

# Surgical Information to Detect Design Problems with MOOSE

Muhammad Usman BHATTI<sup>1</sup>, Stéphane DUCASSE<sup>2</sup>

<sup>1</sup>CRI, Université de Paris 1 Sorbonne, France

<sup>2</sup>Language and Software Evolution Group, Université de Savoie, France

*muhammad.bhatti@malix.univ-paris1.fr; stephane.ducasse@univ-savoie.fr*

## Abstract

*The quality attributes, such as understandability and modularity manifest their importance in the later part of the software life where a lot of resources are required to maintain or reuse software whose quality has been marred by the urgencies of time to market. In this position paper, we present and analyze an existing system meant to be reused on various product lines. We intend to use the MOOSE framework to precisely identify the needs of a reengineer in terms of code-smells, visualization and metrics. In this position paper, we discuss some of the limitations of the existing software system and inadequacy of existing toolkits to automate the task of detection of these limitations. We intend to discuss the appropriateness of MOOSE as a remedy to these deficiencies.*

## 1. Introduction

Software design is an iterative process and it is very difficult to achieve an ideal design in terms of quality in the first iteration. While managers and developers are working to satisfy the marketing needs in a minimum time, software quality is their last preoccupation. Therefore, the use of a certain technology or a paradigm doesn't mean ideal software quality. Reverse Engineering and Reengineering are used to analyze the code of a software having very little or no existing documentation [1]. One of the aims of reengineering activity is to generate high-level models and diagrams from the only artefact that represents the true state of the affairs vis--vis business of an enterprise, along with the technical architecture to support this business, so that these can be visualized and manipulated by reengineers.

The first author is working in a company which produces a range of blood plasma analysis automatons. The overall cost of an instrument includes a huge chunk for software that drives the instruments and managers are always looking to reduce the cost of development, and once devel-

oped, the cost of software maintenance. Development costs can be reduced by reusing the existing artefacts which are developed earlier to be reused in the existing development process.

But the existing software contains a lot of deficiencies in terms of the quality of its components partly due to the lack of knowledge of object-oriented design methodologies.

We would like to use a tool-based, reverse-engineering approach in order to improve the quality of existing software to enable the reuse of individual components. For that purpose, MOOSE [7] seems to be an ideal candidate with its visualization and metric-based plug-ins. One of the limitations that has been encountered during the work is the absence of the research tools supporting Microsoft .NET Framework and associated languages. For this purpose, we are in the process of development of a plug-in to integrate MOOSE with .NET environment.

## 2 Case Study: Blood Plasma Analysis Machines

We are working in a company that builds blood plasma analysis machines. The machine is composed of two main parts: the hardware part concerning mechanics and electronics like the arms, the drawers, and the software part managing the hardware with an aim of analyzing the blood plasma. The user of the machine (operator, biologist, etc), after authentication, loads one or more tubes of plasma, as well as products, in the drawers of the machine, associates a test of analysis each tube, and launches the analysis. The automaton performs the analysis for blood-related diseases and the results of the analyses are displayed on the display device.

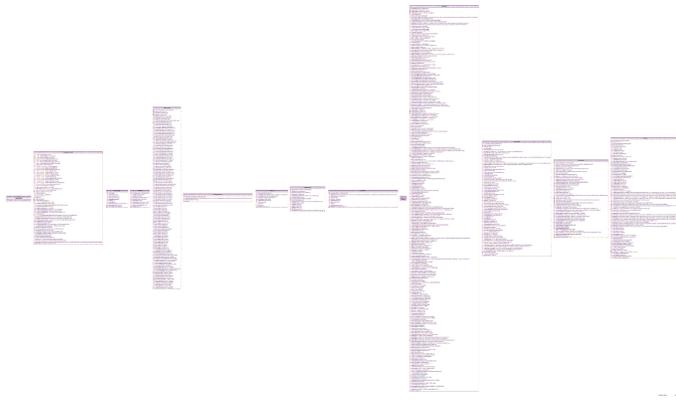
For the sake of precision and clarity, we shall only be talking about the layer that manages the business objects and operates with the database layer to manage the data associated with these objects.

Although the software cannot be considered to be a legacy one, it presents rudimentary examples of object-

oriented code lacking object-oriented design [4]. We discuss this issue in detail in the next section.

### 3 Business Entity Layer

The business entity layer of the software is supposed to support functionality such as patient data, tubes, blood analyses, results, reagents used, etc. An extract for the class diagram of this layer is shown in the figure below.



**Figure 1. Class Diagram Entity Layer**

Although the above diagram is not very clear in terms of its contents, nevertheless it communicates some facts about the business entity layer.

- There is a lack of hierarchical structure, via inheritance one of the tenet of code reuse and a fundamental element of object-oriented design.
- Presence of huge classes encapsulating functionality pertaining to multifaceted objects and entities.

These problems which are visible from a very high-level abstract extraction from the code in question, show that rudimentary restructuring is needed in terms of design and quality of the software.

Unfortunately, most of the tools available for C# language provide this level of view and reengineers are supposed to manually follow the track from these abstract maps. These visualizations are helpful in understanding the overall system architecture but not adapted to fine-grained problems associated with the software architecture. Argument stating that software metrics only represent numbers and provide information that require further interpretation, in this case, seems to carry weight [2]. They do not go farther than the information depicted in the figure above since they do not provide comprehensive information to ease the task of software reengineering. For example, Table 1 is a set of quality metrics for the mammoth class in Figure 1 above, called ServicePatient.

**Table 1. Classification of Aspectized services**

Metric	Value
Lines of Code	4000
# of Methods	262
Depth of inheritance	1
Lack of Cohesion Of Methods (LCOM)	0.8501908
LCOM Henderson-Sellers (LCOMHS)	0.8534483
Cyclomatic Complexity	1368

For a reengineer, the gap between the visualization needed for her refactoring activity and information provided by the metrics remains largely obscure.

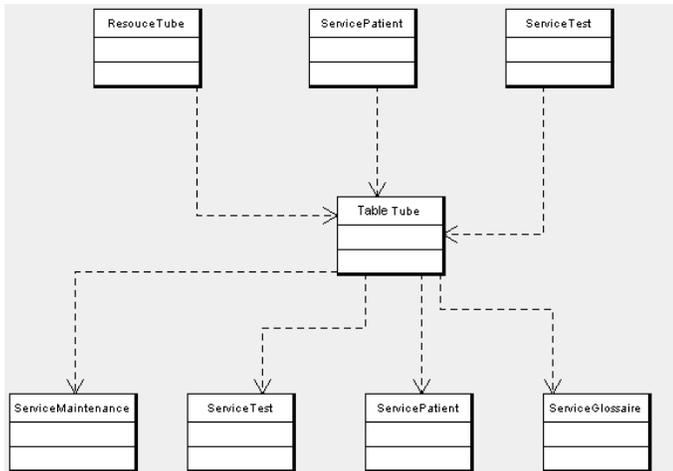
An example may illustrate the things better: While manually analyzing the software in question we found that there is no object related to the patient's tube and the logic to add, search, manipulate and delete the tube data associated to tubes is scattered in other classes of the system. These classes directly operate over the table tube found in the database, and we used manual techniques to identify places where the information was changed in the database as shown in Figure 2. During code inspection, we found that any change to the tube related logic requires changes in several classes spread over two software layers. These are clear code smells for for encapsulation related problems [5]

This bad code smell related to the absence of encapsulation could be identified using visualization techniques [?], it is not straightforward since most visualization techniques do not support the detection of this design anomaly [8, 3]. In addition, it requires a manual effort to understand the effects of absence of this object on the future evolutions of the software.

The extract class refactoring suggested by [5] provides a possible solution consisting of extracting all the methods and attributes related to tubes to be extracted from the main class. This extracted code can then be placed in a class that can encapsulate the logic associated to the tube object and the code can be modified accordingly. The visualization has a great part to play to support unwanted effects of the refactoring exercise. This can be done by identifying all the attributes and methods that are effected by the movement of the code from one class to another. As defined above, the visualization techniques support more legacy code problems such as code duplication and large classes. But there is no visualization technique that supports the actual refactoring process.

A reengineer at this point would find the following information useful:

- Classes containing the logic pertaining to a particular object, in our case tube.



**Figure 2. Classes directly manipulating the Patient’s Tube Information**

- Dependencies of these classes at a coarse-grained level and the methods at a fine-grained level to calculate the impact of restructuring activities on other modules.
- A set of metrics indicating the impact on software quality with the progress of software restructuring.

For this purpose, we intend to exploit the MOOSE system to extract what we call surgical information to guide our reengineering activities. The purpose of this exercise is to find a set of visualizations and metrics that identify problems more precisely than the existing techniques. For this purpose, more details and semantically rich metrics such as FAN in analysis and Formal Concept Analysis should be investigated. This would allow the reengineer to gain an insight into the code which provides much more than mere structural information to detect design defects.

#### 4 Tool Used - Related Work

The number of research studies and tools for C# based systems is inferior to that of Java-based systems due partially to the license and availability issues of the two products. Nevertheless, some research prototypes are still available to study existing C# systems. For this activity, we used Altova’s Class Diagram extractor to extract the class diagrams for the module in question. NDepend provides a useful tools with limited license for academic purposes to calculate various metrics of the software systems developed in C#. These include an extensive set of information for the dependencies of type-based information. But since tube type doesn’t already exist in the system, it already requires a lexical analysis of the system to search for possible candi-

dates methods and properties named tubes. This is the limitation of this tool that the reengineer needs to resort to other tools for lexical analysis of various statements to study the dependency of a software concern deeply embedded in the system. Devmetrics provides a set of high-level metrics for the system developed in C# language but doesn’t provide class diagram extraction functionality. Doxygen is useful tool to generate Javadoc-type documentation for the existing systems. It can be easily inferred that existing applications for the reengineering of C# systems lack all-in-one information needed to guide the reengineering activity.

#### 5 Conclusion

Software quality is understood to be a topic of academic interest in the industrial world. The software generally have inferior quality due to time to market urgencies. Hence, new software is needed to be developed for each new product without reusing the existing components. We are currently working on a system of this type and trying to ameliorate its quality. We are facing a lack of tools which may guide reengineer’s activities. Normally a mix of tools are used to extract this information. In addition, C#-based systems are not studied in research for the problem of restructuring, hence the lack of tools and information is accentuated for this family of systems. We intend to use MOOSE framework to infer what we call surgical information needed to guide reengineering tasks with a set of visualizations and metrics.

#### References

- [1] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
- [2] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS ’99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 18, Washington, DC, USA, 1999. IEEE Computer Society.
- [3] Y. Crespo, C. López, R. Marticorena, and E. Manso. Language independent metrics support towards refactoring inference, jul 2005. <http://pisuerga.inf.ubu.es/lopez/refactoring/>.
- [4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

- [6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [7] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of moose: an agile reengineering environment. volume 30, pages 1–10, New York, NY, USA, 2005. ACM Press.
- [8] C. Parnin and C. Gôrg. Lightweight visualizations for inspecting code smells. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 171–172, New York, NY, USA, 2006. ACM Press.