UNILASALLE
CENTRO UNIVERSITÁRIO LA SALLE
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THE IMPACT OF SOFTWARE QUALITY METRICS IMPROVEMENT OVER PERFORMANCE METRICS IN THE DISTRIBUTED OBJECT-ORIENTED CONTEXT

WAGNER SOARES NOBRES

Canoas, July 2008

WAGNER SOARES NOBRES

THE IMPACT OF SOFTWARE QUALITY METRICS IMPROVEMENT OVER PERFORMANCE METRICS IN THE DISTRIBUTED OBJECT-ORIENTED CONTEXT

Monography presented in fulfillment of the requirements for the degree of Bachelor in Computer Science, under orientation of Prof. DSc. Patrícia Kayser Vargas Mangan.

Canoas, July 2008

TERMO DE APROVAÇÃO

WAGNER SOARES NOBRES

THE IMPACT OF SOFTWARE QUALITY METRICS IMPROVEMENT OVER PERFORMANCE METRICS IN THE DISTRIBUTED OBJECT-ORIENTED CONTEXT

Trabalho de conclusão aprovado como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Centro Universitário La Salle - Unilasalle, pela seguinte banca examinadora:

Canoas, July 2008

ABSTRACT

This text aims to demonstrate how quality object-oriented software metrics relates to the performance metrics, when applied in computational grid development. The result of this work will provide support to software architects and engineers in the decision making process when deciding for using OO (ObjectOriented) paradigm in the development of a grid system. The Appman middleware, an implementation of the GRAND model, is used as case study. Refactoring techniques are used to improve software metrics.
Keywords: Grid Computing, Refactoring, Object Orientation.

O Impacto da Melhoria de Métricas de Qualidade de Software sobre as Métricas de Desempenho no Contexto Orientado a Objetos Distribuído

RESUMO

Este texto visa demonstrar como as métricas de qualidade de software orientado a objetos se relacionam com as métricas de desempenho quando aplicadas na implementação de grades computacionais. O resultado deste trabalho servirá de apoio para arquitetos e engenheiros de software na decisão pelo uso do paradigma OO (Orientado a Objetos) no desenvolvimento de um sistema de grade. Como estudo de caso será utilizado o middleware Appman, implementação do modelo GRAND. São utilizadas técnicas de refactoring para aprimorar as métricas de software.
Palavras-chave: Grid Computing, Refactoring, Object Orientation.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

EXEHDA	Execution Environment for High Distributed Applications
GRAND	Grid Robust Application Deployment
HPC	High Performance Computing
LCOM	Lack of Cohesion on Methods
OO	Object Oriented
PBS	Portable Batch System

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Computational Grid has emerged in the 1990's as a promising alternative to run parallel applications on widely dispersed computational resources, located at different institutions. The rationale was twofold. The first one was to provide a cheaper execution platform for parallel applications than supercomputers. The second was to allow the distributed applications execution using a number of resources that is impossible to achieve using only one supercomputer (Cirne and Neto, 2005).

However according to Foster and Kesselman (1998), grid environments will require a rethinking of existing programming models, most likely, new thinking about novel models more suitable for the specific characteristics of grid applications and environments. New techniques are necessary for expressing advanced algorithms, for mapping those algorithms into complex grid architectures, for translating user performance requirements into system resources requirements and for adapting to changes in underlying system structure and state. Application and system complexity increases the importance of code reuse, and so techniques for the construction and composition of grid-enabled software components are important. Another significant challenge is to provide tools that allow programmers to understand and explain program behavior and performance. Foster and Kesselman (1998) also affirm that technologies that have proven effective in distributed computing, like object-based techniques, have significant software engineering advantages, because their encapsulation properties facilitate the modular construction of programs and the reuse of existing components. However, it remains to be seen whether these models can support performance-focused, complex applications such as teleimmersion (Walton, 2008b) or the construction of dynamic computations that spans hundreds or thousands of processes.

In this case, the treatment of parallelism must be analyzed. Grimshaw (1996) says that is possible to use the object-oriented paradigm in parallel environments of high performance. It also suggests that combining the object-oriented paradigm with compiler-based parallelism detection and management can be performance competitive with hand-coded implementations using send and receive.

On the other hand, Booth (2006) concludes that object orientation while important for controlling code complexity can give rise to performance problems as it constrains the memory layout of application data. However, the encapsulation provided by the object-oriented programming style makes it easier to perform all forms of refactoring including performance optimization.

For Wegner (1990): "The development of object-based and object-oriented environments for high-performance computers and concurrent architectures presents a challenge for the 1990s". Although it is an old reference, until today no definitive work with focus in this study has been done.

With this motivation, this work is going to trace a parallel between object-oriented software quality metrics and performance metrics used in grid systems and thus making possible to measure how much a metric is related to each other.

This text is organized in six chapters. The Chapter 2 gives an introduction about Grid Application Management and Appman (Vargas et al., 2005), which is the software that has been used as a case study for this work. The Chapter 3 talks about the metrics which has been used as a tool to measure the quality and performance between the refactoring cycles. Chapter 4 shows the refactoring techniques that can be used to improve the source code quality and maintainability. It gives some brief comments about performance issues on object-oriented systems and how refactoring implies on it. Chapter 5 demonstrates the refactoring cycles which details, showing class diagrams, software and performance metrics evaluation in each cycle. This chapter also shows the results of this work with some graphs and comments about the changes observed in each cycle. To finish, the Chapter 6 presents the comments of the author about the results of this research.

2 GRID APPLICATION MANAGEMENT

The term grid computing was coined by Foster and Kesselman in late 90's as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.

Nowadays, grid computing efforts can be seen as the third phase of metacomputing evolution as stated in the Meta-computing's paper of Smarr and Catlett in 1992: a transparent network that will increase the computational and information resources available to an application. It is also a synonym of metasystem which "supports the illusion of a single machine by transparently scheduling application components on processors; managing data migration, caching, transfer, and the masking of data-format differences between systems; detecting and managing faults; and ensuring that users's data and physical resources are protected, while scaling appropriately". (Vargas et al., 2004)

2.1 Appman - Case Study

Appman is a Java implementation of the GRAND model detiled by Mangan (2006). GRAND (Grid Robust Application Deployment) is a model of hierarchical application management in grid environment. Its objective is to deal with scalable form of submission and monitoring of applications that execute a great number of tasks (hundreds or thousand) (Vargas et al., 2005).

In this context, Appman (Vargas et al., 2005) implements the submission control (Submission Manager) and execution (Task Manager) of applications using the Java language and EXEHDA system (Yamin et al., 2005). The EXEHDA (Execution Environment for High Distributed Applications) (Yamin et al., 2005) facilitates the implementation and execution of distributed applications. Appman is implemented in Java to allow portability. It uses the JavaCC tool to implement the GRID-ADL parsing and interpretation. GRID-ADL is the application description

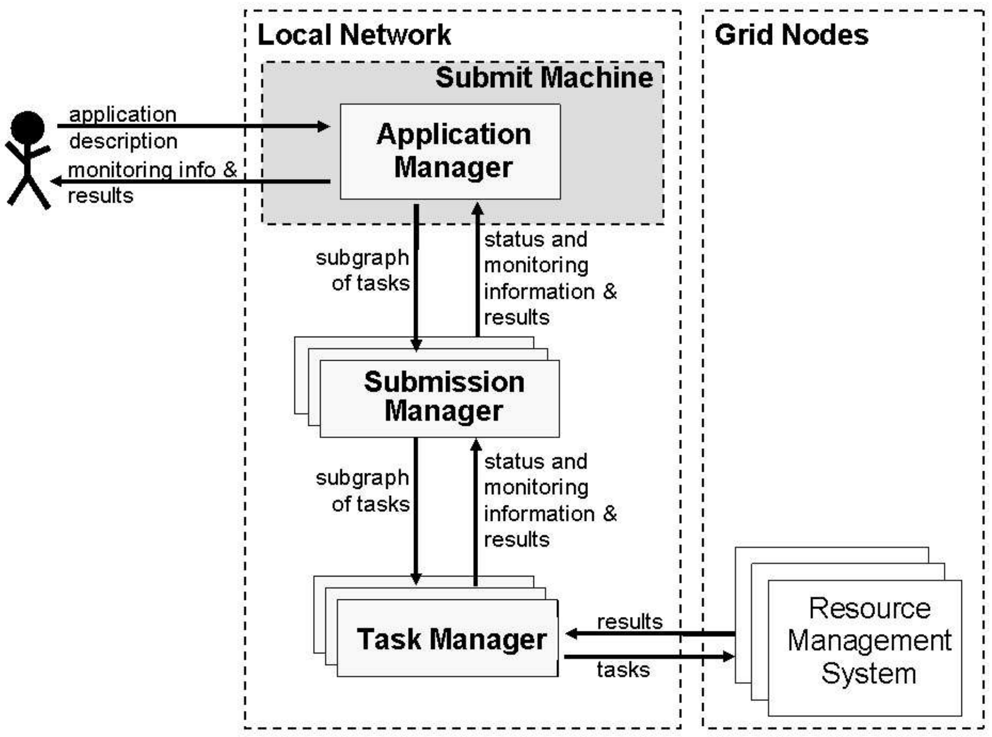

Figure 2.1: GRAND Model: Architecture
Source: (Vargas et al., 2005)

language proposed in GRAND project. The Appman runtime environment uses the services provided by EXEHDA middleware that allows remote execution and monitoring. EXEHDA, which stands for "Execution Environment for Highly Distributed Applications", has facilities to instantiate remote objects and to coordinate their operation, as well as an integrated monitoring architecture.

Appman implements the basic main features of GRAND, including distributed task submission and application monitoring, while giving feedback to the user. Figure 2.1 shows how Appman runs on a grid environment. One instance of Appman and ISAM/EXEHDA needs to be present in every node of the grid. Every machine can submit or run tasks. Users must provide input files in a web server. Step 1 in Figure 2.1 represents the user submitting a description file which is parsed. The application graph is built in memory and the Application Manager (AM) is started. Then, the Application Manager clusters the application graph, instantiates Submission Managers (SM in step 2), and distributes some subgraphs to the SMs. Input files and the executable are fetched from a web server (step 3). In its current version, Appman requires that the user indicates the machines where the Submission Managers will run. With this simplification, a new Submission Manager is instantiated for each application in each machine specified. After creating the SMs, the Application Manager as signs subgraphs to each SM. The SMs report to the AM the tasks progress. Each SM, independently, checks the list of available machines and randomly chooses where to run. Immediately a TM is instantiated, which creates the remote task and monitors it until it successfully completes. Before starting the execution of a task, Appman transfers all input files specified in the GRID-ADL description to a temporary directory in the remote machine where the task will be executed, a sort of sand-box. Each task is only allowed to produce output files int this particular temporart directory. The file transfer is performed automatically, but the user must make the files available in an http accessible area, as mentioned before. Then, each SM gather updated information about available nodes through the EXEHDA information service. Each SM chooses where to run its tasks and instantiates TMs to allocate the remote tasks (step 4).

An graphical interface for online monitoring provides visual execution feedback to users. It runs independent from execution, reading the log files. By using this tool, the user can have an idea of the application execution progress. As the user will probably be running experiments for days or weeks, this becomes a very important and essential tool.

3 COMPUTATIONAL METRICS

This chapter presents two kinds of computational metrics: software quality metrics and metrics related to computer performance.

3.1 Software Quality Metrics

Managing software production requires adequate metrics that shed light on the development progress as well as on the produced assets quality. Useful feedback can then help in adopting better design strategies and techniques for future projects. Measurement is often meant to cover such matters as cost estimation, test management and administrative decision-making. (Benlarbi and Goel, 1998)

Many sets of OO metrics have been proposed in the last few years. In Appendix A, there is a complete list provided by Benlarbi and Goel (1998) of OO metrics that are currently investigated or in use. In the chapter 5 is explained only those metrics that has been choose to measure this work.

3.1.1 Traditional Metrics and OO Software Systems

Rosenberg and Hyatt (1997) has found that there is considerable disagreement in the field about software quality metrics for object-oriented systems. Some researchers and practitioners contend traditional metrics are inappropriate for object-oriented systems. There are valid reasons for applying traditional metrics, however, if it can be done. The traditional metrics have been widely used, they are well understood by researchers and practitioners, and their relationships to software quality attributes have been validated.

In Table 3.1 shows the differences between OO software design and structured software design concepts. For example, coupling is an internal product feature that has been extensively studied since the advent of structured design. However, due to the specific encapsulation and visibility rules in the OO paradigm, coupling in OO systems differs from the module coupling although the notion of interconnection between components holds. The elementary encapsulation level being the class level, most of the basic OO metrics that have been found useful and are being investigated are considered at the class level. Another major difference is that, in the traditional structured software design, methods are separated from data. Therefore, measures such as Halstead's volume and size cannot be directly applied to evaluate OO software systems volume and size because data are fully integrated to methods. (Benlarbi and Goel, 1998)

Table 3.1: Differences between Object-Orientation and Structured Design
Source: (Benlarbi and Goel, 1998)

	OO Features	**Structured Design Features**
Entities	Classes, Agents or Objects (instances of class); Messages (requests for action); Methods (set of operations)	Data types, and variables (instances of data types); Functions and procedures calls (requests for action); functions and procedures (set of operations)
Design rules	Abstraction (class, hierarchy, clusters); Encapsulation (Interface); Modularity (Problem Domain Classification)	Abstraction (Data types, Procedures and Modules); Encapsulation (very little); Modularity (Functional grouping)
Design mechanisms	Inheritance, association, aggregation, polymorphism and message passing	Control flow, data flow, data structure
Coding	Method binding, overriding, reuse, computation as simulation	Function call, data management, computation as process-state

In an object-oriented system, traditional metrics are generally applied to the methods that comprise the operations of a class. A method is a component of an object that operates on data in response to a message and is defined as part of the declaration of a class. Methods reflect how a problem is broken into segments and the capabilities other classes expect of a given class. (Rosenberg and Hyatt, 1997).

3.1.1.1 Cyclomatic Complexity (CC)

Cyclomatic complexity (McCabe) is used to evaluate the complexity of an algorithm in a method. A method with a low cyclomatic complexity is generally better, although it may mean that decisions are deferred through message passing, not that the method is not complex. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. In general, the cyclomatic complexity for a method should be below ten, indicating decisions are deferred through message passing. This metric is used to evaluate the quality attribute Complexity.

3.1.1.2 Size

Size of a method is used to evaluate the ease of understandability of the code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, and the number of blank lines. Thresholds for evaluating the size measures vary depending on the coding language used and the complexity of the method. However, since size affects ease of understanding, routines of large size will always pose a higher risk in the attributes of Understandability, Reusability, and Maintainability.

3.1.1.3 Comment Percentage

The line counts done to compute the Size metric can be expanded to include a count of the number of comments, both on-line (with code) and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. Rosenberg and Hyatt (1997) has found a comment percentage of about 30% is most effective. Since comments assist developers and maintainers, this metric is used to evaluate the attributes of Understandability, Reusability, and Maintainability.

3.1.2 Object-Oriented Specific Metrics

Many different metrics have been proposed for object-oriented systems. The object-oriented metrics that were chosen by Rosenberg and Hyatt (1997) measure principle structures that, if improperly designed, negatively affect the design and code quality attributes. The selected object-oriented metrics are primarily applied to the concepts of classes, coupling, and inheritance. For some of the object-oriented metrics discussed here, multiple definitions are given, since researchers and practitioners have not reached a common definition or counting methodology. In some cases, the counting method for a metric is determined by the software analysis package being used to collect the metrics.

A class is a template from which objects can be created. This set of objects share a common structure and a common behavior manifested by the set of methods. Three class metrics described here measure the complexity of a class using the class's methods, messages, and cohesion as a criteria.

3.1.2.1 Method

A method is an operation upon an object and is defined in the class declaration.

The **Weighted Methods per Class (WMC)** is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement since not all methods are accessible within the class hierarchy due to inheritance. The number of methods and the complexity of the methods 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. This metric measures Understandability, Maintainability, and Reusability.

3.1.2.2 Message

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. The next metric looks at methods and messages within a class.

The **Response for a Class (RFC)** is the carnality of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of this class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time. This metric evaluates Understandability, Maintainability, and Testability.

3.1.2.3 Cohesion

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Effective object-oriented designs maximize cohesion since it promotes encapsulation. The third class metrics investigates cohesion.

Lack of Cohesion of Methods (LCOM) measures the degree of similarity of methods by data input variables or attributes (structural properties of classes. Any measure of separateness of methods helps identify flaws in the design of classes. There are at least two different ways of measuring cohesion:

- Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.
- Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates Efficiency and Reusability.

3.1.2.4 Coupling

Coupling is a measure of the strength of association established by a connection from one entity to another. Classes (objects) are coupled three ways:

- When a message is passed between objects, the objects are said to be coupled.
- Classes are coupled when methods declared in one class use methods or attributes of the other classes.
- Inheritance introduces significant tight coupling between superclasses and their subclasses.

Since good object-oriented design requires a balance between coupling and inheritance, coupling measures focus on non-inheritance coupling. The next object-oriented metric measures coupling strength.

Coupling Between Object Classes (CBO) is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates Efficiency and Reusability.

3.1.2.5 Inheritance

Another design abstraction in object-oriented systems is the use of inheritance. Inheritance is a type of relationship among classes that enables programmers to reuse previously defined elements including variables and operators. Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

The **Depth of Inheritance Tree (DIT)** within the inheritance hierarchy is the maximum length from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number methods it is likely to inherit making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. A support metric for DIT is the number of methods inherited (NMI). This metric primarily evaluates Efficiency and Reuse but also relates to Understandability and Testability.

Number of Children (NOC) is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator 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 abstraction of the parent and may be a case of misuse of subclassing. But the greater the number of children, the greater the reusability since inheritance is a form of reuse. 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.

3.1.3 Metrics Evaluation Criteria

Benlarbi and Goel (1998) searched tools for application of metrics and traced criteria to select the appropriate ones for the context of their work that was the choice of a tool for metric analysis being incorporated to the product of their company.

1. **Covered dimensions and targets:** What are the size and volumn of dimensions (attributes, characteristics) captured by the metrics? The metrics deal specifically the mechanisms of object-oriented design as listed or not? What's the abstraction scope or level covered by the metrics (function, class, module, sub-system, system)?

2. **Completeness and precision of the metrics:** Which phase of the life cycle can the metrics be collected (requirements analysis, high level design, low level design, code)? Is there any theoretical or empirical evidence that allow us to establish the design and code characteristics that the OO metrics are built in a fact contribute to the volumn and size dimension they target? Does the OO metric(s) that is addressing a given size or volume dimension, cover the whole targeted domain or not? Does the measurement method (counts, scale, range values) reflects how much the size or volume dimension is being contributed by the offered OO metrics?

3. **Flexibility and Extendability:** How should we open the tool in order to allow manipulating available OO metrics or adding new basic OO metrics? Does the tool allows aggregating basic OO metrics among the available to

allow building dependability evaluation models? How flexible is the tool to allow constructing OO dependability models?

4. **Other questions which has been adapted to the context of this work:** How difficult is it to set up and run the tool? How long does it take to set up and run the tool to perform the OO metrics extraction on a given project? How long does it take to understand and train a person on performing the OO metrics extraction? What visualization features does the tool have to allow displaying the metrics on design graphs or structural graphs? Given that many metrics will require the visualization of large and complex graphs, can the tool render these in a reasonable time?

In the technical report of Xenos et al. (2000) is carried through a research in literature for traditional and object-oriented specific metrics. In the report the authors make an evaluation of related metrics using meta-metrics and its result serves of support for quality controlling to select the appropriate metric for the project.

Xenos et al. (2000) beyond traditional metrics as LOC (Lines of code) and FUP (Function Points), analyze object-oriented metrics subdivided in:

- Class Metrics
- Method Metrics
- Coupling Metrics
- Inheritance Metrics

Inside of these classes of metrics, they will be chosen at least one metric of each, according to the evaluation of the quality of them using meta-metrics. Below some meta-metrics are listed as defined by Xenos et al. (2000):

- Measurement Scale
- Measurements Independence
- Automation
- Value of Implementation
- Monotocity
- Simplicity
- Accuracy

Following a similar aproach to the previous article, Rosenberg and Hyatt (1997) makes an analysis of the appropriate metrics for measurement of software quality on object-oriented systems. In its article they evaluate the metrics through these criteria:

- Efficiency - Are the constructs efficiently designed?
- Complexity - Could the constructs be used more effectively to decrease the architectural complexity?
- Understandability - Does the design increase the psychological complexity?
- Reusability - Does the design quality support possible reuse?
- Testability/Maintainability - Does the structure support facilities for testing and changes?

3.2 Metrics for Performance Analysis

To measure the effects in performance of Appman we need to get some metrics. Foster (1995) demonstrates in his book some performance models for parallel programs, specifying the following metrics which are detailed in the next subsection:

- Execution time (Computation Time, Communication Time, Idle Time)
- Efficiency
- Speedup

These metrics will be used to measure the performance of our case study. In the bibliographical research made by Fernandes (2003) is demonstrated that these metrics are consolidated and they adjust perfectly to this kind of paralell system.

The next subsection presents metrics of performance according to Foster (1995) definitions.

3.2.1 Execution Time

3.2.1.1 Computation Time

The computation time of an algorithm is the amount of time spent performing computation rather than communicating or idling. If we have a sequential program that performs the same computation as the parallel algorithm, we can determine by timing that program. Otherwise, we may have to implement key kernels.

Computation time will normally depend on some measure of problem size, whether that size is represented by a single parameter N or by a set of parameters. If the parallel algorithm replicates computation, then computation time will also depend on the number of tasks or processors. In a heterogeneous parallel computer (such as a local network of workstations), computation time can vary according to the processor on which computation is performed.

Computation time will also depend on characteristics of processors and their memory systems. For example, scaling problem size or number of processors can change cache performance or the effectiveness of processor pipelining. As a consequence, one cannot automatically assume that total computation time will stay constant as the number of processors changes.

3.2.1.2 Communication Time

The communication time of an algorithm is the amount of time that its tasks spend sending and receiving messages. Two distinct types of communication can be distinguished: interprocessor communication and intraprocessor communication. In interprocessor communication, two communicating tasks are located on different processors. This will always be the case if an algorithm creates one task per processor. In intraprocessor communication, two communicating tasks are located on the same processor. For simplicity, we assume that interprocessor and intraprocessor communication costs are comparable. Perhaps surprisingly, this assumption is not unreasonable in many multicomputers, unless intraprocessor communication is highly optimized. This is because the cost of the memory-to-memory copies and context switches performed in a typical implementation of intraprocessor communication is often comparable to the cost of an interprocessor communication. In other environments, such as Ethernet-connected workstations, intraprocessor communication is much faster.

3.2.1.3 Idle Time

Both computation and communication times are specified explicitly in a parallel algorithm; hence, it is generally straightforward to determine their contribution to execution time. Idle time can be more difficult to determine, however, since it often depends on the order in which operations are performed.

A processor may be idle due to lack of computation or lack of data. In the first case, idle time may be avoided by using load-balancing techniques. In the second case, the processor is idle while the computation and communication required to generate remote data are performed.

This idle time can sometimes be avoided by structuring a program so that processors perform other computation or communication while waiting for remote data. This technique is referred to as overlapping computation and communication, since local computation is performed concurrently with remote communication and computation. Such overlapping can be achieved in two ways. A simple approach is to create multiple tasks on each processor. When one task blocks waiting for remote data, execution may be able to switch to another task for which data are already available. This approach has the advantage of simplicity but is efficient only if the cost of scheduling a new task is less than the idle time cost that is avoided. Alternatively, a single task can be structured so that requests for remote data are interleaved explicitly with other computation.

3.2.2 Efficiency and Speedup

Execution time is not always the most convenient metric by which to evaluate parallel algorithm performance. As execution time tends to vary with problem size, execution times must be normalized when comparing algorithm performance at different problem sizes. Efficiency (the fraction of time that processors spend doing useful work) is a related metric that can sometimes provide a more convenient measure of parallel algorithm quality. It characterizes the effectiveness with which an algorithm uses the computational resources of a parallel computer in a way that is independent of problem size. We define relative efficiency as where is the execution time on one processor and is the time on P processors.

The related quantity relative speedup, is the factor by which execution time is reduced on P processors:

$$E_{relative} = \frac{T_1}{TPp^1}$$
$$S_{relative} = PE_1$$

The quantities defined by the equations above are called relative efficiency and speedup because they are defined with respect to the parallel algorithm executing on a single processor. They are useful when exploring the scalability of an algorithm but do not constitute an absolute figure of merit. For example, assume that we have a parallel algorithm that takes 10,000 seconds on 1 processor and 20 seconds on 1000 processors. Another algorithm takes 1000 seconds on 1 processor and 5 seconds on 1000 processors. Clearly, the second algorithm is superior for P in the range 1 to 1000. Yet it achieves a relative speedup of only 200, compared with 500 for the first algorithm.

When comparing two algorithms, it can be useful to have an algorithm-independent metric other than execution time. Hence, we define absolute efficiency and speedup, using as the baseline the uniprocessor time for the best-known algorithm. In many cases, this "best" algorithm will be the best-known uniprocessor (sequential) algorithm. In this text we frequently use the terms efficiency and speedup without qualifying them as relative or absolute. However, we will always be calculating relative efficiency and speedup.

4 REFACTORING

According to Fowler (2004) refactoring is “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.”.

Refactoring helps improving the design of software, which is the main goal of this work, makes it easier to understand, and, because of it, helps finding bugs too. (Fowler, 2004)

4.1 Refactoring Methods

4.1.1 Composing Methods

A large part of Fowler (2004) refactoring method is composing methods to package code properly.

The main methods used by the author are:

- Extract Method;
- Inline Method;
- Inline Temp;
- Replace Temp with Query;
- Introduce Explaining Variable;
- Split Temporary Variable;
- Remove Assignments to Parameters;
- Replace Method with Method Object;

- Substitute Algorithm.

Almost all the time the problems come from methods that are too long. Long methods are troublesome because they often contain lots of information, which gets buried by the complex logic that usually gets dragged in. The key refactoring is **Extract Method**, which takes a clump of code and turns it into its own method. Inline Method is essentially the opposite. You take a method call and replace it with the body of the code. Fowler (2004) needs **Inline Method** when has done multiple extractions and realize some of the resulting methods are no longer pulling their weight or if he needs to reorganize the way he has broken down methods.

The biggest problem with **Extract Method** is dealing with local variables, and temporary variables (temps) are one of the main sources of this issue. When he is working on a method, Fowler (2004) likes **Replace Temp with Query** to get rid of any temporary variables that he can remove. If the temp is used for many things, he uses **Split Temporary Variable** first to make the temp easier to replace. Sometimes, however, the temporary variables are just too tangled to replace. He needs **Replace Method with Method Object**. This allows him to break up even the most tangled method, at the cost of introducing a new class for the job.

Parameters are less of a problem than temps, provided you don't assign to them. If you do, you need **Remove Assignments to Parameters**. Once the method is broken down, he can understand how it works much better. He may also find that the algorithm can be improved to make it clearer. He then use **Substitute Algorithm** to introduce the clearer algorithm.

4.1.2 Moving Features Between Objects

One of the most fundamental decision in object design is deciding where to put responsibilities. Responsability, in this context, is the coherence between the behavior of one method related to its class. To support moving features between objects, Fowler (2004) offers the following methods:

- Move Method;
- Move Field;
- Extract Class;
- Inline Class;
- Hide Delegate;
- Remove Middle Man;
- Introduce Foreign Method;

- Introduce Local Extension.

Move Method and **Move Field** to move the behavior around. If both are needed, one can use **Move Field** first and then **Move Method**. Often classes become swollen with many responsibilities. In this case **Extract Class** can be used to separate some of these responsibilities. If a class becomes too irresponsible, **Inline Class** can merge it into another class. If another class is being used, it often is helpful to hide this fact with **Hide Delegate**. Sometimes hiding the delegate class results in constantly changing the owner's interface, in which case is necessary to use **Remove Middle Man**. The last two refactorings in this chapter, **Introduce Foreign Method** and **Introduce Local Extension** are special cases. They are used when the source code of a classis not available yet and we want to move responsabilities to this unchangeable class. If it is only one or two methods, we use **Introduce Foreign Method**; for more than one or two methods, we use **Introduce Local Extension**.

4.1.3 Organizing Data

Fowler (2004) also presents several refactorings for data organization and manipulation:

- Self Encapsulate Field;
- Replace Data Value with Object;
- Change Value to Reference;
- Change Reference to Value;
- Replace Array with Object;
- Duplicate Observed Data;
- Change Unidirectional Association to Bidirectional;
- Change Bidirectional Association to Unidirectional;
- Replace Magic Number with Symbolic Constant;
- Encapsulate Field;
- Encapsulate Collection;
- Replace Record with Data Class;
- Replace Type Code with Class;

- Replace Type Code with Subclasses;
- Replace Type Code with State/Strategy;
- Replace Subclass with Fields.

Self Encapsulate Field has long been a matter of debate about whether an object should access its own data directly or through accessors. Sometimes we do need the accessors, and then we can get them with **Self Encapsulate Field**. The direct access makes it simple to do this refactoring when needed.

The **Replace Data Value with Object** allows one to turn data into articulate objects, and if these objects are instances that will be needed in many parts of the program, **Change Value to Reference** may be used to make them into reference objects.

For arrays acting as a data structure, it is possible to make it clearer with **Replace Array with Object**. In all these cases the object is the first step. The author indicates that the real advantage comes when **Move Method** is used to add behavior to the new objects.

Magic numbers, or numbers with special meaning, have long been a problem. We can use **Replace Magic Number with Symbolic Constant** to get rid of magic numbers whenever I figure out what they are doing.

Links between objects can be one-way or two-way. Although one-way links are easier, it may be needed to use **Change Unidirectional Association to Bidirectional** to support a new function. **Change Bidirectional Association to Unidirectional** removes unnecessary complexity that could be found when one longer needs the two-way link.

The author indicates that GUI classes often do business logic that they were not supposed to do. To move the behavior into proper domain classes, one needs to have the data in the domain class and support the GUI by using **Duplicate Observed Data**.

One of the key principles of object-oriented programming is encapsulation. If any public data is exposed, one can create basic data manipulation methods for it using **Encapsulate Field**. If that data is a collection, **Encapsulate Collection** must be used, because that has special protocol. If it is an entire record, one can use **Replace Record with Data Class**.

One form of data that requires particular treatment is the type code: a special value that indicates something particular about a type of instance. If the codes are for information and do not alter the behavior of the class, we can use **Replace Type Code with Class**, which provides better type checking and a platform for moving behavior later. If the behavior of a class is affected by a type code, the use of **Replace Type Code with Subclasses** is recomended. As an alternative, **Replace Type Code with State/Strategy** can be used for flexibility.

4.1.4 Simplifying Conditional Expressions

Conditional logic may also be simplified in order to obtain better results on software quality metrics:

- Decompose Conditional;
- Consolidate Conditional Expression;
- Consolidate Duplicate Conditional Fragments;
- Remove Control Flag;
- Replace Nested Conditional with Guard Clauses;
- Replace Conditional with Polymorphism;
- Introduce Null Object;
- Introduce Assertion.

The core refactoring is **Decompose Conditional**, which entails breaking a conditional into pieces, separating the switching logic from the details of execution. The use of **Consolidate Conditional Expression** when we have several tests, all having the same effect. **Consolidate Duplicate Conditional Fragments** must be used to remove duplication within the conditional code.

Replace Nested Conditional with Guard Clauses is recomended to clarify special case conditionals and **Remove Control Flag** to get remove unwanted control flags. According to the ahtuor, Object-oriented programs often have less conditional behavior than procedural programs because much of the conditional behavior is handled by polymorphism. Polymorphism is better because conditional behavior is transparent for the caller, and it is thus easier to extend the conditions. As a result, object-oriented programs rarely have switch (case) statements. To support that, **Replace Conditional with Polymorphism** may be used. One of the most useful uses of polymorphism is to use **Introduce Null Object** to remove checks for a null value.

4.1.5 Making Method Calls Simpler

One of the main characteristics of objects are interfaces, wich, according to Fowler (2004), are a key skill in developing good object-oriented software.

The main methods used by the author for making method calls simpler are:

- Rename Method;
- Add Parameter;
- Remove Parameter;
- Separate Query from Modifier;
- Parameterize Method;
- Replace Parameter with Explicit Methods;
- Preserve Whole Object;
- Replace Parameter with Method;
- Introduce Parameter Object;
- Remove Setting Method;
- Hide Method;
- Replace Constructor with Factory Method;
- Encapsulate Downcast;
- Replace Error Code with Exception;
- Replace Exception with Test.

Fowler (2004) puts the name of a method as the simplest and most important thing one can do, giving meaning to it, and **Rename Method** can be used to suppport renaming methods. Variables and classes should be renamed. Due to the simplicity of this text replacements, the author does not suggest any extra refactorings for it, but all well known refactoring tool have facilities for that. Parameters are strongly related to interfaces. **Add Parameter** and **Remove Parameter** are common refactorings.

As a solution to long parameter lists, **Preserve Whole Object** must be used to reduce all the values to a single object, that can be created using **Introduce Parameter Object**. Parameters can also be eliminated when data can be obtained from another object to which the method already has access, using **Replace Parameter with Method**. When parameters are used to determine conditional behavior, one can use **Replace Parameter with Explicit Methods**. combination of similar methods may be done by adding a parameter with **Parameterize Method**. To avoid problems when reducing parameter lists on concurrent programming the author usually replace them with immutable objects, but otherwise he recomends to be cautious about this group of refactorings, by clearly separating methods that change state (modifiers) from those that query state (queries). Whenever they get combined, **Separate Query from Modifier** may be used to remove them.

According to Fowler (2004), ”good interfaces show only what they have to and no more”. **Hide Method** and **Remove Setting Method** may be used when we need to make things visible for a while.

Constructors are another important characteristic of OO programming, but since they force the need of knowing the class of an object one need to create, it happens to push against the flexibility of OO paradigm. The need to know can be removed with **Replace Constructor with Factory Method**.

When aiming refactoring on casting, Fowler (2004) suggests that, ”as much as possible try to avoid making the user of a class do downcasting if you can contain it elsewhere by using **Encapsulate Downcast**”.

Most OO programming languages have an exception-handling mechanism to make error handling easier but, according to the author, programmers who are not used to this often use error codes to signal trouble. The use of **Replace Error Code with Exception** is recomended to use the new exceptional features, and **Replace Exception with Test** for a previous test.

4.1.6 Dealing with Generalization

Generalization produces its own batch of refactorings, mostly dealing with moving methods around a hierarchy of inheritance:

- Pull Up Field;
- Pull Up Method;
- Pull Up Constructor Body;
- Push Down Method;
- Push Down Field;
- Extract Subclass;
- Extract Superclass;
- Extract Interface;
- Collapse Hierarchy;
- Form Template Method;
- Replace Inheritance with Delegation;

- Replace Delegation with Inheritance.

Pull Up Field and **Pull Up Method** both promote function up a hierarchy, and **Push Down Method** and **Push Down Field** push function downward. **Pull Up Constructor Body** can be used to promote constructors. Rather than pushing down a constructor, it is often useful to use **Replace Constructor with Factory Method**. When one have a similar outline body but vary in details, **Form Template Method** may be used to separate the differences from the similarities.

Extract Subclass, **Extract Superclass**, and **Extract Interface** allows the change the hierarchy by creating new classes, moving function around a hierarchy by forming new elements out of various points. **Extract Interface** us used to mark a small amount of function for the type system.

Unnecessary classes in the hierarchy, can be removed using **Collapse Hierarchy**.

Delegation may be an alternative when inheritance is not the best way of handling a situation. For this changes, the author offers **Replace Inheritance with Delegation** and **Replace Delegation with Inheritance** methods.

4.2 Refactoring and Performance

The effect refactoring has on the performance of a program is a common concern.

To make the software easier to understand, changes will often cause the program to run more slowly, wich is an important issue. Preformance cannot be ignored in favor of design purity or in hopes of faster hardware. "Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. The secret to fast software is to write tunable software first and then to tune it sufficient speed." (Fowler, 2004)

4.2.1 How Object Orientation impacts Performance

This subsection demonstrates the impact of object orientation in software performance according to Booth (2006) whose focus is on HPC software design.

4.2.1.1 Memory issues

According to Booth (2006), the layout of data in the memory space of a program can have a big impact on the programs performance, especially on the matters of HPC (High Performance Computing) applications. Almost all OO languages will implement an instance of a user defined type as a contiguous region of memory. This makes it very easy for the compiler to allocate space for user defined types. Even though the type is implemented as a contiguous region of memory it is always possible to have member variables that reference dynamically allocated data.

Either way when designing an OO code the choice for types tends to restrict for the memory layout of the application data. (Booth, 2006) Some applications lead naturally to OO modeling, while others have a more procedural approach which makes the use of arrays more natural.

The object oriented approach results in good data locality, leading to good cache use, because all of the data associated with an application element is packed into a contiguous region of memory. On the other hand if a major computational loop only uses a subset of the component data then the unwanted components are still likely to be loaded into the cache because they share cache lines with the required data. This can waste a fraction of the available memory bandwidth and reduce the performance of code sections where cache misses occur. It will also reduce the size of problem that can remain resident in the cache during the computational loop.(Booth, 2006)

This high degree of memory locality may also result in the cache misses being very closely clustered together in time which on some architectures can result in much poorer performance than a traditional array based code.(Booth, 2006)

4.2.1.2 Code structure issues

OO programming style also impacts the structure of the executable code. According to Booth (2006), most naive compilation strategies for OO languages will convert each different operation on a type into a code block roughly equivalent to a subroutine or function in a procedural language. However, the OO programming style encourages the programmer to keep the definition of each operation relatively simple and to build complexity out of multiple levels of types and operations on those types. As result, the naive compilation strategies will generate large numbers of subroutine calls. Consequences are high cost for processing the subroutine calls, inhibiting many of the code optimisations that the compiler would otherwise perform, and much greater demands on the capabilities of the compiler than conventional procedural code. The compiler not only has to be very good at subroutine inlining but also capable of performing additional optimisation steps after the in lining has taken place.

Tunning the memory layout of a type to some extent by changing the order of the component data or by storing some of the data by reference makes possible to arrange data required/not-required by most critical loops live on separate cache lines and may succeed depending on the nature of the loop and the target cache architecture.

4.2.1.3 Pointers

The use of pointers and reference types are widely used on OO programming to store the addresses of other objects and avoid having to make a copy of the object. Booth (2006) afirm that the use of pointers and references can have a detrimental effect on compiler optimisation, because in many cases the compiler cannot determine at compilation time if two pointers may reference the same location or, when calling a subroutine, it may have side effects on the region of memory referenced by a pointer. In these cases the compiler produces additional read and write instructions to ensure that values stored in CPU registers remain consistent with

values in memory. It is an important issue for many OO languages that make heavy use of pointers and references.

This way, it is understood that pointers and reference types must be used parsimony if there is a concern about software performance.

4.2.1.4 High and low level types

Booth (2006) afirm that for a high level type there is little performance disadvantage to the use of OO. On the other hand very low level types like vectors play an integral part in the most time critical parts of the code and have to function efficiently. In particular objects of this type may be created destroyed and copied within the code hotspots. For these performance critical low-level types only those language features that do not impact performance can be used. (Booth, 2006)

However, according to the author, these low-level types are usually very simple and can even be replaced by explicit code using intrinsic types without too much damage to the code. Higher level objects which can have quite complex behaviour. This means that a class takes part in all of the code hotspots so it may be necessary to modify the type interface and do some damage to the encapsulation of the class type to improve the performance of the code. (Booth, 2006)

4.3 Related Work

Stroggylos and Spinellis (2007) analyzed source code version control system logs of popular open source software systems to detect changes marked as refactorings and examine how the software metrics are affected by this process, in order to evaluate whether refactoring is effectively used as a means to improve software quality within the open source community.

The results indicate a significant change of certain metrics to the worse. Specifically he seems that refactoring caused a non trivial increase in metrics such as LCOM (Lines of code), Ca (Afferent Couplings) and RFC (Response for Class), indicating that it caused classes to become less coherent as more responsibilities are assigned to them. The same principles he seems to apply in procedural systems as well, in which case the effect is captured as an increase in complexity metrics. Since it is a common conjecture that the metrics used can actually indicate a system's quality, these results suggest that either the refactoring process does not always improve the quality of a system in a measurable way or that developers still have not managed to use refactoring effectively as a means to improve software quality. To further validate these results, more systems and even more revisions must be examined, because the number examined so far is relatively small. He suggests that using a refactoring detection technique to identify the refactorings performed each time, one could also correlate each kind of refactoring to a specific trend in the change of various metrics and thus deduce which ones are more beneficial to the overall quality of a system.

Another good related work is from Bois et al. (2004) which analyzes how refactoring methods manipulate coupling/cohesion characteristics and how to identify refactoring opportunities that improve these characteristics. They also provide practical guidelines for the optimal usage of refactoring in a software maintenance process.

At the end, they identified specific applications of Move Method, Replace Method with Method Object, Replace Data Value with Object and Extract Class to be beneficial. However, they also experienced that guidelines they created can be insufficiently specific. This was the case for a specific application of Extract Method, which was harmful for cohesion.

They demonstrated in the conclusion that by exploiting the results from coupling/cohesion impact analysis, it is possible to achieve quality improvements with restricted refactoring efforts and that this effort is restricted to the analysis and resolutions of a limited set of refactoring opportunities which are know to improve the associated quality attributes.

The related work from Stroggylos and Spinellis (2007) shows that refactoring may not improve software quality. Besides that, the developers probably are not looking refactoring as a opportunity to improve software quality. In this work the refactoring methods applied must improve the metric and if not this change is rolled back.

Bois et al. (2004) work give a good reference to use to improve this quality metrics because of its guidelines and profiles described. These guidelines can help making the refactoring process more accurate.

5 REFACTORING APPMAN

This work is focused to deal with the application of the object-oriented paradigm in accordance with software quality metrics. A case study, which was demonstrated with more details in the Section 2.1, was used to validate and to extract data. Through the application of quality metrics, we can see that the current implementation of the case study does not use the object-oriented paradigm or that it was not implemented in adequate form. From this process, the performance metrics of the system was extracted and later used to relate with the software metrics after-refactoring.

To prevent distortions in the results, the system only had its class diagrams modified and techniques of refactoring applied in its code, without changing the used logic and technologies to implement the system.

After this refactoring, the quality metrics and performance metrics were analyzed with the new codification. Some cycles of refactoring should to be done to achieve good software metrics for evaluation according to the defined limits in Eclipse Metrics Plugin (Walton, 2007). In this work we present a first refactoring cycle, which presented promising results.

5.1 Appman Software Analysis

In Figure 5.1 is demonstrated a simplified class diagram of the last stable version of the system. AppMan have been developed by a group of researchers, so the SVN version control system is used. We obtained a stable version and created a new branch for our modifications. Bernardo (2007) is working in parallel with these studies, also using the Appman prototype. After development, both works will be integrated through a possible merge between the branch codes.

The main classes, which have responsibilities to control the grid and the main business logic of Appman are: ApplicationManager, SubmissionManager, and TaskManager. They are available as a remote service in EXEHDA, and the reason why

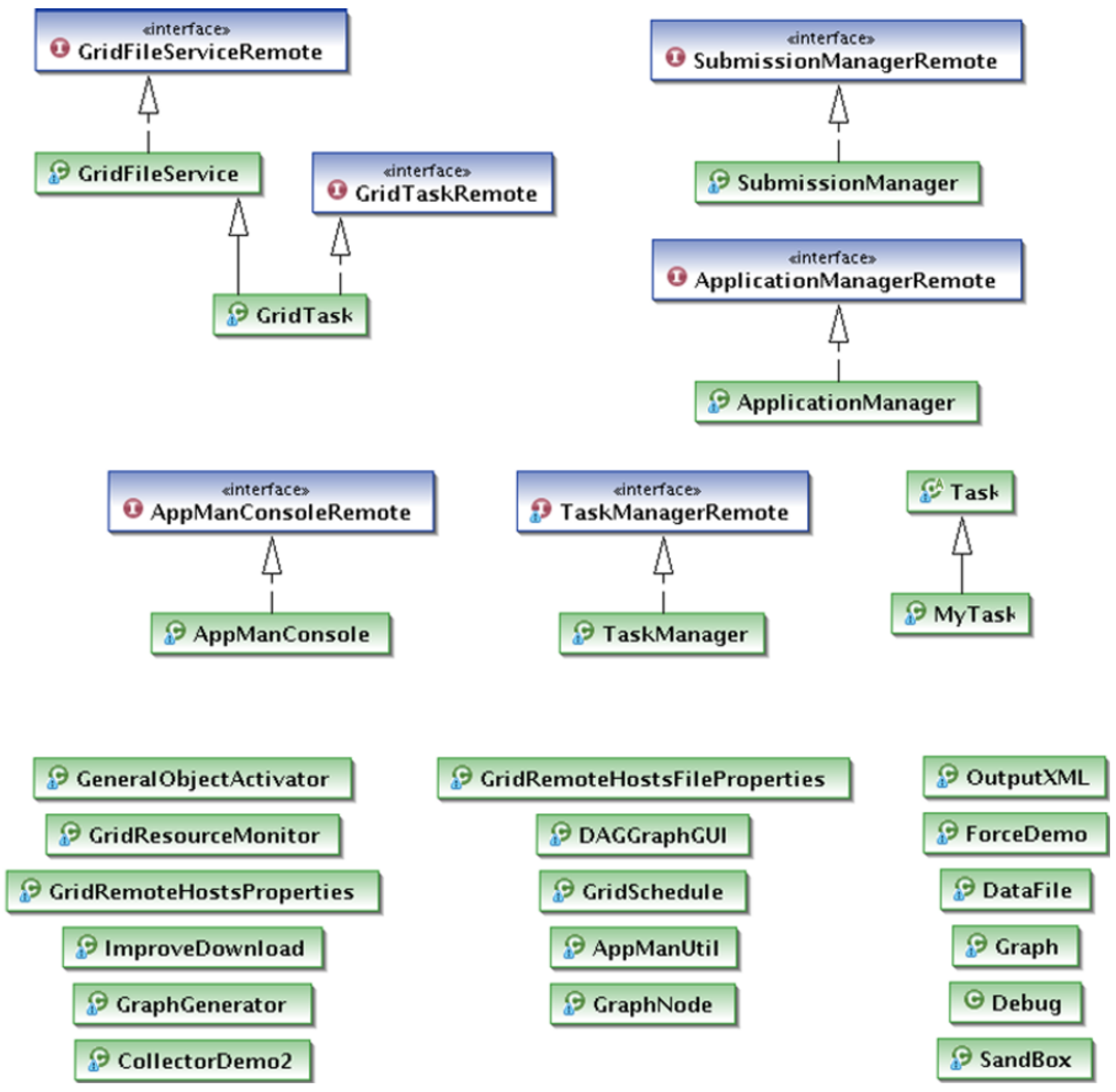

Figure 5.1: Class Diagram - AppMan

they implement some remote interfaces. Other remote services available are: GridFileService, which provides control over the file transmission between the nodes and AppManConsole which is a deprecated form of gathering monitor information from Appman. Other classes have some general responsibilities like utility methods, XML parsing, graph control, graphic interface, file reading and writing, and others.

5.2 Tools

Since AppMan is implemented in Java, using ISAM/Exehda middleware, we choose tools capable of dealing with such kind of language.

5.2.1 Software for Source Code Refactoring

Eclipse Europe 3.3 (Walton, 2008a) is a Java IDE which has many refactoring capabilities. The goal of Java program refactoring is to make system-wide code changes without affecting the behavior of the program.

When performing a refactoring operation, the user can optionally preview all the resultant changes from a refactoring action before choosing to carry them out. When previewing a refactoring operation, the user will be notified of potential problems and will receive a list of the changes that the refactoring action will perform.

The refactoring tools support a number of transformations described in (Fowler, 2004), such as Extract Method, Inline Local Variable, etc. A full list of the refactoring methods supported by this tool is demonstrated in Table 5.1. This information was obtained in the software help documentation.

Besides, Eclipse registers all refactoring history, with details, which can help analysis as well as will help future works.

5.2.2 Software for Metrics Extraction

To extract the software quality metrics from Appman source code, Eclipse Metrics Plugin 3.6 (Walton, 2007) was used. It works as plugin for Eclipse Europe 3.3. The tool can analyze the following metrics:

- McCabes Cyclomatic Complexity;
- Efferent Couplings;
- Lack of Cohesion in Methods;
- Lines Of Code in Method;
- Number Of Fields;
- Number Of Level;
- Number Of Parameters;
- Number Of Statement;
- Weighted Methods Per Class.

This tool shows warning messages notifying the developer about metric limits which has been exceeded making easier the refactoring process. It also allows the user to generate reports showing graphics and tables from the project. The metrics are flexible allowing the user to determine the limits of each metric.

Table 5.1: Refactoring Methods
Source: (Walton, 2007)

Name	**Description**
Rename	Renames the selected element and (if enabled) corrects all references to the elements (also in other files).
Move	Moves the selected elements and (if enabled) corrects all references to the elements (also in other files).
Change Method Signature	Changes parameter names, parameter types, parameter order and updates all references to the corresponding method. Additionally, parameters can be removed or added and method return type as well as its visibility can be changed.
Extract Method	Creates a new method containing the statements or expression currently selected and replaces the selection with a reference to the new method. This feature is useful for cleaning up lengthy, cluttered, or overly-complicated methods.
Extract Local Variable	Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.
Extract Constant	Creates a static final field from the selected expression and substitutes a field reference, and optionally rewrites other places where the same expression occurs.
Inline	Inline local variables, methods or constants.
Convert Anonymous Class to Nested	Converts an anonymous inner class to a member class. Convert Member Type to Top Level Creates a new Java compilation unit for the selected member type, updating all references as needed. For non-static member types, a field is added to allow access to the former enclosing instance, if necessary.
Convert Local Variable to Field	Turn a local variable into a field. If the variable is initialized on creation, then the operation moves the initialization to the new field's declaration or to the class's constructors.
Extract Superclass	Extracts a common superclass from a set of sibling types. The selected sibling types become direct subclasses of the extracted superclass after applying the refactoring.
Extract Interface	Creates a new interface with a set of methods and makes the selected class implement the interface.
Use Supertype Where Possible	Replaces occurrences of a type with one of its supertypes after identifying all places where this replacement is possible.
Push Down	Moves a set of methods and fields from a class to its subclasses.
Pull Up	Moves a field or method to a superclass of its declaring class or (in the case of methods) declares the method as abstract in the superclass.
Introduce Indirection	Creates a static indirection method delegating to the selected method.
Introduce Factory	Creates a new factory method, which will call a selected constructor and return the created object. All references to the constructor will be replaced by calls to the new factory method.
Introduce Parameter Object	Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduce parameter.
Introduce Parameter	Replaces an expression with a reference to a new method parameter, and updates all callers of the method to pass the expression as the value of that parameter.
Encapsulate Field	Replaces all references to a field with getting and setting methods.
Generalize Declared Type	Allows the user to choose a supertype of the reference's current type. If the reference can be safely changed to the new type, it is.
Infer Generic Type Arguments	Replaces raw type occurrences of generic types by parameterized types after identifying all places where this replacement is possible.

5.2.3 Software for Performance Metrics Extraction

No special software was used for performance metrics extraction. Appman has log files which generate performance related information. Data were collected from these log files and imported to an OpenOffice spreadsheet, which has capabilities for generating graphics about these metrics.

The following measures are generated using Appman log files for each execution process: Parser Time, InferDAG Time, Scheduler Time, Download Time and Total Execution Time. Parser Time correspond to input file reading and parsing while InferDAG Time is the elapsed time between receiving the input file information, its representation as a DAG (Directed Acyclic Graph) using a data structure in memory, the clustering definition, and the XML outputs generation. The Scheduler Time aims to get how much time it takes before having a ready cluster to start preparing its actual submission. The Download time correspond to the stage in process. Finally, the execution time, which includes Download Time, measures all stages from stage in, execution, and stage out of the task. Performance metrics was calculated for most of this information, as next sections presents.

5.3 Metrics from Original Source Code

Performance and software metrics were collected from the original source code of Appman. These metrics will be used as the baseline for comparing all refactoring changes in the next sections.

5.3.1 Performance Metrics

Before starting the refactoring process, some performance metrics had to be collected to be used for evaluation at the end of the process, so it can be seen if Appman performance analysis has been affected.

5.3.1.1 Total Execution Time

To evaluate the total execution time, four machines were used to run Appman. Four kinds of tests were made and are represented in Table 5.2:

Table 5.2: Configuration used for Appman Performance Tests

Configuration	Number of Processors	Representation used in Graphs
1 Submission Manager x 1 Task Manager	1 machine	1SM x 1TM
1 Submission Manager x 2 Task Managers	2 machines	1SM x 2TM
1 Submission Manager x 4 Task Managers	4 machines	1SM x 4TM
2 Submission Managers x 2 Task Managers	2 machines	2SM x 2TM

Results presented in Figure 5.2 show that the **execution time** got worst when two submission managers are used and more than 80 tasks submitted. This happens because the number of tasks (amount of workload) is not big enough to justify two submission managers handling submissions.

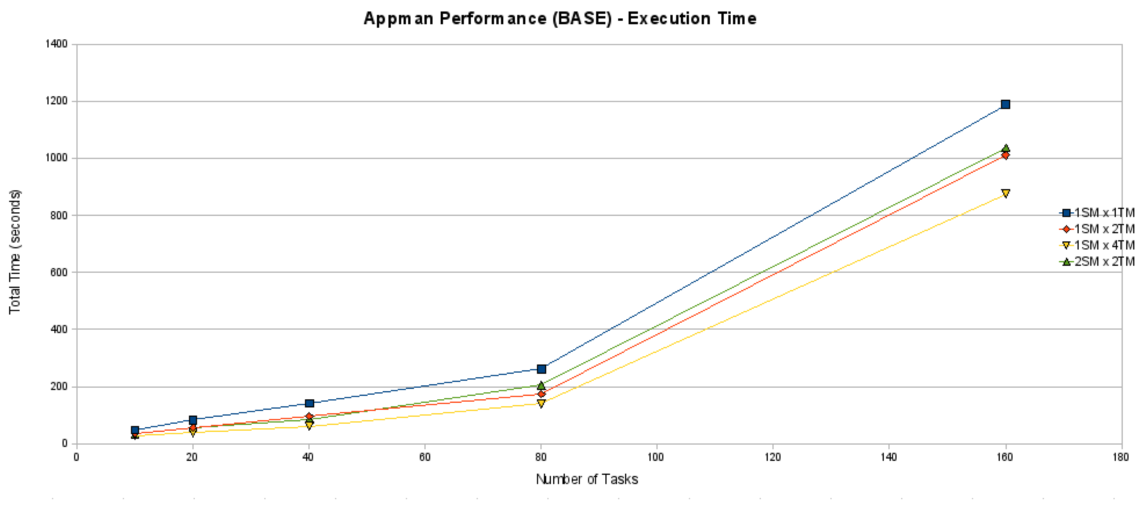

Figure 5.2: Appman Performance Metric - Execution Time

Another information from this graph is that when incrementing the number of task managers, the execution time decreases, at least working with up to four task managers. This behavior was expected, since more processors are executing the tasks in parallel.

5.3.1.2 Speedup

An important performance metric is Speedup. As discussed in Section 3, Speedup allows to visualize the improvement degree obtained in parallel execution.

As observed in Figure 5.3, when increasing the number of processors, some speedup is obtained. But, as expected, for all instances, the speedup is bellow the ideal treshold.

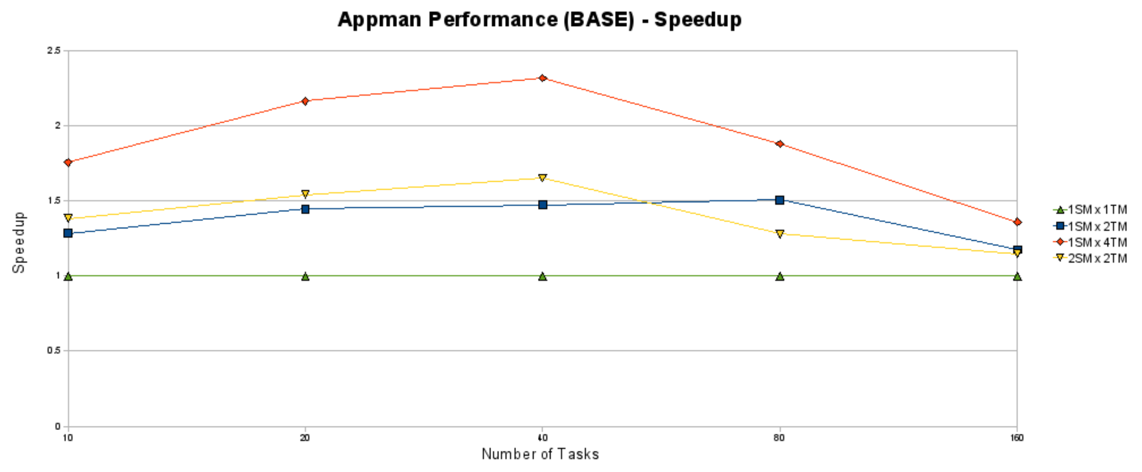

Figure 5.3: Appman Performance Metric (Base) - Speedup

5.3.1.3 Efficiency

Efficiency is the third metric analyzed in this work. This metric allows to visualize how much of the capacity of the system is really used by the system. The results presented in Figure 5.4 were calculated using the simplified formula $E(P) = S/P$, where E is the efficiency for P number of processors and S is the Speedup.

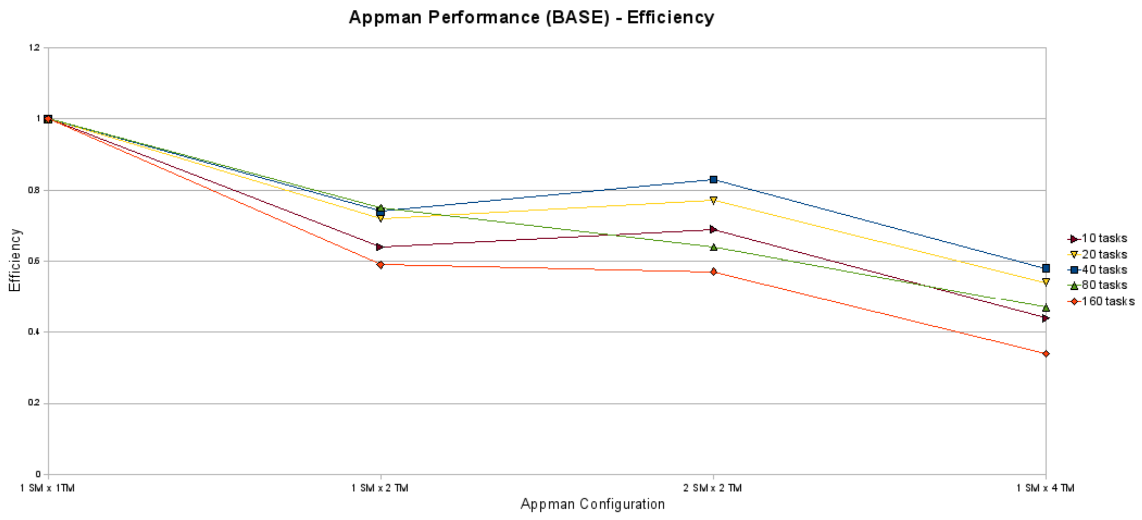

Figure 5.4: Appman Performance Metric (Base) - Efficiency

In the Graph shown on Figure 5.4 is possible to observe a low efficiency for the four machines. This can be justified by the application: there are not enough tasks for sustaining the four CPUs in constant use.

5.3.2 Software Metrics

In this Section is demonstrated the software quality metrics from the original version of Appman.

5.3.2.1 Lack of Cohesion Metrics

There are basically three calculation methods according to Eclipse Metrics Tools (Walton, 2007) to Lack of Cohesion Metric (LCOM), and all three was analyzed for the baseline version of AppMan.

Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field. When this value is negative, the metric value is set to 0. Thus, we can define as following:

- $n1$ = number of pairs of distinct methods in a reftype that do not have at least one commonly accessed field

- *n2* = number of pairs of distinct methods in a reftype that do have at least one commonly accessed field
- *LCOM* = ((n1 - n2)/2 max 0)

The formula presented bellow is the Lack Of Cohesion used in the base version of Appman according to Chidamber and Kemerer metric. The values for such metric are presented in Figure 5.5. As we can observe, the only class that had a big value for this metric is the "Task" class.

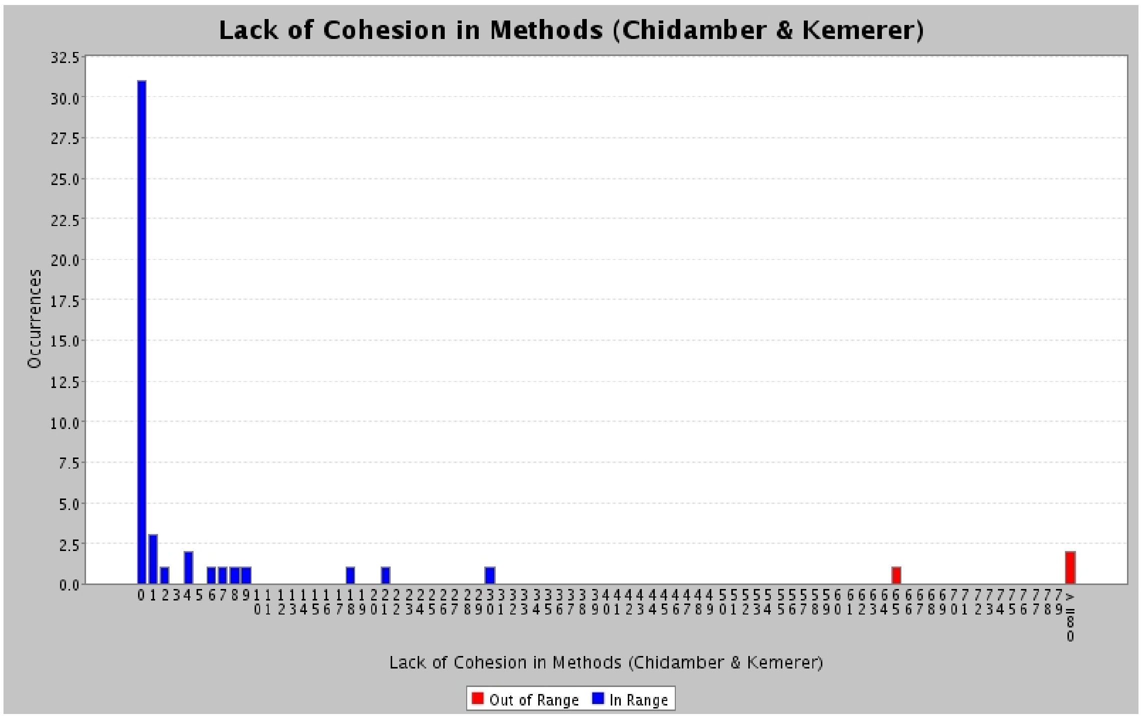

Figure 5.5: Lack of Cohesion - Chidamber Kemerer Method

The second LCOM metric is called **Henderson-Sellers**. Henderson-Sellers defines Lack of Cohesion in Methods as follows:

- *M* = the set of methods defined by the class
- *F* = the set of fields defined by the class
- *r(f)* = the number of methods that access field *f*, where *f* is a member of *F*
- *<r>* be the mean of r(f) over F.
- *LCOM = (<r> - |M|) / (1 - |M|)*

The Figure 5.6 presents the LCOM metric for Henderson Sellers Method in the baseline version of AppMan. With this metric, it is possible to see that at least 50% of the Appman classes do not have a good cohesive index value.

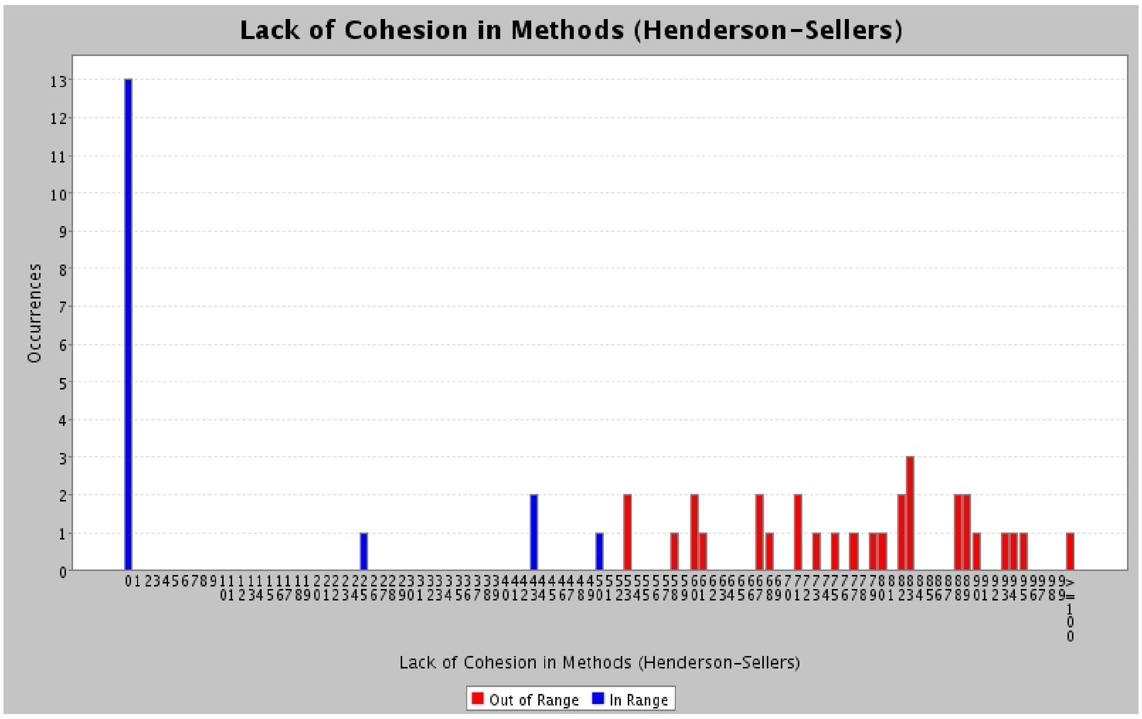

Figure 5.6: Lack of Cohesion - Henderson Sellers Method

Another metric measured by Eclipse Metrics Tool is **Total Correlation**, which is the definition of Lack of Cohesion which uses Watanabe's (1960) generalization of mutual information known as Total Correlation, which determines if a group of variables exhibit redundancy or structure. According to the author of (Walton, 2007), each method in a class makes use of a subset of the fields of the class.

This was used to know whether these subsets exhibit some structure between the fields. If such a structure exists then it can be extracted into one or more other classes in order to remove or reduce the structure. Therefore it can be considered that the use of each field by methods as a 'random' binary variable with a certain probability of occurrence.

The rest follows naturally from the definition of Total Correlation. In this tool, this metric is analyzed including only methods in M if they access at least one field. The reason for this is that methods that do not access fields are often required to be non-static for reasons of polymorphic dispatch. However, these kinds of methods skew the value of this metric in a way that is not helpful. Fields in F are considered only it if they are also accessed by at least one method in the class.

Thus, the Figure 5.7 represents values extracted from AppMan for Total Correlation Method. More than 75% of Appman source code does not present a good cohesive index according to the Total Correlation method.

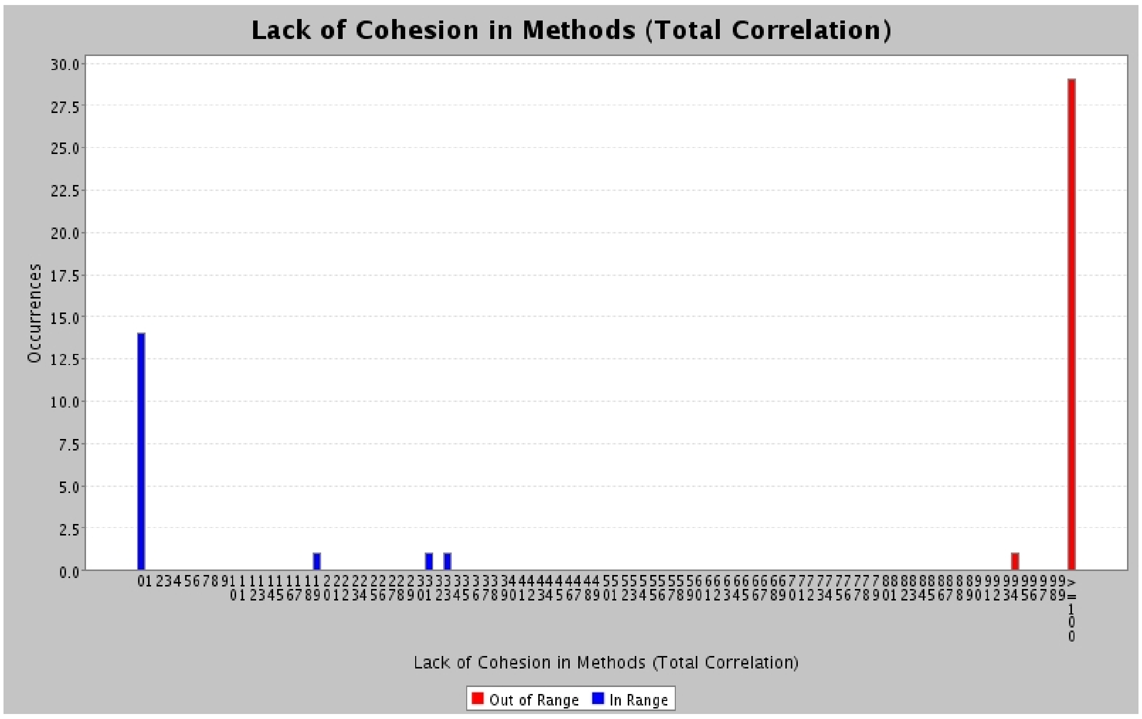

Figure 5.7: Lack of Cohesion - Total Correlation Method

5.3.2.2 Complexity Metrics

The **Cyclomatic Complexity** by McCabe definition is used to evaluate the complexity of an algorithm in a method.

According to Eclipse Metrics documentation, a method with no branches has a Cyclomatic Complexity of one since there is one arc. This number is incremented whenever a branch is encountered. In this implementation, statements that represent branching are defined as: 'for', 'while', 'do', 'if', 'case' (optional), 'catch' (optional) and the ternary operator (optional). The sum of Cyclomatic Complexities for methods in local classes is also included in the total for a method.

We can observe in Figure 5.8 the Cyclomatic Complexity Metric obtained by Method over AppMan original code.

Considering Cyclomatic Complexity, most of Appman classes which have high complexity degree are related to XML parser, which has many recursive methods and loop commands. Since it is executed only one time, it will not be prioritized for initial refactoring.

In Figure 5.9, the same metric is analyzed in the context of class (by type).

Both by method and by type, the same situation occurs, presenting high complexity due to the XML parsing methods.

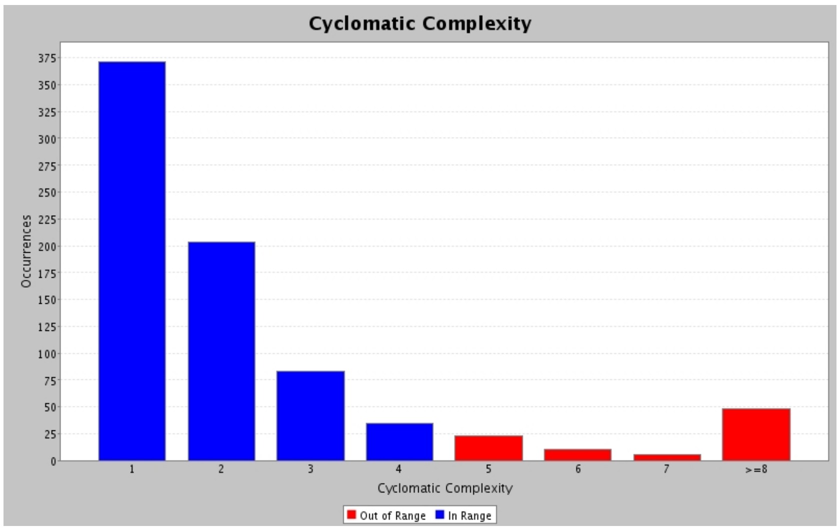

Figure 5.8: Cyclomatic Complexity - by Method

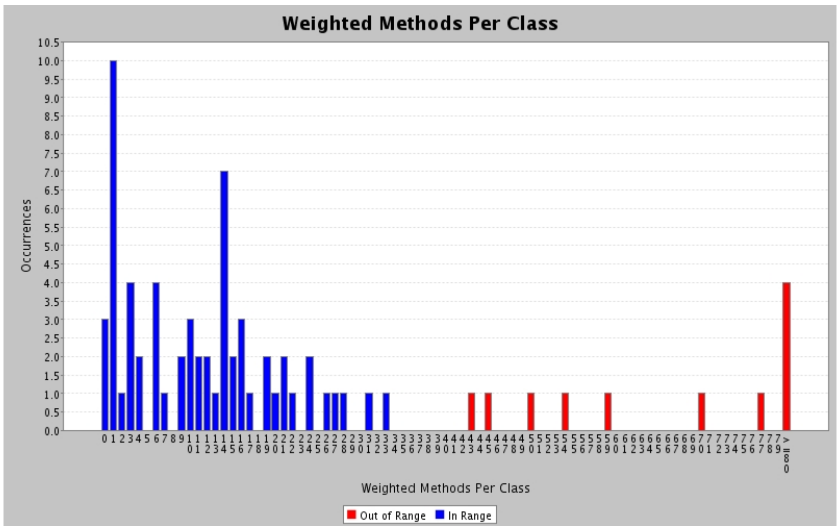

Figure 5.9: Cyclomatic Complexity - by Type

5.3.2.3 Other Metrics

According to (Walton, 2007) documentation, the **Efferent Couplings** metric measures the number of types of the class being measured 'knows' about. This includes: inheritance, interface implementation, parameter types, variable types, thrown and caught exceptions. In brief, all types referred to anywhere within the source of the measured class.

As can be observed in Figure 5.10, Appman have a very low coupling which is very good in the software quality context.

The **Number of Levels** metric is defined by (Walton, 2007) as an indication of the maximum number of levels of nesting in a method. The idea of this metric is that a large Number Of Levels increases complexity and reduces comprehensibility.

Figure 5.11 presents data for this metric. Just a few methods has a high number

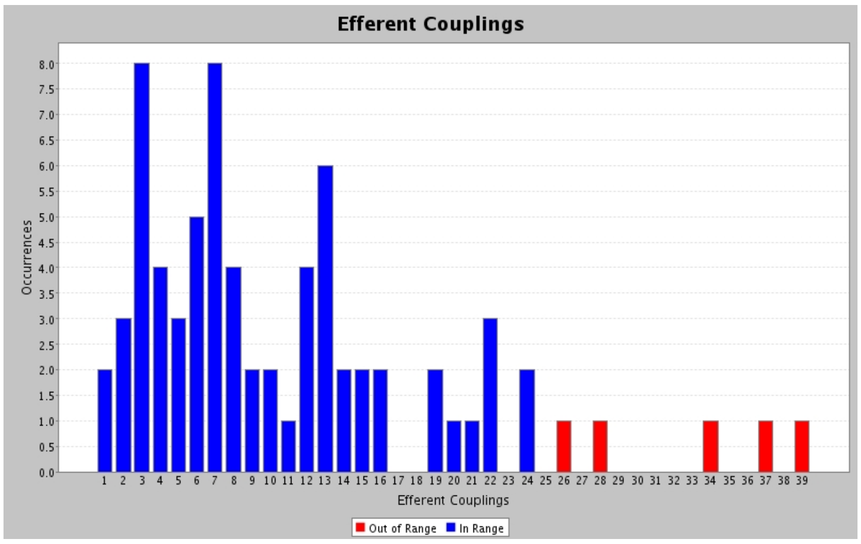

Figure 5.10: Efferent Couplings

of levels, but probably changes in this methods would have effect in the Cyclomatic Complexity metric.

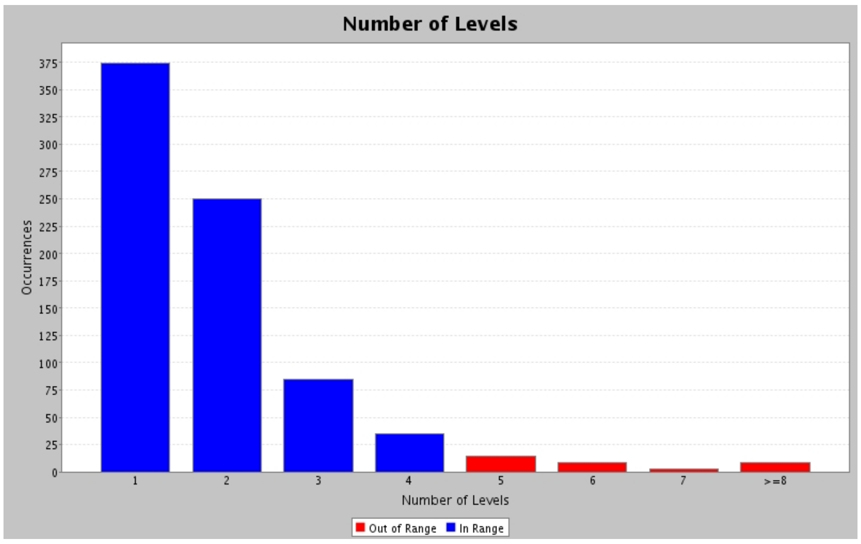

Figure 5.11: Number of Levels

The **Lines of Code** metric indicates the number of lines a method presents. In the context of Eclipse Metrics Tool, a line is determined by the presence of a newline character. It is a very basic measure of size and is susceptible to variation purely on the basis of different source code formatting styles. Although controversial, it is one of the simplest and most known software metrics.

As we can observe in Figure 5.12, almost 50% of the methods in Appman has more than 15 lines of code, which is the default bound of number of lines in a method by the evaluation of the software Eclipse Metrics.

The **Number of Fields** metrics just counts the total of fields in a class, and

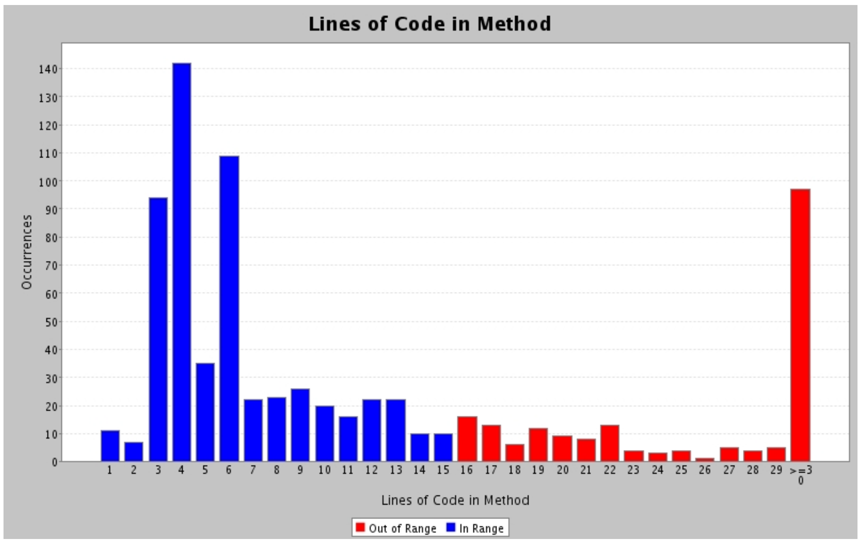

Figure 5.12: Lines of Code

may indicate classes that maybe have more responsibilities it should be. This metric value for original AppMan code is represented in the Figure 5.13.

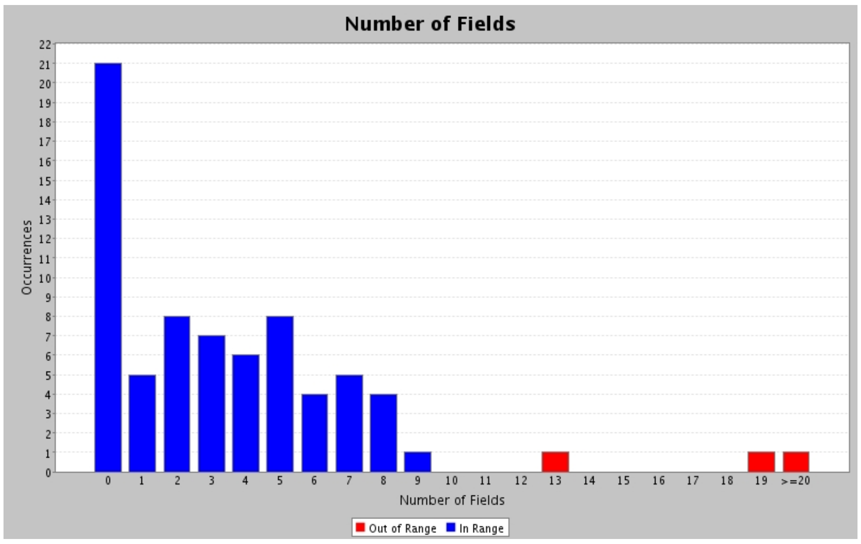

Figure 5.13: Number Of Fields

Only 3 classes had more than ten fields in the class, so these classes could be analyzed to increase the cohesion in the system by breaking them up into more classes when appropriated.

The **Number of Parameters** metric counts parameters in each method, and can give indications on classes that can generate new classes using these set of data in the method parameters. Usually, this new classes attract other behavior that may be lost in another class.

Figure 5.14 have all Number of Parameters metric values. In this case, the number of methods which have more than 5 parameters is almost insignificant and

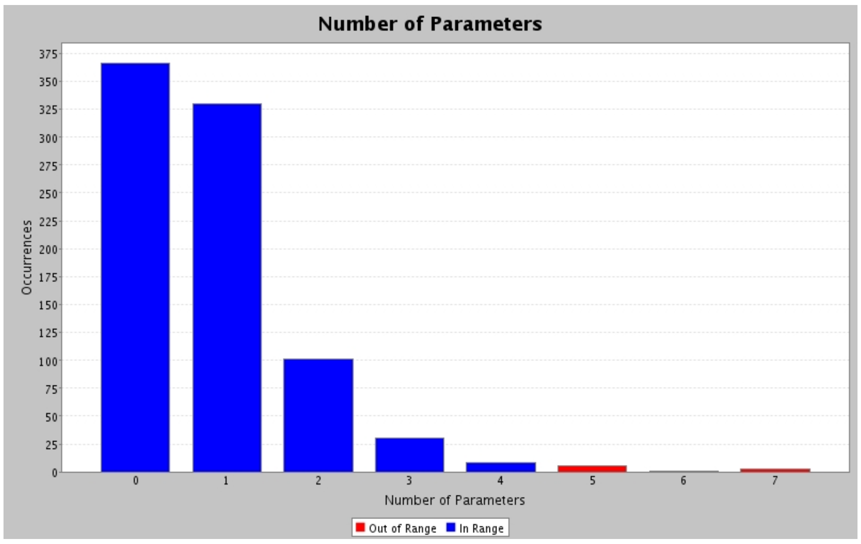

Figure 5.14: Number Of Parameters

probably will not be considered in the refactoring process.

According to (Walton, 2007) documentation this is a more robust measure than the Lines of Code because it does not depend on the source code formatting style. Also, a high number of statements don't mean it is bad but allows to Extract new Methods and make them more cohesive.

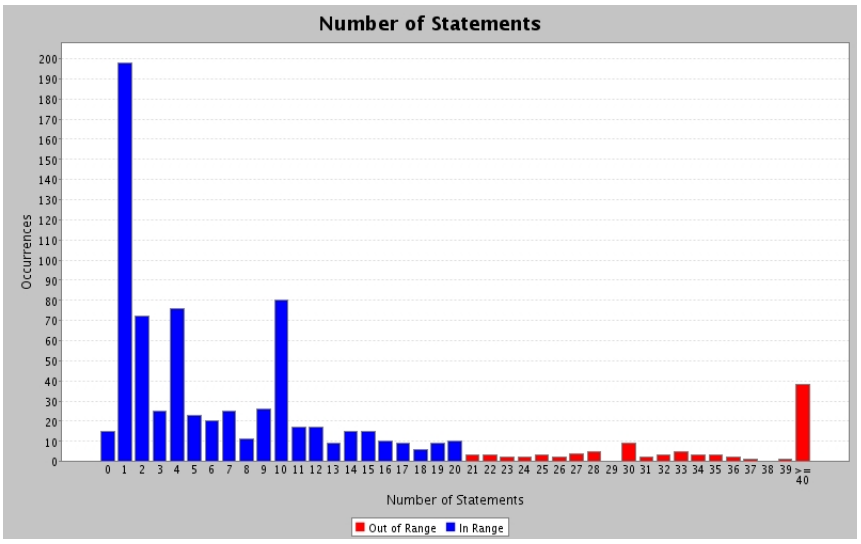

Figure 5.15: Number Of Statements

Appman has a medium number of methods with a high number of statements and probably these methods are not very cohesive so this must be good metric to be considered in the refactoring process.

5.3.2.4 Memory Consumption

Just to enrich the information provided in this work, a collection of memory consumption during the execution of tests was evaluated.

This memory consumption analysis has been done in the Submission Machine which according to the GRAND Model definition (Mangan, 2006) is the user machine used to submit an application. GRAND aims to avoid that Submission Machine became overloaded through distributed submission management. Thus, one of our concerns was to ensure that our refactoring process did not significantly impact in memory consumption.

This data was collected with the method *Runtime.totalMemory()* of Sun Java 2 Platform SE 5.0. According to the documentation (Microsystems, 2008) this method returns the amount of memory currently available for current and future objects, measured in bytes. Although getting slower performance with this, for more reliable results, the Java Virtual Machine was set to reserve the minimum memory possible and will increment as it needs until the limit of 512 Mbytes which is almost the maximum memory available in the machines.

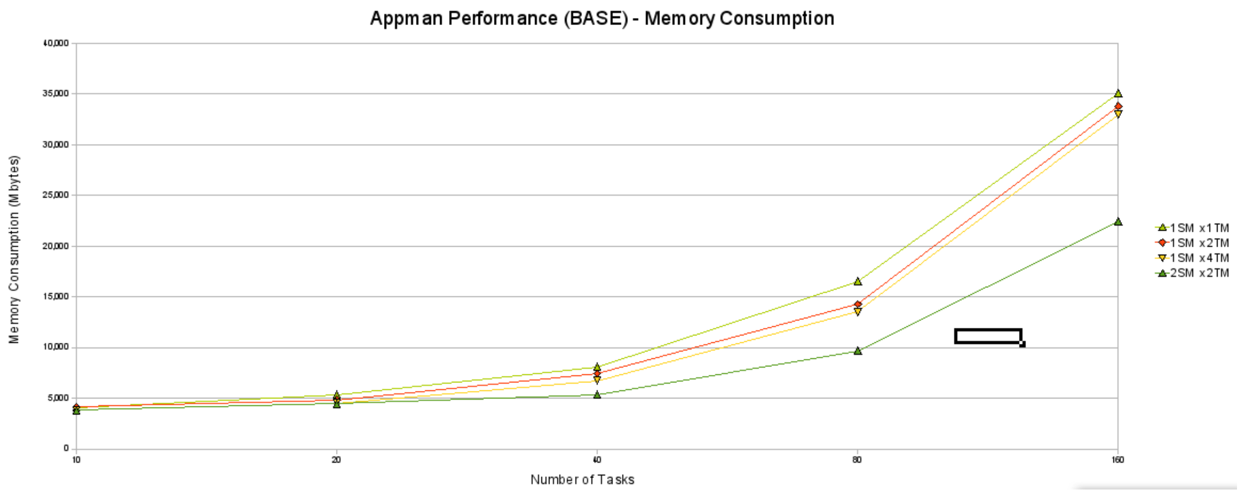

Figure 5.16: Memory Consumption

Currently AppMan does not looks like a very memory consuming software even working with many tasks. Memory, for sure, is not the limitation for the number of tasks it can handle. In the tests, AppMan could not handle more than 320 tasks or even less depending on the resources available. But this kind of analysis is not the focus of this work.

5.4 Refactoring Process

In the previous section, the performance and software quality metrics from Appman original source code was presented and analyzed. Now, with this information, it was possible to define the most appropriated approach to improve some quality metrics using the refactoring methods presented in the Section 3. The main steps and results of the refactoring process are presented in the following subsections.

5.4.1 Identification of Refactoring Points

To extract the refactoring points some considerations must be done.

5.4.1.1 About Software Quality Metrics

Although many metrics were listed in Section 5.3.2, this work aims to focus on object-oriented paradigm analysis. We are going to focus on the improvement of some object-oriented quality attributes which, in this case, is the maintainability and reusability. To improve this quality attributes there are some key concepts in object-orientation we need attention which is the classes cohesion and coupling. So we are going to choose the software quality metrics which can measure these key concepts.

5.4.1.2 About Appman Software Quality

As shown in Figure 5.10 system coupling is not something to worry in Appman software design. On the other hand, the lack of cohesion demonstrated in Figures 5.7 and 5.6 is very high, so this can be a great opportunity for software quality improvement.

As listed in Table 5.4, the top six classes with lack of cohesion are exactly the main classes of Appman for Application Management, Submission of Tasks and the Task itself. So, most of the refactoring decisions will be focusing in improving the cohesion of these classes.

Table 5.3: Appman - Top Lack of Cohesion Classes ordered by Chidamber & Kemerer

LCOM-CK	LCOM-HS %	LCOM-TC %	Type	Package
619	95	255	Task	appman
82	90	515	Application Manager	appman
65	88	162	Submission Manager	appman
30	79	444	OutputXML	appman
21	83	115	SimpleNode	appman.parser
18	82	153	TaskManager	appman

5.4.1.3 About Grid Application Management

This work focus on software quality improvement for grid applications, so another criteria to choose the priority classes to improve is to choose those which have grid application management responsibilities. This criteria is important since any changes on them can impact directly on grid submission and distribution performance.

The main classes with grid application management responsibilities on Appman are from package *appman*: ApplicationManager, SubmissionManager and TaskManager. As can be seen in Table 5.4, it's all in the top six classes with lack of cohesion.

5.4.2 Cycle One - Lack of Cohesion

We present our main contribution in this section: the so called Cycle One of Refactoring. We call it cycle one, since it is implicit that several refactoring aspects can and will be threaded in futures as consequence of this initial source code improvement.

In this cycle the lack of cohesion was analyzed and the main classes of the Appman classes, as previously discussed, are changed, because of the high lack of cohesion detected by the metrics.

To make a good refactoring with focus on software quality metrics, first it is necessary to understand how these metrics works, because any changes in the software modeling can impact positive for one metric but maybe negative to another, even when metrics have the same goal.

5.4.2.1 Approach

According to Gupta (1997) the metric from Chidamber and Kemerer have many critiques. Researchers have proposed various desirable properties of software measures and Gupta selected many of them to create a framework for use to constrain and guide the search for new metrics. His argumentation is based on these properties and he concluded that Chidamber and Kemerer's metrics don't follow this basic properties:

- The value of max varies from class to class as the number of methods varies so the property which defines that the cohesion of a module lies in a normalization range between 0 and some maximum is not satisfied.
- The metric is not able to distinguish between the structural cohesiveness of two classes, i.e., the way in which the methods share instance variable.

In the documentation of the open source tool Eclipse Metrics (Forge, 2008) author refers "I'm unsure of the usefulness of this metric in Java since it penalizes the proper use of getters and setters as the only methods that directly access an attribute and the other methods using the getter/setter methods." The author explains that he could alter the implementation to take this into account, assuming standard JavaBean naming conventions.

This explain why, in many cases, it was very hard to improve the software cohesion metrics when there were many set/get methods. Using set/get methods implies a high number for Chidamber and Kemerer metric. Then, this work will consider Chidamber and Kemerer's as not a good reference to the context of this project. **Henderson and Sellers's metric** which is an evolution of Chidamber and Kemerer's metric is widely used for software cohesion measurement too but was not easy not make refactorings which affects very well this metric because of it's complexity. So will be used Chidamber and Kemberer as reference just because it is a most understandable metric which is based on a Combination Analysis theory.

The Total Correlation metric, created by the Eclipse Metrics's (Walton, 2007) author, is not widely used in scientific papers, so it's not prioritized in this work, too.

5.4.2.2 Refactoring Methods Applied

Many refactoring methods was applied, but the most relevant is presented here. In the Appendix, a Refactoring History Report shows more details about the refactoring process.

For each class we first consider its baseline metric values. For instance, these are the actual metrics for **Task** Class:

- CK (Chidamber Kemerer Method) = 619
- HS (Henderson Sellers Method) = 95%
- TC (Total Correlation) = 255%

To improve metrics related to **Task** class, the following refactoring methods are applied:

- Organizing Data: Replace Type Code with State/Strategy (generating class TaskState and others)
- Simplifying Conditional Expressions: Replace Conditional with Polymorphism

Thus, Metrics Results are:

- CK (Chidamber Kemerer Method) = 583
- HS (Henderson Sellers Method) = 95%
- TC (Total Correlation) = 257%

Then, new Refactoring methods are applied:

- Organizing Data: Replace Type Code with Class (TaskType)

- Composing Methods: Extract Method (initialize(), analyzeFaultTolerance(), updateOutputFiles())
- Moving Features Between Objects: Extract Class (TaskTimer)
- Moving Features Between Objects: Extract Class (TaskFiles)
- Moving Features Between Objects: Move Methods
- Organizing Data: Encapsulate Fields
- Composing Methods: Inline Method (setTaskState(), getTaskState(), getTaskStateString())

Specially the Extract Method and Extract Class refactoring methods had great influence in the metrics above:

- CK (Chidamber Kemerer Method) = 165
- HS (Henderson Sellers Method) = 90%
- TC (Total Correlation) = 303%

For the **ApplicationManager** class, the metric values are the following:

- CK (Chidamber Kemerer Method) = 82
- HS (Henderson Sellers Method) = 90%
- TC (Total Correlation) = 515%

After some refactoring methods are applied:

- Moving Features Between Objects: Extract Class (ApplicationManagerTimer)
- Composing Methods: Extract Method (printTraceInfo(), printFinishTimeInfo())
- Composing Methods: Inline Method (getApplicationStatePercentCompletedRemote(), addGraphRemote())
- Organizing Data: Encapsulate Fields

The metrics results are:

- CK (Chidamber Kemerer Method) = 49
- HS (Henderson Sellers Method) = 88%

- TC (Total Correlation) = 360%

For the **TaskManager** and **SubmissionManager** classes, no many changes was done because they have good metric values and the cohesion of the classes looks fine. But only for test purposes, the set/get methods for the class fields was removed just to see the impact of it in the metric.

Bellow the metrics with set and get methods for each **SubmissionManager** class field:

- CK (Chidamber Kemerer Method) = 60
- HS (Henderson Sellers Method) = 88%
- TC (Total Correlation) = 167%

After removing the set/get methods:

- CK (Chidamber Kemerer Method) = 0
- HS (Henderson Sellers Method) = 76%
- TC (Total Correlation) = 119%

See that Chidamber and Kemerer metric was above the limit (sixty) and just removing the set/get methods became zero. This confirms the influence of this kind of methods in this metric. The encapsulation is a key feature of object-orientation and cannot make that class bad cohesive.

5.4.2.3 Metrics

In this section, we will analyze the metrics after finishing all refactoring, for all classes. Thus, these are the collected metrics after the Cycle One of refactoring.

Software Quality Metrics

At this time, the work will focus on object-oriented and complexity related metrics because they have relevant information for the purpose of this work.

Figure 5.17 presents all values for LCOM's Chidamber and Kemerer. So, we can see that we got a better situation for Chidamber and Kemerer metric.

This improvement can also be observed in LCOM's Henderson-Sellers (Figure 5.18). However, since most of the changes was focused on Chidamber and Kemerer math for metric calculation, the changes we obtained in this metric are not meaningful.

Finally, the Figure 5.19 presents the third LCOM method: Total Correlation.

The set and get methods make a lot of difference in this metric because they do not treat this kind of information separably from others methods. A good cohesive

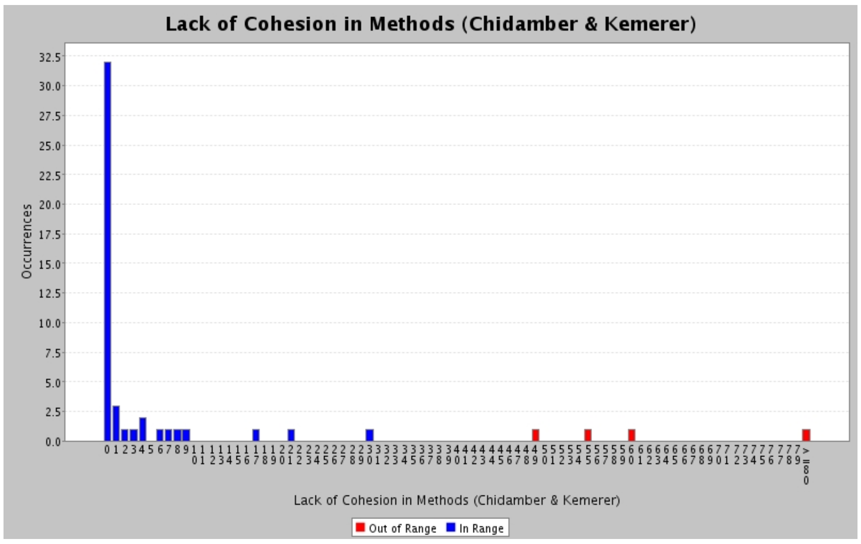

Figure 5.17: Lack Of Cohesion - Chidamber and Kemerer (Cycle One)

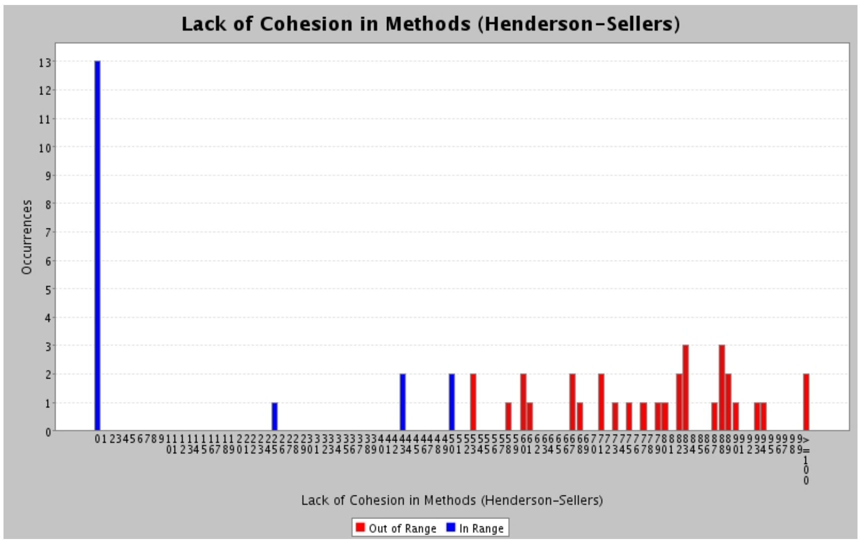

Figure 5.18: Lack Of Cohesion - Henderson-Sellers (Cycle One)

class method can not have just one field access, but many field access in common. This is why probably this metric got worst, because many methods has many set and get operations accessing only one field each.

As we can observe in Figure 5.20, the complexity of the system probably has no effect in system cohesion. Few changes can be observed from the values from baseline AppMan.

The complexity of the system probably has no effect in system cohesion also considering complexity by Type (Figure 5.21).

Another quality software information is presented in Figure 5.22: coupling. Efferent Couplings is almost insignificant in Appman source code and in this cycle had no big changes can be observed.

All these previous discussed graphs presents all classes, which may be difficult

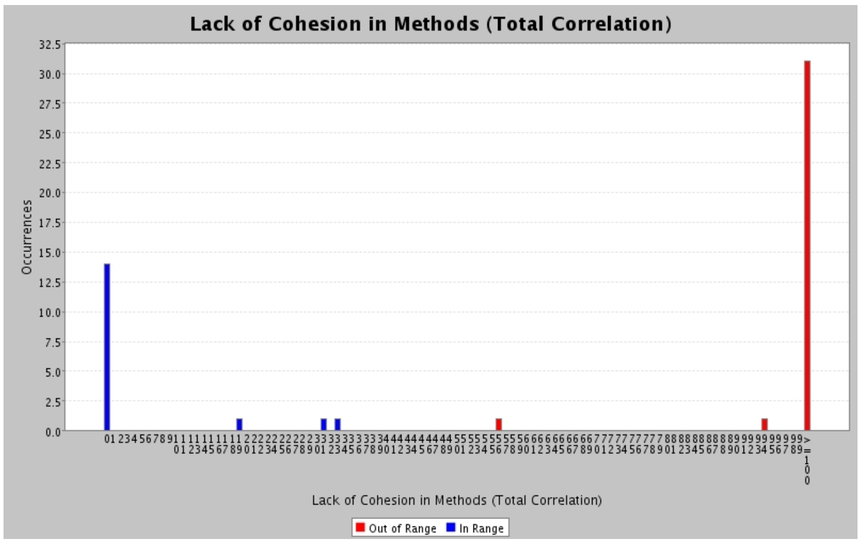

Figure 5.19: Lack Of Cohesion - Total Correlation (Cycle One)

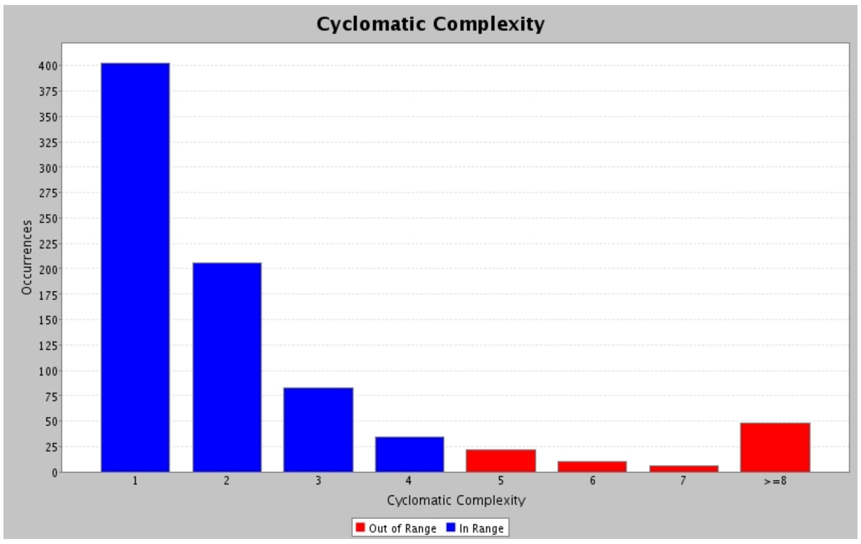

Figure 5.20: Cyclomatic Complexity by Method (Cycle One)

to really get a picture of the new AppMan source code. So, most of the differences in the software quality metrics can see in the Table 5.4. This table presents values only for the target classes for the refactoring process.

Performance Metrics

This section presents some performance metrics for the new refactoring Cycle One.

As we can see in the graph of Figure 5.23, the execution time is almost equal to the baseline version of Appman. The major difference appears when two Submission Managers are used. It may indicate that refactoring in the SubmissionManager and related classes may got a little impact in the grid performance when using more submission managers.

The graph in Figure 5.24 shows that when we used four Task Managers with

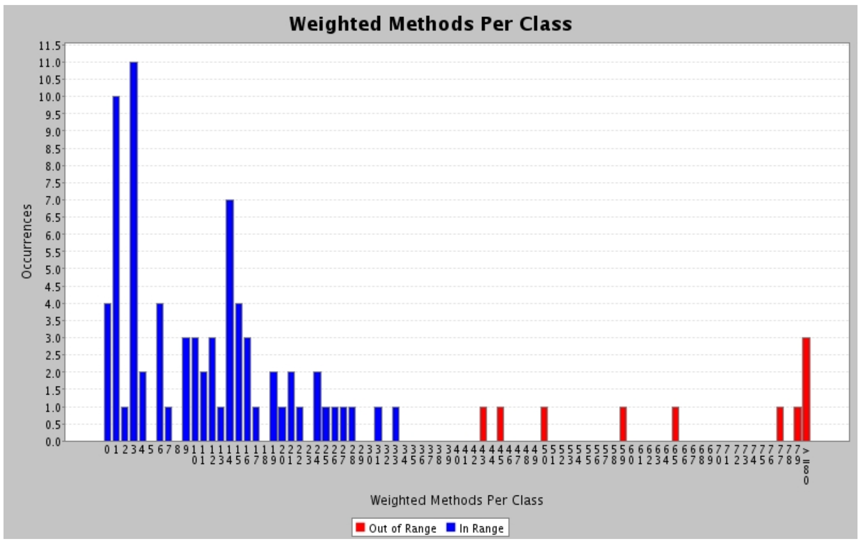

Figure 5.21: Cyclomatic Complexity by Type (Cycle One)

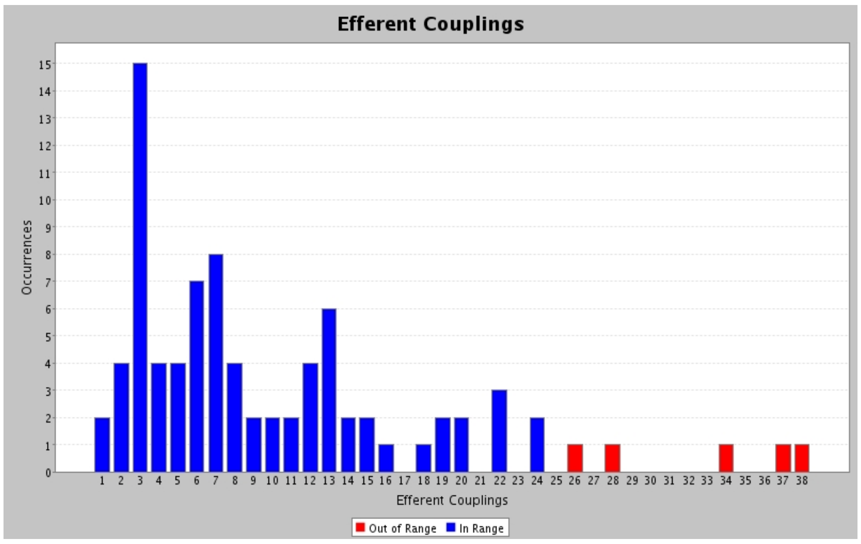

Figure 5.22: Efferent Couplings (Cycle One)

40 tasks we got a better speedup, but the others numbers of tasks was constant following the baseline version of Appman.

Figure 5.25 shows one of the most interesting graphs in this work. The Efficiency of AppMan is somewhat better when using two submission managers and two task managers. In the other cases stay in the same level demonstrating that although the refactoring in the main classes of Appman, no effect in the system efficiency and, even in another condition got a better performance.

The Figure 5.26 shows when using 2 submission managers that the cycle one version waste a little more memory than base version. Although when working with 4 task managers the memory consumption was lower than base version. This can make us conclude that the refactoring process make memory usage of submission managers get higher and for the task managers get lower.

Table 5.4: Appman - Top Lack of Cohesion Classes after Refactoring Cycle One, including TaskManager

LCOM-CK	LCOM-HS %	LCOM-TC %	Type	Package
165	90	303	Task	appman.task
60	88	167	SubmissionManager	appman
55	87	192	TaskTimer	appman.task
49	88	360	ApplicationManager	appman
30	79	444	OutputXML	appman
21	83	115	SimpleNode	appman.parser
17	82	186	TaskManager	appman.task

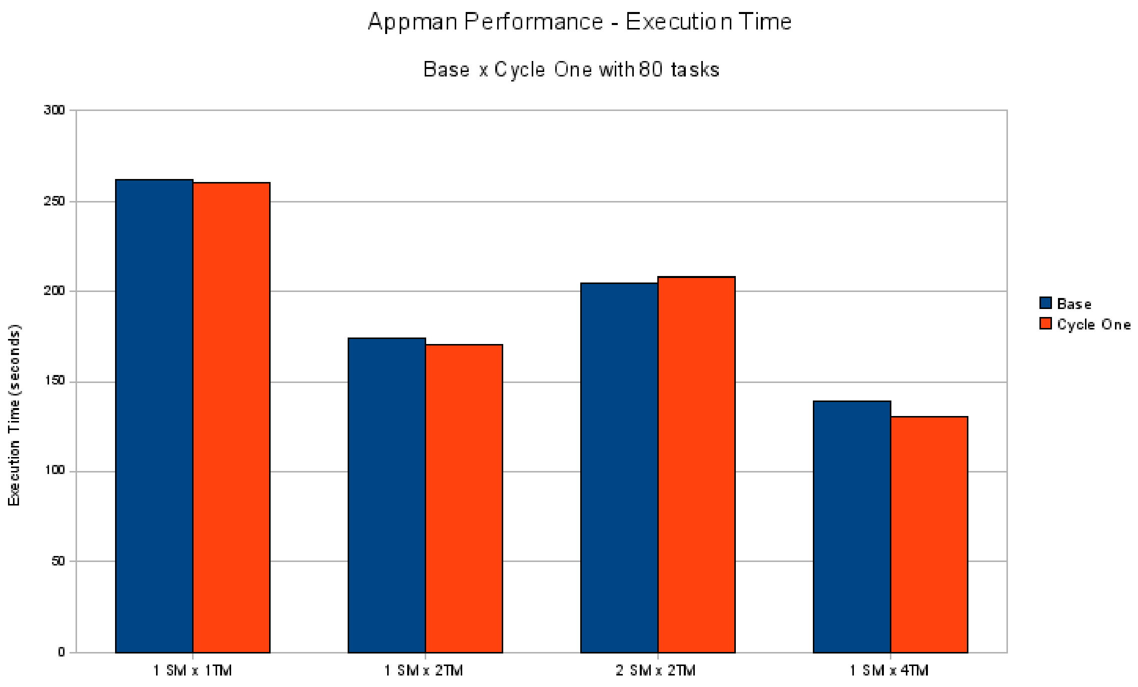

Figure 5.23: Appman Performance - Execution Time (Base x Cycle One)

5.5 Limitations

There are somethings which were not expected to happen during the execution of the refactoring process.

5.5.1 Automatic Refactoring x Manual Refactoring

AppMan is a software written with Java Language and a good Java IDE with refactoring capabilities was needed to apply this refactoring methods without worry with dependencies. For example, if you change the name of a method most of these tools resolve the dependencies of this method in the whole project changing their call to make that project compilable.

The author of this monograph believed that most of the refactoring methods was automated by this tools becoming an easy way to change the software fast with no worries because of its dependencies analyzes. The tools analyzed was Eclipse

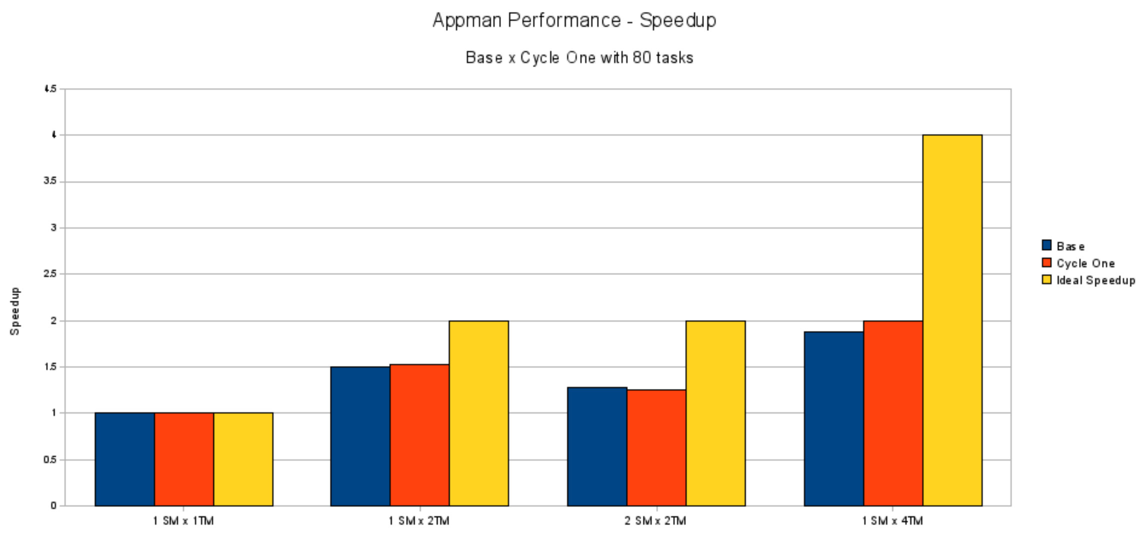

Figure 5.24: Appman Performance - Speedup (Base x Cycle One)

(Walton, 2008a), IntelliJ IDEA and NetBeans.

Has been discovered that there were some refactoring methods, and some very important, which were manual and making the refactoring process become very slow. The most important was the refactoring method *Extract Class* which stands for moving methods and attributes from a class to a new class. In the Section 4 is described the whole process of *Extract Class*.

Another manual refactoring method, but most expected to be manual is *Replace Type Code with State/Strategy* and *Replace Conditional with Polymorphism* because of its complexity.

Although the following refactoring methods are all automated by the tools studied: *Move method*, *Extract Method*, *Encapsulate Field* and *Inline Method*

One week before the conclusion of this work Eclipse (Walton, 2008a) released a new version called *Ganymede* which has *Extract Class* as a new automated refactoring method.

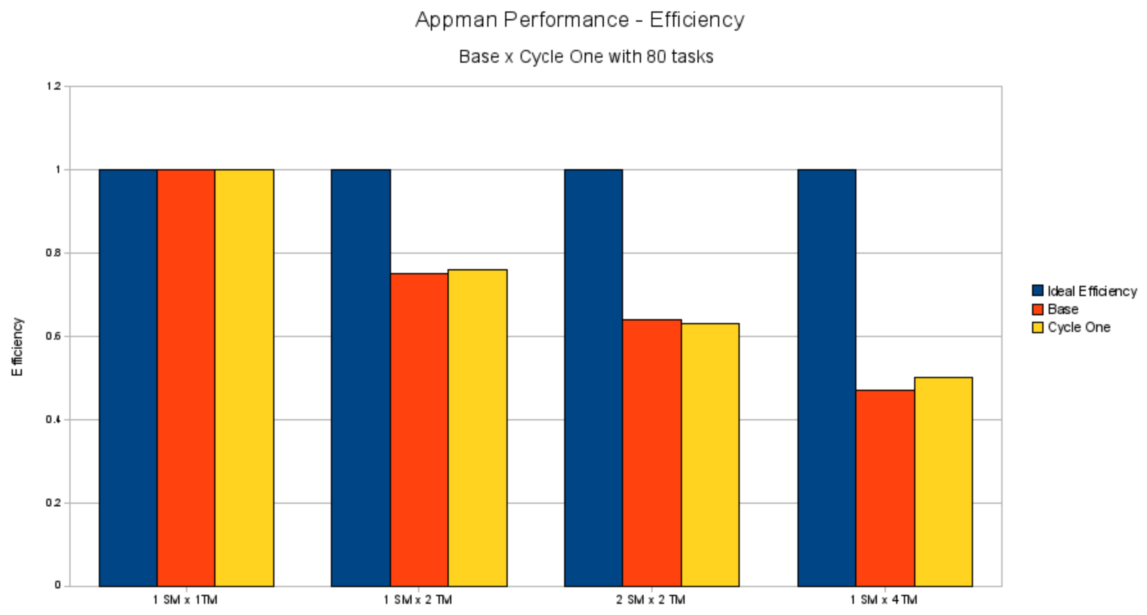

Figure 5.25: Appman Performance - Efficiency (Base x Cycle One)

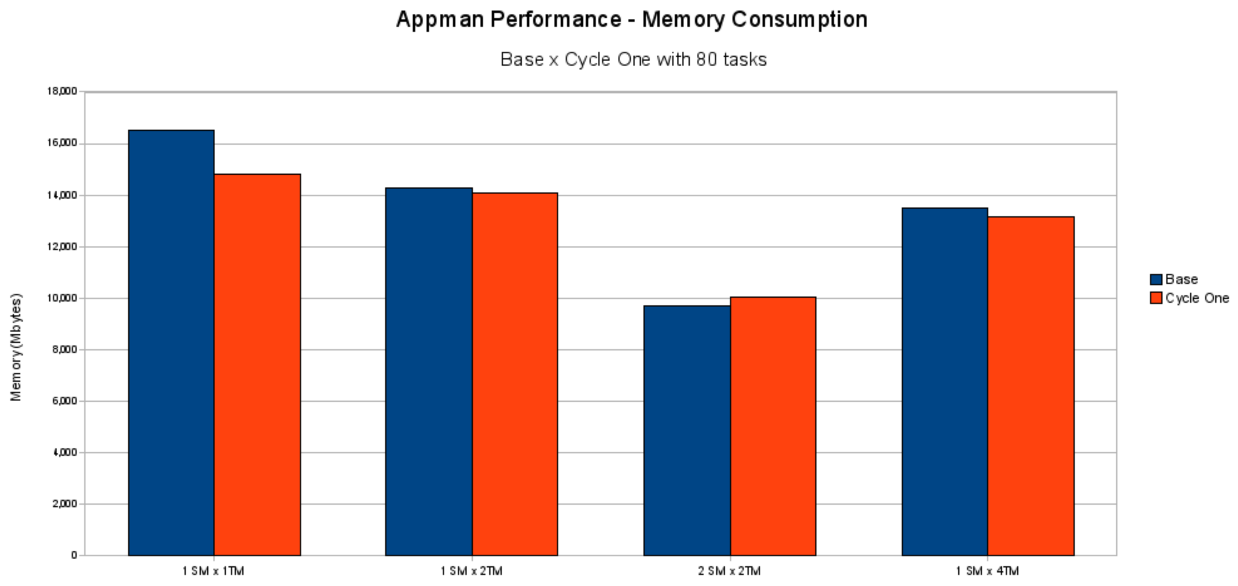

Figure 5.26: Memory Consumption (Base x Cycle One)

6 CONCLUSION

The main goal of this work was to improve software quality attributes using a real software as study case, comparing performance metrics during this refactoring process. Such approach is relatively original, and our preliminar results show it is a really useful and promising research issue.

AppMan was chosen for several reasons. First, this work is part of the GRAND project and thus improving AppMan quality is strongly desired. In spite of being a research project, it was developed by a distributed team, using tools and methods that are currently used both in industry and academy. Finally, the code had complexity enough as well design problems that justify a refactoring for improving maintainability.

The main contribution of this work is the fact that we could show that a good programming practice does not necessarily implies bad performance. Although only preliminary results are shown, our results are promising. For the presented refactoring cycle, we could both improve software quality and performance metrics. Table 6.1 presents the main results concerning quality software improvements. After such improvements, we still get speedup and efficiency as presented in previous chapter.

Table 6.1: Appman - Comparing cohesion classes using LCOM-CK metric

Original	After Refactoring	Type	Package
619	165	Task	appman.task
65	60	SubmissionManager	appman
82	49	ApplicationManager	appman
18	17	TaskManager	appman.task

As future work, our AppMan version can be integrated with current version implemented by Bernardo (2007). This ongoing new version uses a standard application programming interface called DRMAA to allow integrating AppMan with different Resource Management Systems such as PBS. Integrating both versions will provide a more mature and stable infrastructure for future works on GRAND Project.

New refactoring cycles can be implemented in the original version as well as in the integrated version. These new cycles can consider new quality attributes like coupling or complexity. Cohesion can be analyzed again considering other quality metrics and the classes cohesion in a group because this work analyze only level zero (method) and level one (class). This can improve much more this object-orientation key concept.

Finally, new software quality metrics should be used, trying to get more suitable metrics to OO context. As (Gupta, 1997) critique, the cohesion metrics were not suitable to this kind of application because of some limitations in its calculations which required some more effort to make this work happen. GQM (Goal, Quality, Metrics) can be used to evaluate the software quality metrics used in this work. GQM is an approach to software metrics which can be used to choose the right metrics and to drive the results for decision making.

As presented initial chapters, there are several metrics and methodologies that can be explored to get a better software quality. Checking how these new refactoring cycles affect performance metrics will provide even more conclusive data.

APPENDIX A LIST OF OBJECT-ORIENTED METRICS CURRENTLY INVESTIGATED IN THE LITERATURE

[Abreu-94]

- **SC1 - System Complexity (total length of inheritance chain)**
- CC2 - Class Complexity (progeny count) CC3 - Class Complexity (parent count)
- CR1 - Class Reuse (% of inherited methods that are overloaded)
- CR2 - Class Reuse (number of times class is reused "as is")
- CR3 - Class Reuse (number of times class is reused with Adaptation)
- SR1 - System Reuse (% of reused "as is" classes)
- SR2 - System Reuse (% of reused classes with adaptation)
- SR3 - System Reuse (library quality factor)

[Abreu-96]

- **The MOOD set (Metrics for Object-Oriented Design)**
- MHF - Method Hiding Factor
- AHF - Attribute Hiding Factor
- MIF - Method Inheritance Factor
- AIF - Attribute Inheritance Factor
- COF - Coupling Factor
- CLF - Class Clusters
- POF - Polymorphism Factor
- RF - Reuse Factor

[Banker-91]

- RFC - Raw Function Counts
- OC - Object Counts (count of classes)

- OP - Object Points
- RL - Reuse Leverage

[Benlarbi-97a]

- **Static and Dynamic Class Coupling in (A, D,F, IF, O)**
- Static-Import-Class-Attribute
- Static-Import-(Class,Method)-Method
- Dynamic-Import-Class-Attribute
- Dynamic-Import-(Class,Method)-Method
- Static-Export-Class-Attribute
- Static-Export-(Class,Method)-Method
- Dynamic-Export-Class-Attribute
- Static-Export-(Class,Method)-Method

[Benlarbi-97b]

- **Polymorphism Measures in (A, D, O)**
- OVO - Overloading in stand-alone classes
- SPA - Static polymorphism in ancestors
- SPD - Static polymorphism in descendants
- DPA - Dynamic polymorphism in ancestors
- DPD -Dynamic polymorphism in descendants
- NIP - Non Inheritance Polymorphism

[Briand-96]

- **Class-Attribute Interactions (CA)**
- IFCAIC - Inverse Friends CA Import Coupling
- ACAIC - Ancestors CA Import Coupling
- OCAIC - Others CA Import Coupling
- FCAEC - Friends CA Export Coupling
- DCAEC - Descendents CA Export Coupling
- OCAEC - Friend CA Export Coupling
- **Class-Method Interactions (CM)**
- IFCMIC - Inverse Friends CM Import Coupling
- ACMIC - Ancestors CM Import Coupling
- OCMIC - Others CM Import Coupling
- IFCMEC - Friends CM Export Coupling

- ACMEC - Descendents CM Export Coupling
- OCMEC - Others CM Export Coupling
- **Method-Method Interactions (MM)**
- IFMMIC - Inverse Friends MM Import Coupling
- AMMIC - Ancestors MM Import Coupling
- OMMIC - Others MM Import Coupling
- IFMMEC - Friends MM Export Coupling
- AMMEC - Descendents MM Export Coupling
- OMMEC - Others MM Export Coupling

[Chen -93]

- OXM - Operation Complexity Metric (within a class)
- OACM - Operation Argument Complexity Metric
- ACM - Attribute Complexity Metric
- OCM - Operation Coupling Metric
- CCM - Class Coupling Metric
- CM - Cohesion Metric
- CHM - Class Hierarchy of Method
- RM - Reuse Metric (of Classes)

[Chidamber- 94]

- WMC - Weighted Methods per Class
- DIT - Depth of inheritance Tree
- NOC - Number of Children
- CBO - Coupling between Object classes
- RFC - Response for a Class
- LCOM - Lack of Cohesion of Methods

[Coppick-92]

- SSM - Software Science Metrics (Halstead)
- MCC - McCabe's Cyclomatic Complexity Metric

[Lee-93]

- MC - Method Complexity
- CC - Class Complexity
- HC - Hierarchy Complexity of system
- PC - Program Complexity

[Li-93]

- DAC - Data Abstraction Coupling (Number of abstract data types)
- NOM - Number of local Methods
- MPC - Message passing Coupling (Number of send statements in a class)
- Size 1 - number of semi-colons in a class
- Size 2 - number of attributes + number of local methods

[Moreau-90]

- SSM - Software Science Metrics (Halstead)
- MCC - McCabe Cyclomatic Complexity metric
- IL - Inheritance Lattice (stated, but no measure indicated)
- GSDM - Graph of Source and Destination of Message (no measure given)

[Sharble-93]

- WAC - Weighted Attributes per Class
- NOT - Number Of Tramps (count of extraneous parameters)
- VOD - Violations of the law of Demeter [Lieberherr-89]

[Tegarden-92]

- SSM - Software Science Metrics (Halstead)
- MCC - McCabe's Cyclomatic Complexity metric
- LOC - Lines of Code

[Williams-93]

- COU - Count of Uses
- CBC - Count of Base Classes
- CSC - Count of Standalone Classes
- CCR - Count of Contained Relationships

REFERENCES

Benlarbi, S. and Goel, N. (1998). Measuring object-oriented software systems. Technical report, Cistel Technology.

Bernardo, T. R. (2007). Integração do sistema appman de gerenciamento de aplicações para ambiente de grade com diferentes sistemas de gerenciamento de recursos. Trabalho de conclusão (em andamento), Ciência da Computação/UNILASALLE.

Bois, B. D., Demeyer, S., and Verelst, J. (2004). Refactoring "improving coupling and cohesion of existing code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 144–151, Washington, DC, USA. IEEE Computer Society.

Booth, S. (2006). OO performance optimisations of HPC applications. Technical report, EPCC The University of Edinburgh.

Cirne, W. and Neto, E. S. (2005). Grids computacionais: da computação de alto desempenho a serviços sob demanda. Technical report, Universidade Federal de Campina Grande.

Fernandes, C. A. C. (2003). Estudo de algumas ferramentas de coleta e visualização de dados de desempenho de aplicações paralelas no ambiente mpi. Master's thesis, Universidade Federal do Rio Grande do Norte.

Forge, S. (2008). Eclipse metrics. Home page, Source Forge, http://metrics.sourceforge.net/. accessed June, 2008.

Foster, I. (1995). *Designing and Building Parallel Programs.* Addison-Wesley, http://www.mcs.anl.gov/dbpp/. acessado em novembro de 2007.

Foster, I. and Kesselman, C. (1998). Computational grids. Technical report, Argonne National Laboratory and University of Southern California.

Fowler, M. (2004). *Refatoração: Aperfeiçoando o Projeto de Código Existente.* Bookman.

Grimshaw, A. S. (1996). Object-oriented parallel processing with Mentat. *Information Sciences*, 93(1):9–34.

Gupta, B. S. (1997). A critique of cohesion measures in the object-oriented paradigm. Master's thesis, Michigan Technological University.

Mangan, P. K. V. (2006). *GRAND: UM MODELO DE GERENCIAMENTO HIERÁRQUICO DE APLICAÇÕES EM AMBIENTE DE COMPUTAÇÃO EM GRADE*. PhD thesis, Universidade Federal do Rio de Janeiro.

Microsystems, S. (2008). Sun java 2 plataform se 5.0 api. Home page, Sun Microsystems, http://java.sun.com/j2se/1.5.0/docs/api/. acessed in Jun 2008.

Rosenberg, L. and Hyatt, L. (1997). Software quality metrics for object-oriented environments. In *Crosstalk Journal*, page 10(4).

Stroggylos, K. and Spinellis, D. (2007). Refactoring: Does it improve software quality? In Boehm, B., Chulani, S., Verner, J., and Wong, B., editors, *5th International Workshop on Software Quality*. ACM Press.

Vargas, P. K., Dutra, I. d. C., and Geyer, C. F. (2004). Application partitioning and hierarchical application management in grid environments. Relatório Técnico ES-661/04, COPPE/Sistemas - UFRJ.

Vargas, P. K., Santos, L. A. S., Dutra, I. C., and Geyer, C. F. R. (2005). An implementation of the grand hierarchical application management model using the isam/exehda system. In *III Workshop on Computational Grids and Applications*, Petrópolis, RJ, Brazil.

Walton, L. (2007). Eclipse metrics plugin. Home page, State Of Flow, http://eclipse-metrics.sourceforge.net/. acessado em novembro de 2007.

Walton, L. (2008a). Eclipse europe. Home page, Eclipse Foundation, http://www.eclipse.org. acessed in May 2008.

Walton, L. (2008b). Tele-immersion. Home page, Advanced Network and Services, http://www.advanced.org/teleimmersion.html. acessado em junho de 2008.

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. Technical report, Brown University.

Xenos, M., Stavrinoudis, D., Zikouli, K., and Christodoulakis, D. (2000). Object-oriented metrics - a survey.

Yamin, A. C., Augustin, I., da Silva, L. C., Real, R. A., Filho, A. E. S., and Geyer, C. F. R. (2005). Exehda: adaptive middleware for building a pervasive grid environment. In Czap, H., Unland, R., Branki, C., and Tianfield, H., editors, *SOAS*, volume 135 of *Frontiers in Artificial Intelligence and Applications*, pages 203–219. IOS Press.