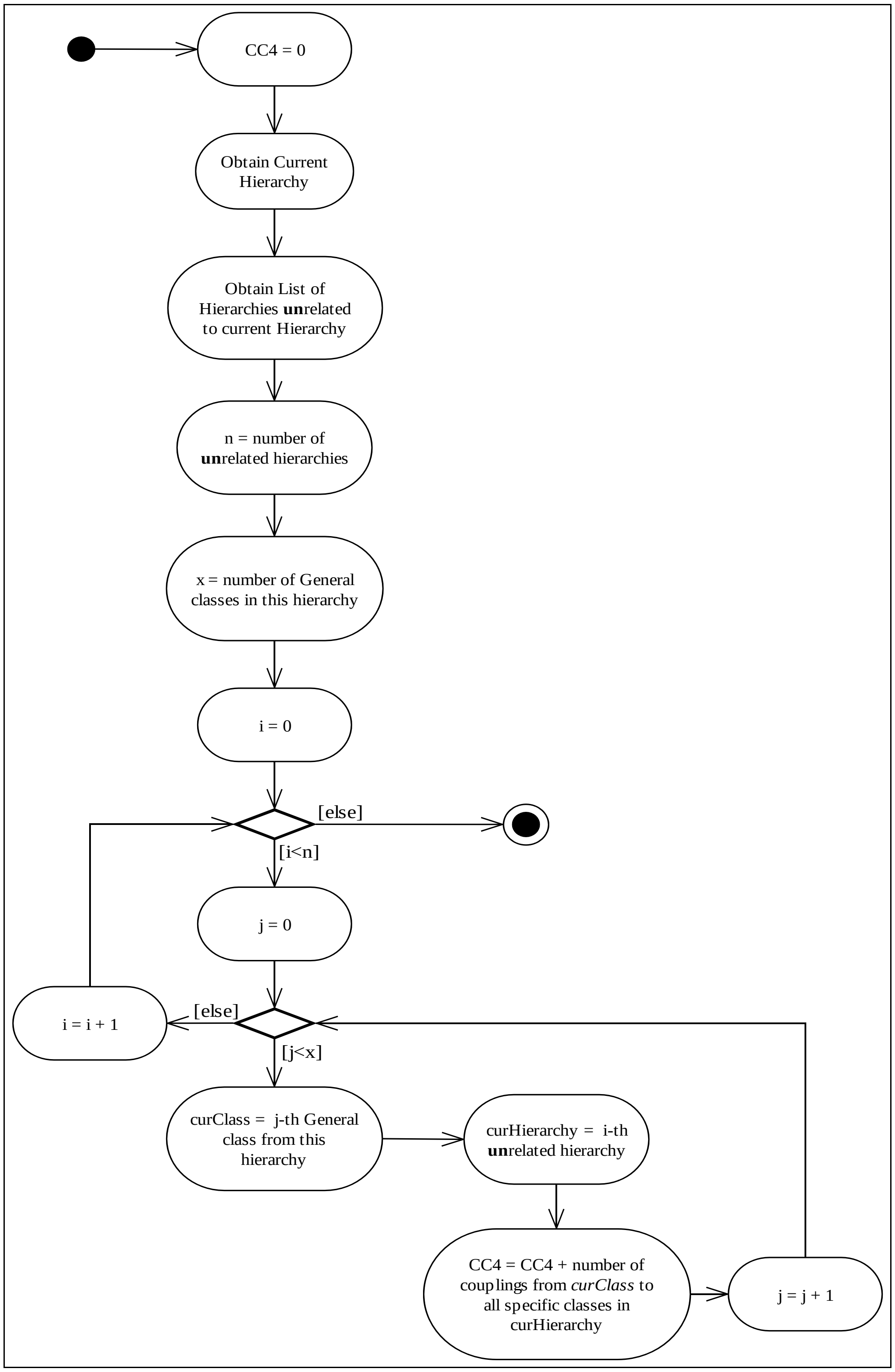

CC4 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies **un**related to current Hierarchy
n = number of **un**related hierarchies
x = number of General classes in this hierarchy
i = 0
[else]
[i<n]
j = 0
i = i + 1
[else]
[j<x]
curClass = j-th General class from this hierarchy
curHierarchy = i-th **un**related hierarchy
CC4 = CC4 + number of couplings from *curClass* to all specific classes in curHierarchy
j = j + 1

Figure 47 - Activity Diagram for calculating CC4

Activity Diagram for Calculating CC5

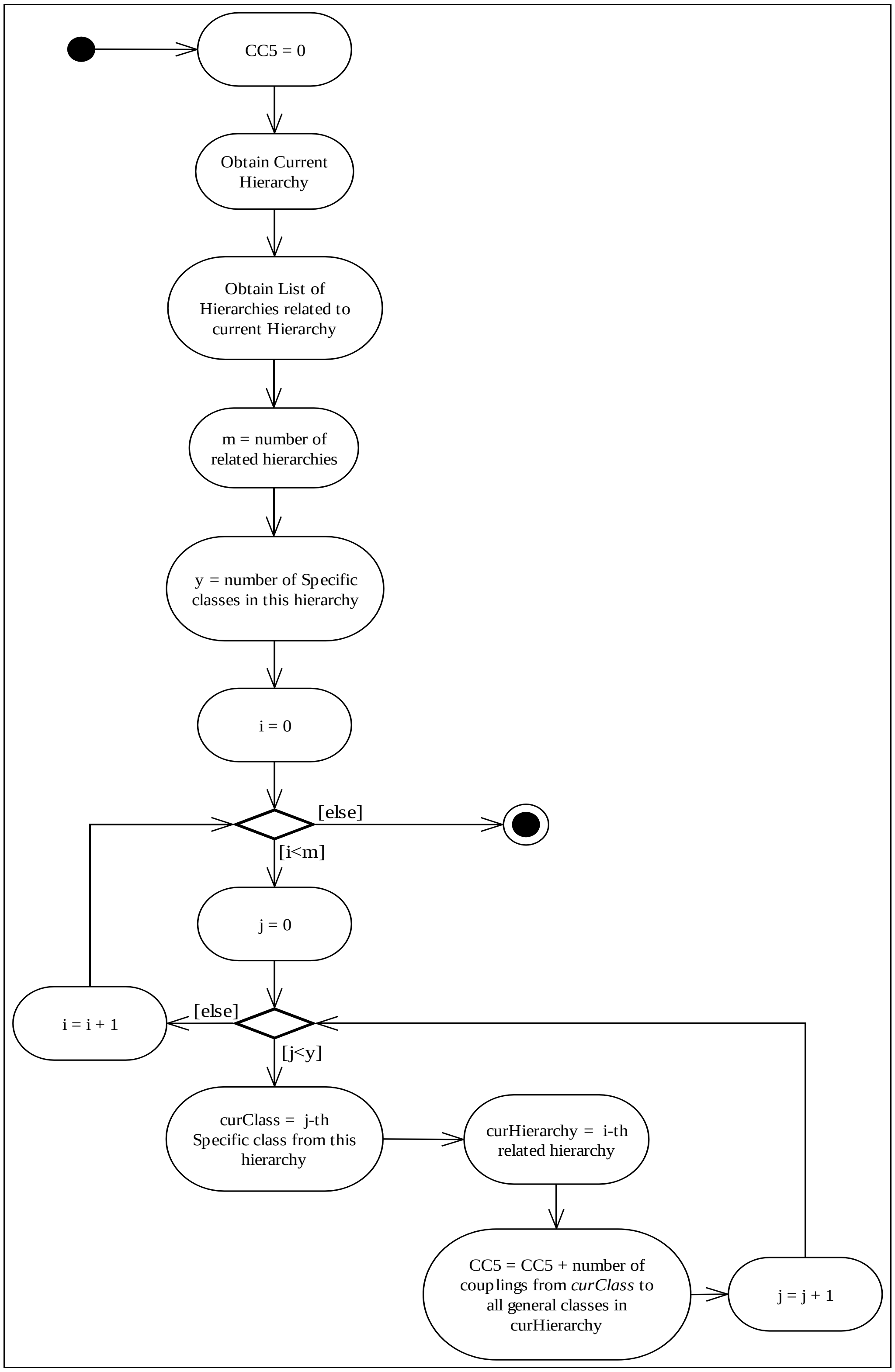

CC5 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies related to current Hierarchy
m = number of related hierarchies
y = number of Specific classes in this hierarchy
i = 0
[else]
[i<m]
j = 0
i = i + 1
[else]
[j<y]
curClass = j-th Specific class from this hierarchy
curHierarchy = i-th related hierarchy
CC5 = CC5 + number of couplings from curClass to all general classes in curHierarchy
j = j + 1

Figure 48 - Activity Diagram for calculating CC5

Activity Diagram for Calculating CC6

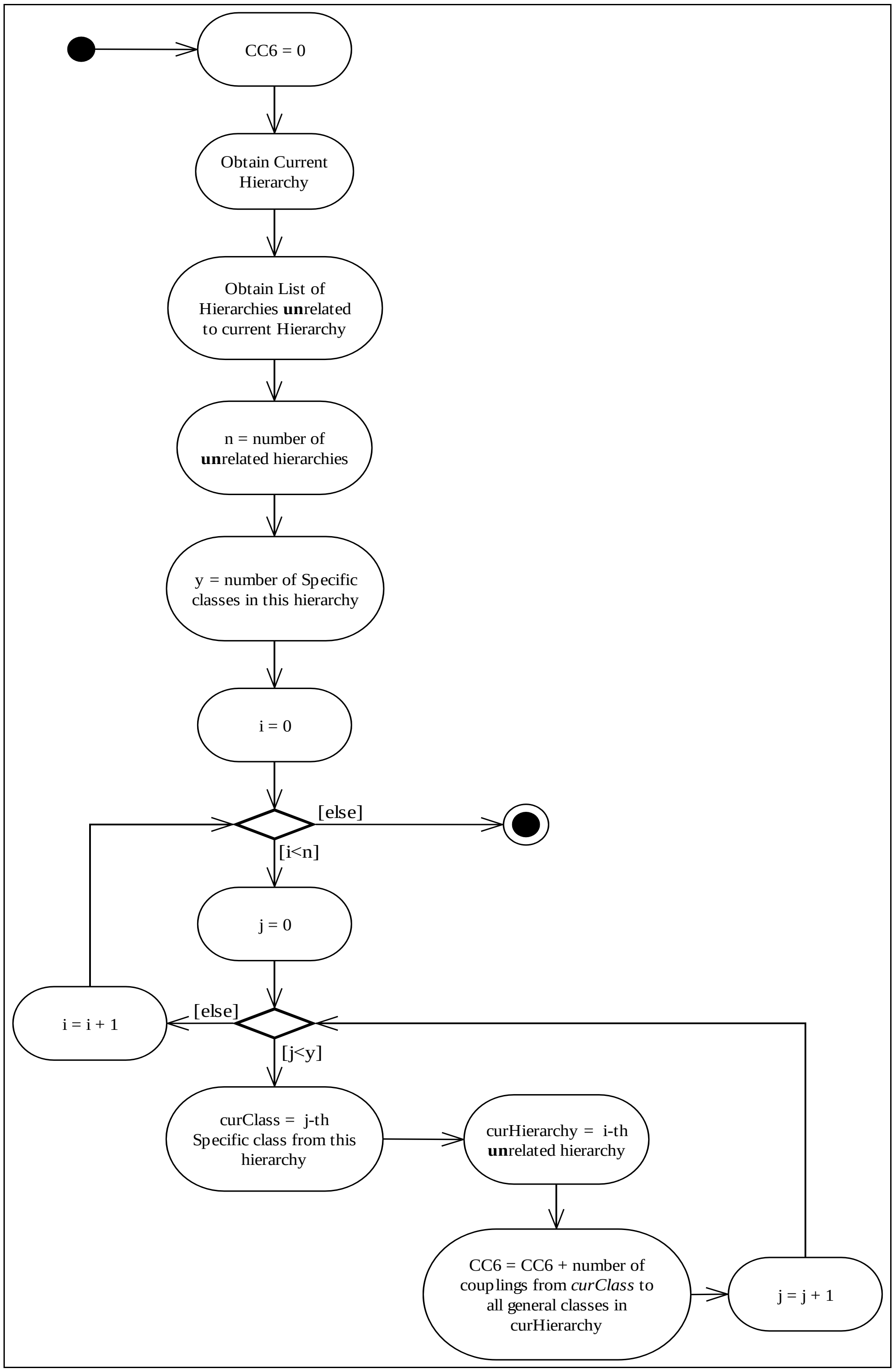

CC6 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies **un**related to current Hierarchy
n = number of **un**related hierarchies
y = number of Specific classes in this hierarchy
i = 0
[else]
[i<n]
j = 0
i = i + 1
[else]
[j<y]
curClass = j-th Specific class from this hierarchy
curHierarchy = i-th **un**related hierarchy
CC6 = CC6 + number of couplings from *curClass* to all general classes in curHierarchy
j = j + 1

Figure 49 - Activity Diagram for calculating CC6

Activity Diagram for Calculating CC7

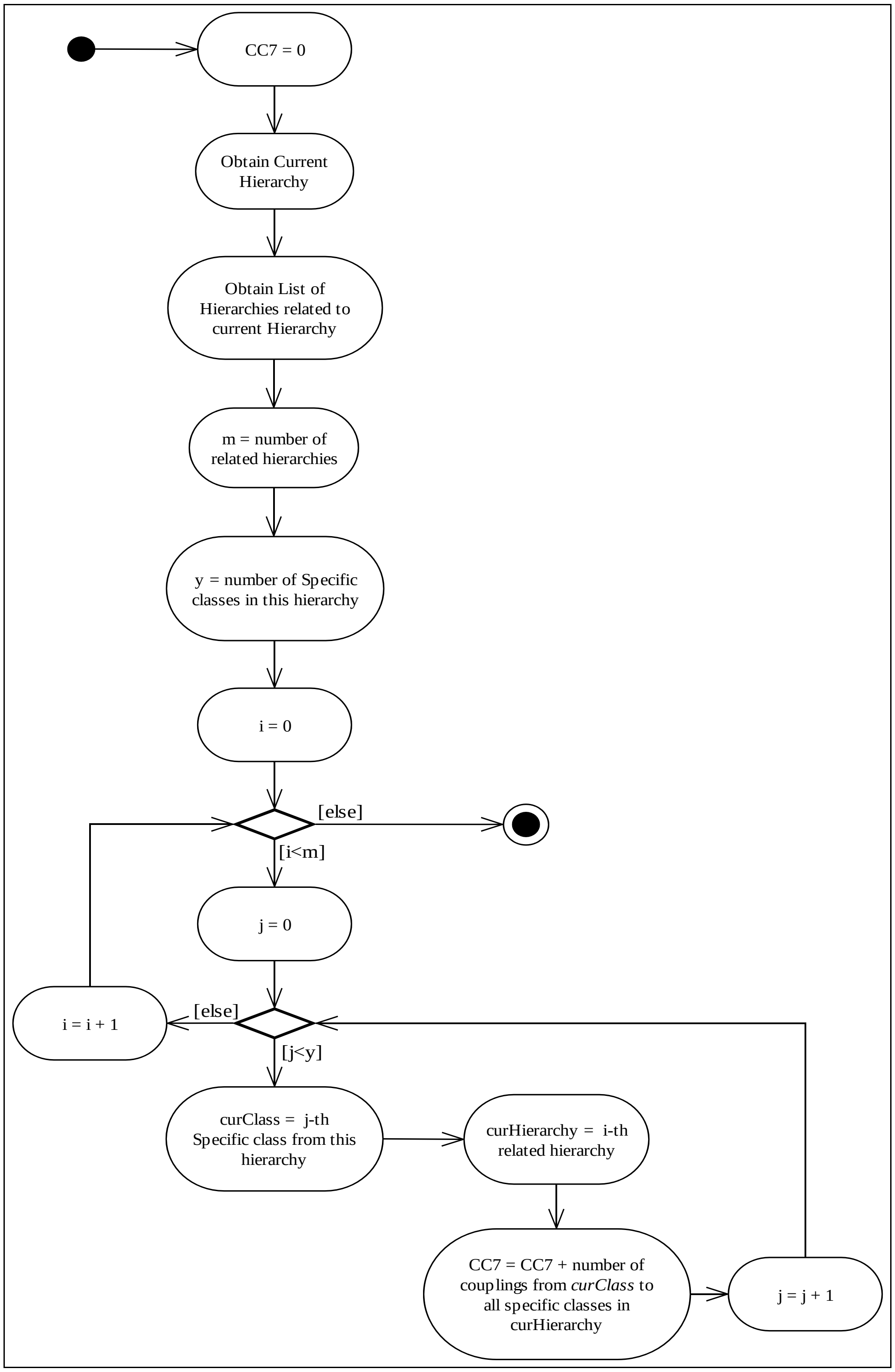

CC7 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies related to current Hierarchy
m = number of related hierarchies
y = number of Specific classes in this hierarchy
i = 0
[else]
[i<m]
j = 0
[else]
i = i + 1
[j<y]
curClass = j-th Specific class from this hierarchy
curHierarchy = i-th related hierarchy
CC7 = CC7 + number of couplings from *curClass* to all specific classes in curHierarchy
j = j + 1

Figure 50 - Activity Diagram for calculating CC7

Activity Diagram for Calculating CC8

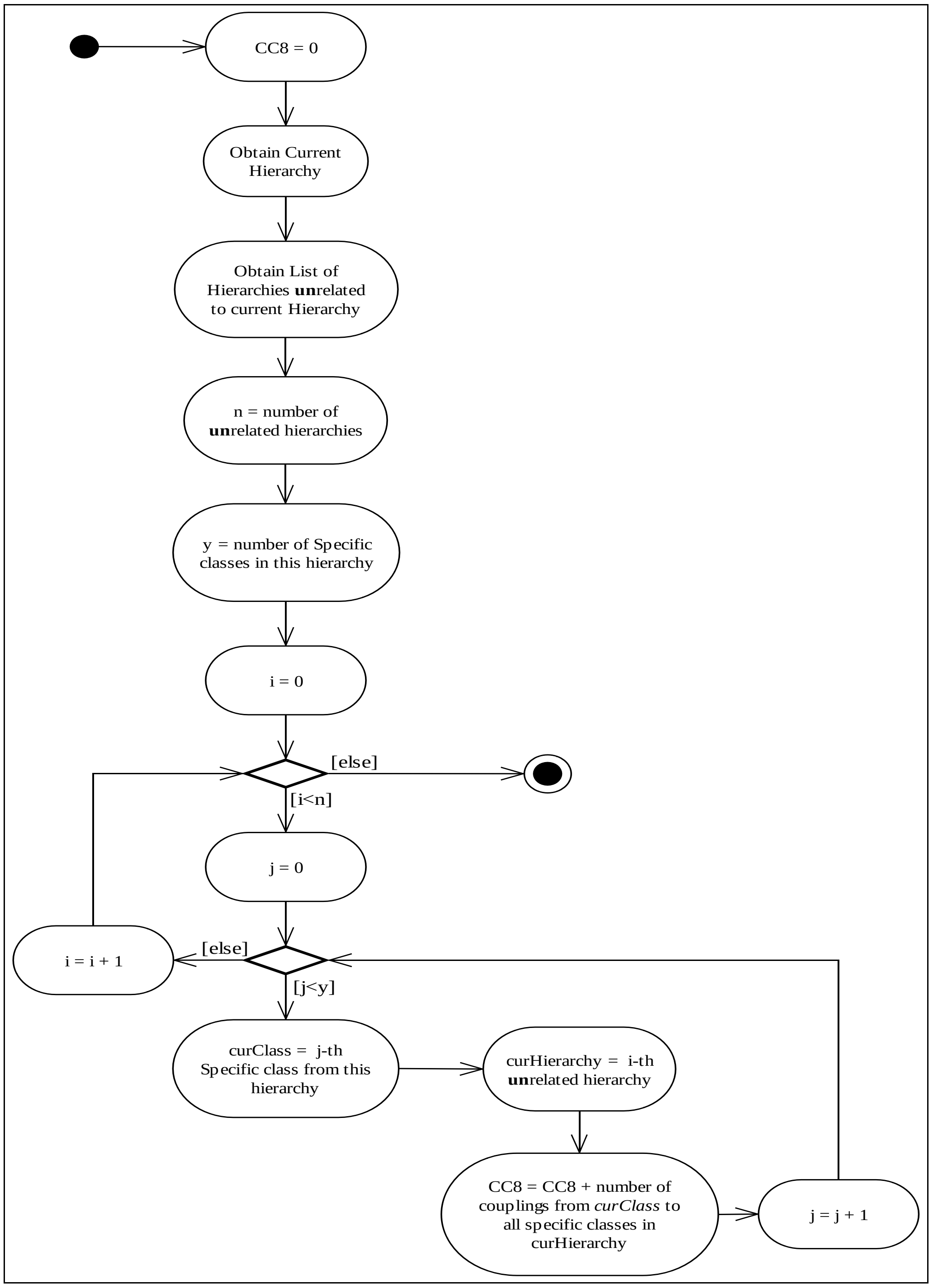

CC8 = 0
Obtain Current Hierarchy
Obtain List of Hierarchies **un**related to current Hierarchy
n = number of **un**related hierarchies
y = number of Specific classes in this hierarchy
i = 0
[else]
[i<n]
j = 0
i = i + 1
[else]
[j<y]
curClass = j-th Specific class from this hierarchy
curHierarchy = i-th **un**related hierarchy
CC8 = CC8 + number of couplings from *curClass* to all specific classes in curHierarchy
j = j + 1

Figure 51 - Activity Diagram for calculating CC8

Appendix B: Bibliography

1. ABO97 - A Data Model for Object Oriented Designs, Joe Raymond Abounader, David Alex Lamb" "Queen's University, Kingston Ontario"
2. ALH98 - UML in a Nutshell, Sinan Si Alhir, O'Reilly
3. ARI96 - Ariane 5 Flight 501 failure report, Inquiry Board, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html
4. BAN97 - Assessing Quality of Object Oriented Designs using a Hierarchical Approach, Bansiya J., University of Alabama
5. BCS00 - http://www.dur.ac.uk/~dcs0elb/reuse, BCS Software Reuse Specialist Group BCS
6. BEN99, Object-Oriented Analysis and Design using UML "Simon Bennett, Steve Mc.Robb, Ray Farmer", Mc Graw Hill
7. BER00- Metrics for Object-Oriented Software Engineering, Edward V. Bernard, www.toa.com/pub/moose.htm
8. BOO94 - Object Oriented Analysis and Design with Applications, Grady Booch, Addisson Wesley
9. CAC97 - Introduction to Software Engineering (Course Notes), Dr. Ernest Cachia, University of Malta
10. CAR97- Assessing Design Quality From a Software Architectural Perspective, "Carriere J, Kazman R.", Software Engineering Institute - Carnegie Mellon University
11. CHI94 - A Metrics Suite for Object-Oriented Design, "Chidamber, Shyam and Kemerer, Chri", IEEE Transactions on Software Engineering
12. CHU95, Towards a conceptual framework for object-oriented software metrics, "Churcher N.I., Shepper M.J. "
13. CNN01a - "Software, hydraulics blamed in Osprey crash", http://www.cnn.com/2001/US/04/05/arms.osprey.02/index.html, CNN
14. CRO79 - Quality is Free: The Art of Making Quality Certain, "Crosby, P.B.", McGraw-Hill

15. GIR93 - Increasing design quality and Engineering Productivity Standards through design reuse, "Girczyc E., Carlson S."
16. GUI99 - Detecting Defects in Object-Oriented Designs, Guilherme H. Travassos et al, ACM
17. HEN96 - Software Metrics, Henderson-Sellers
18. HIT96 - Chidamber and Kemerer's Metrics Suite – a measurement perspective, "Hitz M. , Montazeri B."
19. JAC90 - Risks in Medical Electronics, Jonathan Jacky, ACM
20. KAN99 - Metrics and Models In Software Quality Engineering, Dr. Stephen H. Kan, Addisson Wesley
21. KEL97 - Object-Oriented Design Quality, Rudolf K. Keller and Alistair Cockburn
22. LEW91 - An empirical Study of the OO Paradigm and Software Reuse, John A. Lewis et al, ACM
23. MCL00 - Java and XML, "Brett McLaughlin, Mike Loukides", O'Reilly
24. MIL98 - In Search of the Holy Grail, Don Mills, Software Education Associates
25. PRE97 - Software Engineering - A Practitioner's Approach, Roger S. Pressman, Mc Graw Hill
26. PRI97 - Analyzing and Measuring Reusability in OO Designs, "Price M.W., Demurijian S.A.", ACM
27. ROB99 – Cognitive Support Features for Software Development Tools, Robbins J.E., University of California
28. ROS97a - Software Quality Metrics for Object-Oriented Environments, Rosenberg L. and Hyatt L., NASA Software Assurance Technology Center
29. ROS97b - Applying and Interpreting Object-Oriented Metrics, Rosenberg L.
30. ROS98 - Software Metrics and Reliability, "Rosenberg L., Hammer Ted, Shaw Jack"
31. SHE95 - Foundations of Software Measurement, Martin Shepperd, Prentice Hall
32. WWW01 - XML in 10 Points, W3C, http://www.w3.org/XML/1999/XML-in-10-points
33. YEH97 - Object-Oriented Design Quality, Shai Ben-Yehuda

User Documentation

Introduction

This document explains how to use the features provided by the software quality measurement tool implemented as part of this project. It is impractical to re-print the theoretical basis behind each metric calculated by the tool but a basic description as well as advice on how to interpret graphs will be given. It is recommended that the user read up on the theory behind the tool from the technical documentation.

Using ArgoUML

It is beyond the scope of this document to teach the reader how to use ArgoUML to create and edit UML diagrams. There is a user manual available for ArgoUML at http://www.argouml.org.

Features Provided

The tool provides a number of features that come together to allow the user to gauge the quality of object oriented designs. These are as follows:

Metric Calculation

The tool's functionality centers around the calculation of metrics on UML diagrams. There are two suites of tools that between them contain fourteen different metrics. The *Structural Metrics* suite contains metric that measure the different structures and features present in object-oriented designs (*classes*, *messages*, *coupling*, *cohesion*, and *inheritance*) in order to evaluate a variety of attributes – there are six metrics in this suite. The *Reuse Metrics* suite of metrics consists of eight coupling-based metrics aimed specifically at evaluating the reusability of a design.

For a detailed explanation of the metrics, please refer to the technical manual.

Collection of Metric Information

Some metrics need more information than is available in UML diagrams. The tool provides the functionality for collecting this information.

Metrics Repository

The tool automatically maintains a metrics repository in which statistical information is accumulated over time as the tool is used to analyze different projects. All the user has to do is tell the tool to add a project to a repository and it will be done. The user interface provides functionality for viewing the information stored in the metrics repository.

Function Points Calculation

Function points were developed by Allan Albrecht while working at IBM in the late-1970s. They are offer a way of calculating the size of a system based on the functionality it offers rather than on the size of the resulting code. This has significant advantages over lines-of-code (LOC) measurements since the same system developed in different languages will have different LOC but the same function points. What are the relevance of function points in this project?

It is usually useless to compare projects of different sizes together when it comes to metrics. The tool allows users to compare different projects based on their function points. So if I were designing a large application with 500 function points, I would look up projects with similar function point readings in the repository for benchmarking. The tool automates function point calculation by asking the user to answer questions about the project as described in the original function points method.

Effort Prediction

Having calculated the function points of a system, the user can enter productivity information such as the number of person months it takes to develop one function point, and how much each person month costs. The tool will then provide simple estimates on the length of time the project will take and how much it will cost. This information will be stored when you save the project for future analysis.

Graphical Representation

For each metric, the tool provides one or more graphs that will help the user to visualize the quality situation from the perspective of that particular metric. Each graph is interactive in the sense that it provides project-specific information (such as a list of classes represented by a point on a graph) and in that it provides information on how it should be interpreted. The interpretation advice is not ‘intelligent’ advice. It is simply hard-coded text and does not change from one project to another.

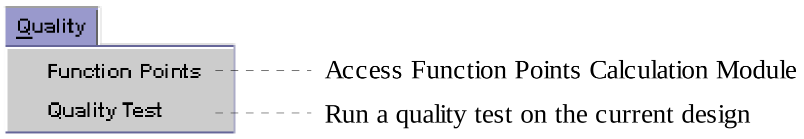
Accessing the Quality Measurement Features

Since the tool is an extension to an existing UML editor, it is worth noting how one can access the features described above.

The *Quality* Menu

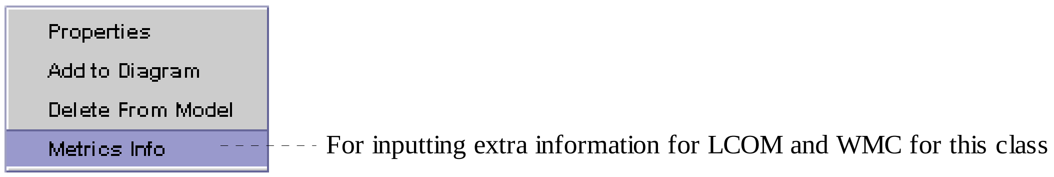
A “Quality” pull-down menu was added to ArgoUML’s menu bar. This provides 2 options:

3. Accessing the function points calculation module and
4. Running a quality test on the current design. This will bring up the results and allow the user to compare with projects in the repository as well as view graphical representations of the data.


Quality
Function Points
Quality Test
Access Function Points Calculation Module
Run a quality test on the current design

The Class Popup Menu

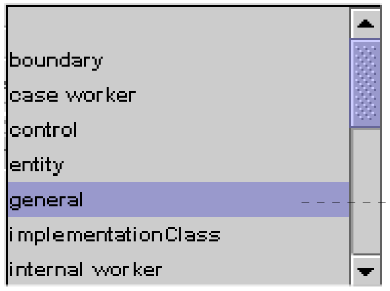
This menu is a feature of ArgoUML but a new option was added to it for the convenience of the tool. A user can right-click on a class and the following popup menu will be displayed:


Properties
Add to Diagram
Delete From Model
Metrics Info
For inputting extra information for LCOM and WMC for this class

The first 3 options were standard ArgoUML options but the last one was added in order to access a module that captures extra information needed for calculating the LCOM and WMC metrics on that class. This is explained in detail later.

The <<General>> Stereotype for Classes

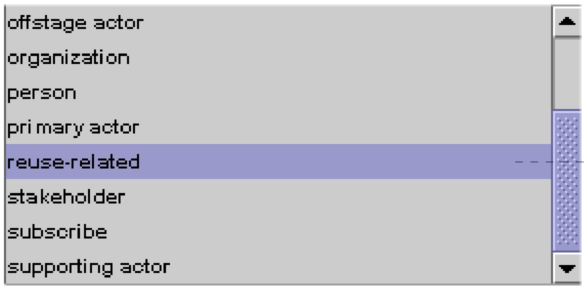
A new stereotype was defined for classes as part of the methodology for reusability metrics. You basically use it to indicate which classes are meant to be reused in other systems.



Stereotype for indicating a General (Reusable) Class

The <<Reuse-Related>> Stereotype for Associations

A new stereotype was defined for use with associations in class diagrams. This stereotype is used to show that two class hierarchies (defined by the two classes at the ends of the association as their roots) are meant to be reused together in future systems. This need not be because they depend on each other, it could be that they are used in the same domains or have complimentary features. It is entirely up to the designer to make this decision.



Stereotype for depicting related hierarchies

Running a Quality Test

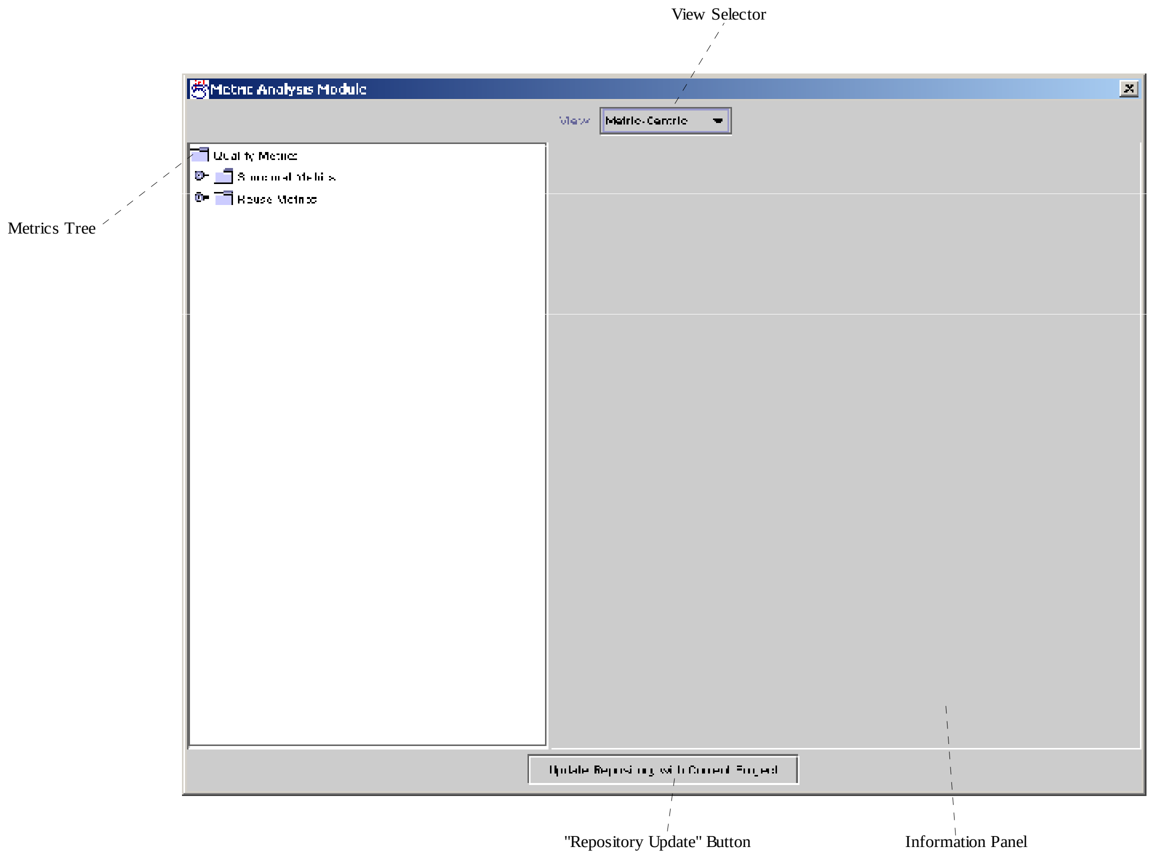
A quality test analyzes the current project using the 16 metrics named above and displays the results to the user in the **Metrics Analysis Module**. The metrics analysis module gives the user access to information about metrics, quality attributes, results, the metrics repository and graphs. Running a quality test can be done by selecting **Quality → Quality Test** from the menu bar.

The Metrics Analysis Module

The *Metrics Analysis Module* is the module responsible for displaying metric-related information. This includes:

1. Descriptions of all the metrics
2. Descriptions of the quality attributes each metric evaluates
3. Suggestions on how to interpret results
4. Metric results of the current project
5. Metric results from other projects in the Metric Repository
6. Graphical representation of results

An Overview of the *Metrics Analysis Module* Interface



The Metrics Analysis Module user interface consists of 4 main components:

1. **The Metrics Tree –** This tree displays metrics in a hierarchical view depending on the chosen view. When the user navigates through the tree, information on the *Information Panel* will be updated accordingly.

2. **The Information Panel –** Serves to give detailed information about the currently selected node in the *Metrics Tree*.

3. **The View Selector –** Lets the user select which view he/she wants the metrics-tree to represent. This will be explained below.

4. **The "Repository Update" Button –** This button lets the user add the current project to the metrics repository. Once in the repository, a summary of the metrics calculated on the project will be available in the repository so that future projects can be compared to it.

The Metrics Tree and the View Selector

There are two views that can be represented by the metrics tree:

1. **Metric-Centric View –** The tree is organized around metrics and the different suites they fall under. There are two suites of metrics: *Structural Metrics* and *Reuse Metrics*. This view is to be used when the user is very familiar with the metrics and what they mean. Each metric is represented by a child-node of it's parent suite. Also, each metric node has a "Project Repository" child-node, which displays different projects in the repository that this metric has been calculated on. This way, the user has immediate and transparent access to the repository with data filtered to the chosen metric.

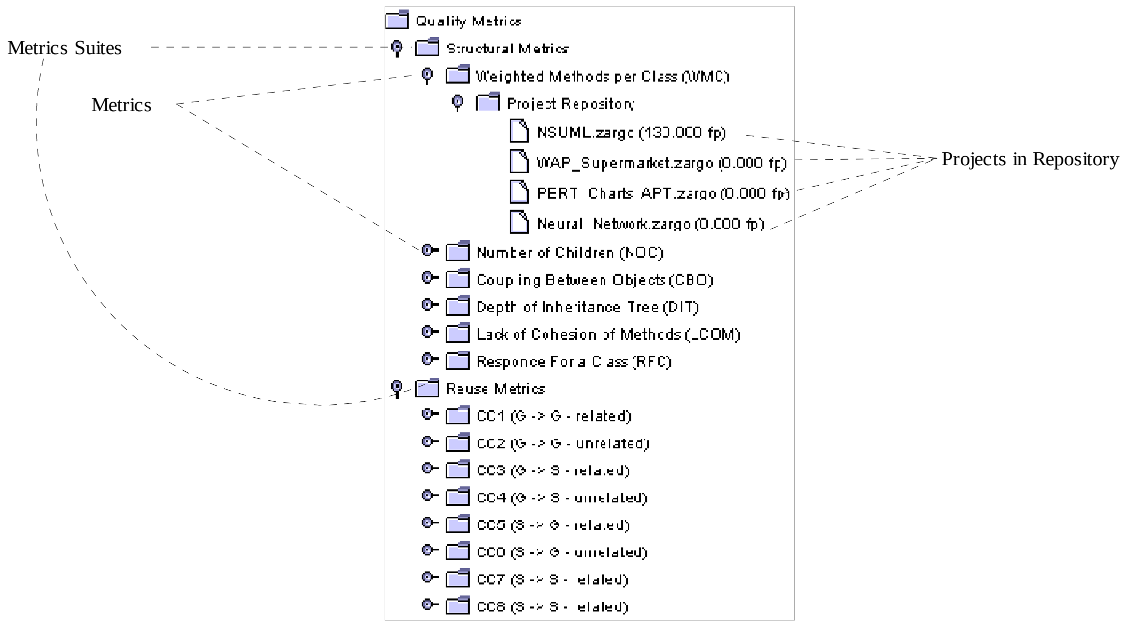


Figure 52 - A snapshot of the metrics tree under the "Metric-Centric" View

2. **Attribute-Centric View –** The tree is organized with quality attributes as the parent nodes. The user can choose a quality attribute node and its child-nodes will be metrics that evaluate that quality attribute. This way user can proceed with quality analysis from a quality-attribute viewpoint.

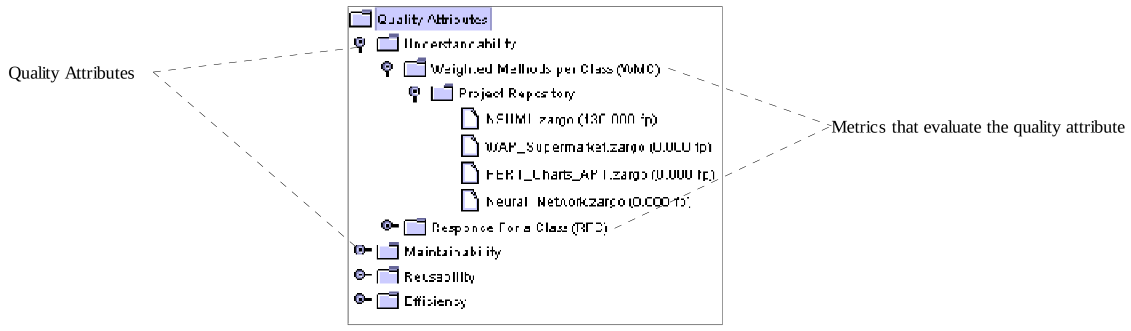

Quality Attributes
Quality Attributes
Understandability
Weighted Methods per Class (WMC)
Project Repository
NSUML.zargo (130.000 fp)
WAP_Supermarket.zargo (0.000 fp)
PERT_Charts_APT.zargo (0.000 fp)
Neural_Network.zargo (0.000 fp)
Metrics that evaluate the quality attribute
Responce For a Class (RFC)
Maintainability
Reusability
Efficiency

Figure 53 - A snapshot of the metrics tree in "Attribute Centric" View

Viewing Metric Results

The user can view all the information related to a metric by selecting the node representing the metric in the metrics tree. This results in the information panel changing its content to display information about the selected metric in 4 tabs:

1. **Metric Information –** Displays information about the metric and the quality attributes it evaluates.

2. **Project Summary –** A summary of the results of the metric on the current project. This tab also provides access to the graphs that are associated with the metric.

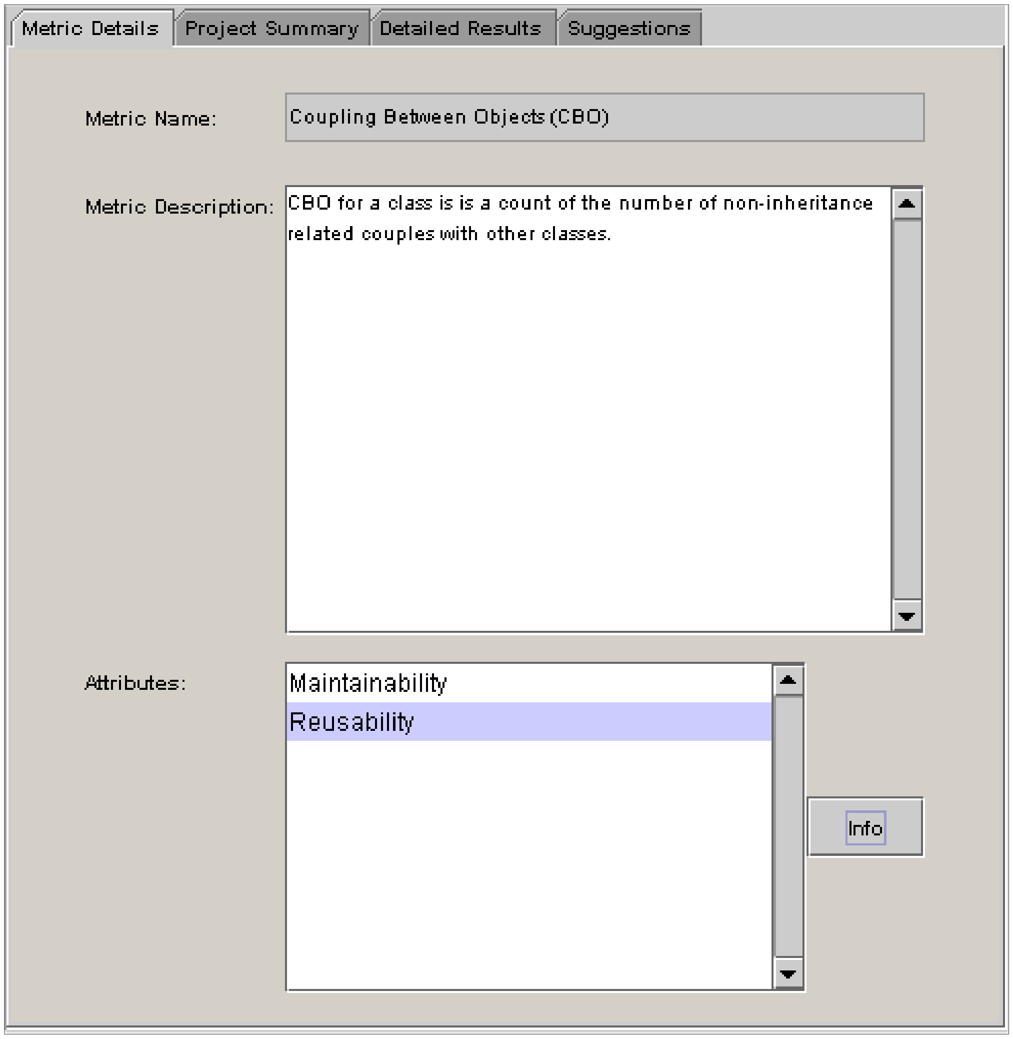
3. **Detailed Results –** Presents detailed metric results for this project in tabulated textual form.

4. **Suggestions –** General information on how to use and interpret this metric.

The *Metric Description* Tab

The *Metric Description* tab provides the following information:

1. Metric Name – The name of the metric
2. Metric Description – A description of the metric
3. Attributes – A list of attributes the metric evaluates. Selecting an attribute from the list and pressing the Info button will bring up a dialog that describes how the results of the metric could indicate the presence (or absence) of the selected quality attribute.

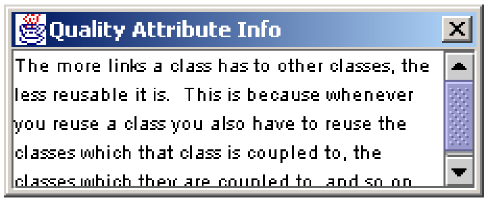


Metric Name: Coupling Between Objects (CBO)

Metric Description: CBO for a class is is a count of the number of non-inheritance related couples with other classes.

Attributes:
Maintainability
Reusability

Figure 54 - The *Metric Details* Tab



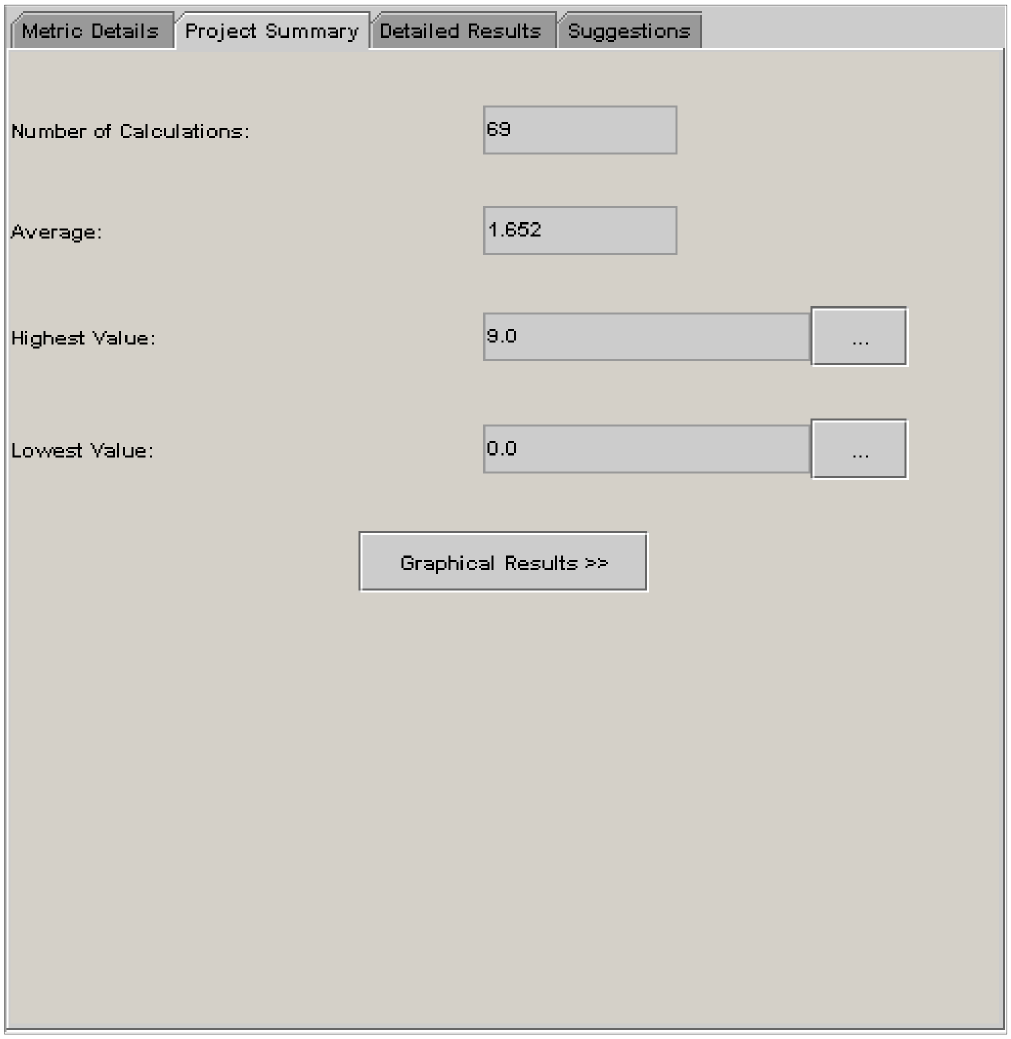
The more links a class has to other classes, the less reusable it is. This is because whenever you reuse a class you also have to reuse the classes which that class is coupled to, the classes which they are coupled to, and so on

Figure 55 - A dialog box describing how the selected metric evaluates the selected attributed.

The *Project Summary* Tab

The *Project Summary* tab provides the following information:

1. **Number of Calculations –** The number of times the metric was used in this project.
2. **Average –** The average value of this metric over all the classes in this project.
3. **Highest Value –** The highest value recorded for this metric in the current project.
4. **Lowest Value –** The lowest value recorded for this metric in the current project.
5. **Graphical Results –** Brings up a popup menu with a list of graphs available for the selected metric. A more detailed explanation of graphs will be given later on in this document.



Number of Calculations: 69

Average: 1.652

Highest Value: 9.0

Lowest Value: 0.0

Figure 56 - The *Project Summary* Tab

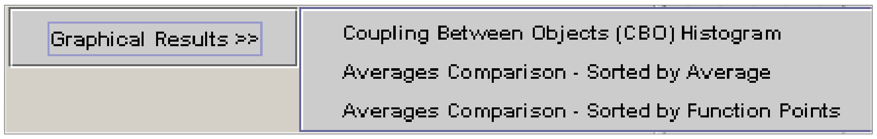
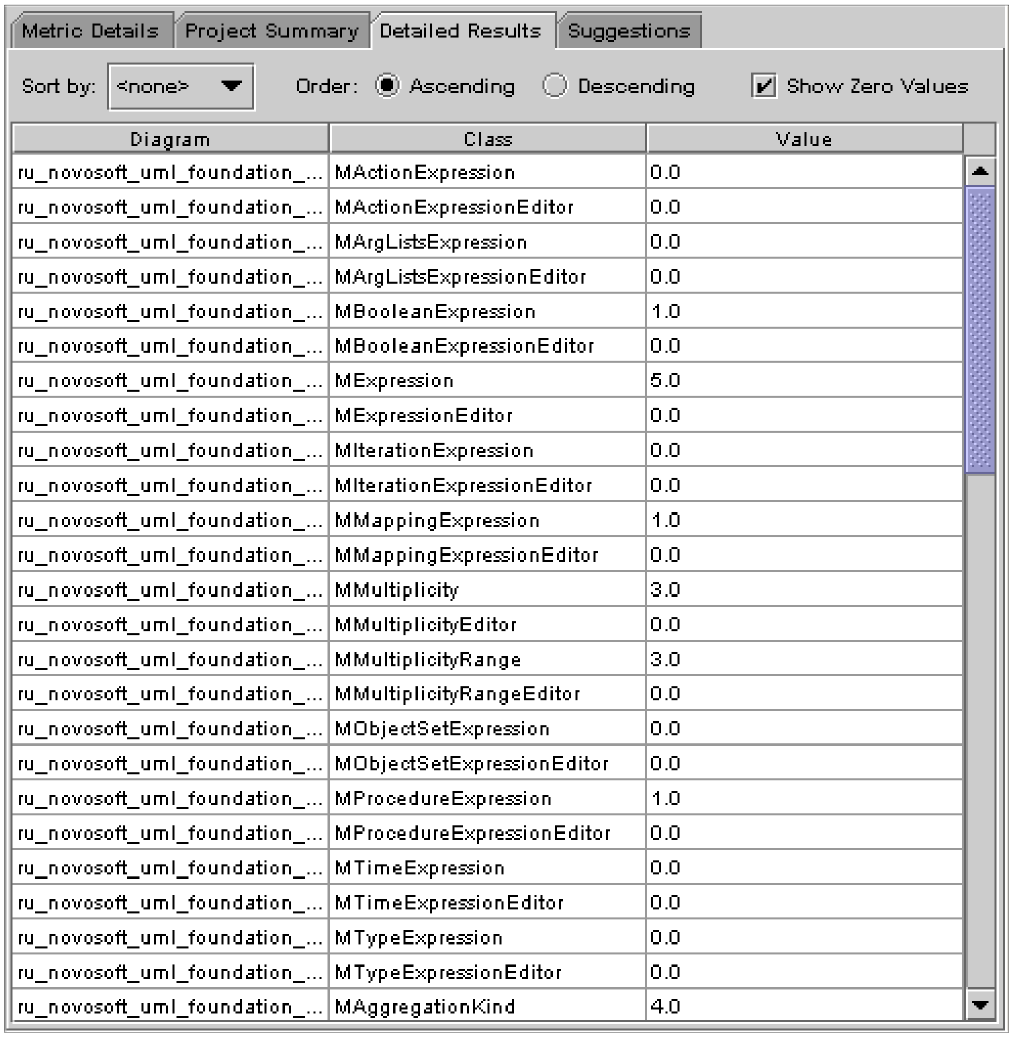

Graphical Results >>
Coupling Between Objects (CBO) Histogram
Averages Comparison - Sorted by Average
Averages Comparison - Sorted by Function Points

Figure 57 - Pressing the *Graphical Results* button will bring up a menu of available graphs for you to choose from

The *Detailed Results* Tab

The *Detailed Results* tab provides a full listing of the classes in the system and the result of the selected metric when applied to each class. The user can sort the values in the table by any of the three fields (*Diagram*, *Class*, *Value*). The user can also choose to eliminate zero-values from the table.



Sort by: <none>   Order: [✓ Ascending] Descending   [✓ Show Zero Values]

Diagram	Class	Value
ru_novosoft_uml_foundation_...	MActionExpression	0.0
ru_novosoft_uml_foundation_...	MActionExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MArgListsExpression	0.0
ru_novosoft_uml_foundation_...	MArgListsExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MBooleanExpression	1.0
ru_novosoft_uml_foundation_...	MBooleanExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MExpression	5.0
ru_novosoft_uml_foundation_...	MExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MIterationExpression	0.0
ru_novosoft_uml_foundation_...	MIterationExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MMappingExpression	1.0
ru_novosoft_uml_foundation_...	MMappingExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MMultiplicity	3.0
ru_novosoft_uml_foundation_...	MMultiplicityEditor	0.0
ru_novosoft_uml_foundation_...	MMultiplicityRange	3.0
ru_novosoft_uml_foundation_...	MMultiplicityRangeEditor	0.0
ru_novosoft_uml_foundation_...	MObjectSetExpression	0.0
ru_novosoft_uml_foundation_...	MObjectSetExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MProcedureExpression	1.0
ru_novosoft_uml_foundation_...	MProcedureExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MTimeExpression	0.0
ru_novosoft_uml_foundation_...	MTimeExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MTypeExpression	0.0
ru_novosoft_uml_foundation_...	MTypeExpressionEditor	0.0
ru_novosoft_uml_foundation_...	MAggregationKind	4.0

Figure 58 - The *Detailed Results* Tab

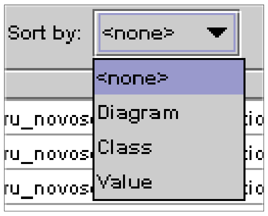
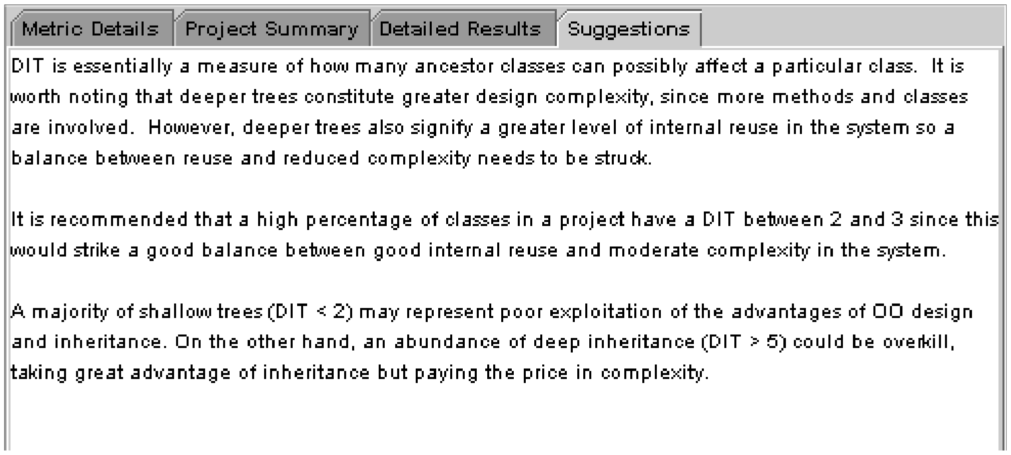


Figure 59 - The user can sort the rows in the table by any of the 3 fields

The *Suggestions* Tab

The suggestions tab provides the user with advice on how to interpret the possible values of the selected metric, stating desirable ranges of the metric and warning about the consequences of extreme values of the metric.



DIT is essentially a measure of how many ancestor classes can possibly affect a particular class. It is worth noting that deeper trees constitute greater design complexity, since more methods and classes are involved. However, deeper trees also signify a greater level of internal reuse in the system so a balance between reuse and reduced complexity needs to be struck.

It is recommended that a high percentage of classes in a project have a DIT between 2 and 3 since this would strike a good balance between good internal reuse and moderate complexity in the system.

A majority of shallow trees (DIT < 2) may represent poor exploitation of the advantages of OO design and inheritance. On the other hand, an abundance of deep inheritance (DIT > 5) could be overkill, taking great advantage of inheritance but paying the price in complexity.

Figure 60 - The suggestions tab for the DIT metric

A Quick Overview of Graphs

Introduction

Graphs provide the user with different views on the numeric results produced by the metrics over a design. Graphs for a metric can be accessed by clicking on the

Graphical Results >> button. This will bring up a menu that allows you to select one of the graphs available for the metric. Once you select a graph, a new window will be brought up with the graph taking up the major portion of the screen and a help button being included at the bottom.

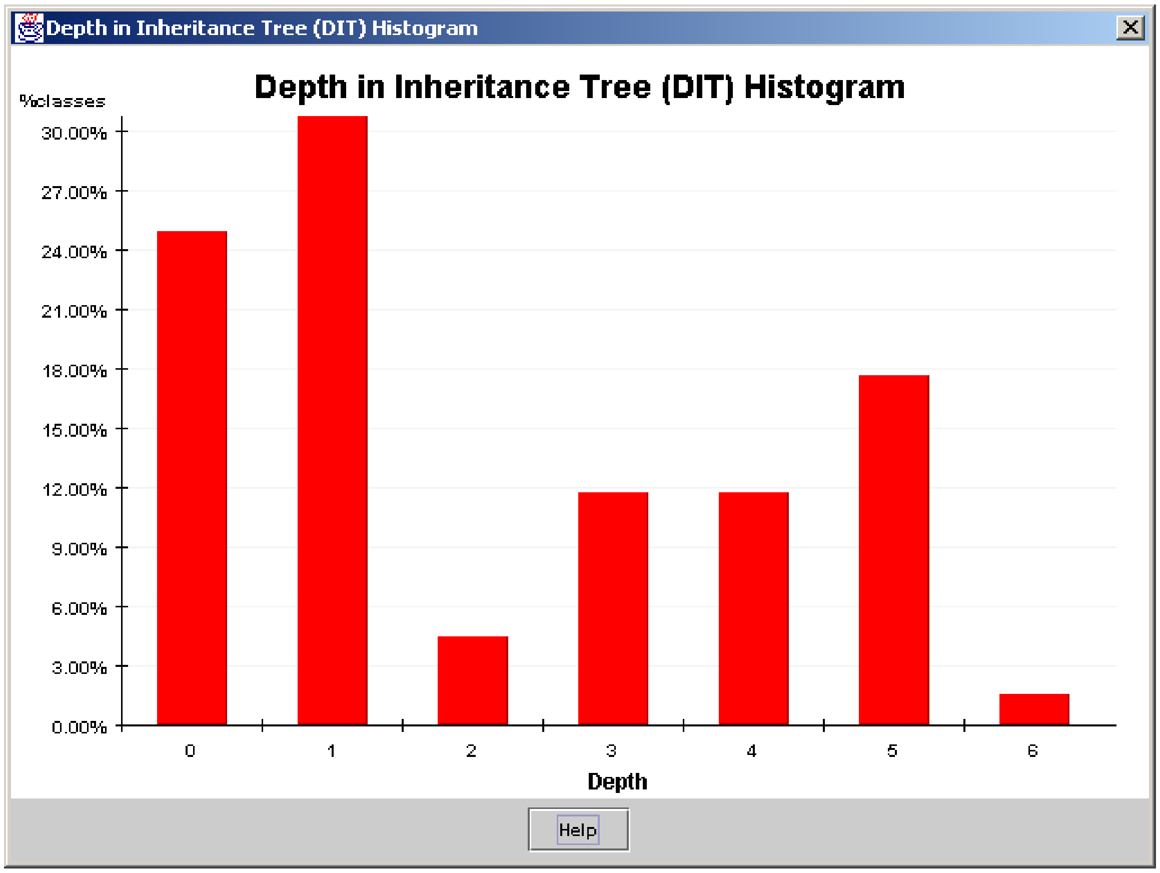
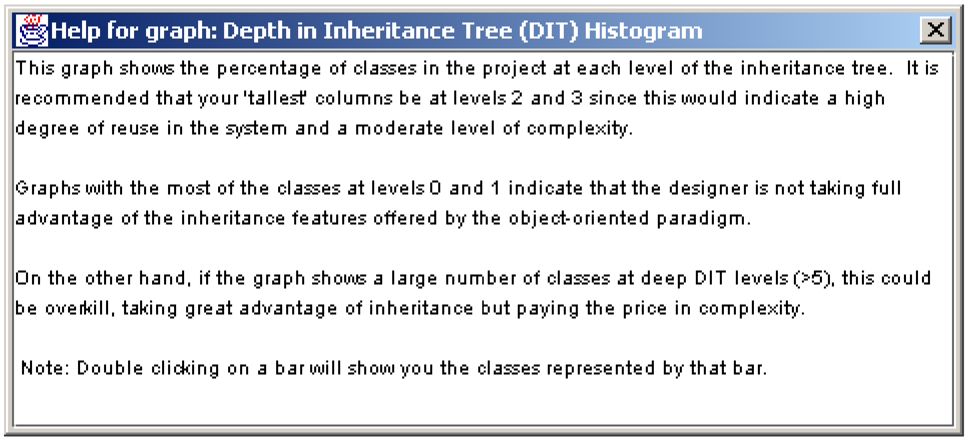


Figure 61 - One of the graphs in the system



Help for graph: Depth in Inheritance Tree (DIT) Histogram

This graph shows the percentage of classes in the project at each level of the inheritance tree. It is recommended that your 'tallest' columns be at levels 2 and 3 since this would indicate a high degree of reuse in the system and a moderate level of complexity.

Graphs with the most of the classes at levels 0 and 1 indicate that the designer is not taking full advantage of the inheritance features offered by the object-oriented paradigm.

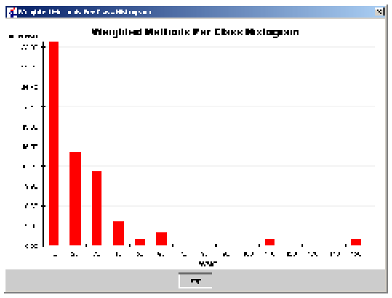
On the other hand, if the graph shows a large number of classes at deep DIT levels (>5), this could be overkill, taking great advantage of inheritance but paying the price in complexity.

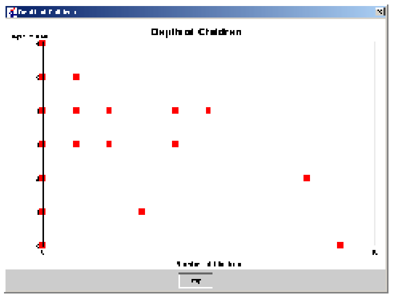
Note: Double clicking on a bar will show you the classes represented by that bar.

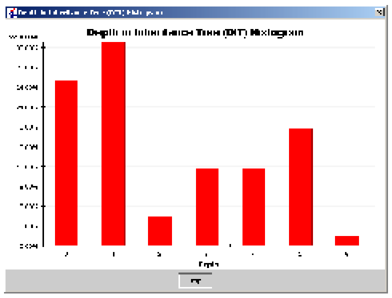
Figure 62 - The help text for the graph shown above (DIT)

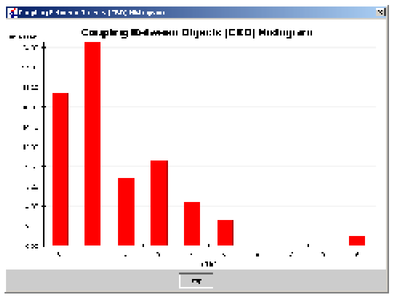
It is beyond the scope of this document to explain the way graphs should be interpreted because this information is available in the tool's help system. The knowledge base regarding metrics, quality attributes and graphs is very detailed and the user should be able to understand how to use the system simply be reading up on the metrics, attributes and graphs using the online help.

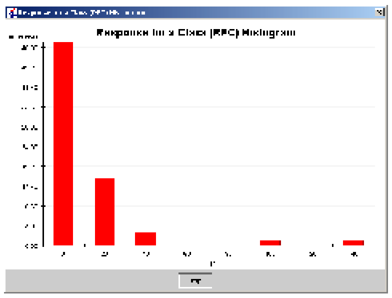
The following are a few screenshots of some of the graphs provided by the system…

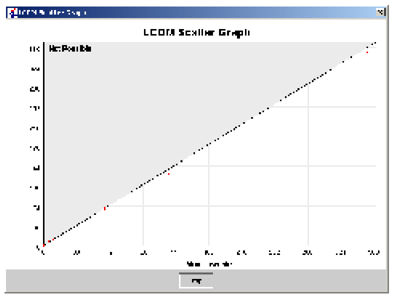


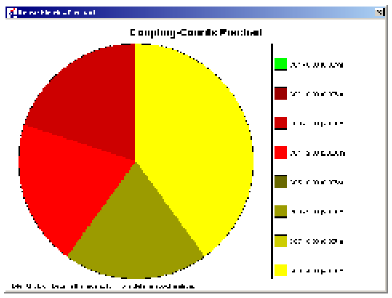


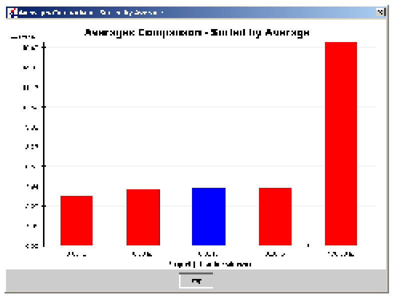

Depth in Inheritance Tree (DIT) Histogram

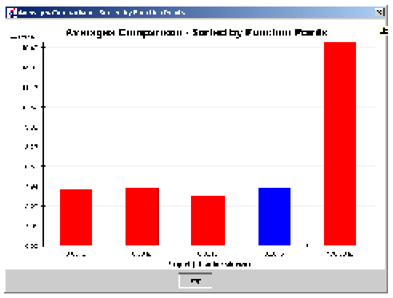

Coupling Between Objects (CBO) Histogram


Response for a Class (RFC) Histogram


LCOM Scatter Graph

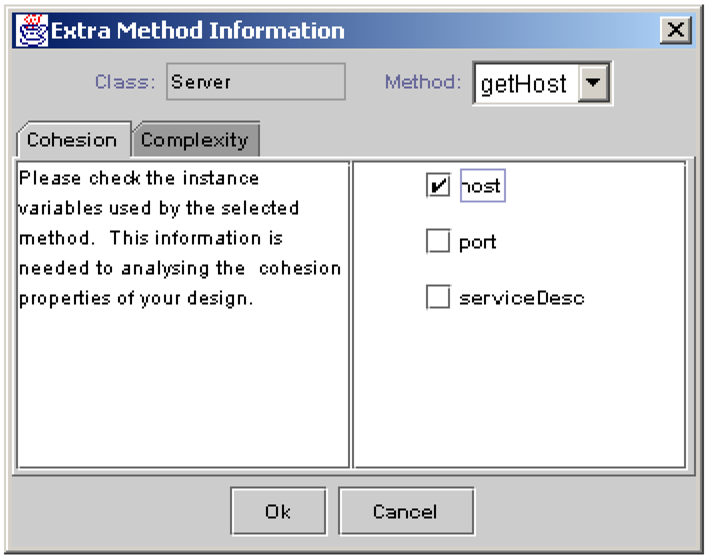

Coupling-Cohesion Piechart


Average Comparison - Sorted by Average


Average Comparison - Sorted by Function Points

Collecting Extra Information for LCOM and WMC

As previously explained, extra information needs to be collected from the user before reliable WMC and LCOM results can be obtained. In the case of WMC, the user needs to link methods of classes with activity diagrams that describe the behavior of those methods. With LCOM, the user has to specify which instance variables each method in a class uses. To access these features, right-click on a class and select *Metrics Info*. This brings up the following window (tailored to the particular class):


Extra Method Information

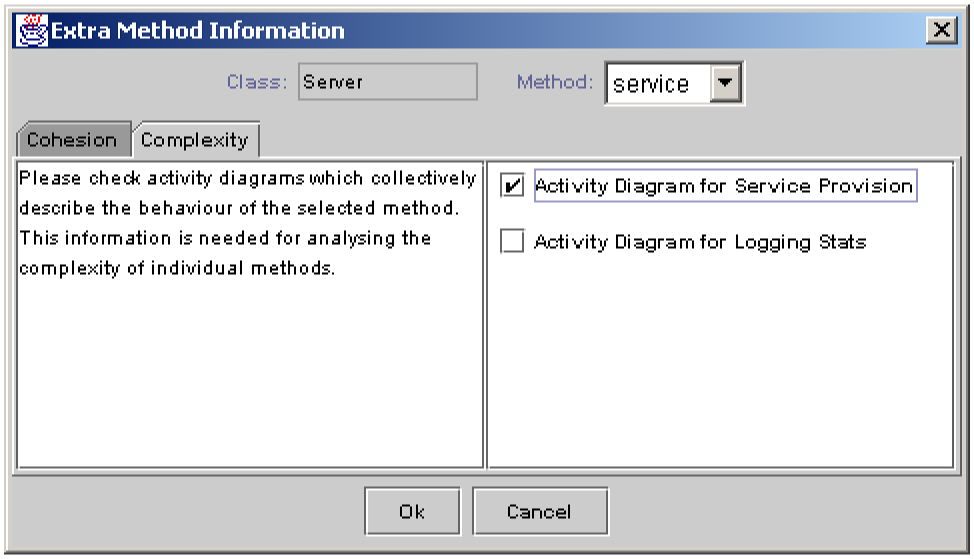
Class: Server Method: getHost

Cohesion | Complexity

Please check the instance variables used by the selected method. This information is needed to analysing the cohesion properties of your design.

[✓] host
[ ] port
[ ] serviceDesc

Figure 63 - Inputting cohesion information



Class: Server Method: service

Please check activity diagrams which collectively describe the behaviour of the selected method. This information is needed for analysing the complexity of individual methods.

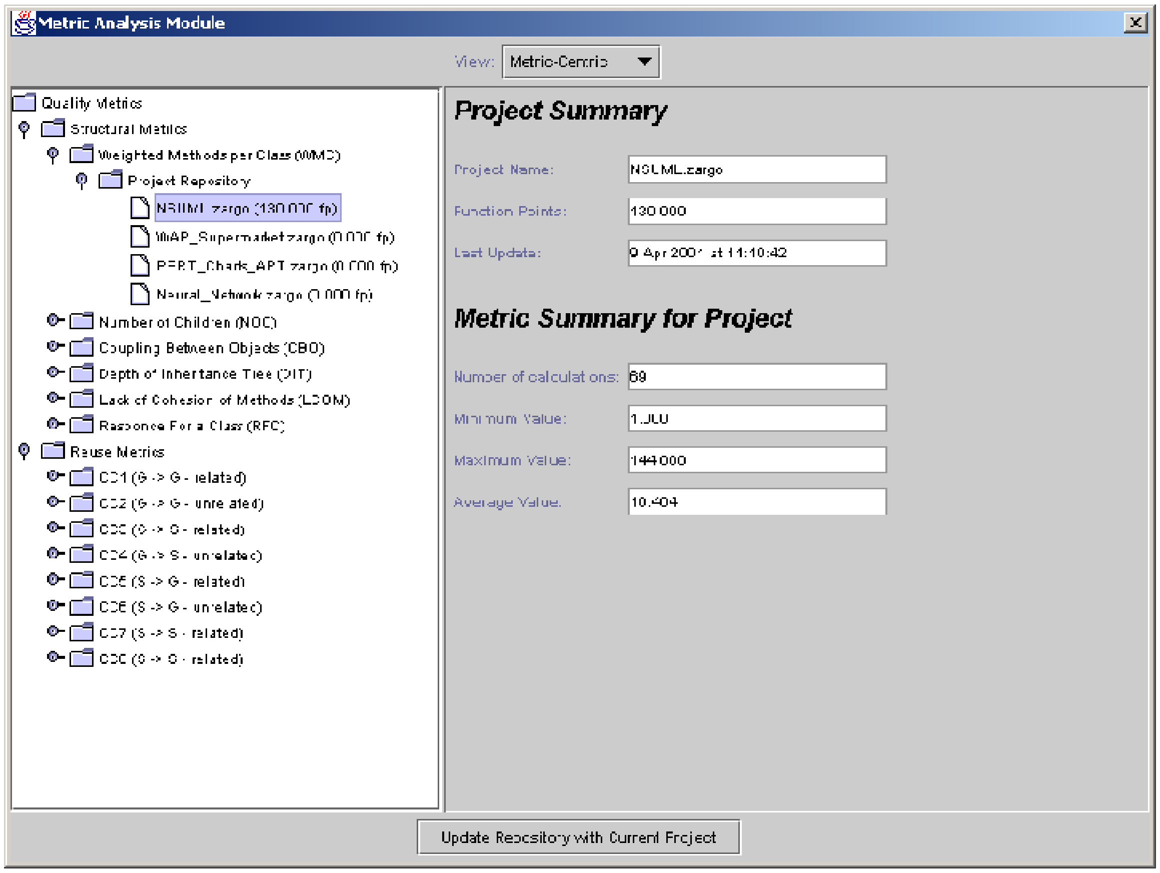
- [✓] Activity Diagram for Service Provision
- [ ] Activity Diagram for Logging Stats

Figure 64 - Inputting complexity information

Assigning methods to activity diagrams or defining which instance variables a method uses, consists of selecting the method from the combo-box and making checking the relevant checkboxes.

Comparing Results with Projects in the Repository

If you want to compare the results of this project with respect to a particular metric, all you have to do is click on the metric in the *metrics tree* and then open up it's *Project Repository* child node. This brings up a list of projects in the repository and selecting a particular project will present a summary of the project (with respect to the selected metric) in the information panel.



View: Metric-Centric

- Quality Metrics
  - Structural Metrics
    - Weighted Methods per Class (WMC)
      - Project Repository
        - NSUML.zargo (130.000 fp)
        - WAP_Supermarket.zargo (0.000 fp)
        - PERT_Charts_APT.zargo (0.000 fp)
        - Neural_Network.zargo (0.000 fp)
    - Number of Children (NOC)
    - Coupling Between Objects (CBO)
    - Depth of Inheritance Tree (DIT)
    - Lack of Cohesion of Methods (LCOM)
    - Response For a Class (RFC)
  - Reuse Metrics
    - CD1 (G -> G - related)
    - CD2 (G -> G - unrelated)
    - CD3 (G -> S - related)
    - CD4 (G -> S - unrelated)
    - CD5 (S -> G - related)
    - CD6 (S -> G - unrelated)
    - CD7 (S -> S - related)
    - CD8 (S -> S - related)

Project Summary

Project Name: NSUML.zargo
Function Points: 130.000
Last Update: 9 Apr 2004 at 11:10:42

Metric Summary for Project

Number of calculations: 69
Minimum Value: 1.000
Maximum Value: 144.000
Average Value: 10.404

Figure 65 - A summary of the project NSUML with respect to the WMC metric

The summary lists the following information:

1. The name of the project
2. The function points of the project
3. The last time the project was updated in the repository
4. The number of times the selected metric was calculated in the project
5. The minimum value of the selected metric in the project
6. The maximum value of the selected metric in the project
7. The average value of the selected metric in the project

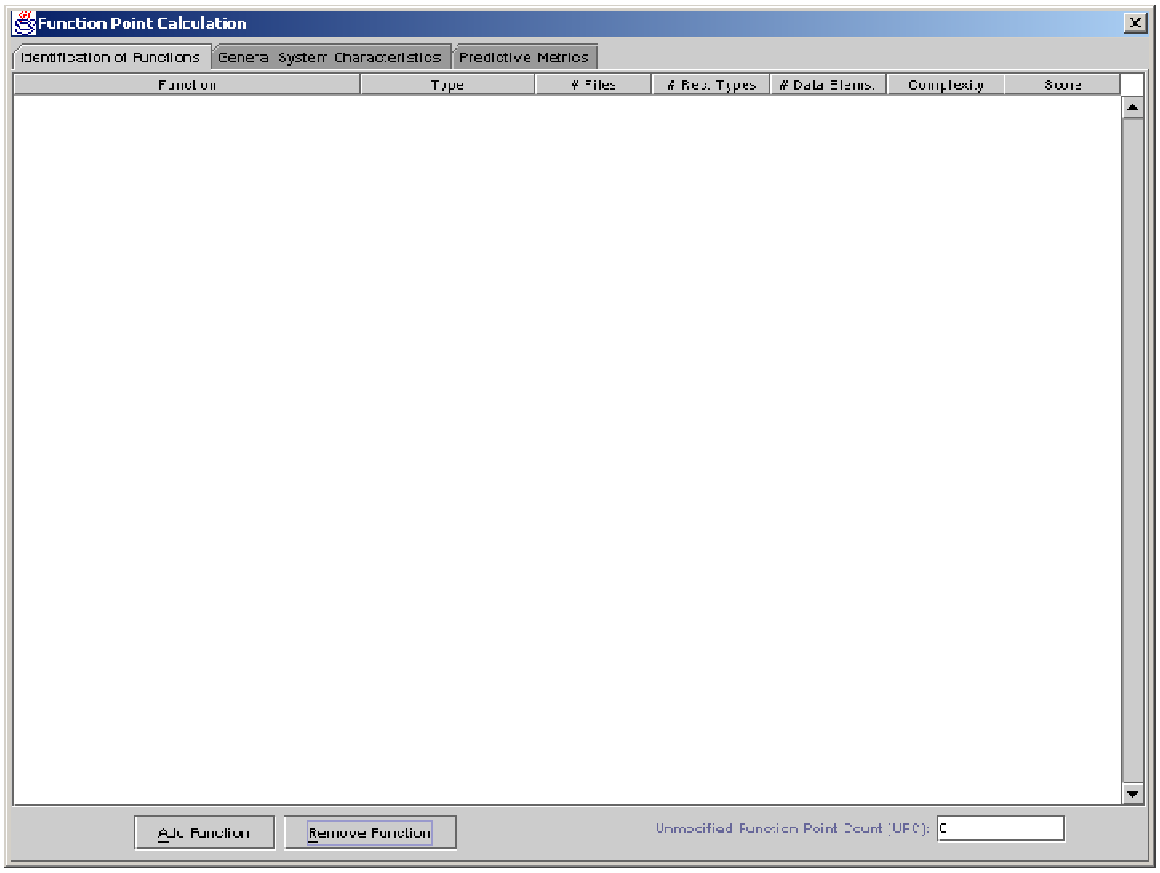
The Function Points Module

The *Function Points* module is responsible for calculating function points for the current projects and providing some rudimentary predictive metrics on the project.

Calculating the Function Points of the current Project

To calculate the Function Points of the current project, simply choose **Quality → Function Points** from the menu bar. This will bring up a window with 3 tabs:

1. **Identification of Functions tab –** for defining the functions offered by your project
2. **General System Characteristics tab –** where you answer a set of questions about your project. The answers to these questions will result in the function point count be adjusted accordingly.
3. **Predictive Metrics tab –** where you can enter your own estimates with regards to how much time it takes to develop a function point and how much it costs. The tool will then come up with predictive results, which will be saved along with the project for later analysis.



Unmodified Function Point Count (UFC): 0

Figure 66 - The Function Points Module

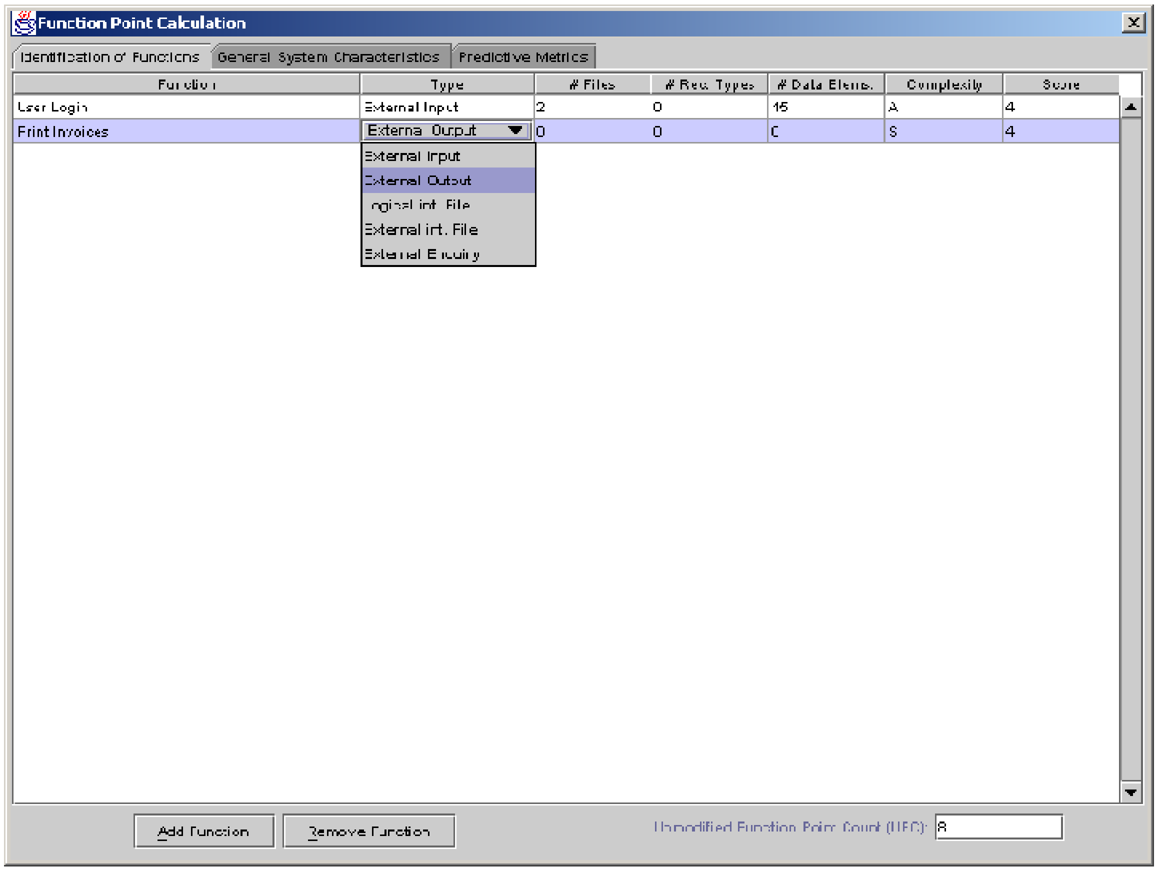
Adding and Removing Functions

To add a function:

1. Click on the Add Function button
2. Fill in the required fields in the newly create row in the table
3. The complexity and score fields will be calculated automatically

To remove a function:

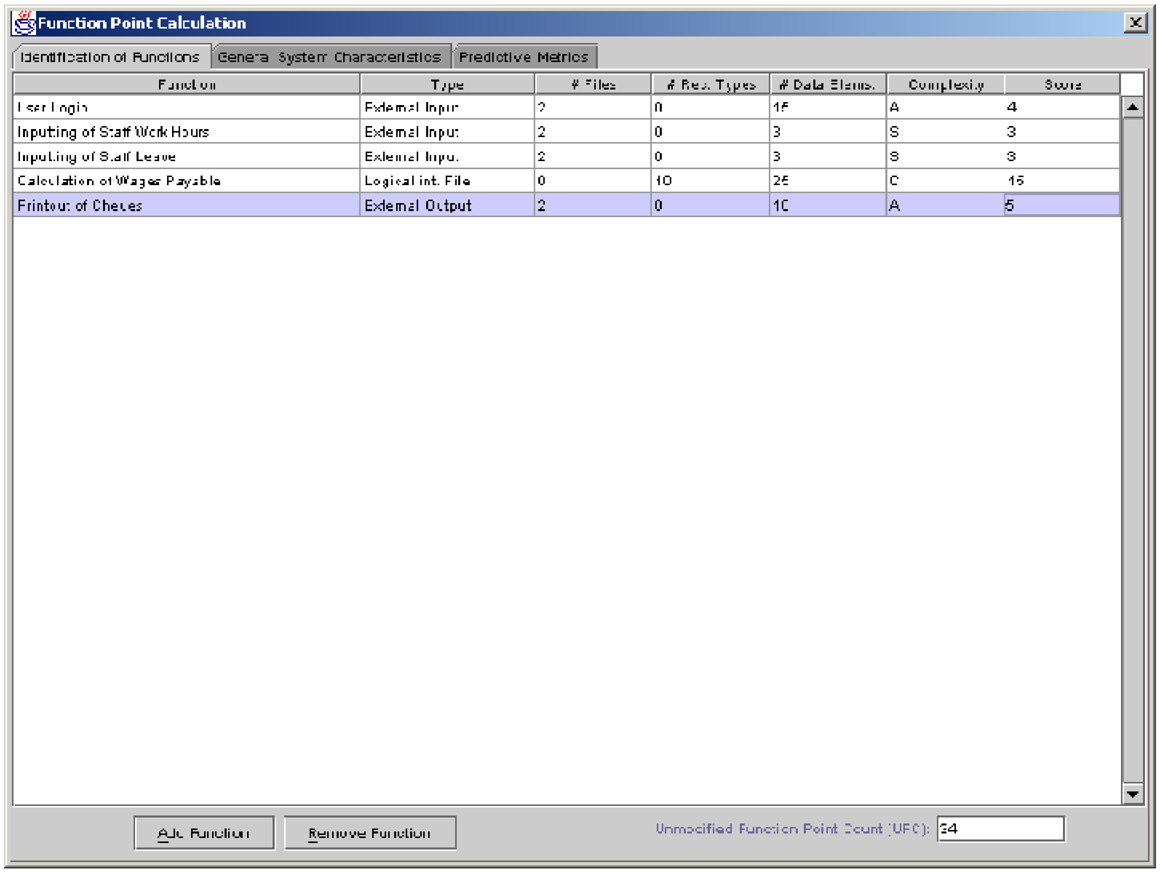
1. Select the function you wish to remove
2. Click on the Remove Function button



Function	Type	# Files	# Rec. Types	# Data Elems.	Complexity	Score
User Login	External Input	2	0	15	A	4
Print Invoices	External Output	0	0	0	S	4

Unmodified Function Point Count (UFC): 8

Figure 67 - Adding a new function


Function Point Calculation
Identification of Functions
General System Characteristics
Predictive Metrics
Add Function
Remove Function

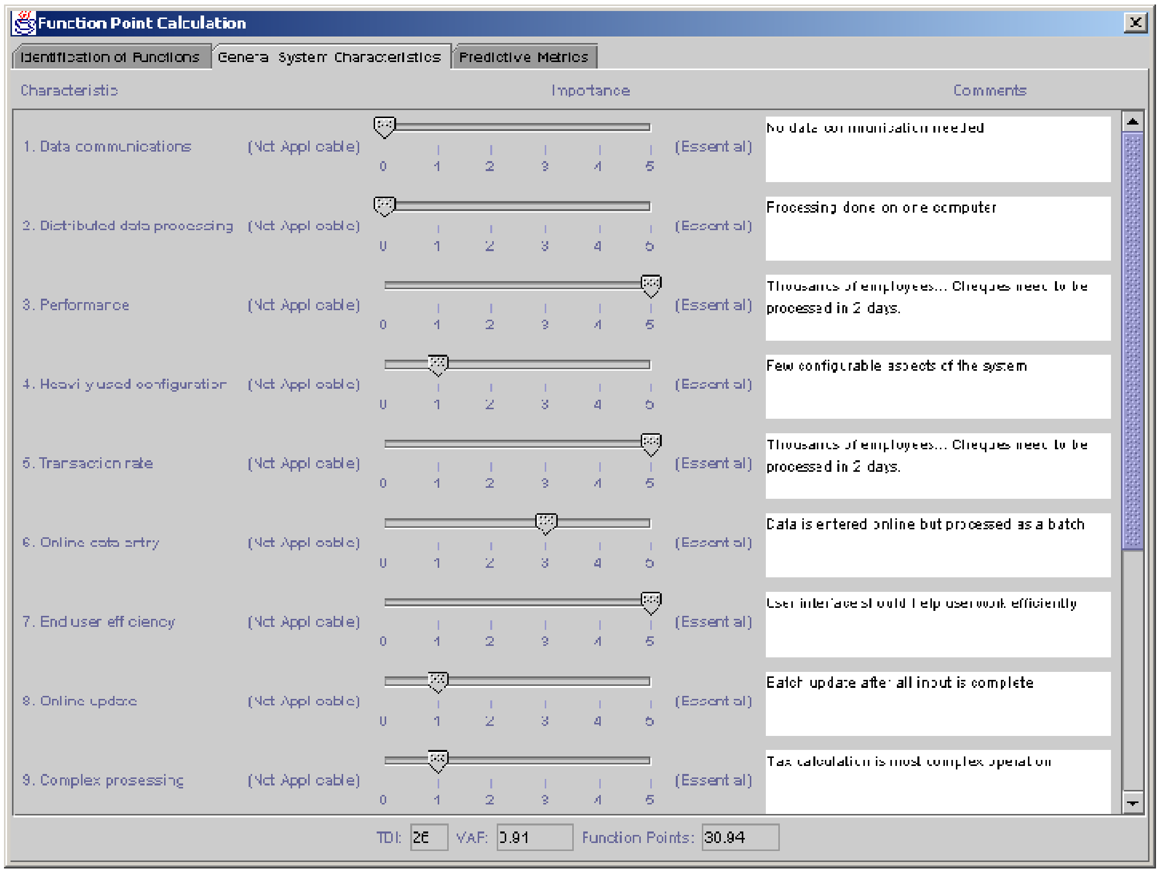
Function	Type	# Files	# Rec. Types	# Data Elems.	Complexity	Score
User Login	External Input	2	0	15	A	4
Inputing of Staff Work Hours	External Input	2	0	3	S	3
Inputing of Staff Leave	External Input	2	0	3	S	3
Calculation of Wages Payable	Logical int. File	0	10	25	C	15
Printout of Cheques	External Output	2	0	10	A	5

Unmodified Function Point Count (UFC): 34

Figure 68 - An example with 5 functions resulting in 34 function points (UFC)

Answering General System Characteristics (GSC) Questions

The GSC questions serve to modify the unmodified function point count by taking into account other generic requirements of the system. There are 14 characteristics and the user has to give each one of them an importance factor between 0 and 5. A comments box is included for each GSC for future reference.


Function Point Calculation

Identification of Functions | General System Characteristics | Predictive Metrics

Characteristic	Importance (Not Applicable 0 – 5 Essential)	Comments
1. Data communications	0	No data communication needed
2. Distributed data processing	0	Processing done on one computer
3. Performance	5	Thousands of employees... Cheques need to be processed in 2 days.
4. Heavily used configuration	1	Few configurable aspects of the system
5. Transaction rate	5	Thousands of employees... Cheques need to be processed in 2 days.
6. Online data entry	3	Data is entered online but processed as a batch
7. End user efficiency	5	User interface should help user work efficiently
8. Online update	1	Batch update after all input is complete
9. Complex processing	1	Tax calculation is most complex operation

TDI: 26 VAF: 0.91 Function Points: 30.94

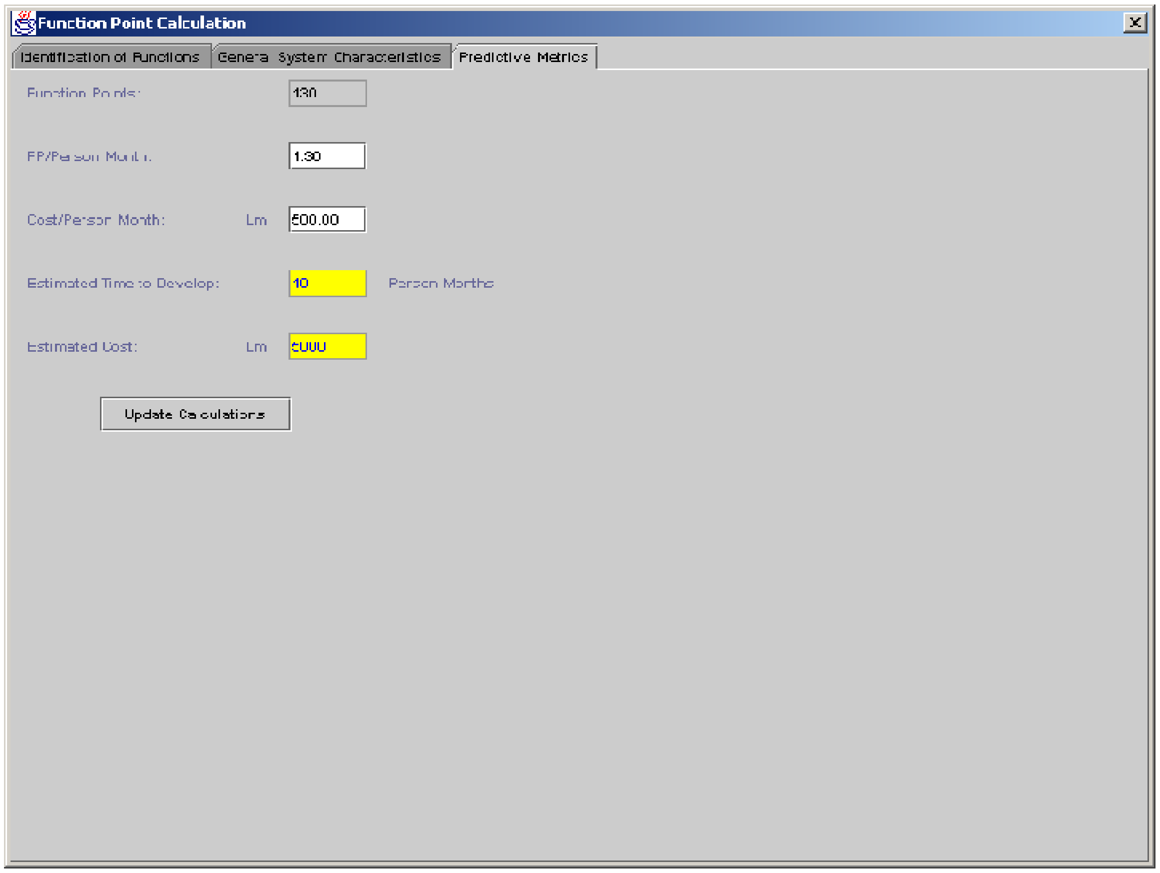
Figure 69 - The *General System Characteristics* Tab

Predictive Metrics

This feature does not offer any fancy computing but it does serve as a way of making estimates and saving them along with your project for future estimates. The user has to:

1. Enter the number of person months it takes to implement one function point (this information would be obtainable from previous experience)
2. Enter the average cost per person month

The system will in turn estimate how long it will take to develop the system and how much it would probably cost. Use of these features will get better with experience.


Function Point Calculation
Identification of Functions
General System Characteristics
Predictive Metrics
Update Calculations

Function Points: 130

FP/Person Month: 1.30

Cost/Person Month: Lm 500.00

Estimated Time to Develop: 10 Person Months

Estimated Cost: Lm 5000

Figure 70 - The *Predictive Metrics* Tab