Heuristics for the Identification of Class Integration Order in Object-Oriented Software Testing

Gladys Machado Pereira Santos Lima
DAdM / Brazilian Navy
Ilha das Cobras – Ed. Almirante Gastão Mota – 20091-000 – Rio de Janeiro – RJ – Brazil
gladys@dadm.mar.mil.br
Fax: +55 21 32342052

Arilo Cláudio Dias Neto
acdn@cos.ufrj.br

Guilherme Horta Travassos
ght@cos.ufrj.br

System Engineering and Computer Science Program – COPPE/UFRJ
P.O. Box 68.511 – CEP 21.941-972
Rio de Janeiro – RJ – Brazil

Abstract

Dependency cycles among components (classes) represent a practical challenge when identifying the class integration order in object-oriented software testing. Classical approaches, such as "top-down" or "bottom-up" integration (or their combination) become less useful due to their acyclic characteristics. Some OO integration testing strategies explore the principle of breaking these dependency cycles, but broken dependencies imply that the target class will have to be stubbed when integrating and testing the source class, which increases the testing effort. This paper describes an integration testing strategy that guides software engineers to identify the class integration order with the minimum number of implemented stubs. This strategy can be directly applied to high level OO design (UML class) diagrams allowing the reduction of extra construction efforts related to additional classes needed in other strategies. The results from experimental studies have indicated its feasibility and effectiveness. The use of such integration testing strategy can be supported by a case tool, also described in this paper.

Keywords: Object-Oriented Software Testing, Integration Testing, CASE, Experimental Software Engineering.

1. Introduction

Software testing is carried out to guarantee software quality, usually evaluating whether it works correctly as specified by requirements. In general, to be considered successful software testing must make the software fail through its execution. By revealing software failures, software engineers have the chance to identify and fix defects prior to software deployment [1].

One of the approaches for preventing or detecting failures produced throughout the software development process is concerned with the definition of different testing levels regarding the diverse software development phases [2]. Integration testing, focus of this work, represents a set of activities aimed at revealing failures from defects in the software structure as defined in the design stage [3], usually related to the modules' interfaces that form the software architecture.

Considering the development of object-oriented (OO) software, the understanding of OO software execution is not easy, rendering traditional testing strategies (acyclic) unable to deal with some of the architecture features of this software development paradigm, such as the dependencies among components (represented by classes), making difficult the accomplishment of the task of identifying the integration order and component testing [4]. So it becomes necessary to review the strategies concerned with integration testing in the context of OO software.

Some strategies aimed at integration testing (Strategies for Integration Testing - SITs) have been proposed to support the identification of class integration order in OO software projects [5, 6, 7, and 8]. These strategies aim at dealing with cyclical dependencies to reduce testing effort if measured by the number of stubs. Stubs represent modules built to simulate the behavior of unavailable (not integrated) modules when accomplishing integration testing. They represent additional development effort that needs to be reduced. Apart from the stubs, these strategies demand additional artifacts and infrastructure to be applied to the software project, increasing testing effort due to the fact that most of them explore low level abstractions (source code) to derive test cases.

Motivated by the idea that it could be possible to support software engineers to identify class integration order for OO software with as little effort as possible, we have proposed and experimentally evaluated a set of heuristics for object-oriented software integration testing. This heuristics set, when applied to a high abstraction level OO software artifact (design represented by UML class diagram) can reduce additional testing effort regarding extras artifacts and can contribute to increase productivity, since class integration order can also represent class construction order, a consequence of the OO paradigm that explores the same set of semantic constructors throughout the software development process.

This paper describes the proposed integration testing strategy, consisting of a heuristics set and an application process. This integration testing strategy has been evaluated in experimental studies. These studies have indicated that at least 80% of the software engineers and software development managers that took part in the studies were able to identify an order to integrate classes with minimum effort applying such approach (heuristics + application process). Besides, it has been possible to observe the feasibility and effectiveness of the proposed integration testing strategy.

The next sections are organized as follows: section 2 describes the heuristics and the corresponding application process. Next, in section 3, the experimental studies to characterize the heuristics set are discussed. To support the application of such heuristics FAROL, a case tool, is introduced. Finally, some conclusions and future works are discussed in section 5.

2. The Heuristics and the Application Process

The UML class diagram represents a static view of OO projects. It describes a set of classes and their relationships (association, generalization, composition, inheritance, and dependency) [9]. This knowledge can usually be applied by software engineers when dealing with issues in the process of identifying which classes should be developed or tested first. However, it is possible to observe that software engineers do not have access to clear guidance on how to explore such knowledge to make decisions about the order of integration. In most of the cases these decisions are taken on an *ad hoc* basis.

Based on this scenario, we have observed that the characteristics of the OO paradigm (inheritance; class method signature; aggregation; navigability; association class; dependency; and cardinality) represented in UML class diagrams can support decision making regarding the class integration testing order. These characteristics can be used to define precedence criteria among classes, as shown in Table 1. To apply such heuristics, we have defined two indicators to support the establishment of integration order among classes: the *Influence Factor* (FI) and the *Delayed Integration Factor* (FIT).

- The *Influence Factor* (FI) quantifies the relation of precedence among classes. It represents the number of classes that must be tested after the current class.
- The *Delayed Integration Factor* (FIT) captures the idea of integration time. It captures the "moment" when the class should be integrated (tested). It is calculated by the sum of the FI obtained for each class that precedes the current class. The higher the FIT, the later the class should be tested. Thus**,** we will look first at the classes with null FIT.

Table 1 - Heuristic: Precedence Criteria.

Criteria	Precedence
Inheritance	1. *Concrete superclass* – the *superclass* should be tested before the *subclass*. 2. *Abstract superclass* – test first the *subclass* with less dependency (measured by the number of all subclasses relationships).
Class Method Signature	3. *Server class* should be tested before the *customer class* (that subscribed the method).
Aggregation/Composition	4. The *"part"* should be tested before the *"whole"*.
Navigability	5. The class that became an attribute should be tested first. 6. If the navigability has two directions, one class has precedence over the other one, and vice-versa.
Association Classes	7. The classes that originated the association should be tested before the association one.
Dependency	8. The *supplier class* should be tested before the *client class*.
Cardinality	9. The class with optional cardinality should be tested before the other class.

The complete heuristics set is shown in [10]. Together with the FI and FIT indicators, this set represents an improvement on the proposal discussed in [11]. Experimental studies allowed the identification of some special scenarios, not previously defined in former research work: class with null Influence Factor, iterations without null delayed integration factor, and deadlock treatment (classes with same FIT value). The proposed solution for these special scenarios guided the heuristics application process definition. The heuristics application process consists of 3 sub-processes, as shown in Figure 1, using the software processes notation shown in [12].

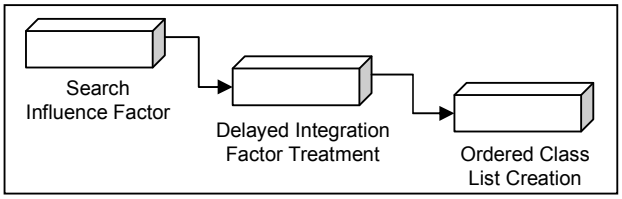

Figure 1 – Heuristics Application Process.

*Search Influence Factor Proces*s (Figure 2): the main activities of this process are concerned with the calculation of FI for all classes present in the design model, identifying null FI classes (which should usually be the last ones to be tested/integrated) and inserting them into a list of dependent classes (LCD), and creating of a list of classes (LCNO – not ordered yet) to be used in the next process.

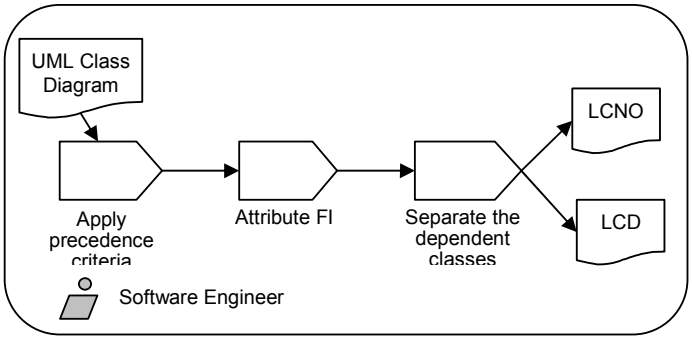

Figure 2 – Search Influence Factor Process.

Delayed Integration Factor Treatment Process (Figure 3): the activities in this process order the classes included in LCNO, producing a list of classes (LCOTI) organized according to

their integration testing order. While there are classes to be ordered, the classes with null or lower FIT must be moved from LCNO to LCOTI. Before starting a new iteration, the FIT must be re-calculated to refresh the influences of the ordered classes. The process is made by the following activities:

(1) identify precedence – identify classes that have precedence over the current class;
(2) calculate FIT – add FI value of precedence classes to current class;
(3) search classes with null FIT – select class(es) for integration order;
(4) or prioritize classes with same lower FIT – if there are no null FIT classes, select and order class(es) using the deadlock treatment criteria;
(5) update lists LCOTI and LCNO – including the selected class(es) in LCOTI (Integration Testing Class Order List) and excluding the same class(es) from LCNO; and
(6) Reduce influences – a new iteration to calculate FIT values for classes included in the LCNO must be done. In this next iteration, the FIT values will be the last calculated FIT less the FI values of the last selected classes if they have precedence over the current class.

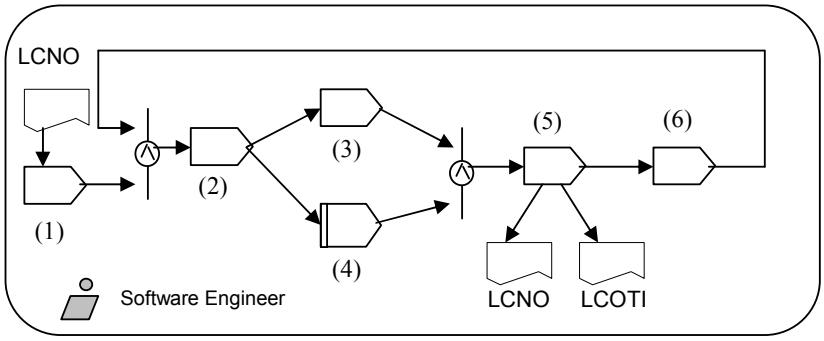

Figure 3 – Delayed Integration Factor Treatment Process.

The "Prioritize classes with same minor FIT" activity (Figure 3 – number 4) represents a composite activity. It represents the deadlock treatment. In this case, priority is set by the sub-activities shown in Figure 4.

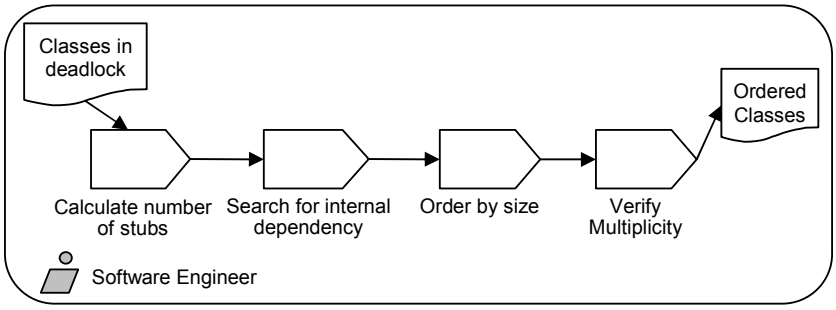

Figure 4 – "Prioritize classes with same minor FIT" Activity.

Ordered Class List Creation Process (Figure 5): Its goal is to create the final ordered classes list for integration testing (LCOTI). The final class order is obtained by combining two lists: LCD and LCOTI, generated in previous process activities.

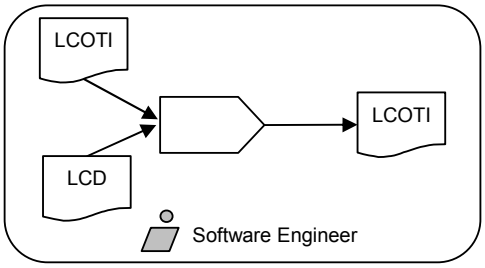

Figure 5 – Ordered Class List Creation Process.

To evaluate the process of heuristics application in different project situations and to try to expand the results described preliminary in [10], we identified the need to elaborate experimental studies on heuristics characterization. These studies are discussed in the next section.

3. Experimental Studies

According to Pfleeger [13], it is necessary to plan strategies to support working with imperfect knowledge and the uncertainties of the models and measures used by software engineers. One of the challenges would be to understand the chances where, under certain conditions, a particular tool or technique could contribute for the improvement of software development. In this sense, experimentation can be considered as determinant for the identification of the feasibility and effectiveness of the techniques applied to software engineering [14].

3.1 Definition of the Objectives

In this research, experimentation was used as a systematic, disciplined and controlled way to evaluate the feasibility and effectiveness of the proposed integration testing strategy. It aims at reducing the uncertainties during the several activities and supplying some indications for better understanding and evolution of the work.

The experimental studies were elaborated according to the methodology for experimentation processes proposed by Wohlin et al. [15], usually applied by the Experimental Software Engineering Group at COPPE/UFRJ, and consisting of the following stages: definition; planning; operation; analysis and interpretation; and presentation and packing, as described in Travassos et al. [16]. The four experimental studies are presented as follows:

- *Heuristics Usability*. It focuses on heuristics usability characterization when used to identify class integration order (extracted from UML class diagrams), and compares the resulting test effort with the effort generated by the strategy proposed by Briand et al. [17].
- *Pre-existing Procedures*. Focuses on learning other (ad hoc) procedures used by participants of the study to identify integration order. Used to compare the testing effort of these procedures with the testing effort employed on the same model when using the proposed heuristics. Used as a pre-testing for the study of heuristics effectiveness.
- *Heuristics self-training*. Aimed at measuring the subjects' heuristics learning, acquiring class integration order as per subject and comparing their results (testing effort) with the one held by the researcher. It has been used as a treatment of the heuristics effectiveness study.
- *Heuristics Effectiveness*. Post-testing for the evaluation of subjects' results when applying the heuristics. The results obtained are compared with pre-testing results (Pre-existing Procedures).

3.2 Preparation of the Studies

The preparation for the execution of the first experimental study (heuristics usability) consists of model selection. The choice was based on two criteria: to be a known model and for which there has been previous publication concerning the application of the strategy of Briand et al. [17] and its results. The selected model was the ATM (Automated Teller Machine) [17], as shown in Figure 6.

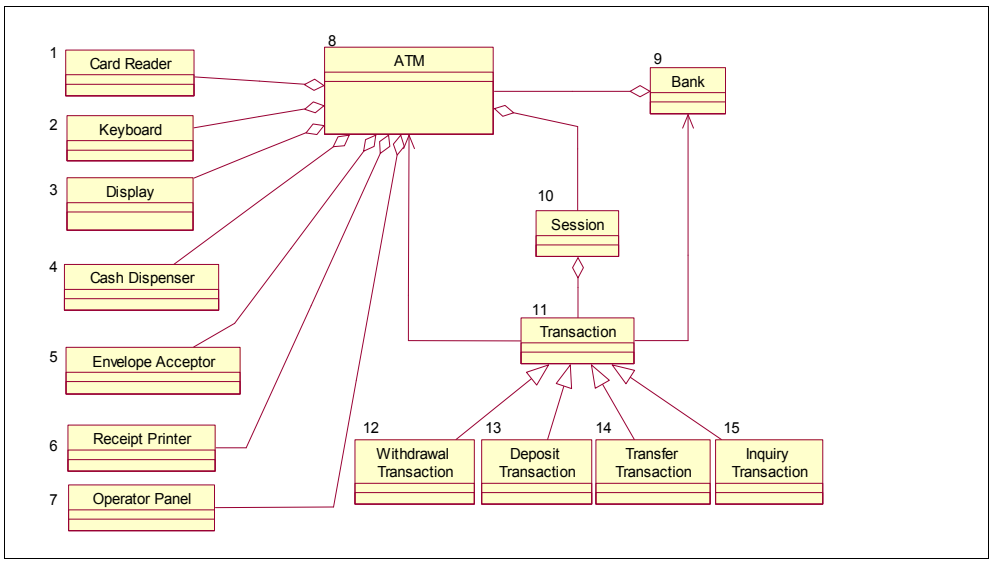

Figure 6 – ATM Model: UML Class Diagram [17]

For the other experimental studies, the preparation consists of the subjects' selection and characterization; and the preparation of the self-training material. This material is provided as a presentation (PowerPoint show) containing a description on how to apply the heuristics application process. It allows individual strategy learning to each participant with no interaction with other participants or the researcher.

Subject selection was limited to two groups, consisting of: COPPE/UFRJ software engineering students (Group 1 – Academic Environment) and software engineers or project managers from the CASNAV/Brazilian Navy (Group 2 – Industrial Environment). The subjects from the industrial and academic environment have an average professional experience in software development of 10.7 and 3.18 years, respectively. Object-oriented knowledge (concepts and UML) was determined through a 1 to 5 scale, where degree 5 shows the subject has used the concepts in more than one industrial project, as shown in Figure 7.

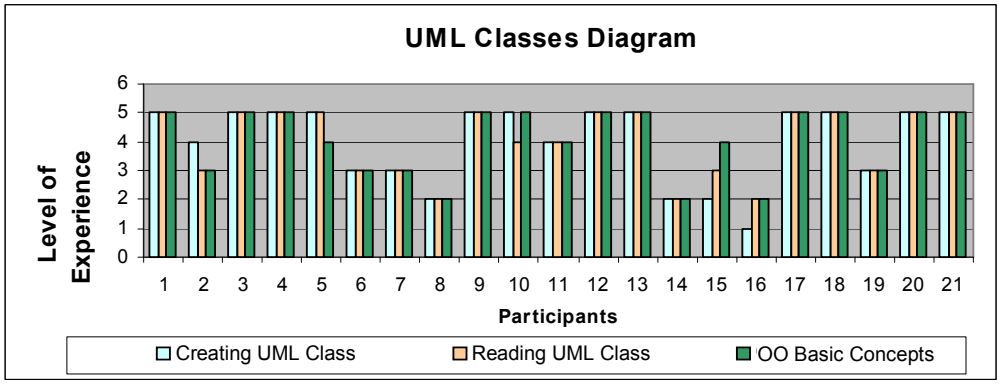

Figure 7 – OO Background of Study Subjects

3.3 Results and Lessons Learned

The strategy proposed by Briand et al. [8,17] obtained better results when compared to other strategies: Kung et al. [5]; Tai and Daniels [6]; and Le Traon et al. [7]. Therefore, the lower testing effort obtained with the heuristics application process compared to the results of Briand et al. published in [17], as shown in Table 2, was important to characterize the proposed strategy feasibility and for the continuity of the heuristics study.

Table 2 – Heuristics Feasibility Study Results.

Strategies	Briand – Labiche - Wang	Lima and Travassos Heuristics
Integration Order	1, 2, 3, 4, 5, 6, 7, 11, 10, 12, 13, 14, 15, 8, 9	1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10, 12, 13, 14, 15
Stubs	Class number 11 is tested using *stubs* for classes number 8 and 9	Class number 8 is tested using a *stub* for class number 10
Testing Effort	2 *stubs*	1 *stub*

The Pre-existing Procedures (pre-testing) and Heuristics Feasibility experimental studies contributed to the analysis of the proposed integration testing strategy effectiveness. The Figure 8 presents the testing effort obtained when subjects applied their ad-hoc (pre-testing) and following the heuristics application process (post-testing) to identify the classes' integration order. We can observe a significantly increasing in the number of subjects that find the minimum testing effort (17 stubs) for the considered model, as shown in Figure 9.

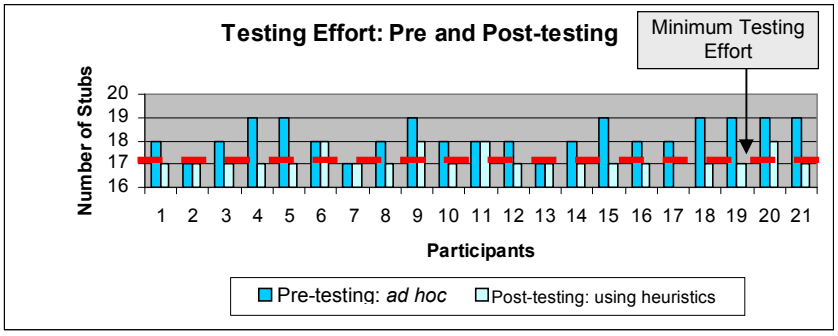

Figure 8 – Results of *Pre-existing Procedures* and *Heuristics Effectiveness* Studies

Some indications of effectiveness from previous studies can be reinforced by heuristics self-training and experimental study analysis. Subjects from the academia (95%) and industry (85%) were able to obtain the minimum testing effort expected for the model.

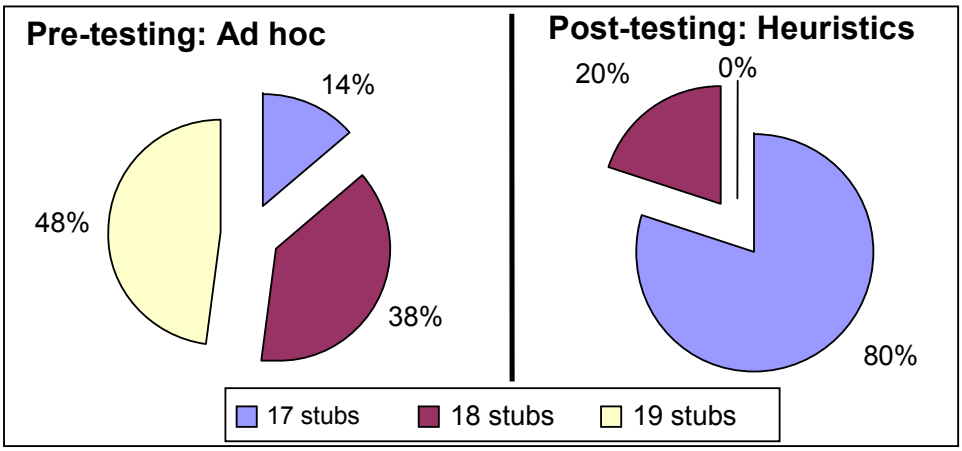

Figure 9 – Comparing the Results

The effort (time in minutes) to identify class integration order was also measured during the experimental studies, taking into account the subjects' groups: academic and industrial environments. Coefficients of variation associated to time decreased in post-testing, as shown in Table 3. This situation indicates lower time dispersion, which could represent an additional contribution of the heuristics application process.

Table 3 – Integration Order Identification Effort – Pre and Post-testing.

Experimental Study	Identification Effort	Academic Environment	Industrial Environment
Pre-existing Procedures (Ad-hoc)	Average	20.91	15.78
	Standard Deviation	7.11	10.57
	Coefficient of Variation	33.98	66.98
Heuristics Effectiveness	Average	30.91	39.78
	Standard Deviation	9.21	20.61
	Coefficient of Variation	29.81	51.81

Other relevant information and suggestions were acquired with the qualitative analysis of the experimental studies, such as:

- The use of OO concepts and UML semantics as factors for decision-making during the application of ad-hoc procedures by participants 2, 7, and 13 allowed them to obtain minimum testing effort during the Pre-existing Procedures (pre-testing) study;
- Subject behavior was the same throughout the experimental studies. It was not possible to identify any correlation between subject performance in the studies and subject characterization (e.g.: academic formation or professional experience in software);
- Most of the subjects indicated they could use the heuristics and its respective application process in future software projects;
- The calculus represents an error prone activity. Some calculation mistakes (not related with the integration testing strategy) happened for two participants during the heuristics effectiveness experimental study. It probably happened due to the large number of classes in the model.

Another contribution from the experimental studies for this research is related to the analysis of their own limitations. It allowed the identification of the need for planning new experimental studies to characterize integration testing strategy adherence in other software contexts, mainly its feasibility when applied to large scale OO software projects (great number of classes in the model).

4. FAROL Tool

Subjects in the study indicated their interest in applying the proposed integration strategy to future software projects. However, the heuristics application process consists of performing many iterations to obtain class integration order. Moreover, the number of interactions increases according to the size of the project. This scenario produced the need to automate the strategy presented in this work.

In this context, a CASE tool (entitled FAROL) was developed aiming at effort reduction in applying the proposed integration testing strategy to OO software projects. As shown in Figure 12, FAROL's main screen has five areas: (A) Main Menu and Toolbar; (B) Class Diagram Panel; (C) Order Sequence Panel; (D) Ordered Class List; and (E) Ordered List Modification Panel.

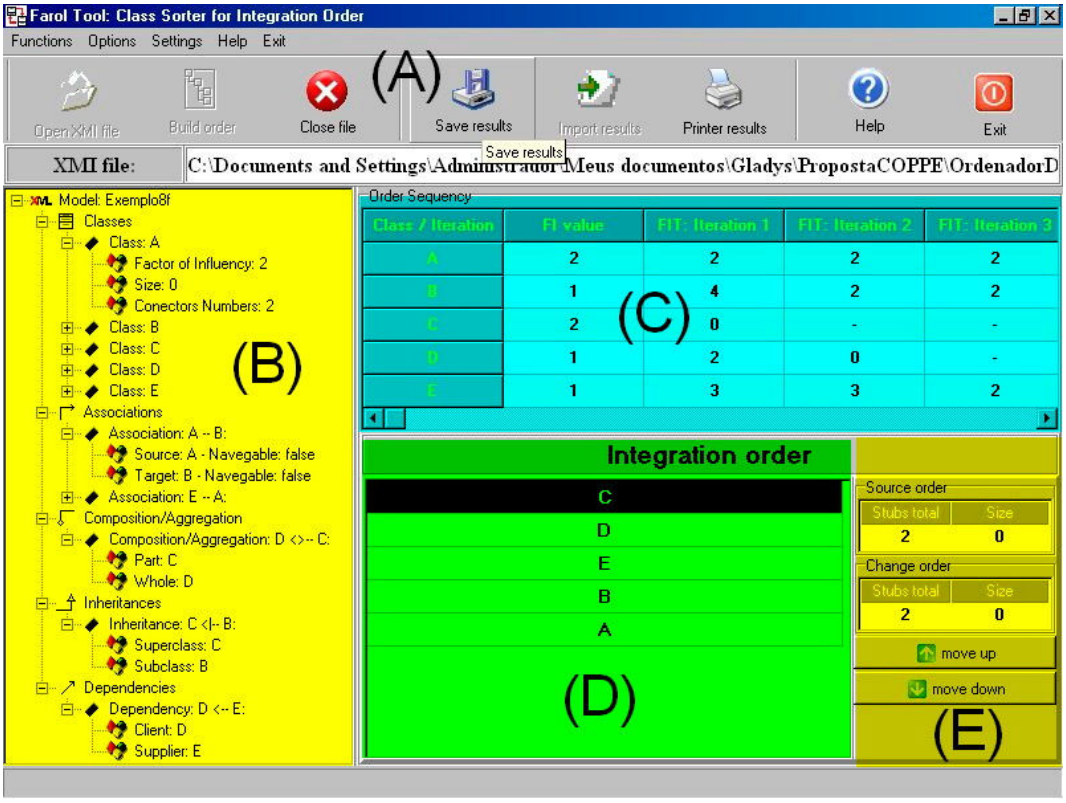

Figure 12 – FAROL's main screen.

The input for the heuristics application process is an UML class diagram. However, there are several commercial UML Modeling Tools. Aiming at maximizing FAROL's usage in software projects, we decided to develop a tool that does not depend exclusively on one UML modeling tool. The solution applied to integrate FAROL and UML modeling tools was to adopt XMI (XML Metadata Interchange), a XML standard that provides a format for the description of UML model elements, available in most of UML modeling tools (i.e. Rational Rose, Together, Poseidon, etc.). However, currently, FAROL is able to import just XMI file describing UML diagrams developed in accordance with the UML DTD (Document Type Definition) version 1.3 published by OMG (Object Management Group). Therefore, any UML modeling tool may generate a XMI file in that format corresponding to the UML class diagram and as a result FAROL must be able to import such files to identify the class integration order. FAROL has four components: XMI Translator; Order Builder; Evaluator; and Exporter, as shown in Figure 13.

The XMI Translator component is responsible for the translation of the added XMI file (generated by an UML modeling tool) and the recognition of the relevant elements concerned with the heuristics (i.e. classes, inheritance, associations, compositions/aggregations, dependencies). This information is grouped and presented according to the elements of the original UML diagram using a tree structure, as shown in the Class Diagram Panel (Figure 12 – area B). Besides, the XMI Translator component is also responsible for generating the precedence matrix. This two-dimensional matrix (N x N) contains the information on the precedence established by the heuristics for all N classes that form the model. In this matrix, cell [X,Y] filled with the value 1 indicates that class X has precedence over the class Y; otherwise, its value must be 0. Thus, depending on the way the precedence matrix is explored. If reading by line, the classes that must be considered after the class defining the line. If reading by column, the classes that must be considered before the class defining the column.

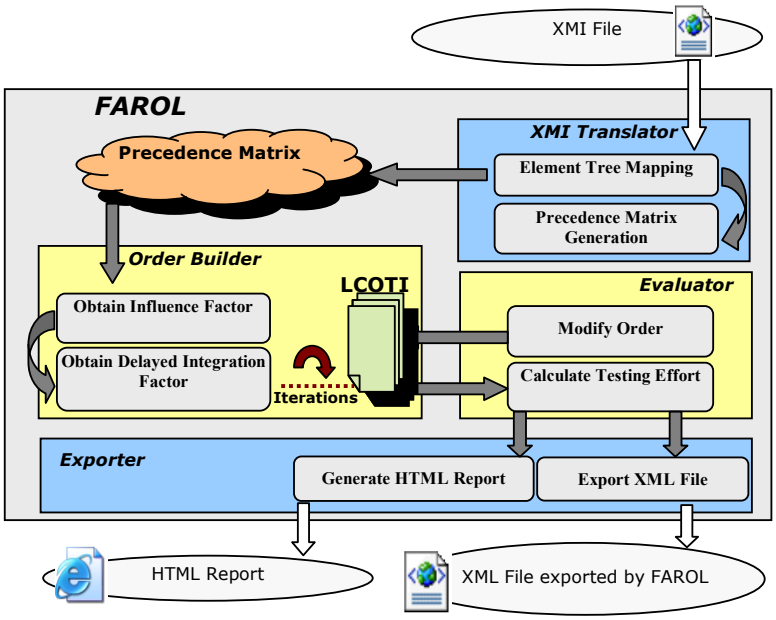

Figure 13. FAROL's Architecture

Table 4 shows the precedence matrix generated from the model presented in Figure 14. For example, in Table 4, the precedence of class A over the B and E classes is represented by the values 1 found in cells [A,B] and [A,E], respectively. Another information extracted from the precedence matrix is the precedence of A and C classes over the class B, observed from the value 1 in the cells [A,B] and [C,B].

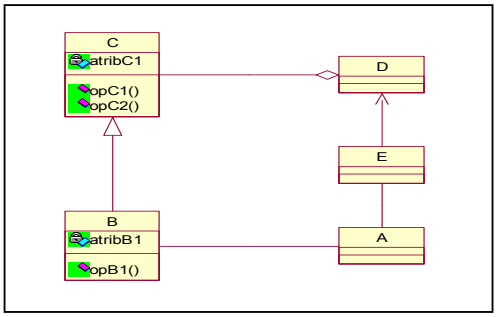

Figure 14 – UML Class Diagram

Table 4 – Precedence Matrix for the Figure 14 Example

Classes	A	B	C	D	E
A	0	1	0	0	1
B	1	0	0	0	0
C	0	1	0	1	0
D	0	0	0	0	1
E	1	0	0	0	0

The preparation of the precedence matrix allows FAROL to start the execution of the heuristics application process. The Order Builder component is responsible for its accomplishment. It determines the Class Order List for Integration Testing (LCOTI) from an UML class diagram entered into the XMI Translator component. The values for FI and FIT (for each interaction) can be calculated from the precedence matrix:

- **Influence Factor.** The FI for one class consists of the total number of cells with value 1 in its corresponding line in the precedence matrix. For instance, class C's FI in Table 4 is 2, because cells [C,B] and [C,D] contain the value 1 and the other cells in the column C are zero ([C,A],[C,C],[C,E]). The same must be done to calculate FI for the other classes:

$FI(C_i) = \sum_{j=1}^{N} PMat(C_i, C_j)$, that is, FI of C_i class is the sum of all cells for the C_i line.

- **Delayed Integration Factor.** The FIT for one class consists of the sum of the FI for all classes marked 1 in its corresponding column in the precedence matrix. For instance, class B's FIT in Table 4 is 4, because in the class B's column, the cells marked 1 are [A,B] (Class A' FI = 2) and [C,B] (Class C' FI = 2). For each iteration, only the classes that have not yet been included in the class integration order list (LCOTI) must be considered for the calculation of FIT:

$FIT(C_j) = \sum_{i=1}^{N} PMat(C_i, C_j) * FI(C_i)$, that is, FIT of the C_j class is the sum of each line of precedence matrix (considering just the classes that have not been included in the LCOTI yet) multiplied by the FI of the correspondent class.

After having all FI and FIT calculated for all classes, the Order Builder component can display LCOTI. The FI and FIT values and LCOTI obtained for the UML class diagram shown in Figure 14 are shown in the Order Sequence Panel (Figure 12 – areas C and D). Next, the Evaluator component calculates the total number of stubs that need to be built considering the LCOTI obtained. Moreover, it calculates the total size of the stubs that need to be built (stub size = number of stub attributes + number of stub methods) for the LCOTI obtained. FAROL also allows the software engineer to simulate new testing efforts (stub number and size) by suggesting modifications to the original class integration order list (LCOTI) obtained by using the strategy presented in this paper. This feature is implemented by the Evaluator component and can be accessed through the "Move Up" and "Move down" buttons (Figure 12 – area E).

To allow software engineers register the information produced by FAROL and apply it to the software development process, the Exporter component makes available devices to produce two types of artifacts. One related to class integration order and the relation of stubs that must be built to support integration testing (reported in HTML format), and the second, a XML based file having all the results on the OO model that can be re-used in future FAROL uses.

5. Conclusions and Future Works

After performing some experimental studies, we believe that there are some indications that allow characterizing the feasibility and the effectiveness of the proposed strategy for class integration order identification when applied to UML class diagrams. All the results pointed towards the reduction of testing effort if considering the number of stubs.

However, the results obtained also indicated the need for automated support in applying the heuristics for large and complex software projects, since the heuristics application process may involve several iterations and calculus, being considered an error-prone task. Thus, the FAROL tool was developed aiming at the automation of the heuristics application process. It imports XMI files describing UML models generated by different UML modeling tools and, from this file, generates a list with the class integration order according to the presented heuristics.

In spite of the heuristics application process evaluation and automation, we observed some limitations of this work. The testing effort measured by the proposed strategy considers

just the number of stubs that need to be built. It assumes that each stub requires the same effort to be built, that is, minimizing the effort to build stubs would be the same as minimizing the number of stubs. However, this assumption is not true, since stubs have different complexity and effort level. This limitation could be solved if a complexity value were associated to the required stubs. Another limitation is related to the validity of the conclusions obtained. They cannot be extended for larger scale projects, because the results presented refer to small project models (maximum of 14 classes). However, all the models used in the study have been extracted from real software projects. Moreover, the FAROL tool has a limitation: it is able just to import UML diagrams described in accordance with the UML DTD version 1.3.

Finally, some perspectives for future work are identified. The first one is the planning of new experimental studies aimed at extending the original strategy to work with other situations (e.g. complexity of the stubs and subsystems integration). The second one is to adapt the FAROL tool to import diagrams in accordance with the UML 2.0 [18] format. Moreover, we believe that another important challenge would be to observe the applicability of the heuristics in different development paradigms (e.g. Multi-Agent Systems – MAS [19] and Component-based Development [20]).

Acknowledgments

We would like to thank the Brazilian Navy and FAPEM for their support. This work has been accomplished in the context of CNPq project 475407/2003-2.

References

[1] HETZEL, W. C., **The Complete Guide to Software Testing**, 2nd edition, QED Information Sciences, 280pg, 1988.

[2] MYERS, G., **The Art of Software Testing**, John Wiley & Sons, 1979.

[3] BRIAND, L.C.; FENG, J.; LABICHE, Y., **Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders**, Carleton University, Technical Report SCE-02-03, Version 3, October, 2002. Available at http://squall.sce.carleton.ca/people/briand/pubs.html#2002 in June 03 of 2006.

[4] BINDER, R.V., **Testing object-oriented systems: models, patterns, and tool,** Addison-Wesley, 2000.

[5] KUNG, D., GAO, J., HSIA, P., TOYOSHYMA, Y., **Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs**, Journal of Object-Oriented Programming, vol. 8 (2), pp 51-65, 1995.

[6] TAI, K.C; DANIELS, F.J, **Test Order for Inter-Class Integration Testing of Object-Oriented Software**. 21st International Computer Software and Applications Conference, pp 602-607, August, 1997.

[7] TRAON, Y.L.; JÉRON, T.; JÉZÉQUEL, J.; e MOREL, P., **Efficient Object-Oriented Integration and Regression Testing**, IEEE Transactions Reliability, Vol. 49, no. 1, Page(s): 12-25, 0018-9529/00, 2000.

[8] BRIAND, L.C.; LABICHE, Y.; YIHONG, W, **An investigation of graph-based class integration test order strategies**, IEEE Transactions on Software Engineering, 0098-5589/03, Vol. 29, Page(s): 594 -607, July, 2003.

[9] BOOCH, G., RUMBAUCH, J., JACOBSON, I., The Unified Modelling Language User Guide, Addison-Wesley Professional, 2nd edition, 2005.

[10] LIMA, G.M.P.S.; TRAVASSOS, G.H., **A Strategy for Object-Oriented Software Integration Testing**, LATW 2004, pp 155-166, Salvador-BA, March, 2004.

[11] TRAVASSOS, G.H.; WERNER, C.; VASCONCELOS, F.M., **Testing Complex Object-Oriented Models: a Practical Approach,** II International Congress on Informational Engineering, Buenos Aires, Argentina, 1995.

[12] VILLELA, K., **Ambientes de Desenvolvimento de Software Orientados à Organização**, D.Sc Thesis., COPPE/UFRJ, Rio de Janeiro, 2004. Available at http://ramses.cos.ufrj.br/taba (publications, in June 03 of 2006).

[13] PFLEEGER, S.L., **Albert Einstein and Empirical Software Engineering. IEEE Computer, pp. 32-38**, October, 1999.

[14] ZELKOWITZ, M.V., WALLACE, D.R., BINKLEY, D.W., **Experimental Validation of New Software Technology**, Lecture Notes On Empirical Software Engineering, Chapter 6, pp 229-263, World Scientific, 2003.

[15] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.C.; REGNELL, B.; WESSLÉN, A., **Experimentation in Software Engineering: an Introduction**. Kluwer Academic Publishers, Massachusetts, 2000.

[16] TRAVASSOS, G.H., GUROV, D., AMARAL, E.A.G.G., **Introduction to Experimental Software Engineering**, In: Technical Report ES-590/02-April, PESC, COPPE/UFRJ, available at http://www.cos.ufrj.br/publicacoes, 2001.

[17] BRIAND, L.C., LABICHE, Y., WANG, Y., **Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles**, 12th ISSRE, Hong Kong, pp 287-296, November 27-30, 2001.

[18] OMG (2005), "**Unified Modeling Language (UML). Version 2.0 Superstructure Specification".** Object Management Group, OMG document formal/05-07-04.

[19] LUCENA, C., **Ongoing Research on the Software Engineering of Multi-Agent Systems**, 18th Brazilian Symposium on Software Engineering, pp 02-09, October, 2004.

[20] CRNKOVIC, I., CHAUDRON, M., LARSSON, S., **Component-Based Development Process and Component Lifecycle**, Proceedings of the international Conference on Software Engineering Advances – ICSEA, IEEE Computer Society, Washington, DC, 44, October, 2006.