

**Software Stability:
Timeless Architectures and System of Patterns
- Proceedings-**

AICCSA 2005 Workshop # 3

<http://www.engr.sjsu.edu/~fayad/workshops/AICCSA-05>

or

<http://www.activeframeworks.com/publications/workshops/AICCSA-05>

In Association with the
3rd ACS/IEEE International Conference on
Computer Systems and Applications

AICCSA05

American University in Cairo (AUC), Cairo, Egypt

January 3-6, 2005

<http://engr.smu.edu/cse/AICCSA-05/>



Mohamed E. Fayad, PhD, San José State University, U.S.A.
Haitham S. Hamza, University of Nebraska-Lincoln, U.S.A

University of Nebraska-Lincoln
Computer Science & Engineering Department
Technical Report No. 04-12-03
December 2004

Preface

This proceedings contains the contributions to the 1st Workshop on Software Stability: *Timeless Architectures and Systems of Patterns*, held in conjunction with the 3rd ACS/IEEE International Conference on Computer Systems and Applications, American University in Cairo, Cairo, Egypt January 3, 2005.

The increasing complexity of software systems coupled with the time-to-market constraints and condensed development budgets all have imposed a real challenge on the future of software development. It becomes a matter of survivability for a business to be able to deliver a high-quality and cost-effective software product in a timely manner. This goal can be greatly precluded by the rapid advances in technology as well as the increasing pace of changes in market needs and customer requirements. Such changes while cannot be avoided; their impact on the system development should be alleviated. A system that requires a major redesign effort in order to adapt to new requirements and emerging technologies is considered to be unstable. A stable system, on the other hand, can handle changes to the system with minimal cost by avoiding unnecessarily changes when redesigning the system. However, developing systems that can evolve gracefully to accommodate necessarily changes without inducing unnecessarily cost is still a challenge in software community.

The motivation of this workshop is to investigate both theoretical and practical aspects of accomplishing stability in the different levels of software development. In this workshop, we have 11 original contributions that highlight the state-of-the-art and practice in developing stable software systems. Contributions span the different phases of software development from requirements and business domain analysis to design and deployment. Some papers address the theoretical aspects of developing stable systems, while some contributions investigate the development of stable patterns, and the use of patterns to architect stable systems.

The web page of the workshop may be found at URL:

<http://www.engr.sjsu.edu/~fayad/workshops/AICCSA05>

<http://www.activeframeworks.com/publications/workshops/AICCSA05>

We would like to thank the conference organizers for their support as well as the authors and participants for their high-quality submissions and engaged contributions during the workshop.

Sincerely,

The Workshop Organizers

Mohamed E. Fayad and Haitham S. Hamza

Organization

Dr. Mohamed E. Fayad –Chair

Professor of Computer Engineering

Computer Engineering Dept., College of Engineering

San José State University One Washington Square, San José, CA 95192-0180

Ph: (408) 924-7364, Fax: (408) 924-4153

E-mail: m.fayad@sjsu.edu

<http://www.engr.sjsu.edu/fayad>

MOHAMED E. FAYAD is a Full Professor of Computer Engineering at San Jose State University. He was a J.D. Edwards Professor, Computer Science & Engineering, at the University of Nebraska, Lincoln and an associate professor at the computer science and computer engineering faculty at the University of Nevada, from 1995 - 1999. He has 15+ years of industrial experience. He has been actively involved in over 60 Object-Oriented projects in several companies using Shlaer-Mellor, Colbert, OMT, Use Case Approach, UML, Design Patterns, Frameworks, Software Process Improvement, Systems & Software Engineering, Internet and Web Applications using Java, OO Distributed Computing using CORBA, and others.

Dr. Fayad is a Senior Member of the *IEEE*, a Senior Member of the *IEEE Computer Society*, a Member of the *ACM*, an *IEEE* Distinguished Speaker, an Associate Editor, Editorial Advisor, and a Columnist for *The Communications of the ACM* and his column is *Thinking Objectively*, and a columnist for *Al-Ahram Egyptians Newspaper* (12 million subscribers), an Editor-In-Chief for *IEEE Computer Society Press - Computer Science and Engineering Practice Press (1995-1997)*, IASTED Technical Committee member on Software Engineering (2001-2004), and a general chair of IEEE/Arab Computer Society International Conference on Computer Systems and Applications (AICCSA 2001), Beirut, Lebanon, June 26-29, 2001

Dr. Fayad was a guest editor on nine theme issues: *CACM's OO Experiences*, Oct. 1995, *IEEE Computer's Managing OO Software Development Projects*, Sept. 1996, *CACM's Software Patterns*, Oct. 1996, *CACM's OO Application Frameworks*, Oct. 1997, *ACM Computing Surveys – OO Application Frameworks*, March 2000, *IEEE Software - Software Engineering in-the-small*, Sept./Oct. 2000, and *International Journal on Software Practice and Experiences*, July 2001, *IEEE Transaction on Robotics and Automation -- Object-Oriented Methods for Distributed Control Architecture*, October 2002, and *Annals of Software Engineering Journal – OO Web-Based Software Engineering*, October 2002. He has published articles in many journals and magazines, such as *IEEE Software*, *IEEE Computer*, *JOOP*, *ACM Computing Surveys* and *CACM* on OO software engineering methods, experiences, aspect-oriented programming, internet & web applications, enterprise and application frameworks, design patterns, and management. He has given tutorials and seminars on OO Technologies and Experiences at many conferences and he has presented various seminars in several countries: Hong Kong (April 96), Canada (10 times), Bahrain, Saudi Arabia, Egypt (12 times), Portugal (Oct. 96, July 99), Finland (July 99), Mexico (Oct. 98), Argentina (3 times), Chile (00), Peru (02), and Spain (02).

Dr. Fayad received an MS and a Ph.D. in computer science, from the University of Minnesota at Minneapolis. His research topic was *OO Software Engineering: Problems & Perspectives*. He is the

lead author of several Wiley books: *Transition to OO Software Development*, August 1998, *Building Application Frameworks*, Sept., 1999, *Implementing Application Frameworks*, Sept., 1999, *Domain-Specific Application Frameworks*, Oct., 1999, and a new book in Progress: *Stable Software Patterns: Analysis, Design, and Applications*.

Haitham S. Hamza – Co-Chair

Computer Science & Engineering Dept
University of Nebraska, Lincoln
256 Avery Hall, P.O. Box 880115, Lincoln, NE 68588-0115
Ph: (402) 472-3485
E-mail: hhamza@cse.unl.edu

HAITHAM S. HAMZA received an MS in computer science, from the University of Nebraska-Lincoln, August 2002 and an MS in Electronics & Communication Engineering from Cairo University, December 2000. His research topic was *A Foundation For Building Stable Analysis Patterns*. He is a co- author of a new book on “*Stable Software Patterns*” with Dr. M.E. Fayad and Dr. M. Cline, Wiley 2005. He has published articles in many conferences and magazines, such as *PLoP*, *ECOOP* and *CACM* on stable model-based architectures, design patterns, stable analysis and design patterns, software stability.

Table of Contents

Architecture Pattern for a Generic Explanation Component.....	1
<i>Susanne Jäger (Universität Klagenfurt, Austria) and Roland T. Mittermeir (Universität Klagenfurt, Austria)</i>	
Stability of Reusable Learning Objects.....	6
<i>Imran A. Zuolkernan (University of Nebraska-Lincoln, UAE),</i>	
Reverse Engineering of Framework Design using a Meta-Patterns-based Approach.....	10
<i>Nuno Flores (Porto, Portugal) and Ademar Aguiar (Porto, Portugal)</i>	
Making Efficient Logging a Common Practice in Software Development.....	17
<i>Osama M. Khaled (American University in Cairo, Egypt) and Hoda M. Hosny (American University in Cairo, Egypt)</i>	
A Novel Approach for Managing Business Rules Evolution and Reuse in Business Architectures.....	22
<i>Haitham Hamza (University of Nebraska-Lincoln, USA), Mohamed E. Fayad (San José State University, USA)</i>	
Architectural Modelling to Understand System Evolution.....	29
<i>Galal H. Galal-Edeen (Cairo University, Egypt)</i>	
Software Stability : A Rewriting Formal Specification Approach.....	38
<i>Mohamed L. Rebaiaia (University of Batna, Algeria)</i>	
Planning Stable Software Applications using Goal Driven Requirements Analysis.....	46
<i>Islam A. M. El-Maddah (Ain Shams University, Egypt)</i>	
Performance Modeling and Analysis of Object Oriented Distributed Software Systems: A Necessary Step Toward Software Performance Stability.....	52
<i>Reda Ammar (Univ. of Connecticut, USA), Amal Abdel-raouf (Univ. of Connecticut, USA) and Tahany A. Fergany (Univ. of New Haven, USA)</i>	
A Stable Architectural Model for Networks with Trajectory-Dependent QoS.....	57
<i>Ahmed M. Mahdy (University of Nebraska-Lincoln, USA) and Mohamed E. Fayad (San José State University, USA)</i>	
Monitoring and Reuse Software Patterns Analysis in Maude.....	61
<i>Mohamed L. Rebaiaia (University of Batna, Algeria)</i>	

Architecture Pattern for a Generic Explanation Component

Susanne Jäger, Roland T. Mittermeir
Institut für Informatik-Systeme
Universität Klagenfurt
AUSTRIA
{susi, roland}@isys.uni-klu.ac.at

Abstract

The paper reports on the architecture of an explanation component for an educational simulation system. This architecture assures tolerance against modifications of the model driving the simulation.

Architectural stability is achieved by extending the principle of state-dismemberment to aspects of application processing. Querying aspects of the model and relating it to aspects of the state of the simulation yields information to construct another query into model aspects and state aspects. Relating the respective information allows providing users with focused feedback, while the software driving this process has just to follow general principles.

1. Motivation

The architecture described in this paper has been developed for *AMEISE*¹, a system that allows students to practice managing software development projects [1, 2]. *AMEISE* consists of a generic simulator, *SESAM* [3], which has been wrapped and extended by various tools to allow instructors to tune the difficulty of a simulation task. Such extensions comprise a *consultant*, which can be asked how to proceed at certain critical points, and a *friendly peer* which intervenes when noticing that students are progressing into a fundamentally wrong direction.

Architecture, in general, defines the evolution interface of systems. Systems are subject to evolution drivers of different sort [4]. A standard strategy to deal with unforeseen change is modularity. In specific situations, though, more powerful approaches can be used. Treating software as data [5] is one of them. At the expense of run-time efficiency, it allows to hard-code only the most

fundamental parts of a system in the classical way. This yields a generic framework which is completed to an executable system by parameters, read as data and properly interpreted. Shaw has formalized this under the term *interpreter pattern* [6].

In *SESAM* this principle is followed as so far as the system consists of a simulator which loads the model of a particular software development scenario. This scenario is expressed in a rule-based language. It describes developers and their qualifications to produce various artefacts by pursuing specific activities. Modelled are: the time needed to produce a specification or to perform a test; the budget consumed during these activities; errors introduced, detected, or corrected during an activity. Performance depends on the task, the person's qualification, and the quality of the base used for performing this task. In *SESAM*, the results of such a simulation need to be interpreted by the instructor.

To relieve instructors from some of the time consuming interpretation of results and to give students intermediate feedback and support, *AMEISE* has to interpret final as well as intermediate results of the simulation. This requirement motivated us to extend the simulation system towards an expert system with an explanation component. The challenge of this requirement is that such an interpretation has not only to be aware of the model but also of the interactions between the model and the user's state space. Both are highly complex if the task is to realistically describe software development. A further challenge is to make this component resistant against future changes of the model.

In the remainder of this paper the interpreter pattern is briefly reviewed. It is shown how this pattern can be applied to the task at hand. Then, the specific architecture, which has to cope with rules of different complexity, is described. To make this approach even more change resistant, a special rule insertion tool has been developed.

¹) Development of *AMEISE*, A Media Education Initiative for Software Engineering, has been supported by bm:bwk, the Austrian Federal Ministry of Science, under grant NML-1/77.

2. Interpreter Pattern, interpreted

2.1. Point of Departure

The key principle behind the interpreter pattern [6] is to split the semantics of a software solution into domain specific problem semantics and system specific solution semantics. Providing this separation on the same level of abstraction will lead to a conventional modularised system. With the interpreter pattern, the solution is raised to a higher, more abstract level. The resulting generic solution is parameterised by an interpretable description of the specific category of the problem domain for which the fully developed system is finally to be used.

In a nutshell one might think about a Universal Turing Machine with the *UTM* being the interpreter, the domain description being the program for the *UTM*, and the actual user data and system-state being whatever rests on *UTM's* tape.

The *UTM*, of course, is an extreme case. When developing stable application systems, the designer's qualifications will ensure that a good trade-off is found between the complexity of the actual interpreter and the one of the domain description. A balanced solution will result in a stable situation, involving as little maintenance effort as possible.

2.2. Recursive Use

With explanation components as needed in *AMEISE* or in similar complex reasoning systems, direct use of the interpreter pattern would be too weak. Both, the program to be simulated as well as the (intermediate) state of a simulation would be too complex to allow a clear decision where to separate between the general solution (interpreter), problem description (program) and data (user state). To be efficient, the user state has to be addressed repeatedly and in a highly focused manner. As additional requirement, the *AMEISE* solution should be robust against changes of the model.

To cope with this, the interpreter pattern is applied recursively, till a message can be constructed which can be finally emitted to the user.

As described in the following section, this requires repeatedly accessing the state space and comparing the results obtained with partial descriptions derived from the model. Based on the results of this comparison and its relation to parameters of the target space (also depending on the particular model used in the simulation), either another query is formulated against the state space, the model description, and result description, or a message to the HCI-component is emitted.

3. Prototypical Realization of this Strategy

3.1. Principles of Evaluation Rules

The information needed to analyse a project's progress is stored in a database. In *SESAM*, knowledge how to analyse this data rests with the instructor. Observing an experienced instructor's reasoning and interpretation led to a list of assessment criteria [7].

"Project cost" is an important criterion which is easy to analyse. It centres on the question "Has the student exceeded the limit of €225.000 for project cost?" The answer will be either *YES* or *NO*. Though simple, this example shows the recurring elements of *AMEISE* evaluation rules. Every assessment criterion consists of one or more **attribute(s)** to be analysed and for each attribute an associated **reference point**. Relating the reference point of an attribute to the respective state of the simulation by a comparison operator yields either *YES* or *NO* as intermediate result. According to this answer, different evaluation texts are emitted.

Therefore, every assessment criterion is represented as a set of so-called specific aids. Each specific aid handles one of these cases. E.g., the assessment criterion "project cost" can be analysed by answering the question "Has the player exceeded the project costs limit of €225.000?". To account for each possible answer, it is represented by two different specific aids.

- 1) "total_cost is greater than €225.000", and
- 2) "total_cost is less or equal €225.000".

Unfortunately, only a few assessment criteria can be described in this simple form. For more complex criteria elementary rules must be combined to chains of rules by logical operators. E.g., "AFP_for_the_specification > 199 AND errors_in_specification < 30" should yield "Your specification is complete and quite correct!". An example dealing with task assignment to developers is "author_specification == Diana AND qualified_for_writing_specifications == Diana". It might yield "Diana was the right choice for writing the specification." This shows that it is also possible to use text as reference point.

Each chain of rules consists of a set of single rules which are called instances. For successful execution of the whole rule, the order in which terms representing elementary rule-instances are combined must be retained.

The evaluation of an assessment criterion is carried out by a rule interpreter. Its task is to get the relevant information (i.e., all specific aids which belong to the selected assessment criterion) from the database and to formulate an automatically generated SQL-query which finally yields the explanations to be given to the student as shown in Fig. 1. All project attributes relevant for the

evaluation of the project are stored in the database and can be obtained by an SQL-statement which gets the value of an attribute for a given simulation run from the database.

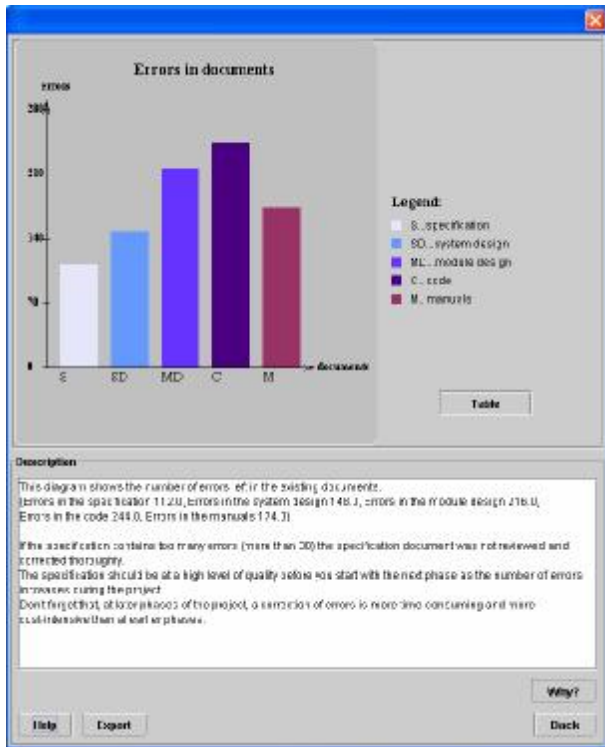


Fig. 1: Evaluation of an assessment criterion with a diagram

The following example describes in detail the execution of the rule interpreter. Let's assume the user has finished the simulation run and wants to know whether he/she was successful. To check, if he/she stayed within the budget line, he/she chooses from the pool the assessment criterion "project cost".

Now the system starts to analyse the actual project situation in reference to the selected criterion. First of all, it searches the database for all specific aids related to this criterion. For "project cost" it finds two specific aids with appropriate information as follows:

	specific aid 1	specific aid 2
attribute	cost	cost
operator	>	<=
reference point	225.000	225.000
evaluation text	"You have exceeded the limit!"	"You have not reached the limit!"

They provide a comprehensive description. Hence, the system executes them in order, until one evaluates to

TRUE. To execute a specific aid implies the rule interpreter to execute an SQL-statement to load the actual value of the attribute "cost" from the database. Assuming the actual attribute value is 216.000, it combines this value, the operator (assume ">") and the reference point (allocated budget of € 225.000) to a new predicate. In the example, this predicate is: "216.000 > 225.000". Its evaluation yields *FALSE*. Thus, the rule interpreter aborts evaluation of this alternative and tries the next specific aid. In this case, this will be composed to the predicate "216.000 <= 225.000" which evaluates to *TRUE*.

For this simple case, there are no more rules in the rule chain. So the examination of this specific aid terminates successfully and the evaluation text associated to this (single element) chain of rules will be passed to the client for display.

In more complex cases the chain is followed as long as the partial term of the composite Boolean expression still evaluates to *TRUE*. If complete evaluation yields *TRUE*, the system emits the message, if not, it progresses to the next chain.

The construction of the specific aids ascertains that every possible attribute value has to be covered by a single rule or a chain of rules. Each rule should contain a meaningful message, to describe the user's mistakes and give hints for further projects.

3.2. Reuse of the Pattern

The architecture of the online evaluation tool has been reused to support users also during the simulation run [8]. The respective components are called *consultant* and *friendly peer*.

The *friendly peer* models a senior colleague using her/his domain knowledge to support the junior. Observing commands entered by the student, hints will be given whenever decisions seem extremely problematic.

The *friendly peer* is realized as an agent, who runs in the background and is therefore completely transparent for the user. When the agent detects a mistake, it contacts the user. Therefore, it has to permanently observe every situation emerging during the whole simulation run. On every simulation step, the agent checks whether problematic effects are conceivable according to the actual situation.

Example: A typical problem in *AMEISE* is to forget the customer during the project. The *friendly peer* can watch for commands like "review the specification" and can ask the student whether he/she forgot inviting the customer to participate.

Another component for user support is the *consultant*. It can be conceived as tutor or advisor. The *consultant* can be asked by the user whenever he/she has

a question with respect to the actual project situation. The user takes the initiative by choosing a specific question that is available in a well structured pool of questions.

Example: The user finishes the specification and wants to start with the design. He asks the *consultant* if the quality of the specification document is good enough to start designing. Inspecting the intermediate state of the project, the *consultant* can give the hint that the quality of the document is not good enough at the moment. Hence, the user should make a review and correct the errors detected before starting with design.

The *consultant* can only answer questions referring to project phases that were already in progress. Otherwise the *consultant* replies that there is no valid answer possible at this moment.

Example: If the user asks something about the code while implementation has not even been started, the *consultant* replies that this question cannot be answered yet.

The knowledge needed by these components is also stored in the database in form of the pattern defined for the evaluation component. The basis is the same model. But *consultant* and *friendly peer* have their own set of assessment criteria, which are also represented by questions like those described for the evaluation component. For the execution of these questions similar rules or chains of rules are needed.

It's due to the generic pattern that all three components are resistant against future changes of the model and can also be used for models developed in the future. The architecture is even resistant against complete changes of the model. Replacing the model will allow to simulate even a marketing campaign instead of software development.

4. User Support and Extensions

To explore whether this architecture is tractable from a performance perspective, the prototypical realization consisted of problem specific data and SQL code properly chopped up, written directly by the developer's of this solution. As the experiments conducted with this solution were satisfactory, a maintenance tool has been constructed to allow specification of explanations on a higher level. The tool constructs the respective SQL queries and properly chops them up into chains of elementary terms [9].

As of current, the system developer using this tool has still to know rules of the model and their first order

interactions with the state space. It also rests with the system developer to assure that each situation is comprehensively covered by the set of rules (chains of rules) related to an attribute or to a particular section of the state space. Automatically deriving explanation rules from rules of the model in a provably correct manner is still subject for further research.

5. Summary / Conclusion

Writing explanation components to give meaningful feedback even at intermediate states of a simulation is a complex development task. Related maintenance activities are cumbersome and costly. By recursively using the interpreter pattern, one reaches a solution, where the maintenance task of the explanation component can be shifted from software developers to model developers, i.e. to application experts.

Thus, even in the presence of application driven change, the software itself remains stable.

References

- [1] Mittermeir R T., Hochmüller E, Bollin A., Jäger S., Nusser M.: "AMEISE – A Media Education Initiative for Software Engineering: Concepts, the Environment and Initial Experiences"; *Electronic proc.: ICL – Interactive Computer Aided Learning*, Villach, Sept. 2003.,(ISBN 3-89958-029-X.)
- [2] <http://ameise.uni-klu.ac.at>, Aug. 2004.
- [3] Drappa A., Ludwig J.: "Simulation in Software Engineering Training"; *Proc. 23rd ICSE*, IEEE-CS and ACM, May 2001, pp 199 – 208.
- [4] Mittermeir R.T.: "Software Evolution: A Distant Perspective"; *Proc. 6th Internat. Workshop on Principles of Software Evolution, IWPSE '03*, IEEE-CS Press, 2003, pp. 105-112.
- [5] Mittermeir R., Oppitz M.: "Software Bases for the Flexible Composition of Application Systems"; *IEEE-Trans. on Software Engineering*, Vol. SE-13/No. 4, April 1987, pp. 440 - 460.
- [6] Shaw, M.; Garlan D.: "Software Architecture: Perspectives on an Emerging Discipline"; *Prentice-Hall*, 1996.
- [7] Jäger S.: „Entwicklung eines elektronischen Tutors zur Analyse von Projektverläufen“; Diplomarbeit Universität Klagenfurt 2003.
- [8] Nusser M.: „Software-Agenten zur Analyse von Projektverläufen“; Diplomarbeit Universität Klagenfurt 2003.
- [9] Gratzner W., Seidl J, Vorraber W.: „Handbuch zum RIETA tool“; ISYS/AMEISE-report 07/2004.

Stability of Reusable Learning Objects

Imran A. Zualkernan
American University of Sharjah
izualkernan@ausharjah.edu

Abstract

In recent years, re-usable learning objects have gained popularity as the primary mechanism for organizing e-learning content. However, little has been said about the “stability” of these objects. Much more so than software, “content” has a problem of obsolescence as information about what needs to be learnt changes very rapidly. In this paper we present some preliminary thoughts on how concepts from software stability can be applied to design more stable re-usable learning objects.

1. Introduction

The idea of re-usable learning objects has been derived from object-oriented design. While it takes many forms [1][2][3], the primary concept is to structure learning content in an object-oriented and instructionally sound manner so that the learning content can be re-used in various e-learning applications. Cisco [1] provides one widely used framework that divides a “learning object” into objectives, re-usable information objects and pre and post-assessments. A course, lecture or a module, in turn, can be composed of multiple re-usable learning objects.

The concept of stability or change applies across a particular learning object. For example, the objectives of the learning objects can change. Similarly, each reusable information object can change as well. The key to designing a stable re-usable learning object is to ensure that the impact of all these changes is minimized.

In the context of stable object-oriented design, Fayed [4] [5][6] has proposed a classification of objects into

- **EBT** – or Enduring Business Themes. These are enduring principles that undergo very little or no change. For example, Friendship, Finance etc. are enduring themes.
- **BO** – Business Objects are objects whose external behavior does not change but the

internal behavior may change. A warehouse is an example of a business object.

- **IO** – Industrial Objects are specific concrete objects that may change with time. For example, a range in a kitchen is an industrial object.

In the context of reusable learning objects, we propose that for stability, learning objects can be analyzed based on the following three categories.

- **Enduring Learning Themes** – Like EBTs, Enduring Learning themes are unchanging aspects of what needs to be learnt.
- **Learning Templates** – Like business objects, learning templates are structures that conveys external learning goals but whose internal manifestation can change.
- **Learning Objects** – Like industrial objects, these are specific learning objects that convey a particular learning content.

2. Case Study

We use a sample module from the MIT’s open course initiative [7] to illustrate these ideas. The specific course is “6.170 Laboratory in Software Engineering.” Within this course, the module being analyzed is the lecture on “Decoupling 1.”

In its “surface” textual form (in pdf format) as presented on the website, this lecture has the following outline.

- 2.1 Decomposition
 - 2.1.1 Why decompose?
 - 2.1.2 What are the parts?
 - 2.1.3 Top down design
 - 2.1.4 A better strategy
- 2.2 Dependence relationships
 - 2.2.1 uses diagrams
 - 2.2.2 dependencies and specifications
 - 2.2.3 weak dependencies
- 2.3 Techniques for decoupling

- 2.3.1 façade
- 2.3.2 hiding representation
- 2.3.3 polymorphism
- 2.3.4 call-backs
- 2.4 Coupling due to shared constraints
- 2.5 Back to Dijkstra: conclusion

An analysis of the text of the lecture, however, yields the Enduring Learning Theme, Learning Templates and the Learning Objects shown in Figure 1.

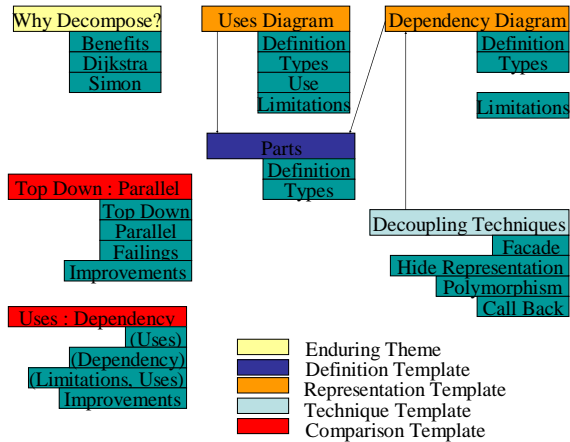


Figure 1 Enduring Learning Theme, Learning Templates and Learning Objects in the “Decoupling 1” module

As Figure 1 shows, the Enduring Learning Theme embedded into this module is really a notion of “why decompose?” This theme is concerned with why one ought to decompose systems at all. This concept is reinforced with examples from Dijkstra and Simon.

Because this specific lecture deals with software design (or design in general), it is not surprising that the Learning Templates (analogous to business objects) are based on conveying design principles. The three templates being used are as follows:

- **Representation Template** – Design often requires representation for specific purposes whether it is for synthesis or analysis. In this example, two representations are being described – Uses and Dependency diagrams; each is defined, its uses are described and limitations are discussed.
- **Technique Template** – Designs also have specific techniques as solutions to design problems. The technique template simply describes the techniques.

- **Comparison Template** – Part of good design is the ability to compare alternatives. Consequently, the two comparison templates in this example compare top-down with parallel decomposition and the Uses with Dependency diagrams. In each case, limitations and improvements are also discussed.

Finally, the learning objects themselves consist of the use of Uses and Dependency diagrams for decomposition. The other learning objects are the various techniques such as façade, hiding representation, polymorphism and call-backs. Note that like industrial objects these learning objects are the most likely to change. For example, an introduction of a new decomposition technique will lead one to add another representation to Uses and Dependency diagrams. The same is true for introduction of new techniques.

3. Discussion and Conclusion

In this paper we have proposed using the framework developed for software stability to reusable learning objects. An initial analysis and a case study suggest that this is a fruitful direction. This analysis also suggests that while instructional designers may use classifications based on Merrill and Bloom’s [1] distinctions between Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation to understand and construct learning content, these may not be the best ways to organize and structure the learning objects from the perspective of stability.

A deeper understanding of stability for learning objects will also have implications for learning standards such as AICC and SCORM [8][9] that represent runtime frameworks for learning objects. For example, the current version of SCORM does not have explicit support for learning templates or learning themes. An explicit inclusion of these mechanisms will have a profound impact on the stability of learning objects.

We are currently extending and applying the concept of stability of learning objects in diverse areas such as knowledge management and fraud detection in the context of commercial e-learning modules that will lead to a further understanding of the issues and implications of this approach.

4. References

[1] “Reusable Learning Objects Authoring Guidelines: How to Build Modules, Lessons and Topics,” Cisco Systems, Inc., 2003.

[2] David Wiley, "Learning Object Design and Sequencing Theory," Doctoral Dissertation, Brigham Young University, 2000.

[3] www.reusability.org

[4] Mohamed E. Fayed, "How to Deal with Software Stability," Communications of the ACM, Vol. 45, No. 4, April, 2002.

[5] Mohamed E. Fayed and Adam Altman, "An Introduction to Software Stability," Communications of the ACM, Vol. 44, No. 9, September, 2001.

[6] Mohamed E. Fayed, "Accomplishing Software Stability," Communications of the ACM, Vol. 45, No. 1, January, 2002.

[7] <http://ocw.mit.edu>

[8] www.aicc.org

[9] <http://www.adlnet.org>

Reverse Engineering of Framework Design using a Meta-Patterns-based Approach

Nuno Flores, Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto
{nuno.flores, ademar.aguiar}@fe.up.pt

Abstract

Object-oriented frameworks are a powerful reuse technique but they are also very complex and difficult to design. Framework's design aims at separating the invariant aspects across several applications in a domain – frozen spots – from the aspects that vary among applications and thus must be kept flexible and customizable – hot spots. The flexibility and extensibility provided at hot spots is usually achieved by following common design patterns, which are often hard and tiresome to identify without proper documentation. This paper proposes a reverse engineering approach to identify the design patterns used in a framework, using a high-level hot spot representation. The goals of this work include: researching a design approach that produces usable intermediate reuse information; defining a representation for design patterns based on meta-patterns; and developing a supporting tool to automate the reverse engineering process.

1. Introduction

"We have experienced a significant increase in software reusability and an overall improvement in software quality due to the application of object-oriented programming concepts in the development and (re)use of semi finished software architectures rather than just single components." [20].

Object-oriented frameworks are a powerful technique for large-scale software development capable of delivering very high levels of design reuse and code reuse [7] [8] [20].

Before being able to efficiently reuse a framework, software engineers must invest time on understanding and learning how to use it.

But frameworks are hard to learn. This difficulty is mainly due to framework design being very complex (abstract, incomplete, highly flexible and obscure), making frameworks hard to communicate, a problem aggravated by the lack of proper design documentation.

Due to its complexity, object-oriented frameworks are also very difficult to design, the reason why, when designing a new framework, developers benefit from grasping the design of other existing frameworks with

the goal of identifying those parts that might be useful to reuse

The main concern of framework design is to separate the aspects that are invariant along several applications in a domain – the frozen spots – from the other domain aspects that vary among applications and thus must be kept flexible and customizable – the hot spots.

Design patterns represent proven solutions to recurrent design problems and are extremely useful to provide the flexibility and extensibility required at hot spots.

Therefore, uncovering the design patterns used in a framework is very useful to improve the effectiveness of framework reuse.

This paper presents research work been carried on to develop a semi-automated approach to extract and reuse design patterns from existing frameworks. The approach uses a high-level hot spot representation based on meta-patterns.

The paper describes the proposed reverse-engineering process and its supporting tool, after an overview of the key design elements of frameworks, from low-level template-hook mechanisms to abstract design patterns,. It concludes with an outline of the work in progress and discusses it in relation to other existing approaches.

2. Framework Design

A framework can be shortly defined as a reusable design of an application together with an implementation [6] [7] [8] [13] [17].

The definitions for a framework are not consensual and vary from author to author. In few words, a framework can be defined as a semi-complete design and implementation for an application in a given problem domain.

Frameworks are firmly in the middle of the reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more concrete and easier to reuse than a raw design (but less flexible and less likely to be applicable).

The design of a framework is typically complex [5]:

- very abstract, to factor out commonality;
- incomplete, requiring additional classes to create a working application;

- more flexible than the strictly needed by the application at hands;
- obscure, in the sense that it usually hides existing dependencies and interactions between classes.

Shortly, frameworks are considered a powerful reuse technique because they lead to one of the most important kinds of reuse, the reuse of design.

The key elements used to design a framework can be divided in three levels of abstraction: template-hook mechanisms, meta-patterns, and design patterns.

2.1. Hot Spots, Template and Hook methods

Frameworks provide their flexibility at hot spots using two essential constructs: templates and hooks. Template and hook methods are two kinds of methods extensively used in the implementation of frameworks. These terms are commonly used by several authors in [9] [19] [20] [26].

Template methods are implemented based on hook methods, and call at least one other method. A hook method is an elementary method in the context where the particular hook is used, and can be either an abstract method, a regular method, or another template method. An abstract method is a method for which only the interface is provided, and thus lacks an implementation. A regular method is a method that doesn't call hook or template methods, but only provides a meaningful implementation.

Generally, template methods are used to implement the frozen spots of a framework, and hook methods are used to implement the hot spots. Opposite to the latter, the frozen spots are aspects that are invariant along several applications in a domain, possibly representing abstract behavior, generic flow of control, or common object relationships.

The difficulty of good framework design resides exactly on the identification of the appropriate hot spots that provide the best level of flexibility required by framework users. More hot spots offers more flexibility, but results in a framework more difficult to design and use, so somewhere in between resides a balanced design.

2.2. Meta-Patterns

The possible ways of composing template and hook classes in the hot spots of a framework were catalogued and presented under the form of a set of design patterns, which were called meta-patterns [20] [21].

Meta-patterns are a useful abstraction that can be applied to categorize and describe framework hot spots on a meta-level.

Template and hook methods can be organized in several ways. Although they can be unified in a single class, in most of the situations it is better to put frozen spots and hot spots into separate classes. When using separate classes, the class that contains the hook method(s) is considered the hook class of the class containing the corresponding template method(s) – the template class. We can consider that hook classes parameterize the corresponding template class. The hook methods on which a template method is based can also be organized in different ways. They can be defined all in the same class, or in separate classes, in a superclass or subclass of the template class, or in any other class.

In [20], a clear distinction between the level of abstraction of design patterns and meta-patterns is made. Design patterns describe the design of specific frameworks on an abstraction level higher than the underlying programming language, albeit they provide no means of capturing the design independently of a more or less specific framework example. By using meta-patterns, the design is captured one step higher, independently of its particular application. Meta-patterns express how the required flexibility – represented by the hot spots – is gained in a particular framework.

Furthermore, seven composition meta-patterns were identified that relate templates with hooks, as depicted in Figure 1. These repeatedly occur in frameworks and, thus, in its constituent design patterns.

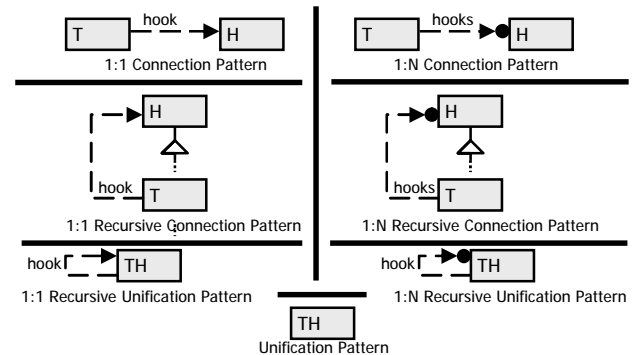


Figure 1 – Catalogued Meta-Patterns

2.2. Design Patterns

Frameworks and design patterns are concepts closely related, representing two different categories of high-level design abstractions [14]. A single framework typically encompasses several design patterns. Patterns provide an intermediate level of abstraction between the application level and the level of classes and objects.

A design pattern is commonly defined as a generic solution to a recurring design problem that might arise in

a given context [2] [4] [10]. The relationships between individual patterns unfold in the application domain naturally and form a high level language, called a pattern language [2]. A pattern language represents the essential design knowledge of a specific application domain, i.e. the experience gained by many designers in solving a class of similar problems.

Design patterns can be viewed as a valuable means to document existing frameworks and develop new reusable object-oriented software architectures [20]. Design patterns and pattern languages are particularly good to document frameworks because they capture design experience and enclose meta-knowledge about how the flexibility was incorporated. Pattern languages help document the application domain of the framework, the design of the framework in terms of classes, objects and their relationships, and also the specifications of important framework classes. The combined use of frameworks with patterns and components is very effective, significantly helping to increase software quality and reduce development effort [8].

Although meta-patterns can be directly used to document the roles of framework participants, they are much more useful to document the roles of the participants involved in a design pattern. Generally, it is preferred to attach meta-patterns to design patterns, and design-patterns to concrete participants that instantiate them, instead of attaching meta-patterns directly to concrete participants, mainly due to the redundancy introduced, and the level of detail being too fine to be useful.

3. Reverse Engineering of Framework Design

As in any high-level software reverse engineering process, the main goal is to begin with the source code (in this case), being it a snippet, a module, package, component or framework and progressively evolve it until we've reached the desired level of information, namely, the design. From the source code to the constituent design patterns it implements, several consecutive steps are performed, each providing results for the next step, like a Pipes & Filters architectural style.

The approach presented here relies on capturing and analyzing information regarding the flexibility and reusability proneness of the source code, grouping it into composing structures of increasing level of abstraction and obtaining the design patterns identified with some level of certainty.

3.1. Goals

The work in progress stated in this paper thrives to reach three goals.

To emphasize reuse information by adopting an approached based on meta-patterns, where each step of the process should produce “deliverable” relevant reuse information for the user.

To define a meta-patterns-based design pattern representation through a representation of design patterns based on its meta-pattern composition. It should be generic enough to be able to represent any design pattern, whether already discovered or yet to unfold.

To provide automation and tool support, by developing a tool to automate the reverse engineering process, flexible, interactive and capable of visually deliver its results to the user.

3.2. Meta-Patterns approach

As described earlier, meta-patterns capture how templates and hooks are composed together. Also stated, design patterns can be characterized by these meta-patterns, emphasizing the flexible aspects of these framework composing elements. Taking these two facts into consideration, a reverse engineering process was devised in order to detect design patterns using these reusability-based properties.

The phased process unfolds as follows, each step leading into the next level of abstraction.

- *Source Code parsing*: at a first stage, the source code is parsed and all relevant information is gathered concerning candidate template and hook methods.
- *Hot spots detection*: a second step groups the templates and hook methods into hot spots. At this level, information regarding the framework flexibility areas is available to use (An already co-developed tool by the authors called “HotSpotter” exists that performs this step. Their intention is to adapt it to this process).
- *Meta-Patterns Detection*: the templates and hooks are analyzed and grouped into the known existing seven meta-patterns.
- *Micro-Architectures Detection*: the meta-patterns are then grouped themselves into possible micro-architectures. These are the candidate design patterns.
- *Pattern Recognizer*: this step tries to match the micro-architectures with a design pattern representation based on meta-patterns. This representation was previously stored in a repository or knowledge-base, through a Pattern Loader. This Pattern Loader is a semi-automated tool that allows a new discovered design pattern to be added to the repository, by producing its meta-patterns-based representation.

- *Documentation production*: the final step will consist on producing documentation of the framework flexibility areas, possibly in several formats, such as UML 2 [24], XSDoc [27] or Javadoc [12].

One of the main advantages of this approach is that, despite the final results concerning the detection of the design patterns, the data produced at intermediate steps may be “delivered”, as it is already imbued with reusability information. Notice that the emphasis goes to the reusability aspects of the framework. Already at the Hot spots detection step, there is useful and relevant information concerning the adaptable parts of the source code, namely the hot spots. Thus the process is, itself, flexible, leaving the user with the decision to proceed, or not, to the next level of abstraction.

3.2. Tool support

One the pursued goals is to develop an automated (where possible) tool to support this reverse engineering process. Existing available tool solutions don't address the reusability issue in such a straightforward manner and are rather available for use. The main concerns regarding this tool are:

- It should be flexible, that is, it should be able to plug into an integrated development environment.
- It should be interactive. The user should be able to control its process and customize its behavior and preferences.
- It should be able to generate visual information, whether graphics or text, to store the produced data.

The work plan proposes to develop a supporting tool as an Eclipse plug-in [28] that parses Java source code and produces the earlier described results. Figure 2 depicts an overview of the tool architecture.



Figure 2 – Tool architecture

4. Related Work

The subject of reverse engineering design patterns has already undertaken several approaches in recent years. From automated detection tools to pattern extraction heuristics, most have a variable range of results regarding availability, flexibility, effectiveness

and efficiency. Next, a brief review is made over the existing approaches catalogued in the literature.

Pat [16]: the Pat system is a design recovery tool for C++. Extract design information from C++ header files and stores it in a repository. Patterns are defined as PROLOG rules and the design information is translated into facts. The actual matching work is done by a PROLOG engine. Its limitations lie on dealing with behavioral patterns, since too much semantic information is required.

KT [3]: KT is a tool that can reverse-engineer design diagrams from Smalltalk code and use this information to detect patterns. It supports both static and dynamic modeling information of design patterns. The methods used for the detection of patterns are hard-coded directly into the KT source, thus constraining its flexibility.

Seeman-Gudenberg [22]: this approach introduces several ideas on how to recover design information from Java source code. The method proceeds along successive steps, revealing different layers of abstraction. In a first step, a graph is generated after a source code parsing to collect information about inheritance, method calling and naming conventions. Next, this graph is transformed by graph grammar productions until pattern detection is finally applied. This approach relies on a Pipes & Filters architectural style.

SPOOL [15]: the SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project is a joint industry/university collaboration. The SPOOL reverse engineering environment has a three tier architecture: object-oriented database, repository schema and end-user tools. The lower tier provides physical storage of the reverse engineering model and design information. The middle tier contains the object-oriented schema of the reverse engineering model, comprising static structure and dynamic behavior. The upper-tier consists of end-user tools implementing domain specific functions such as source code capturing and visualization analysis.

Albin-Amiot & Guéhéneuc [1]: this approach bases itself on the definition of a meta-model to represent design patterns. This representation allows design patterns to be detected relying on their structural and behavioral aspects. The source code is parsed and an instantiation of the meta-model is produced accordingly. This instance is then matched against previously stored design pattern instantiations with the same meta-model.

JBOOTRET [18]: JBOOTRET (Jade Bird Object-Oriented Reverse Engineering Tool) uses a parser to extract the higher-level design information and conceptual model from system artifacts. The version for C++ consists of three major components: a data extractor, a knowledge manager and an information

presenter. The approach intends to separate data extraction from information representation, thus preventing repeating the analysis process for each higher-level model extraction. This design gives an enhanced degree of flexibility, enabling an easy adaptation to other programming languages.

Heuzeroth-Holl-Hogstrom-Lowe [11]: this approach presents a way to automatically detect patterns by combining both static and dynamic analysis. The former restricts the code construction and the latter the runtime behavior. This analysis does not depend on coding or naming convention. A pattern instance is defined by a tuple of program elements such as classes, methods or attributes. These elements must conform to the rules of certain design patterns. A step-by-step process filters those elements more likely to be identified as design patterns. Nevertheless, this automated process doesn't eliminate human intervention to ascertain the reliability of the results.

SPQR [23]: a Pipes & Filters based approach where the source code is progressively filtered and converted into intermediate notations until it finally reaches a state suitable for Formal Proof assertions of design patterns. This process requires an extensive formalization of the design patterns made a priori, based on its structures and relationships between its components. Its representation relies on production rules. It is relevant to point out the reasonable high-abstraction level of this approach, similar to the meta-pattern level.

DPVK [25]: reverse engineering tool to detect design patterns in Eiffel source code. This approach relies on a phased process starting on a static behavior analysis of candidate design patterns and ending on a dynamic behavior analysis. It relies on information stored in a repository where the design patterns have previously been catalogued according to those two aspects of behavior. It presents itself as a plug-in for IBM's Eclipse development environment.

All these approaches deal with the aspect of flexibility and software reuse in a very shallow way, if at all. The authors are convinced that a design recovery approach relying on flexibility and reusability aspects should bring a new insight to the matter.

5. Conclusions

One of the main goals of framework development is software reuse. In order to efficiently reuse a framework, one must be aware of its design. Most commonly, the accompanying design documentation of frameworks is poor or even inexistent, thus leading to a steep learning curve to the framework (re)user.

Uncovering the design of an existing framework is a hard and tiresome task, whereas an automated aiding tool comes in order. Several solutions exist to this

problem, yet none deals with the aspect of reusability in a clear, straightforward manner, and are unable to provide useful intermediate results.

The approach proposed focuses purely on identifying the elements responsible to provide framework reusability and flexibility, at various levels of abstraction. Aimed at obtaining the same results as other existing approaches it provides however usable intermediate results (hot-spots, template-hooks, and meta-patterns), even if the final results (design patterns) aren't sufficiently accurate.

The proposed multi-phase process supports itself on the concept of meta-patterns, a meta-level representation of an abstract design pattern that describes the relationship between the elementary template and hook methods.

The intended automated tool supporting the proposed approach should be flexible, interactive and visually deliver its results to the user. An Eclipse plug-in was decided to be the preferred way to satisfy such requirements.

It is the authors' conviction that this approach will bring new insights to the subject of reverse engineering of design patterns used in frameworks.

6. References

- [1] Hervé Albio-Amiot, Yann-Gaël Guéhéneuc, "Meta-modeling Design Patterns: application to pattern detection and code synthesis", Workshop on Adaptive Object-Models and Metamodeling Techniques, ECOOP 2001.
- [2] Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language*. Oxford University Press, 1977
- [3] Kyle Brown, "Design reverse-engineering and automated design pattern detection in Smalltalk", Master's thesis, North Carolina State University, 1996.
- [4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern Oriented Software Architecture - a System of Patterns*. John Wiley & Sons, 1996
- [5] Butler, G. "A reuse case perspective on documenting frameworks". 1997. Available online at <http://www.cs.concordia.ca/faculty/gregb>.
- [6] Campbell, R., Islam, N., Johnson, R., Kougiouris, P., and Madany, P. *Choices: Framework and refinement*. 1991
- [7] Fayad, M. E. and Schmidt, D. C. (1997). "Object-oriented application frameworks". *Communications of the ACM* 40(10), 1997, pp.32-38.
- [8] Fayad, M. E., Schmidt, D. C., and Johnson, R. E. *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1999.

- [9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (). "Design patterns: abstraction and reuse of object-oriented design." In Proceedings of the ECOOP'93 Conference, Springer-Verlag.
- [10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (). *Design Patterns - Elements of reusable object-oriented software*, Addison-Wesley, 1995
- [11] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, Welf Lowe, "Automatic Design Pattern Detection", 11th IEEE International Workshop on Program Comprehension, 2003, pp. 94-103.
- [12] Javadoc home page. <http://java.sun.com/j2se/javadoc>.
- [13] Johnson, R. E. and Foote, B. "Designing reusable classes". Journal of Object-Oriented Programming 1(2), 1988, pp.22-35.
- [14] Johnson, R. "Documenting frameworks using patterns." In Paepcke, A., editor, OOPSLA'92 Conference Proceedings, ACM Press, pp. 63-76.
- [15] R.Keller, R.Schauer, S. Robitaille and P.Page, "Pattern-based reverse engineering of design components", Proceedings of 21st Conference on Software Engineering, Los Angeles, USA, IEEE, 1999, pp. 226-235.
- [16] Christian Kramer, Lutz Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software", Working Conference on Reverse Engineering, 1996, pp. 208-215.
- [17] Lewis, T., Andert, G., Calder, P., Gamma, E., Pree, W., Rosenstein, L., Schmucker, K., Weigand, A., and Vlissides, J. M. *Object-Oriented Application Frameworks*. Manning Publications Co. / Prentice-Hall, 1995.
- [18] Hong Mei, Tao Xie, Fuqing Yang, "JBOORET: an Automated Tool to Recover OO Design and Source Models", 25th Annual International Computer Software and Applications Conference, 2001, pp. 61-76.
- [19] Pree, W. "Object-oriented versus conventional construction of user interface prototyping tools". PhD thesis, University of Linz, 1991.
- [20] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley / ACM Press, 1995.
- [21] Pree, W. "Hot-spot-driven development". In *Building Application Frameworks - Object-Oriented Foundations of Framework Design*, John Wiley & Sons, 1999. pp. 379-394.
- [22] J.Seemann and J.W. von Gudenberg, "Pattern-Based Design Recovery of Java Software", ACM SIGSOFT Software Engineering Notes, ACM Press, 1998.

Making Efficient Logging a Common Practice in Software Development

Osama M. Khaled
The American University in Cairo
okhaled@aucegypt.edu

Hoda M. Hosny
The American University in Cairo
hhosny@aucegypt.edu

Abstract

This paper proposes a new approach for making logging an efficient mechanism within the software development lifecycle from the design to the maintenance phase. Our approach looks into logging from an architectural point of view. In this respect, we introduce new roles and responsibilities for the designer and the implementer. We also try to adopt some of the known aspects of logging which will lead to more efficient and stable software systems such as logging rotation and event handling.

1. Introduction

Logging, most of the time, is a solution to write certain informative statements in an external media, like plain text files, in order to record system problems or track system behavior. The log files are useful to both technical and non-technical people because they show errors and track the system behavior. The importance of logging is most significant when there is no way to debug a live version of the application or when it is running in a distributed environment.

You can find logging solutions in many systems and applications. Operating systems, like Windows, Solaris, and Linux adopt various logging solutions. Windows has a special logging application that keeps information, warnings and errors of the system. On the other hand, Solaris and Linux adopt logging solutions like sending an email to the system administrator with the events that occur within their environments.

All well-known HTTP Servers like apache, Sun IPlanet, IBM http server, or Microsoft IIS use logging techniques. They have a standard logging format for the logs which allow other reporting applications to analyze them and generate valuable reports.

There are many logging API solutions as well. Jakarta Log4j is one of the famous examples. It has a logging framework that is easy to adopt and implement [4]. The Java Development Kit recently adopted new logging APIs into the language [3]. Java APIs came in recognition of the importance of logging in building systems.

To understand the importance of log files, imagine an application without logging and this application suffers from a certain problem. To fix the problem, it would require a thorough investigation of the application code to know the source of the problem. The investigation may require adding ad hoc logging statements, which may be left after the problem is resolved to introduce other side

effects to the application. Then, the developer has to test the application again not only against the failure point but also against all other related cases in order to ensure that the fix did not corrupt other components in the application.

On the other hand, if the application is enabled with a logger, then the source of the problem would have been most properly identified in details. This logged information provides the technical people with the guidance to investigate, if investigation is required, and may not require investigation at all.

In the next sections, we discuss how to plan for logging by introducing a new artifact which we call the *logging roadmap*. In section 3, we discuss the logs lifetime. In section 4, we express our solution in terms of logging fundamentals that are already known to most developers. In section 5, we show how to maximize the benefits of logging by discussing the common types of event handling.

2. Planning for Logging

Logging is usually delayed until the development phase and it is usually done on individual basis which do not conform to certain logging standards. This leads at the end to messy log files that contain a lot of messages from many sources. This unprofessional development practice happens mostly because: -

1. Product time-to-Market is critical and logging is not one of the main functional activities.
2. The over-trust in the final product, especially if it is well tested.
3. Unprofessional developers do not anticipate future runtime problems. This is why they do not plan for them.

Planning for logging should start from the design phase. A separate iteration should be conducted over the design after it is completed to mark the locations that will hold logging points to generate a *logging roadmap*. One of the roadmap benefits is that it emphasizes and clarifies the duties of the developers regarding logging. Moreover, it prevents individual logging techniques by providing standard logging guidelines.

The *logging roadmap*, as suggested in [5], involves the following activities: -

1. Visiting all the collaboration diagrams and depicting where the message should be printed.

2. Deciding on the message or at least the type of information that should be logged (e.g. error, message, or performance) at every location.
3. Delivering a separate artifact to indicate the locations, log level, message, and whether there is a performance logging or not.

For example, Figure 1 cites an example which depicts typical methods that are needed to log events, errors, and performance behavior. Notice that the diagram does not show how logging will occur as it is not the issue discussed in the logging roadmap.

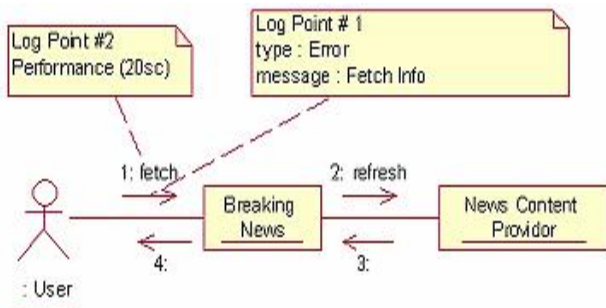


Figure 1 A Collaboration Diagram Showing the Logging Roadmap

Table 1 shows a suggested logging roadmap document that can be delivered to the development team to start working. This document can evolve to contain more roadmap points. Actually, the logging roadmap is not a waste of time. It can be added to the documentation and handed to the system administrator as part of his/her manual documents [5].

Table 1 A Logging Roadmap

Log Point	Class Method	Type	Message
1	BreakingNews.fetch()	Error	Fetch info
2	SportsNews.fetch()	Performance (20sc)	Performance problem in database
3

This standard logging technique opens the door for log analyzers to run over the logs at runtime and generate valuable reports about the system. These reports can be very important for the administrators, developers, and even the system owners. Bug fixing and enhancement can be partially fetched from such reports [5].

3. Logs Lifetime

All application activities could be written into a single log and it would still be readable. Sometimes it is very important to distinguish between the different types of logs for readability and understandability. Determining log types is one of the most important points in making logging a successful process. The application may have a

log to track activities, and other logs to track errors, and there could be a special log to monitor the application's resource utilization of memory, disk, and CPU. Such classification helps in generating reports with minimal complications.

For example, web servers can generate more than one type of logs. Apache HTTP Server has two basic log types, access and error logs. Access information can be generated in a single log file or distributed on more than one file according to need. For example, an access log can generate information using *common* format and put *referrer* and *agent* information in two separate files [1].

Rotation of logs is as important as the types. Rotating logs means that, a log file is archived according to a certain criteria like time or size. So, an application can generate a daily log file instead of putting all information in a single file. The rotation of the log may start before the next day if the log file exceeds a certain limit. This guarantees easy to open log files all the time. It is also important to have a limited number of rotated logs and backup the rest of the files. Log rotation is very important because it guarantees that the application does not exceed the operating system capabilities regarding the number of files in a single directory and the maximum file size, otherwise it will lead to unstable environments.

4. Logging Solution

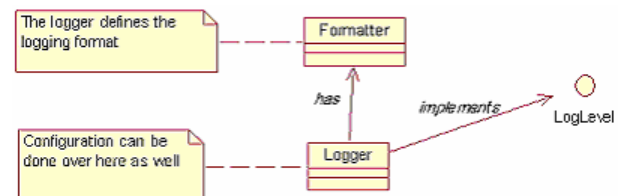


Figure 2 Logging Solution Main Structure

Figure 2 shows the main components that the logging solution is providing which are the logging formatter, logging Levels, and the logger itself. The logging formatter is used to render the message in a standard format, usually in an external media, e.g. a plain text file. The format has basic attributes by which the log line is understood [5]: -

1. *Timestamp*: the time the message is printed.
2. *Log Level*: The type of the message, e.g. (Critical, Error, Warning, Information, Debug, etc).
3. *Logging Point*: The place where the message is logged. This should have been shown in the logging roadmap.
4. *Message*: The message is most probably defined from the logging roadmap.

These attributes have to exist even if the solution will be modified or extended. The *Timestamp* shows clearly

when an event occurred during the execution process. The *Log Level* gives a high level abstraction about the type of the problem. This attribute should be configurable by the administrator according to the business needs. The *Logging Point* could be abstractly understood according to its context. For example, if the log is for tracing, then the logging point could be the method signature. However, if the log is designed to track users' activities, then the logging points may be the system functionalities. The *message* is the core of the logging event. It shows what happened at a certain moment. Further explanation may be needed.

Log Level needs more attention here for its importance in the system stability. It is the method that makes the system administrator control the amount of information logged out of the application. It is the responsibility of the system administrator to determine the kind of information that should be logged, through the log levels, according to the system stability and the business need. However, the log level tuning must not put a burden over the system itself and cause a performance drop. For example, if the system has a CRITICAL log level at one end and DEBUG log level at the other end, then the administrator may choose to start with the DEBUG level in the soft launch of an e-business application. A DEBUG level means that the application will log all the messages for this level and for the other levels. When the customer visits increase and the system may risk instability, the level can be set to CRITICAL to log only the CRITICAL messages [5].

5. Event Handling

Event handling is simply a post-event action after an event occurs. An event may range from a simple activity to a severe problem. Event handling may generally be classified into four types: -

1. Logging the information about the event to be an output of the system.
2. Capture information about the running environment when the event occurred and log it as an extra information about the event
3. Notify a certain recipient with the event in order to take a quick decision.
4. Take an automated action by changing the application environment.

The first handling type is the easiest and simplest of the ones discussed in this paper. The decision is left to the administrator or the system owner on how to utilize this piece of information. The reader may take or may not take a decision regarding this information even if there are alerts for errors.

The second type provides the log reader with supportive information in order to help him/her take the proper decision. For example, if the event was a critical problem in the application memory, then the system's

basic information would capture items such as the number of running processes, heap size information, physical memory information, swap memory information. Such information may change after the problem occurs, if not properly logged, and the log reader may get lost about the true reason for the problem.

The third type is considered a quick signal to a certain recipient about a certain event of the system. This alert may be sent using email, SMS, or whatever kind of communication means. Such an alert may be considered another type of logging. The information sent in the alert could be a combination between the first two types. A very famous example is the new electrical equipment that are connected to the Internet and that send maintenance requests to their manufacturers in case of problems.

The fourth type is about designing automated actions that the system should take in case of certain events. In other words, automatic actions to change in the system environment are already designed and fed to the system. For example, some hardware devices are designed to switch off the power at a certain temperature or switch on a cooler fan. However, software is more complex than hardware because of the inherited complexities of the its systems and the research work is still not so mature. IBM has developed a set of autonomic computing toolkits for different varieties of its products [2].

6. Conclusion

Logging is considered to be an infrastructure practice to any system although it may increase the development time. Small systems may survive without logging, but they will not grow well into bigger systems without logs. Complex systems and applications usually have logs to monitor resource utilization and system behavior. However, If a product is overwhelmed with logging points, its performance will suffer greatly. So, logging must be classified into levels which would allow the application administrator to change them at runtime.

The choice is for the developer to either log or not log. As mentioned earlier if he/she chooses to log then he/she must plan for logging from the design phase if not earlier in the software development lifecycle. This will inevitably result in that the system runtime behavior will be very clear in the end. Moreover, by composing a logging roadmap, developers will be obliged to use common messages and by using the same logging format logs are no longer messy. On the other hand, if the developer chooses not to log from the beginning, system failures which will be encountered when the system goes in operation, will always be difficult to explain.

7. Acknowledgment

We would like to acknowledge Ahmed Mohamed Ismaiel from Vodafone Egypt for reviewing the paper.

7. References

- [1] *Apache HTTP Server Log Files*.
<http://httpd.apache.org/docs/logs.html>
- [2] Autonomic: <http://www.alphaworks.ibm.com/autonomic>
- [3] Java™ 2 Platform Std. Ed. v1.4.2.
<http://java.sun.com/j2se/1.4.2/docs/api/index.html>
- [4] Log4j Project.
<http://logging.apache.org/log4j/docs/index.html>
- [5] Osama Mabrouk Khaled (2004). *Capturing Design Patterns for Performance Issues in Database-Driven Web Applications*. M.Sc. Thesis, Computer Science Department, the American University in Cairo.

A Novel Approach for Managing and Reusing Business Rules in Business Architectures

Haitham S. Hamza
Computer Science & Engineering Dept
University of Nebraska-Lincoln
Lincoln, NE 68588-0115, USA
hhamza@cse.unl.edu

Mohamed E. Fayad
Computer Engineering Dept.
San José State University
San José, CA 95192-0180, USA
m.fayad@sjsu.edu

Abstract

Business architectures are models that describe the business entities, their relationships, their dynamics, and the rules that govern their interaction. These rules are usually called business rules. A major challenge in developing effective business architectures is to define and implement business rules effectively so that the architecture can be adapted whenever new requirements evolve. Current approaches that address business rules do not provide a balance between effective management and reuse of the business rules within the architecture. In this paper, we discuss current solution to manage business rules, and then we propose a new approach for developing business architectures and their business rules. The proposed approach allows for both managing and reusing business rules. The approach is demonstrated through the means of a case study.

1. Introduction

Business architecture, in its simplest form, can be viewed as a set of resources that interact under defined rules through a set of well defined processes to achieve certain goal(s). These rules are known as *business rules*. This simple view of business architecture does not indeed clarify what business rules are. In fact, there exists no formal or standard definition for business rules; nonetheless, several definitions have evolved over the last decade. In the following we give some of such definitions. A business rule can be defined as:

- Units of business knowledge [6].
- A statement that defines or constrains certain aspects of a business [7].
- Declarations of policies or conditions that must be satisfied [8].
- A statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business [9].

In this paper, we use the last definition of business rules.

The rapid changes in requirements and market demands have made the *flexibility* of the business to tune and evolve to meet these requirements a crucial factor to the success of any business. In fact, today, a successful business is characterized by a high-quality product and a fast time-to-market response.

When business architectures change, their structural as well as logical elements may change as well. Among the changeable aspects in the architecture are its business rules. When objects of the business architecture are changed, updated, or removed, the related business rules should be changed accordingly to avoid inconsistencies in the business logic. Inconsistent business rules may lead to disasters. Nonetheless, there is a huge limitation on the time in which developers can identify and update business rules to evolve the architecture. Worst, after few updates, architecture development documents are not effectively updated; a fact that leads to a considerable mismatch between what is documented and what is actually running inside the business. In addition, business modeling techniques are not mature enough to capture the dynamics and details of business rules interaction. All the above coupled with the increasing complexity and cost of business architectures, all have made understanding and managing business rules an important yet a very challenging research topic.

While effective management for business rules and business architectures seems to alleviate part of the problem, it might not be sufficient by itself. Therefore, we foresee that business rule reuse will be an interesting topic to investigate in the future, and hence, we do address this issue in this paper.

Several researches are focusing on improving the practice of dealing with business rules. These efforts range from developing efficient techniques to identifying business rules from the code, while others are focusing on improving the business architecture modeling techniques and theories to ensure the development of business rules that are easy to change and are well defined.

In this paper, we propose our perspective in the problem of managing and reuse business rules and develop an approach based on software stability concepts [5] that can enable a systematic and an automated management and reuse of business rules.

2. Related Work

The problem of extracting business rules was addressed in the literature [1-4]. For example, in [1], a framework for extracting business rules from large legacy systems is proposed. The proposed framework consists of five main steps: slicing program, identifying domain variables, data analysis, presenting business rules, and business validation. In [2], another solution to the business rule extraction problem is proposed that combines variable classifications, program slicing, and hierarchical abstraction. In [3], data output identification and program stripping techniques are used to identify and extract business rules.

In [11], the author proposes the separation of business process aspects from essential business rules. Such separation of concerns leads to natural hierarchal business architecture that separates domain modeling concepts from the business process modeling concepts.

In [12], the authors argued that a set of software metrics need to be developed in order to characterize and understand the impact of business rule evolution and its impact on software systems.

3. Business Architecture Evolution Scenarios and Consequences

When a business is updated or changed, several scenarios might take place. We classify these scenarios into three broad categories:

- **Change Structure Only.** It is possible that we update the business architecture with new or modified objects that do not require a change in business rules. For example, if we have a Payment object in the business model, business rules might remain the same even if we update the payment method. For instance, a business rule such as: *a late fee should be imposed after the payment due date*, may not be affected by the payment method itself. Such changing scenario requires flexibility in the architectural update, that is, the business architecture should be able to accommodate the change without affecting other objects in the architecture.
- **Change Business Rules Only.** In such scenario we do not want to change any architectural aspects; however, we want to update, extend, or remove some rules from the model. This is a critical change scenario and the development approach should

provide tools and techniques by which rules can be identified and changed.

- **Change Business Rules and Objects.** This scenario represents the extreme changing Scenario in the business architecture. The requirements that are imposed by this scenario are similar to those discussed above; but in addition, a new requirement that we call *rule coupling* should be addressed. In rule coupling we need to identify what are the current objects and rules in the system will be affected by the new added rules.

Our objective is to develop a systematic approach for managing business rule evolution and enable their reuse. A successful business architecture development approach should effectively deal with the three above evolution Scenarios.

We identified a set of requirements for an effective business architecture development approach. Besides the effective management of business rules, our approach should allow for business rules reuse. The reuse space of a business rule has two dimensions. The first dimension is the ability of reusing business rules within the same architecture when new requirements evolve. The second dimension is to reuse a business rule in developing similar or related architectures, in other words, the organization can develop a *business rule catalog* or *library* that can be used in future developments. Both reuse dimensions are complex and hard to achieve. However, we argue that to accomplish such broad reuse, business rules should be abstracted and generalized *semantically*.

4. The Proposed Approach

Extracting business rules from the code is error-prone and could involve a considerable risk of misidentification and hidden dependency between business rules that can be very hard to detect in the code level. Therefore, we believe that business rules should be extracted, identified, and validated at a higher-level phase than codes, and later on, code can be used to validate and ensure correctness in the run time. Addressing business rules in a higher level than the code has also the benefit of facilitating the communication between developers and business decision-maker who may not necessarily aware of the technical aspects of the code.

On the other hand, separating business rules from business processes [10] may not be effective as new intermediate layers to bridge the two layers need to be introduced.

We argue that any business architecture has certain entities (objects, processes, business rules, etc.) that can be classified as stable, partially stable, or unstable. By stability we mean the frequency by which the component is likely to change in the context of the business.

This argument motivates the use of techniques that can differentiate between these three stability levels. We adopt software stability approach [5] to accomplish this goal.

4.1. Software Stability Concepts

Software stability approach is a layered approach for developing software. In this approach, the classes of the system are classified into three layers: the *Enduring Business Themes* (EBTs) layer, the *Business Objects* (BOs) layer, and the *Industrial Objects* (IOs) layer. Based on their nature, classes are classified into one of these three layers. EBTs are the classes that present the enduring aspects of the underlying business. Therefore, they are extremely stable and form the nucleus of the stable model. BOs map the EBTs of the system into more concrete objects. BOs are internally stable and can be externally adapted through hooks. IOs map the BOs of the system into physical objects. Figure 1 depicts the concepts of software stability and the relationship among them.

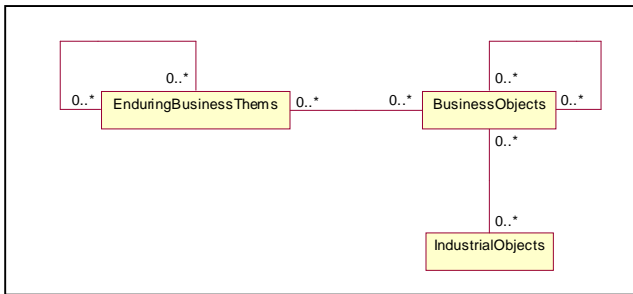


Figure 1. The relationship between software stability concepts

4.2. Business Architectural Views

Business architectures are complex and involve many issues that cannot be represented in a single view. Therefore, usually different architecture aspects are represented with different views, in which every view emphasizes certain aspects of the business.

In that sense, we can use a functional view, logical view, deployment view, conceptual view, etc. While our approach supports different views to illustrate different aspects in the business, we focus in this paper on the structural view of the business architecture- a view that depicts the relationship between the architecture components² and the rules that governs the interaction of these components.

² We use the word component in a rather loosely way in this paper. By component we mean any entity that constitutes the business architecture, e.g. classes, objects, patterns, etc.

4.3. Rules Categories

Business rules can be divided into three different kinds: *Derivations*, *Constraints*, and *Existence* [10]. Derivations business rules define the why by which some information is derived from other information. Constraints rules impose certain constraints on the structure or behavior of different elements in the business. Existence rules define the rules under which an element in the business should be created or destroyed. We define new categories of business rules: Enduring business rules, encapsulated business rules, propagating business rules, and unstable business rules.

In our approach, while using the above categorizes is still valid, we define an orthogonal set of rules categorizes to match the different object types that are used in software stability approach. A rule in our approach is a generic term that refers to a formal or informal statement that imposes certain constrain(s) on the relationship between two processes, or within a single process. Based on this definition, we differentiate between four different types of rules:

- *Enduring Rules:* An enduring rule is a rule that governs the interaction between EBTs.
- *Process Rules:* These rules define the constraints that govern the internal interaction of the process.
- *Business Rules:* are rules that govern the interaction between two business objects or processes.
- *Propagating Rules:* constraints the interaction between objects or processes across different layers.
- *Industrial Rules:* constrains the relationships between IOs.

Figure 2 summarizes the different rule categorizes. It is worth to note that these categorizes are applicable to different views of the business architecture. For instance, in the *process view*, process rules are more pronounced than in the *structural view*. In the *structural view*, on the other hand, all rules are represented; however, some rules are not presented in details such as the process rules. In fact, we aim at providing the overall picture of the rules of the architecture, while try to present the details of the relevant rules on the relevant architecture view. We feel that separating the views completely, like what is used in [10], may lead to confusion as the overall architecture picture is not maintained in any single view even.

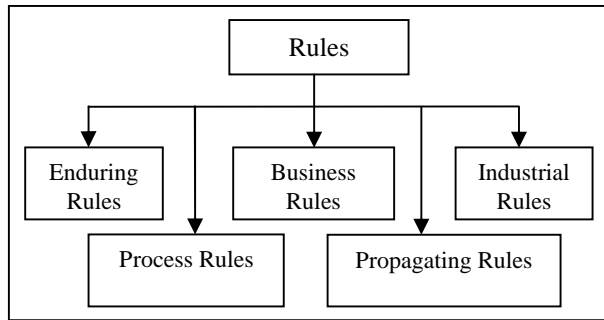


Figure 2. Different rules categories in the proposed approach

4.4. Rule-Dependency (RD) Diagram

We introduce the concept of *Rule-Dependency (RD) diagram* as a tool in our approach to facilitate the study and identification of rules dependency in a given business architecture.

RD diagram depicts the semantic of rule dependency between objects in the business architecture. Figure 3 shows an example of a generic RD diagram. In this Figure, rules are represented by circles while objects are represented by squares. Every rule points to the objects that it governs or constraints. Note that this diagram does not show the relationship between the objects or how the rule constraints their relationships. RD diagram tries just to maintain the knowledge of what rule affects which objects. Rules can be connected with each other if they influence the same or related objects. For example, the Figure below indicates that the rule R_i impacts the rule R_j , but not the other way around. On the other hand, R_j and R_k both affects each other and hence each rule points to the other one.

For large system, DR diagram will be more complex; however, we believe that manipulating RD diagrams can be automated, and hence, time should not be a major concern.

5. A Case Study

In order to illustrate the concepts in our approach, we use a simple case study of a renting business. Figure 5 shows the EBTs and BOs in the business of renting. The model has two EBTs: Renting and Negotiation. Objects AnyEntry, AnyAccount, Receipt, AnyParty, AnyLog, AnyAgreement, and LineItem are each externally stable and internally adaptable; they are the system's BOs. As shown in the Figure, few rules of different types are defined. Note that the process rule associated with the AnyAgreement BO is not shown in details as the details of this rule will appear in the process view of the

architecture; however, we still point them out in the given structural view for completeness.

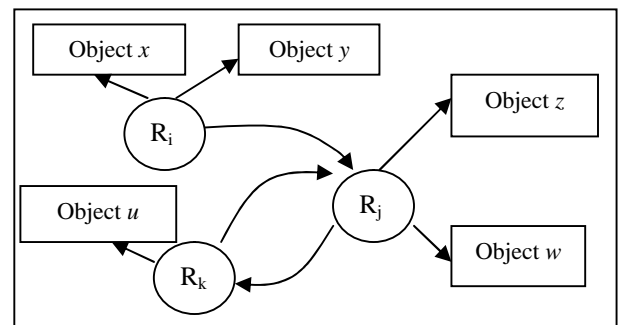


Figure 3. An example of the RD diagram.

Figure 6 shows the object diagram of the *car renting* architecture. Each BO is mapped to a real life object using the appropriate IO. For instance, the BO “LineItem” is physically represented with the IO “Car” in this system. The BO “Receipt” is realized in the model by the IO “Payment”, and this payment can take any form as cash, credit card, etc. Changing the payment option should not affect the core aspect of the business itself. The concept of Receipt exists whether you pay with credit card or cash. However, the Payment procedure for both paying options differs, in the credit card payment an ID card may be required which is not the case when the cash option is used. Figure 4 gives a part of the RD diagram for the renting business architecture.

6. Conclusions and Future Work

In this paper, we discuss the current approaches in managing the evolution of business rules within the business architecture. We argue that current approaches may not support effective evolution of business rules when the business encounters changes in requirements. We propose an approach that views the business architecture as a hierarchical structure, and we define a set of new rules that can be used in this hierarchical structure. In addition, we propose the concept of rule-dependency diagram that can be used to automate to facilitate the management and the reuse of rules. Future work includes the use of OCL and the development of architectural specification languages to describe the business architectures and their rules. Also, the formalization of the RD diagram should be investigated as an essential step to validate and automate the process of rule management and reuse.

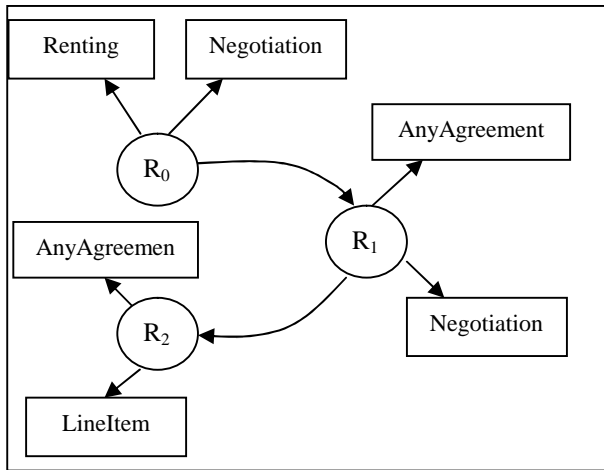


Figure 4. Part of the RD diagram for the renting business architecture example.

7. References

- [1] X. Wang, J. Sun, X. Yang, Z. He, and S. Maddineni, "Business rules extraction from large legacy systems," In Proc. of IEEE 8th European conference on Software Maintenance and Reengineering, 2004.
- [2] H. Huang, W.T. Tsai, S. Bhattacharya, X.P. Chen, Y. Wang and J. Sun, "Business rule extraction from legacy code," In Proc. of IEEE 20th Computer Software and Applications Conference (COMPSAC 96), 1996, pp. 162-167.
- [3] H.M. Sneed, K. Erdos, "Extracting business rules from source code," in Proc. of IEEE 4th International Workshop on Program Comperension (IWPC 96), 1996, pp. 240-247.
- [4] J. Shao, S.M. Embury, G. Fu, X. Liu, and W.A. Gray, "Unlocking business rules for maintaining information systems," In Proc. of the 1st International Workshop on Database Maintenance and Re-engineering (DBMR 02), 2002, pp. 17-30.
- [5] M. E. Fayad and A. Altman, "Introduction to Software Stability", Communications of the ACM, Vol. 44, No. 9, September 2001.
- [6] J. Odell, "Advanced object-oriented analysis and design using UML," New York: SIGs Books, 1998.
- [7] D. Hay and K. Healy, "Defining business rules-what are they really, GUIDE Business Rule Report. Available on line at: <http://www.businessrulesgroup.org/2000>.
- [8] OMG, "Analysis and design reference model," Framingham, MA: OMG, 1992.
- [9] T. Morgan, "Business rules and information systems," Boston: Addison-Wesley Publishing, 2002.
- [10] H-E. Eriksson and M. Penker, "Business modeling with UML: business patterns at work," John Wiley & Sons, Inc. 2000.
- [11] M. Snoeck, "Separating business process aspects from business object behavior," In "New directions in software engineering," by L. Amicorum, M. Verhelst, J. Vandernbulcke (Edt.), and M. Snoeck (Edt.).
- [12] L. Lin, S. Embury, and B. Warboys, "Business rule evolution and measures of business rule evolution," In Proc. of the 6th International workshop on Principles of Software Evolution (IWPSE 03), 2003.
- [13] H.S. Hamza and M.E. Fayad, "An architectural pattern for developing renting systems," In Proc. of 3rd Latin American Conference on Pattern Languages of Programming (SugarLoafPLOP04), 2004.

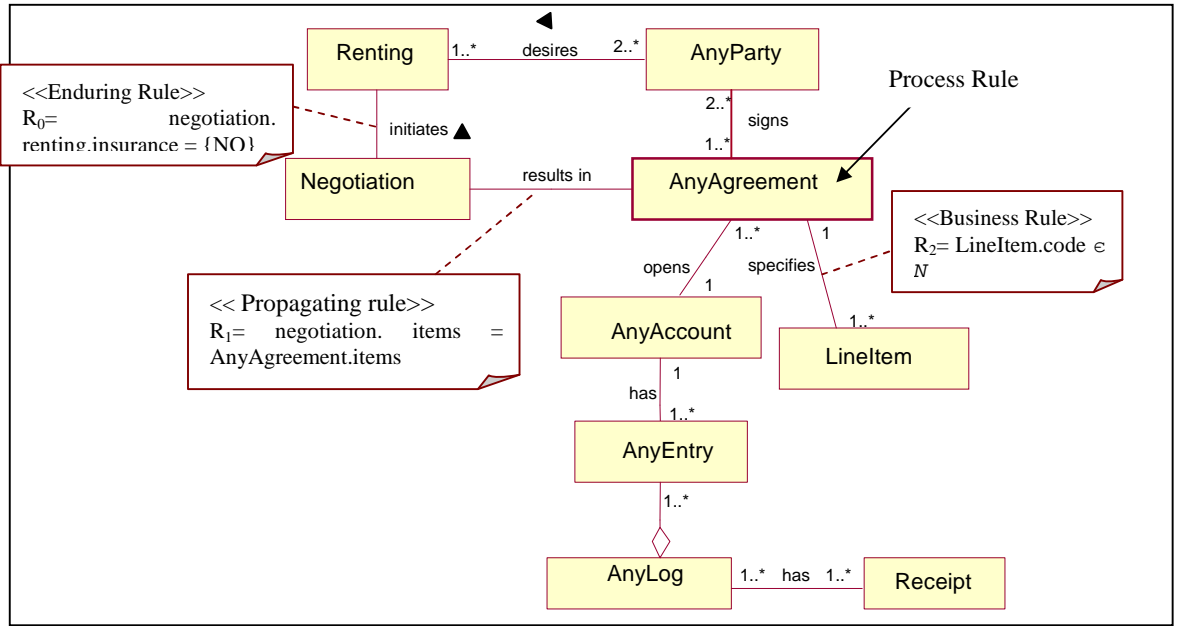


Figure 5. Renting business architecture annotated with different types of rules.

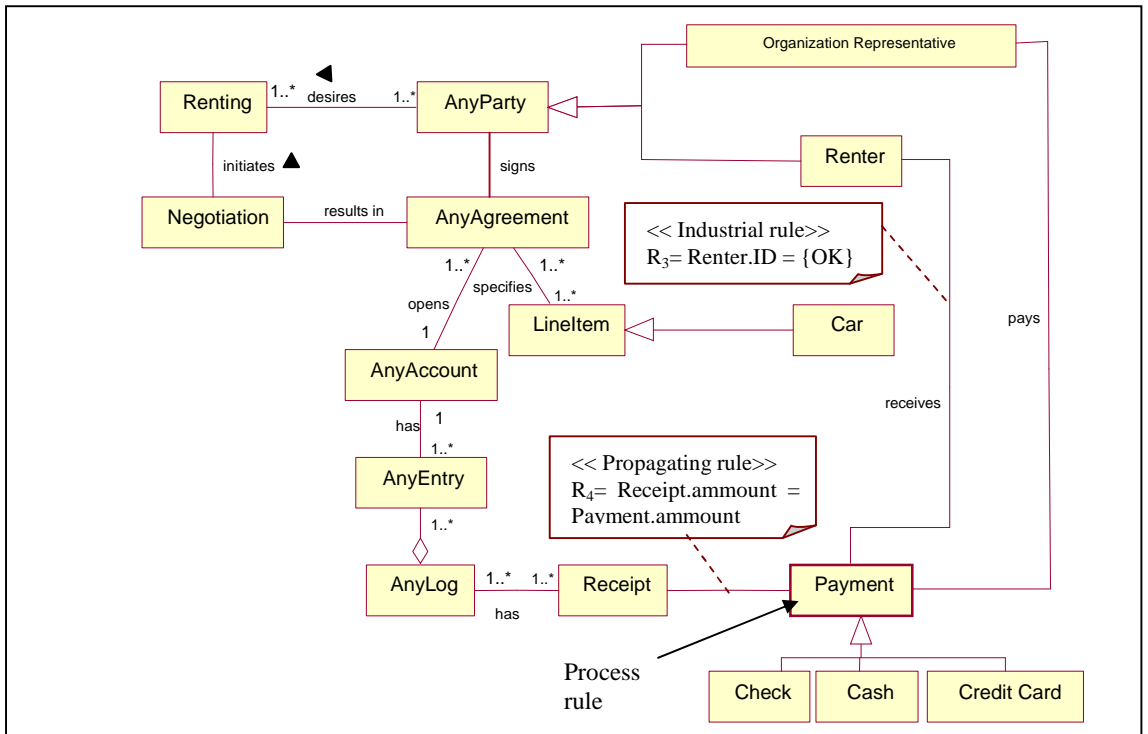


Figure 6. Car renting business architecture annotated with different types of rules.

Architectural Modelling to Understand System Evolution

Galal H. Galal-Edeen

Faculty of Computers and Informatics, Cairo University, Orman, Giza 12613 - Egypt

Email: Galal@cu.edu.eg

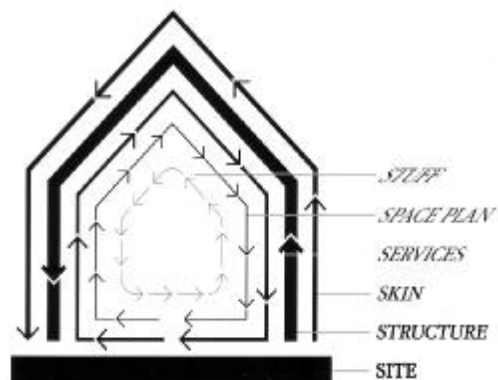
Abstract

This report outlines the research carried out with the support of a grant by the EPSRC (Engineering and Physical Sciences Research Council) of England, United Kingdom, under the Systems Integration initiative within EPSRC. We have analysed a problem domain using a qualitative research methodology to model its main architectural features. Our model formed an input to an associated EPSRC grant under the same initiative. Links between the results of systematic qualitative analysis and architectural software analysis have been shown, demonstrating the feasibility of further research into the idea of architectonic isomorphisms between a problem domain and the adaptability of associated software artefacts. We point out the technical developments resulting from this investigation, and point a way forward.

Background/ Context

The issues for devising software and systems architecture that are adaptable in the long term has come to the fore of the interests of the software and information systems communities [11]. A main issue is how to bring an understanding of some problem domain to bear on the technical architecture in a way that fosters the emergence of desired non-functional properties such as adaptability and robustness. This grant together with the associated grant GR/R11919/01 (at the University of Portsmouth) formed an integrated programme of research that sought to explore the relationships between a software artefact and the original problem domain³ that required it in the first place. Our research collaborators in UoP concentrated on the analysis and modelling of number of extant software artefacts, and we concentrated on the exploration, modelling and analysis of one of the

relevant host domains. Our working hypothesis was that a relationship, at the architectural level⁴, could eventually be discerned between the problem domain and its related software artefact. The nature of the architectural link is what we previously used the concept of *Architectonic isomorphism* to characterise [1]. The use of the term *architectonic* to describe a certain architectural view of software systems [6, 8, 10, 16] is based on a certain use in architecture [4] which refers to the differentiation of constructional elements according to their relative stability. This view is also borne out by the concept of shearing layers [2], which relates the adaptability of a building to the degree to which *layers* of its constructional elements are de-coupled in a way that accommodates, in relative terms, new or unforeseen requirements. The reasoning being that constructional layers can *slip past* each other: changes to one layer do not necessitate changes to others. Note here that the low coupling is not at the level of individual constructional elements (such as bricks for instance), rather, the de-coupling referred to is at the level of categories of such elements. The constructional elements are categorised according to the degree of susceptibility to, or speed of, change that they share. Figure 1 below illustrates this view.



³ We use the term “Domain” in the sense that features in Jackson’s usage (Jackson 2001), to refer to the specific application domain, rather than a generic activity domain such as air traffic control.

⁴ We strongly demarcate the architectural level from the structural one. See (Galal 1998) for an elaboration.

Figure 1. Shearing Layers of Change (Courtesy of Phoenix Illustrated)

Since this view is fundamentally normative, i.e. it is based on a study of the types of changes that typically affect buildings, we must ground any such analysis in the cultures that give rise to any specific architectonic profile. The fundamental question of this research is whether revealing the relative embeddedness of domain concepts can help in structuring software (or indeed another technical artefact) in a way that makes it more adaptable and in tune with the dynamics of the change profiles of the original problem domain.

Investigating architectonic isomorphisms

2.1 Applying GSEM to modelling the problem domain

We applied our qualitative analysis-based framework, the Grounded Systems Engineering Methodology (GSEM)[8, 7] to try to identify the architectonic profile of the problem domain for which certain software artefacts have been developed. One of the fundamental ingredients of GSEM is the rigorous qualitative analysis of a problem domain, typically using the Grounded Theory method from the social sciences [18]. We have built GSEM around procedures to operationalise Grounded Theory to describe a problem domain, define its core requirements and arrive at a system architectural design [9]. Figure 2 below describes the core procedure upon which GSEM domain analysis and modelling was based:

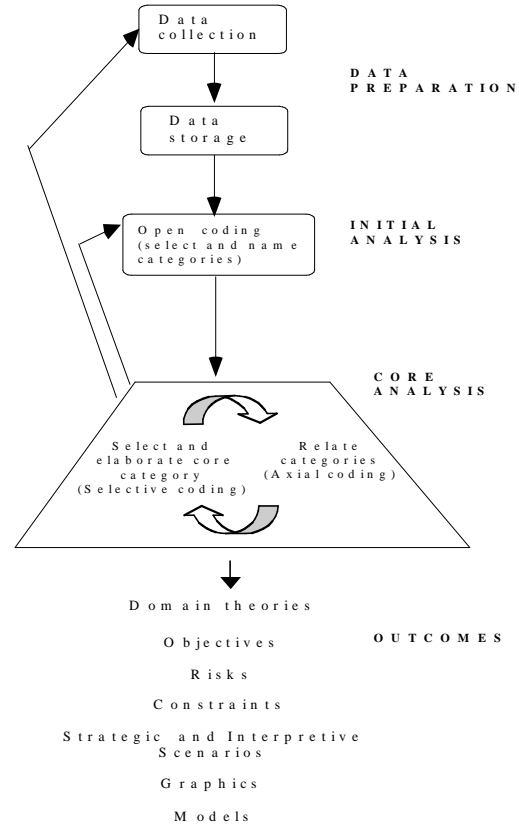


Figure 2. Core steps in Grounded Theory analysis (adapted from [17]).

GSEM increases the fidelity of domain analysis and structuring activity by bringing the subjectivity of the analyst under greater discipline, through applying the principle of giving rigorous consideration to the data, all the data and nothing but the data. The selection process that is a fundamental part of any analysis activity is brought into the open and under improved visibility and control. The process of exploring, understanding and modeling a problem domain can be a lengthy process and so we chose just one example, SUEZ, to investigate in detail. SUEZ was written for Ship Analytics to model sea fluid cargo dynamics and is the largest and most complex project of the four Fortran programs (which were analysed by our research collaborators in UoP). It is also the newest of the ones provided, written around 1997.

2.2 A GSEM Analysis of the domain.

In the course of investigating the Ship Analytics domain, we interviewed a number of key individuals (domain experts, software architects, chief programmers and technical managers) who knew about the domain, its environment and history. Table 1 gives a summary of the main persons interviewed.

Name	Function	Organisation
Edgar Ernstbrunner	Main programmer	Ship Analytics
Alan Witcher	Lecturer (Petrochemical Division)	Warsash
Mike Barnet	Lecturer (Petrochemical Division)	Warsash
Ray Gillett	Program Manager	Ship Analytics
George Angus	Ex-Warsash Director	Warsash
Mike Turner	Ex-Ferranti project manager	Ferranti
Jack Ponton	Prof. Of Chemical Engineering	Edinburgh

Table 1. Roles, function and affiliation of persons interviewed

After approximately seven iterations of the analysis, supported by the Atlas.ti qualitative analysis software, we reached the model of the domain illustrated in Figure 3.

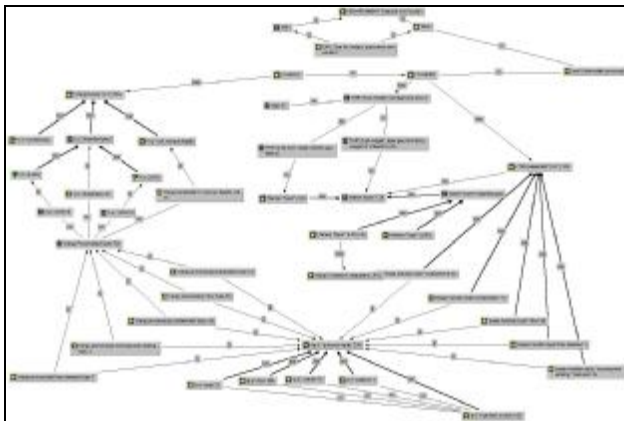


Figure 3 (A8 Overview): A qualitative model of the ship simulation domain.

To render the model in Figure 3 above comparable to the ones our colleagues in Portsmouth University have produced for the software, it was necessary to normalise it. One of the key advances of this project is the way in which the qualitative model has had to be normalised to render it comparable to the outcome of the keystone analyses carried out at Portsmouth on the corresponding software.

2.3 Matching architectonic⁵ levels

Our initial analysis is simply numeric, based on counts of the concept occurrences and similar ones in the software artefact. The early stages of the analysis revealed the following domain concepts in order of relative weight or dominance:

- Tanker
- Cargo
- Plan
- Compound
- Risk

The frequency order of the two top concepts (Tanker and Cargo) occupied high relative positions that corresponded to the relative size of the keystone groups revealed by our colleagues in UoP. Further analysis of both the domain (this time focusing on the density of structural associations) and the related software confirmed this observation as explained below.

We provided our research collaborators at UoP (Grant GR/R11919/01), with a detailed analysis of one of the projects' problem domains. This analysis was provided in the form of a diagram with various relationships between the conceptual units revealed, along with numerical counts expressing the density of inter-associations between concepts. Our research collaborators normalised and reduced the model into a form that was more comparable with the results of their analysis. The result of this normalisation process is shown in Figure 4.

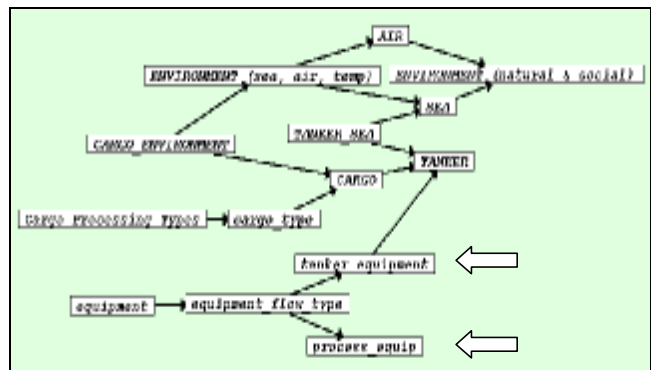


Figure 4. Normalised graphs of problem domain concepts and relationships.

The architectonic structure of the associated software artefact extracted by the analysis tool is shown in Figure 5, which shows four major keystone groups at the root context of the artefact. Relating these groups to the structure as given in Figure 4 (resulting from the problem domain analysis) reveals immediately a match

⁵ Architectonics is used to indicate the differentiation into levels of flexibility or likelihood of change. See (Galal & Paul 1999).

between the two largest (or heaviest) groups with specific concepts in Figure 5 as indicated by the wide arrows in both figures. The other groups in the root context of the context tree are also related to the artefact domain, but with a lower degree of substantiality to the domain.

The significance of the isomorphic match can best be explained in terms of the propagation of behavioural change. This can only travel up an arrow, towards the dependent and never down and as a result of this, knowing the conceptual match between both the problem domain and artefact domain indicates which changes in the problem domain could be mirrored to the artefact domain. This is a significant result from the point of view of the designer who would need to determine the base architectural layer for the software artefact in a way that fosters greater adaptability.

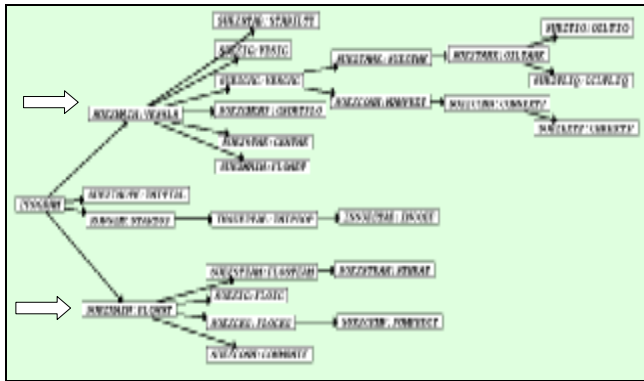


Figure 5. Pruned context tree of related software artifact.

However, the exact nature of these interface-links (wide arrows in Figures 4&5), their mechanisms and what kind of influence they have on a given variety of desired changes are still unknown. This, we believe, needs analyses of various snapshots for the same software artefact through a few adaptation cycles. It is unlikely there will be just a single mechanism that applies to all cases.

Since the isomorphic match is only between few (but relatively more significant) concepts, the original idea of a full match between the program structure and problem domain was evolved into the idea of an *interface* between the two. This interface has the unique property that it resembles parts of the architecture of both the artefact- and problem-domain. At this interface level, we expect a good program to have the same architectonic levels as the problem domain and vice versa. Indications for this can be seen in Figure 4 and Figure 5 where the concepts of the isomorphic groups in the artefact match

(i.e., are semantically close to) concepts in the problem domain.

2.4 Applying configurational space analysis techniques to conceptual models.

To get to the architectonic profile of the domain, we conducted further analyses in which we used methods and tools from the set of spatial analysis techniques known as Space Syntax. Space Syntax refers to a collection of inter-related representational tools and analytic techniques, supported by software tools and by a philosophy as to how spatial configurations operate through influencing societies and vice versa [12, 13, 14]. Space syntax has been successfully used to explain the emergence of a number of pedestrian and land use phenomena in the built environment, such as footfall, crime and other anti-social behaviour as well as the location of retail activity. The philosophical foundation of Space Syntax is that of Structuralism [3] and its fundamental principle that it is the relationships between elements of the language that constitute the meaning of the signs that it contains. We consider that the same principle applies when reasoning about problem domains too, thus we can base domain modelling on similar structural principles. Reflecting the resulting structural models into the software or other technical artefact has always been one of the aims of software and systems engineering activities. We add to this that adaptable artefacts need to reflect domain structures that address adaptability (to us this means likelihood of change) as a core value. However, to us the nature of language in general, and by implication languages to describe specific domains in particular, also embodies architectonic principles. This is the sense that there are concepts that are core to a domain in that they have a high impact on rendering it cognisable. Following Frampton's [4] use of the term *stereotomic*, we say that such core concepts play a stereotomic (or heavy, or solid) role in constituting a given domain. Our analyses show that both the problem domain architecture and artefact architecture will have concepts such as Tanker (or Vessel) and Cargo as core, both are stereotomic concepts towards which the model will tend to gravitate.

We used configurational analysis to get to the conceptual elements that more fundamental to understanding the domain: these are concepts that are simply the most referenced by other concepts, and thus they provide a higher relative privilege in understanding the domain in question. The results of configuration analysis can be depicted as a series of coloured lines (Figures 6&7). Each line represents a concept uncovered through the qualitative analysis conducted. The colours represent the degree of integration that the line has within the constellation of concepts connected to it. The

closer the colour is to red, the more integrated it is; red being the most integrated, relative to other lines. Figure 6 shows the results of carrying out configuration analysis on the normalised conceptual diagram of the problem domain. The corresponding integration values are given in Table 2. Figure 7 shows the results of carrying out configuration analysis on the software artefact related to the same domain, whilst the corresponding integration values are shown in Table 3. In both figures (6&7) the core concepts of the domain, these that closer to the colour red (or the darkest shade on a grey scale print), have a strong integrative influence on the remaining concepts within the same analysis. The closeness of the various colours to red is borne out by the values in tables 2 & 3. Highly integrative concepts are those that are more fundamental to understanding the domain; in other words these are more critical to understanding (or interpreting) the constellation of the remaining concepts in the qualitative domain model that we offer. This is a novel application of the concept of integration from urban morphology [12] to domain and software analyses.

In both figures, it can be seen that the most integrated lines⁶ have remained more or less the same (we ignore the vertical red line (no. 1) in Figure 7 since it simply stands for the root module for the whole diagram, a purely synthetic and artefact-bound concept). The line no. 1, representing the “Program” node is actually *not* the most integrating one, but the line representing the “Vessel” concept is. Table 3 gives the global integration measures for the context tree diagram of the software. Line identification numbers are printed next to each line to facilitate linking them to the Line ID columns in the associated tables. Dark arrows point to the “heaviest” concepts.

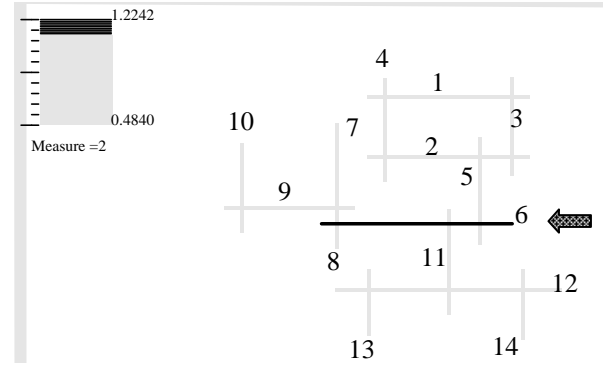


Figure 6. Integration (Rad n) map of the normalised domain model⁷

Line ID	Order	Concept name	Integ Rad=n
6	1	TANKER	1.2241853
8	2	CARGO	1.0953236
5	3	TANKER_SEA	0.9910071
7	4	CARGO_ENVIRONMENT	0.9048326
11	5	tanker_equipment	0.9048326
2	6	SEA	0.832446
4	7	ENVIRONMENT (sea, air, temp)	0.7707833
9	8	cargo_type	0.7176259
12	9	e.f.t. [equipment flow type]	0.6713274
3	10	ENVIRONMENT (natural & social)	0.5946043
1	11	AIR	0.5624635
10	12	Cargo Processing Types	0.507589
13	13	equipment	0.4839802
14	14	process equip	0.4839802

Table 2. Global Integration (Rad n) values for the problem domain model

⁶ In urban morphology, the most integrated lines are the most key lines to navigating and understanding the urban grid.

⁷ Rad n abbreviates Radius n, or Global Integration, in which the integration of a line is calculated with respect to all other lines in the system by travelling to all other lines in every possible direction. Integration can be also calculated for various radii. For example Radius 3 is calculated by travelling up to only three lines away from each line in every possible direction.

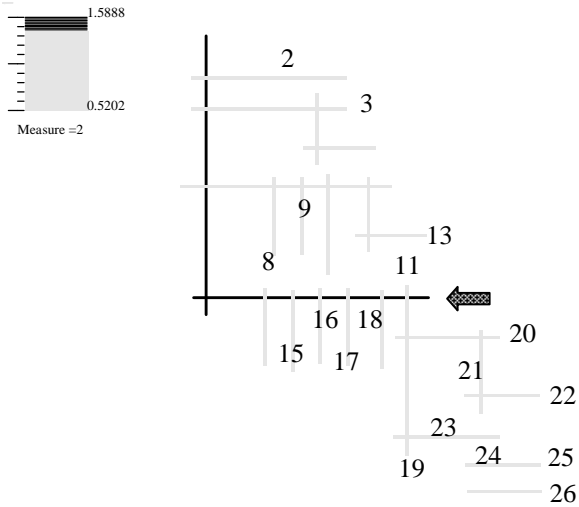


Figure 7. Global Integration (Rad n) map of the context tree diagram of the SUEZ software

Line ID	Order	Concept name	Integ Rad=n
5	1	SUEZMAIN:VESSL	1.5887512
1	2	PROGRAM	1.5072768
19	3	SUEZCRG:VESCRG	1.250719
4	4	SUEZMAIN:FLOWST	1.1526234
3	5	RUNSIM:STARTUP	0.9963355
14	6	SUEZSTAB:STABILITY	0.9636687
15	7	SUEZIG:VESIG	0.9636687
16	8	SUEZGMONF:GSDETFLO	0.9636687
17	9	SUEZMAIN:FLONET	0.9636687
18	10	SUEZSTNK:GENTNK	0.9636687
2	11	SUEZINCTK:INITIAL	0.9330761
23	12	SUEZTANK:SUEZTNK	0.9043661
20	13	SUEZCONN:MANPEXT	0.8773701
10	14	SUEZSTEAM:FLOSTEAM	0.8052574
12	15	SUEZCRG:FLOCRG	0.8052574
8	16	SUEZIG:FLOIG	0.7837839
9	17	SUEZCONN:CONMANIF	0.7837839
6	18	INSUEZTAB:INIPROP	0.7257259
24	19	SUEZTANK:OILTANK	0.691574
21	20	SUEZCONN:CONNEXTP	0.6604921
11	21	SUEZSTEAM:STHEAT	0.6060184
13	22	SUEZCPMP:PUMPEDCT	0.6060184
7	23	INSUEZTAB:INCOEF	0.5598457
25	24	SUEZTLIQ:OILTLIQ	0.5393009
26	25	SUEZTIO:OILTIO	0.5393009
22	26	SUEZEXTP:CHKEXTP	0.5202106

Table 3. Global Integration (Rad n) values for the software artifact context tree.

The core elements of the domain have evidently been propagated into the technical artefact, and more importantly, remained recognisable after the software has undergone various adaptations, and with a relative

weight that was mirrored in the software artefact. But we note that the said artefact has been subject to adaptation and evolution for some time. In a sense we have been looking into the structure that resulted from the artefact’s adaptation to accommodate evolving requirements. This work reveals the feasibility of the qualitative/ structural analysis of the domain with the aim of relating it to a corresponding technical artefact.

Difficulties encountered

The original plan allocates three months to the work involving using GSEM to analyse the domain. The work involved considerably longer due to the following factors:

- a) A considerable period of training and orientation for the research fellow appointed to carry out the qualitative domain analysis, both in architectural thinking in general and the particulars of Grounded Theory analysis on which GSEM is based, was needed. This consumed a substantial amount of time and leading to the interview transcription and analysis period taking eight months instead of the originally anticipated three. On reflection, three months would have probably been too short anyway and six months would have been a more realistic estimate to cater for the difficulty of actually carrying out interviews and pre-processing their content.
- b) Although the collaborators were willing to provide access, actually setting up access for the necessary interviews took a considerable time (about three additional months) to discuss the nature of data needed and to sign the necessary confidentiality agreements in a form that satisfied the organisation that owned the software and employed the domain experts. Interviews with the domain experts started only after these preparations were complete.

Research impact

This research project was for a one-year feasibility study into the extent to which the concepts of architectonics and architectonic matches apply to problem domains and a corresponding software artefact; investigating the existence of a link between what we termed “domain architectonics” and “software architectonics”. This grant, along with the associated grant GR/R11919/01, held by our research collaborators at the University of Portsmouth developed research tools and concepts that can be applied to revealing the core architectonic isomorphisms between artefacts and corresponding domains. The two grants, taken together, have revealed

the feasibility of investigating the concept of architectonic correspondences between domains and artefacts. A set of appropriate analytic tools have also been developed, and a development cycle that clearly links the domain investigation and software architecting cycles has been proposed (see the final report for the linked grant GR/R11919/01 at UoP). We have established that further research into this area is both feasible and promising. We outline this in the further research section below.

Further research or dissemination activities

To build on the work already done, we believe that further work is needed in the following areas:

- a) Investigate successive versions of the same software, preferably using additional examples from other domains, in various stages of its evolution, to see the degree of stability of the architectonic layers that we predict would be stable. The crystallisation of these notions would aid designers in designing software (or other) artefacts that reflect the architectonic profile of original problem domains, into software and systems artefacts.
- b) Developing measures of software artefact adaptability that can be compared to degrees of architectonic matches. If we are able to show that flexible software has a high degree of architectural correspondence to a model of the domain, then this is again will prove a substantial aid in designing adaptable systems and software.
- c) Refining our domain modelling methods to facilitate comparison with artefact architectural models.
- d) Developing a measure of closure for the process of qualitative/ structural domain modelling.

Acknowledgements

We would like to thank the EPSRC (Grant GR/R11919/01, GR/R12152/01) for their support in this feasibility study under the Systems Integration programme. We are grateful for Prof Bob Malcom, the Systems Integration programme co-ordinator for spotting the link between the two research ideas and brokering their integration. We have also had considerable help from Ship Analytics and in particular Dr Edgar Ernstbrunner who has spent much time in describing the problem domain and the associated programs. We are also grateful for RelQ for providing valuable feedback on our results.

References

1. Addis, T. R., and G. Galal, (2001) Using Problem-Domain and Artefact-Domain Architectural

- Modelling to Understand System Evolution European Conference on Information Systems, 2001, June 27-29, Bled, Slovenia.\
2. Brand, S. (1994). How buildings learn: What happens after they're built. London, Phoenix Illustrated.
3. Sussure, F. (1966 (1915)). Course in General Linguistics, McGraw Hill.
4. Frampton, K. and J. e. Cava (1995). Studies in Tectonic Culture - The Poetics of Construction in Nineteenth and Twentieth Century Architecture. Cambridge, Massachusetts, The MIT Press.
5. Galal, G. H. (1998). Software architecting: from requirements to building blocks within an architectural style. 12th European Conference on Object-Oriented Programming: ECOOP'98, 20-24 July , 1998. Workshop W2: Techniques, Tools and Formalisms for capturing and assessing Architectural Quality in Object-Oriented Software, Brussels, Belgium, <http://www.emn.fr/borne/>.
6. Galal, G. H. (1999). "On the Architectonics of Requirements." Requirements Engineering Journal **4**(3): 165-167.
7. Galal, G. H. (2001). "From Contexts to Constructs: the use of Grounded Theory in Operationalising Contingent Process Models." European Journal of Information Systems **10**(1): 2-14.
8. Galal, G. H. and R. J. Paul (1999). "A Qualitative Scenario Approach to Managing Evolving Requirements." Requirements Engineering Journal **4**: 92-102.
9. Galal, G. H. and R. J. Paul (1999). Systems Architectonics. Fifth Americas Conference on Information Systems (AMCIS'99), University of Wisconsin-Milwaukee. Milwaukee, WI. August 13-15, 1999., Association for Information Systems.
10. Galal-Edeen, G. H. (2002) Reverse Architecting: seeking the architectonic. Invited Keynote paper in Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE 2002) 29th October – 1st November, 2002, Richmond, Virginia, USA. IEEE Computer Society Press. Pp.141- 148.
11. Galal-Edeen, G. H. (2003) *Systems architecting: the very idea*, *Logistics Information Management*, volume 16, no. 2. Pp. 101-105.
12. Hillier, B. (1996). Space is the machine - A configurational theory of architecture. Cambridge, Cambridge University Press.
13. Hillier, B. and J. Hanson (1984). The Social Logic of Space. Cambridge, Cambridge University Press.
14. Hillier, B. and A. Penn (1994). "Virtuous Circles, Building Sciences and the Science of Buildings:

- using computers to integrate product and process in the built environment.” The International Journal of Construction Information Technology **1**(4): 69-92.
15. Jackson, M. (2001). Problem Frames: Analysing and structuring software development problems. New York, Addison-Wesley.
16. Maccari, A. and G. H. Galal (2002). Introducing the Software Architectonic Viewpoint. WICSA 3: The Working IEEE/ IFIP Conference on Software Architecture 2002, Montreal, Canada.
17. Pidgeon, N. F., B. A. Turner and D.I. Blockley (1991). “The use of Grounded Theory for conceptual analysis in knowledge elicitation.” International Journal of Man-machine Studies **35**(2): 151-173.
18. Strauss, A. and J. Corbin (1998). Basics of Qualitative Research, Techniques and Procedures for developing Grounded Theory. California, Sage.

Software Stability : A Rewriting Formal Specification Approach

Mohamed Larbi Rebaiaia
Computer Science Department, University of Batna, Algeria
E-mail: rebaiaia@arabia.com

Abstract

The software patterns are new software engineering concepts used to give a description solution to the software design problem, which occurs continuously in our immediate environment (industries, services, banking etc.) Such techniques remain general, not well defined, ambiguous and hard to be apprehended by software design practitioner's.

This paper is a simple endeavor, and at this stage, the content try to orientate software patterns researchers towards the association of formal methods especially algebraic specification within Stability Software Systems. To illustrate our proposition, the formal models derived from specifications are written using abstract data-type and the algebraic Rewriting Logic materialized in the style of Maude language according to UML Statecharts and Sequences diagrams.

1. Introduction

It is widely accepted by the software engineering community that almost software code contains portion of programs or informal sub-problems previously written or designed in previous projects. In such case, this fact is called a recurring problem. Developing new software embedding recurring portion of designs or codes is unnecessary and it is a costly and a time consuming process. To avoid such unpleasant situations in conjunction to the software reengineering, we need to use flexible design architectures, handing procedures and strategies to organize system patterns according to the notion of Software Stability Model (SSM) as introduced by Fayad et al. in [1],[2],[3]. In the SSM, the model of the system is composed by three particular classes of objects—the basic level, is the principal, timeless and is a stable core which represents the concepts of a representation in term of oriented objects; it is called the Enduring Business Themes (EBTs) class. The middle level, called the Business Objects (BOs) class, it is a semi-concrete part of the model, according to Fayad [1], the objects are externally stable and internally adaptable. The last one is the metalevel core, it contains what we call the Industrial Objects (IOs) class; they are a more concrete objects, which can be removed, added and replaced without affecting the systems. Such type of objects represents the interface of

the SSM model and depends directly according to the specialization of the application.

Despite the positive aspects of the software patterns and in particular software stability models to describe meticulously complex systems [4], [5] and [6], using informal phrases or graphical notations like UML language, certainly, they are very useful to trace up all the aspects of the problem. Meanwhile, they lack in capturing the fundamental and mathematical identity of the object system specialization, the model execution to simulate its behavior and in some cases to avoid error designs using formal verification techniques, plus, the absence of a proved strategy to organize the model into the three parts (EBTs, BOs and IOs objects classes). To be more precise, software patterns need to be explained using rigorous static semantics (e.g. UML and SDL Statechart diagrams [16], abstract data types), plus, a well founded dynamic semantics to express the behavior of the model (rewriting logic, temporal logic, automata ...), and, an executable of specifications tool like Maude, CafeObj, or Elan, without to push for the moment the expectation until to demonstrate the foundation of the theoretical aspects of the SSM.

The paper is organized as follows : section 2 presents the theory of rewriting. In section 3, Maude language is briefly described using the notion of functional and object theories. Section 4 outlines the algebraic specification theory as an approach to formalize the SSM models. Finally, section 5 presents some concluding remarks and the work in progress contents.

2. The Theory of Rewriting Logic

The Rewriting logic is a unified and reflexive logic based on both rewriting theory and equational logic [8]. A signature in this logic is a pair (\hat{a}, E) with \hat{a} a ranked alphabet of function symbols and E a set of \hat{a} -axioms (equations). The sentences are rules of the forms $[t_1]_E \rightarrow [t_2]_E$ with t_1 and t_2 are \hat{a} -terms and a theory is a rewrite system $S = (\hat{a}, E, L, R)$ where L is a set of labels and R is a collection of labelled and possibly conditional rules. This generalises the usual notion of theory, which is typically defined as a pair consisting of a signature and a set of sentences. The semantics of the rewriting

logic has been defined by Meseguer et al. in [8] as an inference system producing some well-known mathematical axioms (reflexivity, symmetry, transitivity and others.). In rewriting logic, if we suppose that S is a rewrite theory, we say that S entails a sequent $[t_1] \rightarrow^i [t_2]$ and write $S \vdash [t_1] \rightarrow^i [t_2]$ if and only if $[t_1] \rightarrow^i [t_2]$ can be obtained by finite application of the deduction rules as defined in [8], and thus in that sense we can define sequential or concurrent steps. We note that the parameter r behind the arrow means the label (name) associated with the rule.

Rewriting logic support membership equational logic which is a Horn logic whose atomic sentences are equalities $t = t'$ and membership assertions of the form $t : S$, stating that a term has sort S . Such logic extends order-sorted equational logic and supports sorts, subsort relations, subsort polymorphic, overloading of operators, and definitions of partial functions with equationally defined domains [9].

3. Maude Language

Maude language [10] is an object-oriented and executable specification based on the rewriting logic, it is a flexible and expressive abstraction in which different models and concurrent systems can naturally be specified. In Maude, distributed object-oriented systems can be defined using three kinds of modules: **(1)** a functional module introduced by the key-word *f-mod* **(2)** a system module introduced by the key-word *mod* **(3)** and an object-oriented module introduced by the key-word *omod*. A module contains sort and subsort stating that different data-types could be manipulated by the module and their functional relationships. Maude's type of structure is order-sorted [9]. The natural law of Maude as a language in the expressiveness of the concurrent and dynamic systems interpreting (specifying) the configuration (state) as a collection and messages using the **A.C.I** (Additive, Commutative and Identity) operator and the neutral element .

The relationship between an object and its own input and output messages is represented by the following rule:

$$M_1, \dots, M_n \langle O : C \mid a_1 : v_1 \dots a_n : v_n \rangle Mn_1, \dots, Mn_m$$

The object in Maude is a term $\langle O : C \mid a_1 : v_1 \dots a_n : v_n \rangle$ with O as object name, C as object class, a_i as the name of the attribute number i and v_i the value of the attribute i . The system evolve by a parallel and conditional rewriting rules which take the following form:

$$\begin{aligned} & \text{crl } [r] : M_1, \dots, M_n \langle O_1 : C_1 \mid \text{atts}_1 \rangle \dots \langle O_m : C_m \mid \text{atts}_m \rangle \\ \Rightarrow & \langle O_{i1} : C_{i1} \mid \text{atts}_{i1} \rangle \dots \langle O_{ik} : C_{ik} \mid \text{atts}_{ik} \rangle \langle Q_1 : D_1 \mid \text{atts}_1 \rangle \dots \\ & \langle Q_p : D_p \mid \text{atts}_p \rangle M'_1, \dots, M'_q \quad \text{if } \textit{Cond}. \end{aligned}$$

The above rule expresses a communication event in which n messages and m distinct objects participate, where r is the rule label. In a computing phase the messages M_1, \dots, M_n are then consumed and new messages M'_1, \dots, M'_q may be created by the system and sent to the referred objects or modules. The objects O_1, \dots, O_m occurring only on the left-hand side are deleted, objects $Q_1 \dots Q_p$ occurring only on the right side are created and those on both sides change their local states. The conditional *Cond* is an optional rule condition or a guard controlling the application of the rule. When several objects or messages appear in the left-hand side of a rule, they need to synchronize. Thus, such rules are called synchronous, while rules involving just one object and one message in their left-hand side are called asynchronous rules. Maude functional modules are executed by interpreting the equations of a specification as a left-to-right term rewriting system as a reduction process. This fundamental operation terminates when a term is reached to which no reduction is possible and it is called normal form associated to the term algebra modulo the congruence generated by the deduction rules.

As in object oriented philosophy, class inheritance is directly supported by the notion of order-sorted type structure [9]. A subclass declaration $C < C'$ is a particular case by which all attributes, messages, and rules of the superclasses characterize its structure and behavior and multiple inheritance is supported. For instance the following functional module (Figure 1) declares the abstract data-type of a bank account :

```

fmod bank_operation is
protecting Int .
sort Int, Nat, Account, Msg
subsor Nat < Int
class Account .
att solde : Account → Nat .
msgs credit, debit : Account Nat → Msg .
    msg transfer_from_ to _ : Nat Account Account
    → Msg
vars C1 C2 : Account, var mont : Nat .
endfm

```

Figure 1. Algebraic specification of a bank account

The example (Figure 1) presents bank operations in term of balancing from credit to debit and vice-versa of two accounts say Custum1 (C1) and Custum2 (C2). To be more accurate, the information contained in such example could be manipulated using two kinds of rules -- synchronous and asynchronous rules where (transfer Mont From C1 to C2), credit(C1, Mont) and debit(C2, Mont) are messages.

4. Formalizing State Stability Model Description

The state stability model as defined previously, seems to be a well-formed model and can be approached as a concurrent system which possesses a number of elements used to identify its structure by means of static abstract data-type (ADTs). In such case, the ADT is identified formally as an algebraic specification. Our formal development is based on the ideas of the order-sorted algebraic specification as developed by meseguer et al. in [9]. Such formalism is a well-founded mathematical and has been proved sound and complete using category theory, plus its facility to be computationally interpreted. The other side of the state stability model not introduced until now by the SSM community is capable to give a certain tonicity to the SSM, and then to take it away from its structural rigidity to beneficiate from formal theories. This renewal stage can be performed using the dynamic part of the ADTs which is performed using equations and axioms and the reduction operations of the rewriting logic.

The SSM representation (static part) of a system could be substantially described using David Harel's Statecharts [15] extended with object oriented capabilities. A statechart diagram can be defined formally as a rewriting specification theory, whose configuration are particular states, a state can be simple or composite. A composite state consists of either concurrent substates or disjoint substates, commonly called as the AND and the OR composite states. The following example describes graphically such elements what we call the Crossing Train-Gate system (Figure 2).

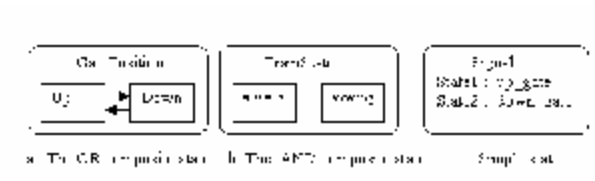


Figure 2. Some Harel's Statecharts States

The formal meaning of Harel's Statecharts diagrams, an object that remain in a state may be performing some activities and thus an action is an abstraction of a computational procedure by means of sending a message or modifying a value of an attribute and its signature is a label written as the name of a transition expressing for instance the name of an event and its relative parameters, a guard condition or some activation actions.

The following section will help in understanding some basic notions regarding the formalism used here in this paper as for example the definition of polymorphism and the modules encapsulation used in the Object Oriented paradigm and UML notations [12], [13], [14] and [15]. We may understand by the notion of sort and subsort defined in the Abstract data type theory exactly as the notion of type and subtype. For example an integer number is declared as "sort Int."

Given a sort set S , an S -sorted set A is just a family of sets A_s , and for each sort $s \in S$ we write $\{A_s | s \in S\}$. We define an S -sorted function as the relation between two s -sorted sets A and B and an S -sorted family is written $f = \{f_s: A_s \rightarrow B_s | s \in S\}$.

In the order-sorted theory, S is a partially ordered set, or poset, i.e., there is a binary relation \leq on S that is reflexive, transitive and antisymmetric.

If we consider the extension of the ordering on S to strings of equal length in S^* by $s_1 \dots s_n \leq s'_1 \dots s'_n$ iff $s_i \leq s'_i$ for $1 \leq i \leq n$. Similarly, \leq extends to pairs in $S^* \times S$ by $\langle w, s \rangle \leq \langle w', s' \rangle$ iff $w \leq w'$ and $s \leq s'$ and we may write $\sigma: w \rightarrow s$ to emphasize that σ denotes a symbol function with arity w and sort s . If w is the empty string, σ denotes the constant of sort s .

Definition 1. A many-sorted signature is a pair $(S, \hat{\alpha})$, where S is called the sort set and $\hat{\alpha}$ is an $S^* \times S$ -sorted family $\{\hat{\alpha}_{w,s} | w \in S^* \text{ and } s \in S\}$. Elements of $\hat{\alpha}$ are called operation (or functions) symbols.

Definition 2. An order-sorted signature is a triple $(S, \mathcal{L}, \hat{\alpha})$ such that (S, \mathcal{L}) is a many-sorted signature partially ordered set of sorts. Each sort denotes a collection of distinguished and homogenous data items and the operations satisfy the following monotonicity condition,

$$s \hat{\alpha}_{w_1, s_1} \hat{\alpha}_{w_2, s_2} \text{ and } w_1 \mathcal{L} w_2 \text{ imply } s_1 \mathcal{L} s_2.$$

Example : The following code (Figure 3) is a part of a Maude functional module, it presents a formal description of the lists of Integers data structure. The subsort NeList defines the nonempty lists and Int is pre-defined Module. Nil is the constant of type List, head and tail define the first and the last element of the list. The poset operator is written $<$ for typographic convenience.

C : Controlling, A : Automation, M : Monitoring, A_M : AnyMechanism, A_Med : AnyMedia, A_P : AnyParts.

<A : EBT | A_i : attributeA_i> <C : BO | C_i : attributeC_i> <M : BO | M_i : attributeM_i> <A_M : BO | A_M_i : attributeA_M_i> <A_P : IO | A_P_i : attributeA_P_i> <A_Med : IO | A_Med_i : attributeA_Med_i> .

Individual form according to the SSM description:

First relation : between basic level and the middle level,

<A : EBT | A_i : attributeA_i> <C : BO | C_i : attributeC_i> <M : BO | M_i : attributeM_i> <A_M : BO | A_M_i : attributeA_M_i>

Second relation : between the middle level and the metalevel (third one),

<C : BO | C_i : attributeC_i> <M : BO | M_i : attributeM_i> <A_M : BO | A_M_i : attributeA_M_i> <A_P : IO | A_P_i : attributeA_P_i> <A_Med : IO | A_Med_i : attributeA_Med_i> .

These types of rule authorize the communication only between the objects belonging to the basic level with those belonging to the middle level, and, the middle level object will be authorized to communicate with those of the metalevel.

The following code gives an overview of the sequence diagram written in Maude as a rewriting rules system.

<A : EBT | A_i : wait> $\dot{\Rightarrow}$ <A : EBT | A_i : wait> (specifyLevel A to A) .
(specifyLevel A to A) <A : EBT | A_i : wait> $\dot{\Rightarrow}$ <A : EBT | A_i : active> .

<A : EBT | A_i : active> $\dot{\Rightarrow}$ <A : EBT | A_i : active> (over_ride A to C) .

(over_ride A to C) <C : BO | C_i : wait> $\dot{\Rightarrow}$ <C : BO | C_i : activate> .

<C : BO | C_i : activate> $\dot{\Rightarrow}$ <C : BO | C_i : activate> (proceed C to A_M) .

(proceed C to A_M) (A_M : BO | A_M_i : wait) $\dot{\Rightarrow}$ <A_M : BO | A_M_i : Activate> .

<A_M : BO | A_M_i : Activate> $\dot{\Rightarrow}$ <A_M : BO | A_M_i : Activate> (hold A_M to A_M) .

(hold A_M to A_M) <A_M : BO | A_M_i : Activate> $\dot{\Rightarrow}$ <A_M : BO | A_M_i : Activate> .

<A_M : BO | A_M_i : Activate> $\dot{\Rightarrow}$ <A_M : BO | A_M_i : Activate> (operate A_M to C) .

(operate A_M to C) <C : BO | C_i : activate> $\dot{\Rightarrow}$ <C : BO | C_i : activate> .

<C : BO | C_i : activate> $\dot{\Rightarrow}$ <C : BO | C_i : activate> (defineRole C to A_P) .

(defineRole C to A_P) <A_P : IO | A_P_i : wait> $\dot{\Rightarrow}$ <A_P : IO | A_P_i : activate> .

5. Conclusion

It is well known that Pattern Software engineering is a recent field not well explored by the software engineering community. Among such new ideas, Stability Software models, are a promised concepts.

The main objective of this paper is to show that SSM models could be approached mathematically using formal algebraic specification theories. The idea explored here, is to describe SSM model objects classes using what we call the order-sorted algebra and thus to make them executable. The executability has been provided using an automatic deductive system based on the rewriting logic, a well known formal tool called Maude.

Perspective and future works according to the formal description of the SSM model is to show that some properties as a graphical signification could be verified using temporal logics as for example Linear Temporal Logic [11], [12] which is broadly related within Maude capabilities.

6. References

- [1] M.E. Fayad, and A. Altman, "An Introduction to Software Stability," *Communications of the ACM* (44:9) 2001, pp 95-98.
- [2] M.E. Fayad, and S. Wu, "Merging Multiple Conventional Models in One Stable Model," *communications of the ACM* (45:9) 2002, pp 102-106.
- [3] M.E. Fayad, "Accomplishing Software Stability", *Communications of the ACM*, Vo. 45, No. 1, January 2001, pp 95-98.
- [4] A. Mahdy and M.E. Fayad, "A Software Stability Model Pattern", *Proc. of the 9 th Conference on Pattern Language of Programs (PLoP02)*, Illinois, USA, Sept. 2002.
- [5] A. Mahdy, H. Hamza, M.E. Fayad, and Marshall Cline, "Identifying Domain Patterns Using Software Stability Paradigm," *International Conference on Information Reuse and Integration (IRI)*, Nevada, USA, 2003.
- [6] M.E. Fayad, G. R. Cangiano, and H. A. Sanchez, "The Automaton Analysis Patterns", *Technical Report No. 04-03-02*, University of Nebraska-Lincoln, March, 2004.
- [7] M.E. Fayad, V. Stanton, and Hamza, H. "A New Look At the CRC Cards." <http://www.activeframeworks.com>

[8] J. Meseguer, “A Rewriting Logic as a Unified Model of Concurrency”, Springer LNCS 458-1990.

[9] J. Meseguer and J.A. Goguen, “Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operation”, Technical Report, SRI-CSL-89-10, July, 1989.

[10] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada. A, “Maude Tutorial, March 2000”, <http://maude.csl.sri.com/tutorial>.

[11] K. Havelund, G. Rosu. Testing Linear Temporal Logic Formulae on Finite Execution Traces. RIACS Technical report, <http://ase.arc.nasa.gov/pax>, November 2000.

[12] M.L. Rebaiaia and J. M. Jaam, “VALID-2: A Practical Modeling, Simulation and Verification Software for Distributed Systems”, in the proceeding of the IEEE-IPDPS 2004, Santa Fee, New Mexico, ISBN 0-7695-2132-0, April 2004.

[13] G. Booch, J. Rumbaugh ,and I. Jacobson. Unified Modelling Language User Guide .Addison-Wesley,1999.

[14] Object Management Group, Unified Modelling Language Specification, version 1.3, June 1999, <http://omg.org>.

[15] D. Harel, “Statecharts: a Visual Formalism for Complex Systems”, Science of Computer Programming, vol. 8, pp. 231-274,1987.

[16] M. Broy, “Towards a Formal Foundation of the Specification and Description Language SDL”, Formal Aspects of Compting, vol.3, pp. 21-57, 1991.

Planning Stable Software Applications using Goal Driven Requirements Analysis

Islam A. M. El-Maddah

Ain Shams University, Faculty of Engineering, Computer and Systems Engineering Dept.

1 Saryat St, Abdo Basha Square, 11517 Cairo, Egypt

Email: islam_elmaddah@yahoo.co.uk

Abstract

The ever-increasing user needs entail constantly modifying/extending existing software applications. However, the effort and cost required to satisfy the new user- needs usually control the decisions made in software industry. Thus, the need for developing stable software applications has emerged. These stable applications should be less expensive to extend and require less effort and time to develop/test. Since the new demanded user needs make the existing application unstable, the stability treatment should start at the requirements level.

Thus, a goal driven requirements analysis method (GOPCSD) has been adopted to develop stable applications. This goal driven requirements analysis method has a well defined path for correcting the requirements and tracing the user needs to the formal specification and implementation levels. This controls the effort required to maintain the applications. Furthermore, the requirements reuse and automatic code-generation concepts used in the GOPCSD method support developing stable software applications

Keywords: *Goal driven Requirements Analysis, Traceability, Stability, Requirements Validation*

1. Introduction

After more than five decades, the software has reached a considerable mature level in satisfying the initial user needs. However, the user requirements are constantly increasing and changing. This restricts the speed and ease in which the software producer can modify the existing applications to meet the new user needs. Thus, the process of making the extension or modification decisions is mainly controlled by cost and effort feasibility studies. A practical solution is to plan stable [12, 13, 14] software applications that are clients-agreed, as well as can be easily extended/slightly-changed to meet the possible future needs.

Although software engineering differs from other engineering disciplines, software products can be termed stable when it is neither difficult nor expensive to perform some minor changes on them. On the other hand, existing software applications can become unstable when a bug is discovered in the application and decided to be removed or when a function is planned to be added to the application.

The effort required to modify a software application does not depend only on the new functions to be added or bugs to be removed, but it is also affected, to big extent, by the structuredness and organizedness [6] of the existing application. Some software programs can be easily altered or extended than others. For example, a little function, such as manipulating one more output variable may result in making the whole program unreliable and even require reengineering the whole application from scratch. This depends on how well the application was initially developed, how much effort has been devoted to prepare the stage for extending the application in the future, and whether it is possible to trace the implementation back to the user needs, as well as to produce implementation corresponding to some abstract requirement.

The cost paid for obtaining a well-structured and organized software application is not directly paid back, but at later stages, such as when evolving the software application, adding a new function to it, or when removing a bug (discovered either by the user or the implementer) from the software application after possibly producing an implementation. This means a cost- feasibility study should be carried out to invest in long-term software applications (that usually need to be maintained over long period of time) [4, 5]. Although the production cost [15] at the early stages will be high, the overall cost required to develop and maintain the application will be less. Developing stable software applications does not come without a price. Such a price mainly depends on the components, architecture, design patterns used to develop the stable applications. The relationship between the application's architecture and its stability has been studied in many literatures, such as [3, 16, 17, 19].

Thus, we were motivated to plan developing stable software application, as early as at the requirements analysis stages and ensuring the user needs are traced to the implementation level, and vice versa. This guided us to adopt the goal-driven concepts of the KAOS method [20] and adapt it for the process control systems. The developed method (GOPCSD [7] (goal oriented process control systems design)) refines the user needs into formal specifications that can be easily mapped to implementation code. We supported this method by

developing an integrated environment (GOPCSD) [8] that refines, reuses [10], debugs, reasons about and validates the requirements. In addition, the corrected requirements will be translated into corresponding formal specifications in B [1] and usecases.

In this section, we have introduced the research area. The paper consists of three more sections. In section two, we discuss some issues about developing stable software applications starting from the user needs and system requirements stages. In section three, we describe the software stability support in GOPCSD. Finally, in section four, we provide some conclusions and suggestions for future work.

2. Developing Stable Software Applications

The software stability term indicates how much (quality and quantity) changes are required to satisfy a new user need to be added to the software or possibly to remove a detected bug after developing the application (which is practically probable because the image of the software application does not become complete until it has been developed/tested and possibly used over a period of time). Thus, it is not uncommon to start tracing the stability at the requirements level.

The development of stable applications can be considered as a process that has two main activities. The first is to develop software applications free of bugs and user-agreed (thus less probable to be changed in future); whereas, the second is to prepare the developed applications to be easily extended and enhanced according to the future user needs. Although these two activities are related, we discuss each of them in the following sub-sections.

2.1. Planning Stable Software Applications

Software instability sources can be diminished before implementing the application by avoiding developing unchecked or user-disagreed applications. Such situations could happen because of agreeing to ambiguous requirements/user needs, formalizing unchecked requirements, or delaying the user-feedback to the late implementation stages. When the software producer avoids all of these situations, the chances for developing stable applications would be higher.

2.2. Maintaining Software Stability

Since the changes in user needs cannot be totally avoided neither expected in advance, other decisions

should be undertaken to reduce the effort and cost required to extend/ evolve the software applications or remove the discovered bugs from them. These decisions can include structuring the requirements/specification/implementation, achieving a level of independence between the design elements and their inter-relationships, relying on software reuse and automatic code generation/translation from one stage to another.

Grouping the related software applications' constructs will reduce the effort and cost required to extend/evolve the software applications. Thus, a well structured application would need less effort to extend [6] because it will be partially changed not entirely in order to meet the new user needs. Achieving a considerable level of independence between the design elements and their inter-relationships can reduce the effort required to extend the application. For instance developing the applications out of pre-fabricated components, objects and design patterns [23] are examples of achieving this independence.

Moreover, adopting the software-reuse [18] and automatic code-generation concepts should result in more stable applications. The automatic generated software applications should be more reliable and having less bugs because of the checks and tests they already passed during the early development stages. But, the generated code may be less understandable if it is expected that a human element is still required in the development lifecycle. Moreover, software reuse decreases the risks that the developer faces when developing or extending a part of the system from scratch.

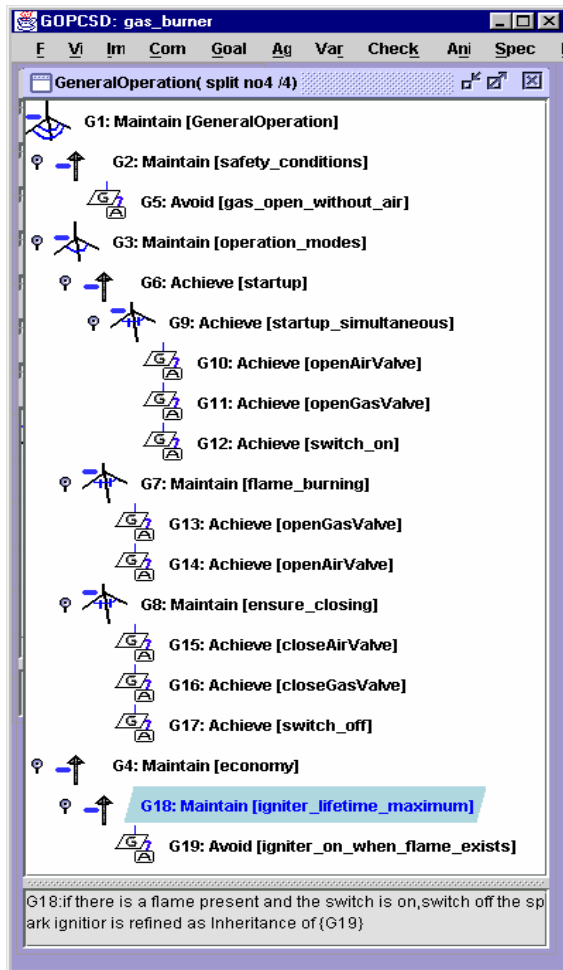


Fig. 1, the goal model of the gas burner system

3. Software Stability Support in GOPCSD

3.1. Introduction to GOPCSD

The GOPCSD method [7] has been developed to manage rationally the gap between the process control systems engineer's perspective, as the client, and the formal specification level, supplied normally by a software engineer. To accomplish this; we have developed an integrated requirements development environment [8], where the process control requirements can be constructed using a provided library, structured in terms of hierarchies of goals, and then checked, corrected, validated and finally automatically translated to a B formal specification. GOPCSD uses the following elements to construct process control requirements model:

The Components. Components represent the physical parts of the applications, such as valves, robots, and deposit belts. The detailed specifications of each component, including its variables, agents and goal-models, are stored in the GOPCSD library.

The Variables. Variables are considered as the essential part of formalizing the user requirements. In the GOPCSD tool, the application's global state is described by a set of variables. The variables are associated with the high-level goal-model templates or the components, each of which the user can import from the library.

The Agents. Agents are software, hardware or human that control the application's output variables and possibly the local environment. Some of the agents can be part of the application to be built, or, alternatively, they can be software programs or hardware devices that will be responsible for accomplishing pre-defined goals to fulfil the overall application operation.

The Goal-models. Goal-models are tree-like structures where requirements reside; they represent the requirements in a hierarchy of goals. Each goal-model starts with a main goal that traces one aspect [9] and has general scope; this main goal is usually refined to a number of sub-goals describing sub-parts, or operation-modes of the application. The GOPCSD tool supports informal descriptions of goals as well as formal descriptions based on temporal logic [22].

Fig. 1 shows an example of a goal model for a gas burner system; the high-level goals, such as G2, G6, G7, G8 and G4 accommodate the user needs and are traced (using different goal-refinement patterns [7]) to terminal functional goals, such as G5, G10, G11, G14, G15 and G19. These terminal goals form a basis for the formal specification, and hence the implementation of the application.

3.2. Developing Stable Applications

The construction of process control applications in GOPCSD covers the early development lifecycle stages including requirements elicitation, formalizing requirements, user needs validation, and specification generation. These development processes are accomplished in three consecutive phases.

In phase one, the abstract user needs are refined and formalized in a tree-like that starts with the abstract goals and ends with fully specified and formalized requirements specifications. In phase two, various checks and tests are applied on the constructed model to provide a feedback to the user to draw his/her full agreement as well as provide a means for correcting the wrong requirements decisions; thus phase two early removes the software bugs before it is too expensive and tiring to discover them. After the requirements are fully checked, corrected and user-agreed, the tool automatically translates them into a B formal specification that can be used within Btoolkit environment [2] in the readiness of a software engineer to complete the development of the process control applications.

The goal-driven requirements analysis of GOPCSD can contribute to the development of stable software applications in two different ways: reducing the chances for producing user-dissatisfied applications and producing a traced user needs [9, 11] that can be mapped into formal specification constructs (so that any future modifications can be easily added to the application).

The consistency, completeness and reachability checks offered in the GOPCSD tool [8] provide a strong analysis stage to capture the hidden bugs that can be transmitted from informal requirements to the specification stages. As shown in fig. 2, the tool provides an early feedback path for the user to check whether the application will satisfy his/her main needs or not, rather than leaving these chances to the post-implementation and test stages, where it will be difficult and expensive to correct the wrong requirements decisions. This increases the separation between the requirements and specification concerns. And put the client in a charge of correcting the requirements as early as possible. Thus, the possibility that the client will disagree about the implemented applications would dramatically decrease.

The GOPCSD tool guides its user to develop a software requirements model, which can be extended in future to accommodate new user needs and/or enhance the existing ones. It makes it possible to automatically translate the checked and corrected requirements into formal specifications and hence an implementation in the B toolkit [2] environment or similar environments.

The structure of the requirements within the GOPCSD tool can be arranged in aspect-based fashion

[9], in which the requirements related to one aspect will be grouped together. This can be useful for separating the treatment for each aspect, such as safety, operation, productivity. Thus, for example, when the user demands a change in the safety aspect of the developed application, it would be much easier to modify a single aspect view without altering the rest of the requirements model.

4. Conclusions

Having realized the importance of the requirements stage [21] and its effect on the stability of the developed software applications, the GOPCSD method focuses on requirements analysis. The software development in GOPCSD is based on some stability-support concepts. These concepts include refining user needs, tracing them to the implementation level and grouping the system requirements in a fashion very similar to the expected way, in which they would be possibly extended or altered in future.

The GOPCSD method structures the requirements in a hierarchy of goals, each of which concerns a local requirement, as shown in fig. 1. This makes the possibility of altering or changing the whole model decreases. The checks that can be performed to make the model consistent, complete and validated are already supported by an automatic tool that can perform the goal-conflict, completeness and reachability analyses.

Some research should focus on adopting the concepts of defining stability meters and integrating such meters into the early stages of the software development lifecycle. Thus, it would be possible to judge how much stability is provided by some specific development tool or method and whether one software program is more stable than another program.

The effort paid towards having a stable software application should be carefully planned, depending mainly on the expected lifetime of the application; thus, a short-term application which may be completely changed, should not attract much effort in planning its stability. Whereas, long-term software applications that are supposed to remain for long period of time, should be planned to be stable and easily extended.

References

- [1] J. R. Abrial, (1995), "The B Book: Assigning Programs to Meaning", Cambridge University Press.

- [2] B Core UK limited, (1998), B Toolkit <http://www.b-core.com/btoolkit.html>
- [3] R. Bahsoon and W. Emmerich, (2003), "ArchOptions: A Real Options-Based Model for Predicting the Stability of Software Architecture", Proc. of the ICSE Fifth Workshop on Economics-Driven Software Engineering Research.
- [4] B. Boehm (1981), Software Engineering Economics. Prentice Hall.
- [5] B. Boehm and K. J. Sullivan, (2000), "Software Economics: A Roadmap" In: Finkelstein, A. (ed.): The Future of Software Engineering.
- [6] A. M. Davis, (1993) "Software Requirements: Objects, Functions and States", Prentice Hall PTR.
- [7] I. A. M. El-Maddah and T. S. E. Maibaum, (2003), "Goal-oriented Requirements Analysis of Process Control Systems", Proc. Of 1st MEMOCODE, Mont St Michel, France.
- [8] I. A. M. El-Maddah and T. S. E. Maibaum, (2004), "The GOPCSD Tool: An Integrated Development Environment for Process Control Systems Requirements and Design", FASE 04, Barcelona, LNCS vol. 2984, Springer-Verlage, Heidelberg
- [9] I. A. M. El-Maddah and T. S. E. Maibaum, (2004), "Tracing Early Aspects in goal driven Process Control Requirements", Early Aspects Workshop, Aspect Oriented Software Development Conference, Lancaster, UK.
- [10] I. A. M. El-Maddah and T. S. E. Maibaum, (2004), "Requirements Reuse: Component-based development using GOPCSD", Proc of ICSR8, Madrid, Spain.
- [11] I. A. M. El-Maddah and T. S. E. Maibaum (2004) "Towards a Software Engineered Road for Developing Process Control Systems", Proc of ICEEC04, Cairo, Egypt.
- [12] M. E. Fayad, (2002), "How to deal with software stability", *Commun. ACM* 45(4): 109-112.
- [13] M. E. Fayad, (2002), "Accomplishing Software Stability", *Communications of the ACM* 45(1), Jan. 2002.
- [14] M. E. Fayad and A. Altman, (2001), "An Introduction to Software Stability", *Communications of the ACM*, Vol. 44. No. 9, September 2001.
- [15] J. M. Favaro, K. R. Favaro and P. F. Favaro, (1998), "Value-Based Software Reuse Investment", *Annals of Software Engineering*. Vol. 5. (1998) 5 – 52
- [16] A. Finkelstein, (2000), "Architectural Stability, Some Preliminary Comments", <http://www.cs.ucl.ac.uk/staff/a.finkelstein>.
- [17] G. H. Galal-Edeen, (1999), "On the Architectonics of Requirements – Viewpoint", *Requir. Eng.* 4(3): pp 165-168.
- [18] H. Hamza, M. E. Fayad, (2003), "Engineering and reusing stable atomic knowledge (SAK) patterns", Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 26-30, 2003, Anaheim, CA, USA.
- [19] M. Jazayeri, (2002) "On Architectural Stability and Evolution", *Lecture Notes in Computer Science*, Springer Verlag.
- [20] A. van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy, (1991) "The KAOS Project: Knowledge acquisition in automated specifications of software", proceeding AAAI Spring Symposium series, Track: "Design of Composite Systems", Stanford University, pp 59-62.
- [21] B. Nuseibeh and S. Easterbrook (2000) "Requirements Engineering: a Roadmap", *Future of Software Engineering*, Limerick, Ireland
- [22] Z. Manna and A. Pnueli (1992) "The Temporal Logic of Reactive and Concurrent systems", Springer-Verlag.
- [23] S. Wu, A. Mahdy, and M. E. Fayad. (2002), "The impact of Stability on Design Patterns Implementation", Proc. of the Conference on pattern Languages of Programming (PloP).

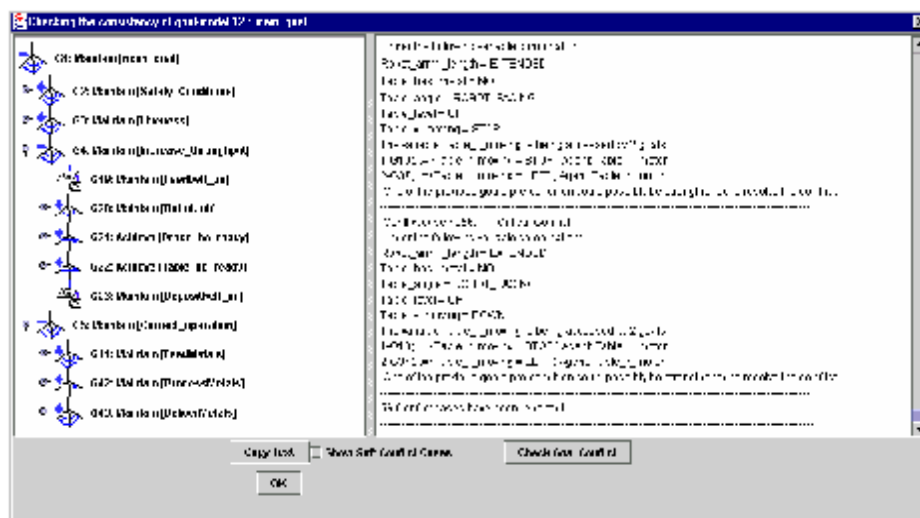


Fig. 2, consistency check within the GOPCSD tool

Performance Modeling and Analysis of Object Oriented Distributed Software Systems: A Necessary Step Toward Software Performance Stability

Reda Ammar and Amal Abdel-raouf
Computer Sc. and Engr. Dept.
Univ. of Connecticut,
Storrs, CT 06269-3155
amal@engr.uconn.edu

Tahany A. Fergany
Computer Sc. Dept,
Univ. of New Haven
West Haven, CT 06516
tfergany@newhaven.edu

Abstract

Advances in object-oriented (OO) technology and distributed computing generate distributed object oriented software systems. Choosing an efficient design of such software is a multi-criteria decision problem. Performance is a key criterion that makes software developers be able to select the system that best fit their requirements and achieve longer design stability. Classical techniques of performance analysis are either unsuitable or unnatural to capture performance behavior of OO systems. In this paper, we present Performance-Based Model for Distributed OO Software (DOOS) and a methodology to analyze and evaluate its performance. The new model evaluates the time cost of DOOS system considering the communication overheads while preserving the OO features such as encapsulations and inheritance.

Keywords: software performance stability, performance modeling and analysis, Object-Oriented software

1. Introduction

The OOS has become one of the leading techniques in problem solving for being more natural and reliable way [1, 3, 5]. It also has a high potential for reuse, and a relatively quick and easy way to implement and maintain. OO programming is widely accepted for designing and implementing software systems for various application domains such as: air traffic control, high-energy physics, e-commerce, and military systems. However, these systems need to meet performance objectives. Most often, performance doesn't become an issue until software implementation and integration are complete and critical paths within the system fail to meet timing requirements [9]. In these cases, software is redesigned to make its performance acceptable. Performance analysis during the software life cycle will ensure that the system satisfies its performance

objectives. Performance analysis will not only evaluate the resources utilization but also will allow comparing different design alternatives to select the most efficient (and hence stable) design, detecting and solving performance bottlenecks as early as possible to avoid future problems, minimizing maintenance needed and re-use existing efficient components [11]. All of these activities are critical to provide longer life to the software and increase its stability.

Most OO techniques studying the performance of OOS are based on either measurement [4] or mapping to a conventional performance model [7] and hence no way to preserve OO features during the analysis. In this paper, we introduce an analytical modeling approach for DOOS system that represents the system behavior and evaluates its performance. The model will capture the behavior of the overall system, preserve the OO features, properties and relationships between objects, analyze the system performance and take into consideration the resulting communication activities among objects. The model is a necessary tool to achieve software performance stability via increasing efficiency and re-usability, and minimizing maintenance.

2. Model Description

The model of a node in DOOS system is divided into two phases; one represents the execution process and the other stands for the communication process (see Figure 1). A hierarchical model is necessary here to represent the distinct abstraction levels of the OOS system. The hierarchical approach allows the designer to overcome the complexity of modeling a large application with multiple levels of abstraction and large numbers of interacting objects. This modeling framework helps the system designer to derive the information required to construct a software application that meets a set of performance requirements and to make the design amenable to future changes without major cost. The proposed model consists of:

2.1. Input queue

Arrivals to a software node can be classified into three categories. The first one is the *external user request (EUR)*, which submits a request to execute the entire software driver. Therefore, driver's modules will be processed on this target node and initiate communication requests. The communication requests may include: an object data update, request for Object Procedure Call (OPC), and/or send information to other nodes. The second type of arrivals is *Remote Request (RR)*, which is a request from another node in the distributed system to perform a computation activity within this target node. The computation activity may include: a request to create or update an object, to invoke a local object-related method, and/or sent information to this target node. The third type is *response*, which carries information representing the results of an OPC sent earlier by the target node to others. This type of arrivals requires no communication activity and hence has no effect on to the communication process.

2.2. Execution server

Requests arrive to the input queue activates methods through the software execution driver. It executes the methods, calculates the execution time cost equations using all the information in the upper layers of the model, and then forwards the required communication to the communication server. The time spent in the execution process is evaluated through the model using the traditional CSM model [6, 8].

In order to manipulate the structure and all the properties of the OOS, we are trying to imitate the design phase of an OO system. The natural way of handling the execution of an OO application is to divide it into three main phases. As shown in figure 1, there are three levels of abstractions each corresponds to one phase. First is to identify the classes in the system, each with its own attributes and methods through the *class level*. The second phase is to create objects as instances of the existing classes and hold their information in the *object level*. The third phase is to execute all the software modules as represented by the *master level*. Therefore, our hierarchal performance model consists of three levels of abstraction together in addition to the driver level. The model maintains the structure of the application itself by using the *Performance Image (PI)* structure. The PI will be automatically generated for all classes and objects defined within the system. The PI itself is a multi-layers representation as shown in figure 1; each of these layers is reflecting one of the main properties that can't be neglected in any OOS. Each layer

will also hold the performance cost that is partially added by the use of this layer related information in the performance evaluation process. This layered approach allows separating the time cost of each layer and each individual component. The PI consists of three layers and is generated for all *classes* and *objects*.

2.2.1. Object-creation Performance Layer (OPL),

Object creation method has a conceptual significance in the OO paradigms. In contrast to other methods, it can only be executed once and it has some implicit effects due to the objects relationships. Therefore, OPL is responsible for the performance calculations of object creation. At the class level, the OPL layer will include the cost equations of all the possible constructors; default and others. This layer will also keep all other information about classes (such as private, public, attributes, .. etc.). At the object level, The OPL will include only one cost equation for the creation of this Object. It will be abstracted from the PI of the original class at the class level. This layer deals only with the basic object (being created explicitly at this point), but any other object created implicitly due to any relationship with this object will have its own PI developed at the creation time.

2.2.2. Related Classes / Objects Performance Layer (RPL).

The relationship between classes and objects is a basic concept in OOS. Therefore, we have a specific separate layer (RPL) to include all the relationships information. At the class level, RPL will have all the information about classes related to this class by composition, inheritance or access relationships. However, at the object level, RPL is designed mainly to handle the implicit creation of objects due to the relationships among the original classes. In case of composition relationship, the creation of the PI of the whole-object will lead to the creation of the PI of the part-object. RPL of the whole-object will include a link to the PI of the part-object. This provides a way to include the cost of creating this object and access any of its information. In case of inheritance relationship, we will have one PI that has all the performance information for both the parent and derived classes. RPL simplifies the multi-dimensional relationships commonly exist in OOS. It also helps if this model is used in concurrent OO application, to reduce the occurrence of inheritance anomaly [2].

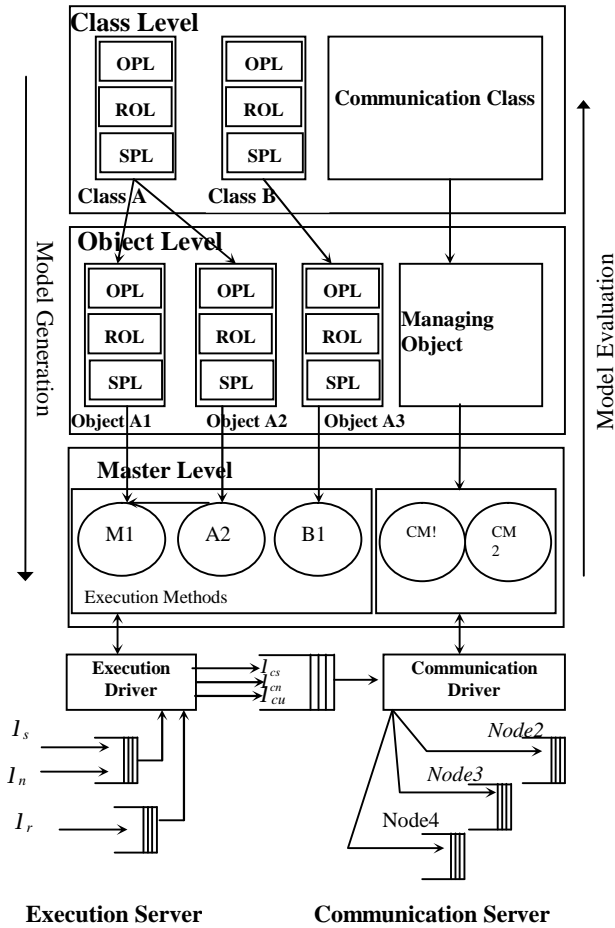


Figure 1: The DOOS Performance Model Node Structure

2.2.3. Service Performance Layer (SPL). SPL represents a list of the methods defined within the class/object. The detailed behavior and analysis of the methods will be specified in SPL. Using CSM (Computation Structure Model), defined in [6] gives each method cost equation that contributes in the overall estimated execution time calculation.

2.3. Execution-to-Communication Buffer

To ensure better system performance, a buffer is used in between the execution and communication servers. Therefore, if the execution server is faster, the data and requests sent to the communication server can be kept in the buffer. The arrivals to communication buffer or queue are the data and/or information that need to be sent to other nodes within the distributed systems. There are two arrivals categories to the communication queue. First is the portion of the EUR that need to be processed on or propagated to other nodes (i.e. OPC). The second

category represents the communication activities due to RR, which may create a message for another OPC (i.e. nested OPCs), for updating, or sending back the results of the RR.

2.4. Communication Model Description

A built-in communication class is generated in the class level; the managing object is an instant of the communication class, which is constructed in the object level as shown in figure 1. In the master level, the entire driver communication-related methods are included.

The Communication Class is mainly concerned with the internal and the external communication activities that take place among objects in a DOOS computation. It holds information about all the cooperating objects, their locations (node holding each object), number of object copies, the size of data exchange between objects methods, links between nodes, and others. This information will be used to calculate the cost of communication processes and also the cost of updating the objects' data. Methods in this class are designed to enable a node to communicate with other nodes in the system. The first core method is *Add_Object* to add a new object and defining its parameters. *Remove_Object* removes any superfluous object. This method is necessary to ensure better resources utilization. *Send_Message* sends data from one object method to another using both the physical and logical link information. *Update_BW* is necessary for dynamically changing bandwidth values. *Send_Update* is used when a change in an object state occurs to update all its copies and keep system consistent.

The Managing Object is created automatically in the Object level with the creation of any node in the system. This object is the only instance that can be created from the communication class and it is unique in every node. This object is responsible for the communication between its node and all other nodes. All the communication methods that are defined in the communication class can only be invoked through this object. The attributes of the object hold the communication parameters that will be used in evaluating the communication time cost such as: the message size, message multiplier, communication arrival rate, and others. The Communication Methods that are defined in the communication class can be used within the Master level. Only the required methods are invoked depending on the driver structure and the type of communication arrivals. *The communication driver* will cooperate with higher layers to retrieve the information necessary to calculate the overall communication cost.

2.5. Output Queues

The output queues represent the input queues to other nodes in DOOS communicating with this node. Communication is performed in a point-to-point fashion. The output of the communication server will go through the physical communication link connecting nodes, and then to the corresponding input queue. The communication server may submit requests and/or information to one or more communication links.

3. Model Evaluation

We extended the communication queuing model presented in [10] to adapt the DOOS model defined in the previous sections. This model provides an accurate representation for the communication activities among nodes in distributed system. It also provides a way to evaluate the execution cost of the software modules and their related communication activities. The performance model consists of two main parts: the execution server and the communication server. The execution server and its related analysis represents the execution process of the software modules reside on the node and provides a way to evaluate the execution cost. The communication server provides the analysis representing the communication activities (including objects updating) of this node with other nodes and the evaluation process for communication cost. Total cost will be the summation of both the execution and communication costs. In the analysis, three classes of arrivals enter the input execution queues. First is EUR and second is RR. Both types arrive to the same queue but each will be processed in accordance to its own requirement. The third type is the response that will be received by a separate queue. Responses are receiving higher priority in processing since they require short execution service time.

4. Conclusion

In this paper, we introduced a performance-based model for DOOS systems. The model considers both the computational and communication aspects. The model

consists of two phases; model generation and model evaluation. In the first phase, the model generates a performance image structure for each class and object in the system. Each structure will contain all the information required to calculate the time cost function. All the required communication activities will be handled by the matching methods of the communication class. In the second phase, the hierarchical model simplifies the complexity of evaluating the total time cost while preserving the concepts of the OO design. Both the computational and communication costs are considered as well as the cost of data updating among objects within the distributed system. In conclusion, this model provides a simple way to design and develop efficient DOOS system with performance stability.

5. References

- [1] P. Coad, and E. Yourdon, "Object- Oriented Analysis", 2nd Edition, Yourdon Press, Englewood Cliffs, NJ, 1991.
- [2] S.i Matsuoka and A. Yonezawa. "Analysis of inheritance anomaly in OO Conc. prog. Lang.". I. Agha, P. Wegner, and A. Yonezawa, editors, Research Direct. in Conc. OO Prog., pages 107#150. MIT Press Cambridge, Mass., London, England, 1993.
- [3] Bertrand Meyer, "Object-Oriented Software Construction", Prentice-Hall International (UK), Ltd, 1988.
- [4] M. O'Riordan, "Technical Report on C++ Performance", Doc No. 02-0013/N1355, March 2002.
- [5] B. Ostereich, "Developing Software with UML: OO Analysis and Design in Practice", Addison Wesley, June 2002.
- [6] H. Sholl and T. Booth "Software Performance Modeling Using Computation Structures", IEEE transaction on Software Engineering, VOL. SE-1, No.4, Dec 1975.
- [7] D. Smarkusky, "A Performance-Engineered OO Design Framework", PhD Thesis. University of Connecticut, 2000.
- [8] D. Smarkusky, R. Ammar, I. Antonios and H. Sholl, "Hierarchical performance modeling for distributed system architectures", Proc. of the 5th IEEE Symp. on Comp.& Comm., 2000.
- [9] C.U. Smith, "Performance Engineering of Software Systems", Addison-Wesley, Boston, Mass., 1990.
- [10] K. S. Trivedi, "Probability and Statistics with Reliability, Queueing and Computer Science Applications", second edition, John Wiley & sons, Inc., New York, 2002.
- [11] M. Williams and C.U. Smith, "Performance Solutions", Addison Wesley, June 2002.

A Stable Architectural Model for Networks with Trajectory-Dependent QoS

Ahmed M. Mahdy
Computer Science and Engineering Dept.
University of Nebraska-Lincoln
Lincoln, NE 68588 USA
amahdy@cse.unl.edu

Mohamed E. Fayad
Computer Engineering Dept. San José State University
San José, CA 95192 USA
m.fayad@sjsu.edu

Abstract

QoS-based networks have been a focus of much research for the last several years. The wireless Internet as well as cellular networks are major domains where QoS is a viable option. QoS support is tied to the ability of predicting user movements. Resources that are required to maintain a desirable QoS need to be known a priori. In this context, two basic approaches based on the user location and user history have been developed and propagated. In this paper, we present a hybrid approach for prediction and show a stable model for path prediction in networks with trajectory-dependent QoS.

1. Path Prediction

Two major approaches to this problem have evolved. The first approach is *location update* in which the system periodically records the coordinates of the mobile user and predicts a possible trajectory based on the physical location [1]. The second approach is *location prediction* in which the system predicts the user movement depending on some movement model [2].

Path prediction based on the location update approach is a function of the user location with no consideration to its history. A cell is assumed the next destination of a user if the location of the user is nearest to this cell than any of the other neighbor cells. On the other hand, a location prediction based network anticipates the destination of a user based on its history. The thought is that if a mobile user uses a route through the network, it most probably will eventually reuse it, for example it maybe the work route. By maintaining a history profile for

each mobile user, the system can anticipate the destination of a user. User history can be easily logged since the user ID (i.e. the electronic signature of the mobile card) is invariable. In general, the predicted destination is informed of a possible admission of a new user so that required resources to maintain a certain QoS are reserved.

2. Hybrid Prediction

We present a hybrid approach that uses both the current location and user history approaches to predict a destination. This applies the location prediction approach as long as the current route of the user matches one of its stored paths. If the current route does not match any route in the user history profile, the system calculates, for every neighbor cell, an integer value that is a function of two components; namely *shortest distance* and *favorite destination*. Shortest distance is the distance between the current location of the user and every cell. The cell with neighbors, usually “6” in cellular networks, and the farthest cell is assigned “1” (i.e. every cell has 6 neighbors except boundary cells, which are dealt accordingly). Favorite destination is an ordering of neighbor cells based on the history of a user; a cell that was the most selected destination for this user gets highest priority and is assigned a high value. Based on these two values, a decision is made according to some criteria.

Hybrid Approach

- search user history using current route
- if a match found, predict destination according to history.
- if no match
 - calculate distance to all neighbors
 - order neighbors according to distance
 - order neighbors according to history
 - predict destination with highest combined order

2. Stable Model for Path Prediction

We model the prediction of trajectory in QoS-based networks using software stability [3]. A SSM is presented in Figure 1. In this model, we identify *QoS Guarantee* and *Performance Optimization* as the Enduring Business Themes (EBTs) of the model. These two objects reflect the purpose of the prediction process. *Prediction* and *Criteria* are the Business Objects (BO) of the model. While *Prediction* can also be seen as a BO pattern, *Criteria* is a regular BO. The model Industrial Objects are *Decision*, and possible instances of *Criteria*; *History*, *Hybrid*, and *Location*. The process of path prediction aims at *optimizing* the network performance by providing the network users with a *guarantee* on QoS. The EBTs of the model accordingly reflect this fact. QoS is guaranteed based on the *prediction* of user movements and its possible destinations. The logic used for the prediction process depends on the prediction *criteria* of the network. Some networks may implement the user *history* approach while others could adopt the user *location* or the *hybrid* approach. The outcome of the prediction can be seen as a *decision* on what is the most probable destination of the user and the actions that the network should accordingly take.

The Prediction object can be viewed as a recurring element that represents a pattern that can be applied independently to similar applications that belong to the same domain. We identify *Path*, *User*, and *Network Model* as the BOs. The Destination, Location, Profile are the corresponding IOs.

References

- [1] B. A. Akyol and D. C. Cox, "Signaling alternatives in a wireless ATM network," *IEEE Journal on Selected Areas in Communications*, vol. 16, pp. 35–49.
- [2] J. Ho, Y. B. Lin, and I. Akyildiz, "Movement-based location update and selective paging for PCS networks," *IEEE/ACM Trans. Networking*, vol. 4, pp. 629–639.
- [3] M.E. Fayad, and A. Altman, "Introduction to Software Stability," *Communications of the ACM*, Vo. 44, No. 9, September 2001, pp 95-98.

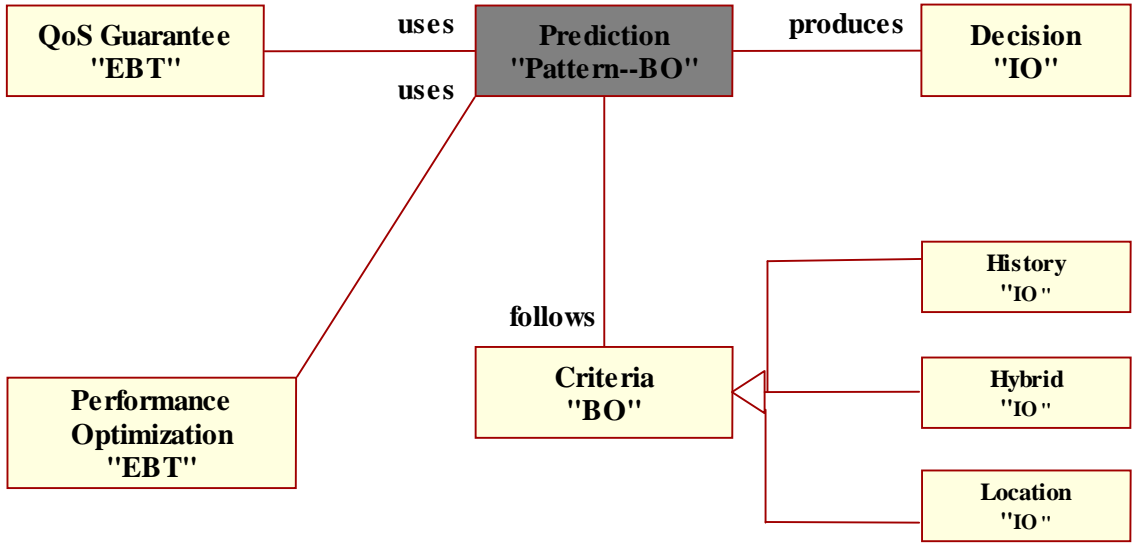


Figure 1. Path Prediction Software Stable Model

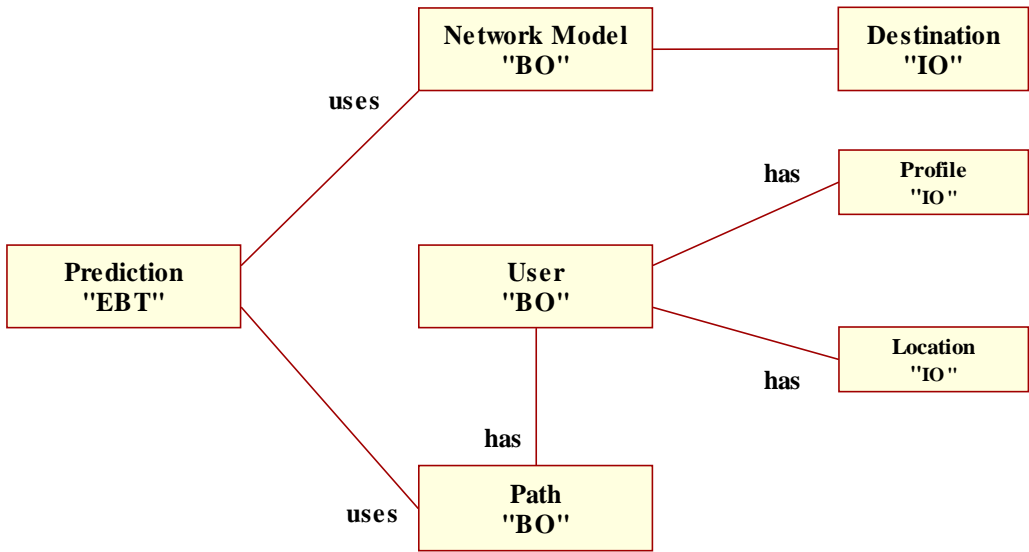


Figure 2. Prediction Pattern

Monitoring and Reuse Software Patterns Analysis in Maude

Mohamed Larbi Rebaiaia

Computer Science Department, University of Batna, Algeria

E-mail: rebaiaia@arabia.com

Abstract

Reusability is considered by the software engineering community to be one of the main challenges for the next near future. During many decades reusability was treated as a concept joining object-oriented inheritance and modularity. There were many approaches trying to find relations between a concrete part of a software and its abstract representation to facilitate the modules reusability. Unhappily, such thinking approach fail to find a correlation between class of objects which can be reused and those in contrary could fail to be reused.

To facilitate reuse in object-oriented views, we consider problems from their patterns analysis side rather than from the inheritance side.

In this paper we demonstrate the similitude between patterns modeling and concurrent languages like Maude. We show that the algebraic interpretation of a class of objects could guarantee a reusable core. This, is due to the formal methods based essentially on a mathematical and proved concepts. The demonstration is based on the description of a naïve problem using pattern diagram classes and their transcription in Maude language. This fact, encourage the achievement of an executable form of the specification to well understand its behavior. Such behavior is insured using the notion of rewriting rules.

1. Introduction

In [9], James O. Coplien defines a pattern as a piece of literature that describes a design problem and a general solution for the problem in a particular context. He take back some words of Alexander that : Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

The pattern is, in short, the description (model) of a design, the rules which give the capacity to create the design, and when we must create it. The pattern that models a specific problem should be easily reused to represent the same problem regardless of its context

There are two major schools in accomplishing software analysis of a product (say the product P for example). The first one details the design of the product

according to the famous V or Y software engineering phases, from requirement definitions until the release phase. This study concerns only the product P, and if, one, wants to design another product, say, Q, he has to repeat the engineering phases for the product Q, even, if the product Q is approximately different from the product P. Thus, a lot of time and money are lost stupidly. The second situation, try to study the product (Q for example), using the pattern analysis and especially the Stability Software Model (SSM) introduced by Fayad et al. [1, 2]. Since the product Q resemble to the product P previously designed, it is more interesting, instead of designing Q, the designer try to notice all the similitude's and then, redesign just the different parts of the product by reusing the common elements. This, is what it can be approached by software patterns.

The stability software pattern try to construct, first, the stable core (EBT) of the design, which can be reused for another problem. Second, the middle one (BO), which plays the role of the interface between the core and the outside elements of the design. In general, the middle part didn't change except in a rare opportunities and the last one, which is the outside interface composed by what we call IO objects. These elements can be removed without disturbing the system functioning.

The idea behind the stability patterns is to design the system by considering the unchanged elements, and those, which can be replaced by other one's. For more details, suppose we want to design (as above) the product Q, which is similar to the product P (already designed) except for some elements. If we want to reason according to the stability pattern, the P design is decomposed into the famous three parts (EBT, BO, and IO). Since the Q design is almost similar to the P design, thus we have to arrange ourselves in the sense, that the difference will be pulled outside. Thus, instead to redesign the product, all we have to do, is just to design the IO elements part and then to reuse the EBT and the BO parts. Therefore, separating analysis models from design and implementation issues is one of the keys for accomplishing pattern stability and generality.

2. The Maude Language

This section provides a brief introduction to our specification language, Maude [5]. Maude has two parts: one which defines the basic data types using order-sorted equational specification [3, 4] and Membership equalities [6], and another which specifies states (so-called configurations) and state changes.

In the state-dependent part of Maude one writes object-oriented specifications consisting of an import list (protection item), a number of class declarations, message declarations, equations and transition rules. An object of a class is represented by a term comprising an object identifier (of sort `ObjectId`), a class identifier and a set of attributes with their values; e.g., $\langle A : \text{Account} \mid \text{bal} : N \rangle$ represents an object of class `Account` with identifier `A` and attributes `bal` with value `N`. A message is a term of sort `Message` (in mixfix notation) that consists of the message's name, the identifiers of the objects. The message is addressed to and, possibly, parameters; e.g., the term $(\text{transfer } M \text{ from } A \text{ to } B)$ is a message. A configuration is a multiset of objects and messages. Multiset union is denoted by juxtaposition. State changes are specified by transition rules (keyword `rl` or `cr`), e.g. :

```
(transfer M from A to B) < A : Accnt | bal : N > < B :
Accnt | bal : N' > =>
< A : Accnt | bal : (N - M) > < B : Accnt | bal : (N' + M)
>
if N > M .
```

is the expression of a pure concurrency between two processes, say, two accounts, named `A` and `B` respectively. The money transfer operation is managed by the message $(\text{transfer } M \text{ from } A \text{ to } B)$. The result of the operation is expressed by the right hand-side of the rewriting rule.

As an example of a specification, let us give the specification of bounded buffers and explain it subsequently. The specification `CONFIGURATION` specifies the basic data types of objects, messages and configurations. The empty state, i.e., the element of sort `CONFIGURATION` is denoted by `empty`. `LIST` specifies the sort `List` of finite sequences together with a juxtaposition operation where adding an element `E` to a list `C` on the left is written `E C` and a list consisting of a list and a single element is written `C E`. `NAT` contains the specification of natural numbers (`Nat`) and the sort `Nat` for natural numbers strictly greater than zero.

```
fmod BUFFER .
protecting NAT .
protecting LIST .
protecting CONFIGURATION .

class Buffer .
max : Nat .
cont : List .
op get _ replyto _ : ObjectId ObjectId -> Message
op to _ answer is _ : ObjectId Elem -> Message
op put _ into _ : Elem ObjectId -> Message }

vars B R : ObjectId .
var E : Elem .
var C : List .
var M : Nat .
var ATTS : Attributes .
cr [P]: (put E into B) < B : Buffer | cont : C, max :
M, ATTS > =>
< B : Buffer | cont : E C, max : M, ATTS >
if length(C) < M .
rl [G]: (get B replyto R) < B : Buffer | cont : C E,
max : M, ATTS > =>
< B : Buffer | cont : C , max : M, ATTS >
(to R answer is E) .
```

Figure 1. A Functional Module Called `BUFFER`.

The class `Buffer` has two attributes, `max` is the capacity of a bounded buffer and `cont` stores the buffered elements. The variable `ATTS` collects according to the syntax supported by the Maude system-attributes not mentioned in a rule or additional attributes.

A buffer as a storage device may react to two messages: `put` and `get`. The `put` operator stores an element in the buffer, and `get` removes the oldest element being stored in the buffer and sends it to a "receiver". The transition rule with rule label `P` says that an object of class `Buffer` can react to a `put` message only if the actual number of objects being stored, `length(C)` is smaller than the capacity `max`. Sending a `get` message triggers not only a state change of buffer `B` but also initiates an answer message to `R` which contains the result (an element). Note, that a `get` is only accepted if the buffer is not empty, i.e., if attribute `cont` contains a structure "`C E`" indicating that there is at least one element part of the list.

3. Writing Patterns using a Specification and Executable Language Maude

The class diagram as shown in the Figure 2, represents a fictive Library *Pattern* of a university. The Figure 2 didn't include all the details of a real representation of the library functionalities, we restrict our paper to just some parts.

We will start our presentation by an informal specification followed if necessary by a description using Maude language. As the UML diagram (Figure 2) [7], depicts classes details of a model : the class name, its attributes and the methods, these elements will be written in Maude plus some others distinctive operations and rules to enrich the semantics of software Pattern from static behavior to a dynamic one.

From the diagram (Figure 2), we can identify some main roles, say, *reader*, *calendar*, *Item*. There are three kinds of readers, namely Academic staff, Undergraduate students and Postgraduate students and two kinds of items, namely book and periodical. This kind of example has been proposed by different papers like [8]. Such example needs some introductions.

All the readers (academic staff, students) have some rights (e.g. borrowing books and periodicals) and obligations (e.g. borrowing items must be less than 10 books and 5 periodicals), this is restricted to the attributes *maxLoans* and *loanPeriods* of the class *Library*. Reader who retard to return an item in the required time may be penalized according to the number of days of the delay, for example each day is equivalent to one point and the total of points corresponds to a certain sum of money. The reader may be suspended by the librarian if he don't want to pay the infraction, the information concerning this situation is stored in the attributes *finer* and *suspendedUsers* of the class *Library* and the payment order is insured by the object *Pay_fines*, so the librarian can read the name of the reader, the *pay_date* (payment date due) and the *amount* of the fines. Thus the number of borrowed items and the currents fines are kept respectively in the attributes *borrowedItems* and *finer* of the class *reader*. The attribute status of class *Item* defines the status of an item with possible values *OnLoan*, *free* or *disposed*. The attribute *location* show the arrangement of an item in the library shelves. When an item required by a reader is not available, the reader can reserve it. This operation is provided by the class *BookReservation* or *PeriodicReservation* and the information is stored in the attributes *reservationDate* and *observation*. The reader may cancel its reservation.

This presentation of the problem is just an informal overview and could be extended to others parts. Thus, we are only concerned by showing that software pattern

modeling could be approached by formal specification and, thus we can pass from a semi-informal description which is a static one and we surpass this rigid situation by imposing a certain dynamic behavior. Finally, we want that software pattern could be managed using a dynamic semantic.

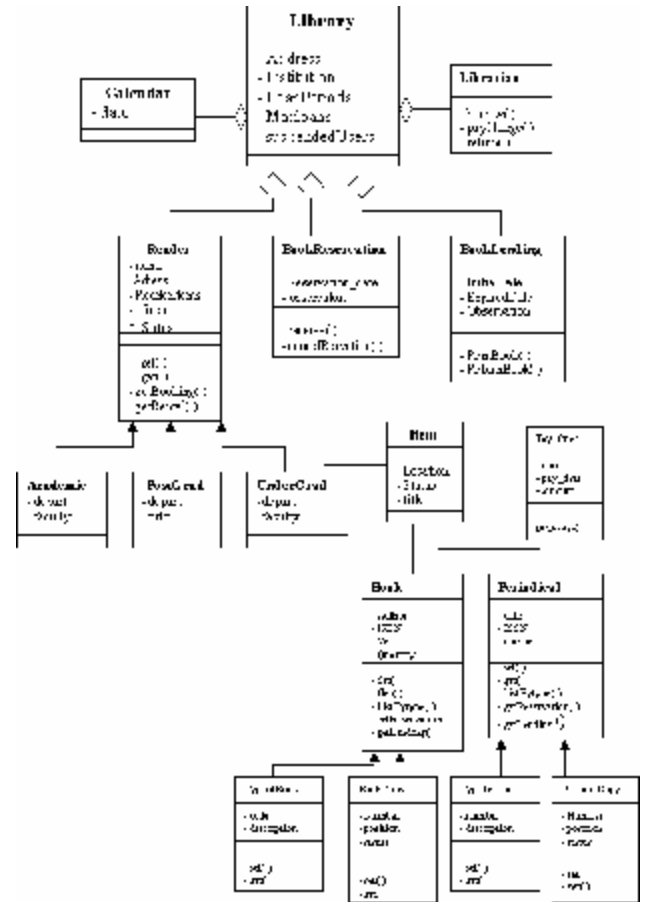


Figure 1. Structure of a Rental Pattern for a Library Community Service

Suppose that we want to describe a part of the library system using Maude language, this could be done by the following representations.

First we present the reader and we can take for example an instance of this class, say, for example Ryan.

```
class reader | code : Nat, name : string, address : string,
status : ReaderStatus, ReaderItem : Nat, finer :
pointsNumber .
```

```
Including MACHINE-INT .
Including Qid .
Including CONFIGURATION .
sort Reader .
subsort Reader < Cid .
```

```

var Academic-staff PostGrad-stud UnderGrad-stud :
Reader .
var reader : à Reader .
var Ryan : Oid .
rl [change_status] <Ryan : reader | reader_Status : nil>
<L : Librarian >(set_Suspended) =>
<Ryan : reader | reader_status : suspended><L :
Librarian | suspendedUsers : reader_name> .

```

The above rule shows traduce the concurrent behavior of the objects belonging to the reader class and those of the librarian class. The interaction influenced by the message “set_suspended” changes the reader Status from “nil” to “Suspended”, and, add the name of the reader to the database list of the suspended readers.

As the reader belongs to one of the following classes, they are declared in Maude as follows :

```

class academic | dept : string, faculty : string .
class postGrad | dept : string, author : string .
class undergrad | faculty : string, dept : string, tutor :
string .

```

The above three classes are declared as subclasses of the class reader :

```

subclasses academic postGrad undergrad < reader .

```

One of the importance classes which is the interface between the reader and the hard means—the books and the periodicals is the class item. The three are declared as follows, with a simple difference that the book and the periodical classes are systematic to be considered as subclasses of the item class :

```

class Item | location : string, status : ItemStatus, title :
string .
class Book | author : string, ISBN : string, quantity : Nat,
year : Nat, code : string, observation : string .
class Periodical | ISSN : string, date : Date, number : Nat

subclasses Book Periodical < Item .

```

The following operation permit to decrease the qtyAvallable of a book by one if the book has just been rent by a reader. Such rule expresses the easiness of Maude system to deal with a certain number of concurrent objects, here, for example four classes synchronize themselves to decrease the number of items in the library :

```

Rl [decreaseqtyavailableOfaBook] <R : reader | code :
?, ISBN = ?>(getBookin) <BookLend : BookLending |
InitialDate : date, ExpiredDate : date + 15, Observation
: “nil”>(RentBook)<book : Book | Author : “Peter

```

```

Moses”, ISBN : “#=1252514”, Year : 2004, Quantity :
N> (getLending) =>
<book : Book | Author : “Peter Moses”, ISBN :
“#=1252514”, Year : 2004, Quantity : N - 1>

```

The above rule didn’t gives all the information about the interaction between reader object, book object and the relation book lending, we can for example extend the interaction toward the subclass TypeOfBook belonging to the class Book and to convey the concurrency until the librarian and the library objects to set the databases.

It is possible to extend both book class and periodic class and to detail their characteristics, for example the type of books and the number of instance of a book. By similarity we declare the same things for the periodical items as follows :

```

class TypeofBook | code : string, description : string .
class BookCopy | number : string, position : string,
status : bookStatus .
subclasses typeOfBook BookCopy < Book .

```

```

class TypePeriodic | number : string, description :
string
class PeriodicCopy | number : string, position : string,
status : PeriodicStatus .

```

```

subclasses typePeriodic PeriodicCopy < Book .

```

```

class BookReservation | reservationDate : date, reader :
Oid, item : Oid, observation : string .
class Booklending | lendingDate : Date, initialDate :
Date, Observation : string, reader : Oid, item : Oid .

```

```

class PeriodicalReservation | reservationDate : date,
reader : Oid, item : Oid, observation : string .
class PeriodicalLending | lendingDate : Date, initialDate
: Date, Observation : string, reader : Oid, item : Oid .

```

```

class Calendar | date : Date .
rl [new-date] : <C : calendar | date : D> => <C :
Calendar | date : D + 1> .

```

```

class Library | intitution : string, address : string,
loanPeriod : maxLoans : Pfun(Cid,Nat),
suspendedsers : set(Oid) .

```

```

class Pay_fines | reader : string, pay_date : Date,
amount : money .

```

In the diagram class (Figure 2), the class Librarian does not have any attributes, but it offers three methods declared as messages created to authorize the

communication and to exchange information and notifications.

```
class Librarian .
msg borrow return : Oid Oid Oid => Msg .
msg payCharges : Oid Money Oid => Msg .
```

Each message will be named after the operation it models, and the first and last arguments of the message will be the identifiers of the called objects, respectively. The rest of the arguments correspond to the arguments declared for the operation. For example the following declaration (as above) :

```
msg payCharges : Oid Money Oid  $\hat{e}$  Msg .
```

determines the charges to be paid by the reader to the Library and the order is given by the Librarian.

We stop now, our explanation concerning the direct correspondence between Pattern software modeling and Maude specifications.

4. Reuse facility in Maude

Suppose we would like to reuse the object buffer already written in Maude specification, to construct a new extend buffer, globally similar to the first one with some minor modifications of its properties and call it `bufferNew` that accepts `put`, `get` the replication of the original messages, and `getNew`, a new message that triggers a retrieval of two elements from the bounded buffer.

Assume, that we would like to start with the specification of the first buffer (maybe because it is implemented in the standard library) and implement the other second buffer (`bufferNew`) by reusing the specification of the first buffer.

The specification `BUFFER_NEW` containing a class `Buffer` is the starting point.

```
fmod BUFFER_NEW in
protecting BUFFER .
```

```
class BufferNew { cont : List }
op get _ replyto _ : ObjectId ObjectId  $\hat{a}$  Message
op to _ answer is _ : ObjectId Elem  $\hat{a}$  Message
op put _ into _ : Elem ObjectId  $\hat{a}$  Message
vars B R : ObjectId
var E : Elem
var C : List
var ATTS : Attributes
```

```
rl [P]: (put E into B) < B : Buffer | cont = C , ATTS >
```

```
=>
< B : Buffer | cont = E C , ATTS > .
```

```
rl [G]: (get B replyto R) < B : Buffer | cont = C E , ATTS
> =>
< B : Buffer | cont = C , ATTS > (to R answer is E) . } }
```

First, `BdBuffer` has been implemented as in [10] by reusing `Buffer`. Since the bounded buffer is more restricted to acting and to reacting than the buffer, we employ subconfiguration for reuse. We establish the relation by:

```
eq < B : BdBuffer | max : M , cont : C , ATTS > = < B :
Buffer | cont : C , ATTS > .
```

to verify that `BUFFER_NEW` simulates `BUFFER`. Thus, class `BdBuffer` can be implemented by reusing `Buffer`, more precisely, the value of attribute `cont` of `BdBuffer` can be replaced by an object of class `Buffer`. The reuse relation is `BdBuffer cont : Subconfiguration of Buffer`.

The second step is to implement `get2` by subconfiguration as a sequential composition of two `get` messages and defined as follows :

```
eq (get2 B replyto R) < B : Buffer | ATTS > = ((get B
replyto R);;(get B replyto R)) < B : Buffer | ATTS >
```

```
eq (to R answer is E1 and E2) = (to R answer is E);;(to
R answer is E) .
```

Then relates all configurations containing a `get2` to configurations containing two `get` messages.

5. Conclusion

Maude is an executable rewriting language well suited for the specification of the diagram classes associated with stability software model. The semantic of Maude language is based on the rewriting logic and the equational logic, two well proved concepts, which permit the expression of the dynamic behavior of concurrent models and thus interpret the communication between objects as a executable sequential steps.

In this paper, we show that Maude system and its language make easier to achieve the integration of different models and patterns in a rigorous way. We are sure that Maude is based on a Universal algebras and then can be accommodated to different problems and strategies. In addition to the integration of Maude as a

programming language for the patterns forms and for the reusability of different parts of a specification. Another important way, is to use the reflective capabilities of the rewriting logic, a very interesting concept which can construct a theoretical plan to deal easily to design software components and to reuse them.

6. References

[1] M.E. Fayad, and A. Altman, "An Introduction to Software Stability," *Communications of the ACM* (44:9) 2001, pp 95-98.

[2] A. Mahdy and M.E. Fayad, "A Software Stability Model Pattern", Proc. of the 9th Conference on Pattern Language of Programs (PLoP02), Illinois, USA, Sept. 2002.

[3] J. Meseguer, "A Rewriting Logic as a Unified Model of Concurrency," In Proceeding of the Concur'90 Conference, Amsterdam, August 90, Page 384-400, Springer LNCS 458-1990.

[4] J. Meseguer and J.A. Goguen, "Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operation", Technical Report, SRI-CSL-89-10, July, 1989.

[5] J. Meseguer, "Membership Algebra as a semantic framework for equational specification", in F. Parisi-Presicce, ed., Proc. WADT'97, pp. 18-61, Springer LNCS 1376, 1998.

[6] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada. A, "Maude Tutorial, March 2000", Tutorial distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. Presented at the European Joint Conference on Theory and Practice of Software, ETAPS2000, Berlin, Germany, March 25, 2000. <http://maude.csl.sri.com/tutorial>

[7] G. Booch, J. Rumbaugh, and I. Jacobson, "Unified Modelling Language," User Guide .Addison-Wesley,1999.

[8] F. Duran and A. Vallecillo, "Writing ODB Information specification in Maude", Available at <http://www.lcc.uma.es/~av/Publicaciones/01/ITI-2001-10-pdf> .

[9] J O. Coplien, "Software Patterns", ISBN 1-884842-50-X, SIGS Books &Multimedia, New York, 2000.

[10] U. Lechner, "Constructs, concepts and criteria for reuse in concurrent object-oriented languages. In Fundamental Approaches to Software Engineering," 1996.

