ELSEVIER

The Journal of Systems and Software 43 (1998) 85–91

The Journal of Systems and Software

A case study of evolution in object oriented and heterogeneous architectures

Vaclav Rajlich *, Shivkumar Ragunathan

Department of Computer Science, Wayne State University, 431 State Hall, 5143 Cass Avenue, Detroit, MI 48202, USA

Accepted 24 March 1997

Abstract

In order to properly understand a technology, it is important to know not only its strengths but also its limits. In this paper, we investigate the limits of object oriented technology. We present a case study in which we compare two different architectures for the same program – a student registration system which belongs to the domain of reactive repositories. We started our case study with a model produced by Object Modeling Technique (OMT), and used it for two different implementations: a homogeneous one (HOA) and a heterogeneous one (HTA). The HOA results in an object oriented system consisting of classes and their dependencies, while the HTA combines traditional imperative programming with a relational database. We evaluated the evolvability of both resulting programs by adding new use cases. HTA proved to be easier to evolve than HOA. In the paper, we discuss possible reasons for this finding. © 1998 Elsevier Science Inc. All rights reserved.

1. Introduction

Object oriented technology is currently a very popular software technology. The programs implemented with this technology have a structure similar to the world they model and interact with, and achieve a high level of reusability. The reuse is credited as the primary reason behind the improvements in programmer's productivity (Lewis et al., 1991). Against this background, we are asking the following question: Is the object oriented technology a universally superior technology, or are there any specific areas where the object oriented technology performs worse than other technologies? As a partial answer to the question, we conducted the case study reported in this paper.

The case study belongs to the growing field of software architectures (Shaw and Garland, 1996), which investigates how various application domains project themselves into the structure of the programs. In our view, the optimal architecture is determined not only by the application domain but also by the programming processes employed, both during the development (Rajlich and Silva, 1992) and later during the maintenance and evolution (Rajlich and Silva, 1996). We believe that it is possible that the program, where the primary emphasis is on software reuse, will have a different architecture than the program where the primary emphasis is on the evolution.

Our case study deals with a student registration system. It belongs to a domain of reactive repositories, i.e. programs which are data intensive, and have a set of predefined operations, sometimes called "use cases" (Jacobson, 1992). The emphasis is on the software evolution, i.e. the modifications of the system in response to the changing requirements. In our case study, we are investigating how difficult it is to add new use cases into an existing repository system. The process of adding additional use cases is a very common process for reactive repositories, where the users often request new use cases which enhance the functionality of the repository.

We started our case study with an OMT model (Rumbaugh et al., 1991), one of the most popular methodologies for modeling object oriented systems. The methodology requires the user to build a model of the system consisting of three parts: the object model, the dynamic model, and the functional model. This model is a starting point for the design, during which the three models are combined and a design document is produced. In this case study, we used the shared OMT model and produced two different designs and two different architectures: Homogeneous Architecture (HOA) which consists of classes of C++, and Heterogeneous Architec-

* Corresponding author. Tel.: +1 313 577 5423; fax: +1 313 577 6868; e-mail: rajlich@cs.wayne.edu.

0164-1212/98/$ – see front matter © 1998 Elsevier Science Inc. All rights reserved.
PII: S 0 1 6 4 - 1 2 1 2 (9 8) 1 0 0 2 4 - 9

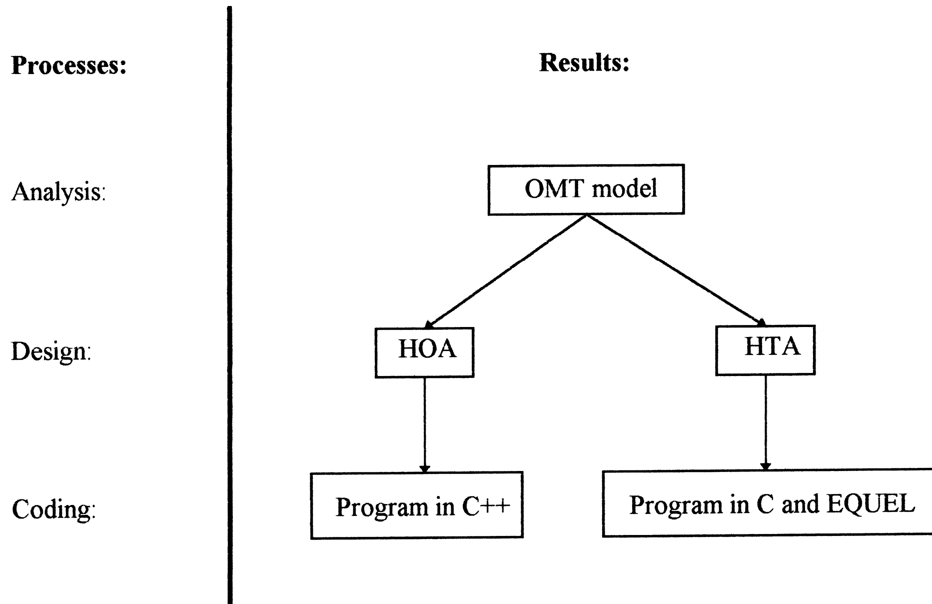

Fig. 1. Processes and results of the case study.

ture (HTA) which combines imperative programming in C with a relational database. For the overview of basic notions and processes of the case study, see Fig. 1.

After both HOA and HTA programs were developed, we evaluated them for their evolvability, i.e. we asked which of the two architectures provides better support for adding additional use cases. For a more complete description of the case study, see Ragunathan (1995). The reader may also be interested in an earlier study of Rajlich and Silva (1996), where they investigated evolution of another architecture for reactive repositories.

In Section 2 we discuss the OMT model for our case study. Sections 3 and 4 cover the HTA and HOA designs and implementations, respectively. Section 5 compares the evolvability of both architectures. Section 6 contains the discussion and conclusions.

2. OMT model of student registration system

The OMT model of a student registration system consists of three parts: The object model, the dynamic model, and the functional model. Following are brief descriptions of the three models.

Object model: The object model of Fig. 2 consists of 10 classes. The superclass Person has three subclasses: Instructor, Administrator, and Student. Class Department represents the department offering the course. Class Course has the course information. A course can have many sections, therefore class Section and class Course have an aggregation association. A classroom can have many sections assigned to it, so we have a one-to-many association there. Each section is taught by an instructor, while an instructor can teach many sections, resulting in a one-to-many association. Each course is assumed to have only one prerequisite. Class Holds stores the information on student holds. Hold is any circumstance which may stop a student from registering a course, like unpaid tuition, or lack of academic progress. An Administrator checks if a hold exists for a student, before registering him for a course. There is thus a ternary relationship between the classes Administrator, Student, and Holds. Each student has a transcript showing performance to date. There is a one-to-one association between the classes Student and Transcript. A student may register for many courses, and for each course the student is allotted a section. A one-to-many ternary association thus results between the classes Student, Course, and Section. An administrator can add course information for courses. A one-to-many association results here.

Dynamic model: The dynamic model shows the time-dependent behavior of the system. Fig. 3 shows the state transitions for the Student Registration System. The user selects a mode of operation: student or administrator. The system asks the user for his identification number and password. It verifies if the user is a valid user and allows him into the system. After that, the user is allowed to select operations of the system. The students are allowed to select only a subset of all operations, while administrators are allowed to select all operations.

Functional model: The functional model shows how values are constructed without regard for sequencing, decisions, or object structure. There is a data flow diagram for each use case of the system. As an example, Fig. 4 shows the data flow diagram for "Initial Registration". The student enters his student identification number. The system checks for holds against the student,

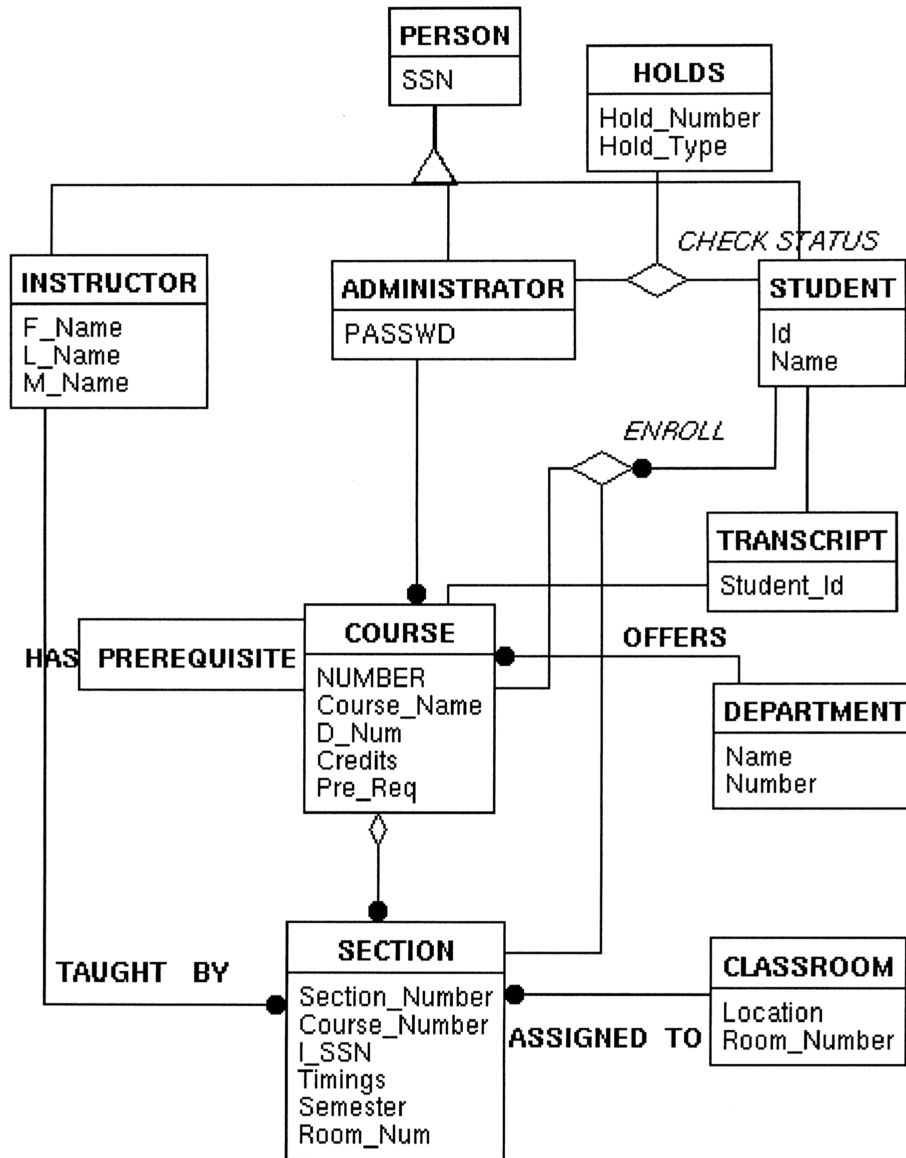

Fig. 2. Student registration system object model.

and also checks if the student has registered earlier. The student then enters the course he wants to register for. The system checks if the course is offered and if it is open. It also checks whether prerequisite courses have been completed by the student. Then the course is registered and a section allotted for the student. The appropriate databases are updated. This model is converted into a design and implementation of HTA and HOA programs, as described in the following two sections.

3. Heterogeneous architecture

In the first part of the case study, we developed the HTA. HTA uses both relational and imperative concepts, hence the name "heterogeneous". The design stage consists of mapping the object model of Section 2 to a relational database, and then mapping the functional and dynamic models to the functions of the system.

Mapping the object model to the relational database: In this mapping, we followed the process described by Rumbaugh et al. (1991). Each object in the object diagram maps to a single table in the relational database. A many-to-many binary association also maps to a distinct table. The primary keys for both related classes and any link attributes become attributes of the association table. A one-to-one association does not need a distinct table for itself: one table has an attribute which is also the key attribute of the other table. Similarly a one-to-many association does not have a distinct table. Here the table for the object on the many side requires the key attribute of the other table. A ternary association al-

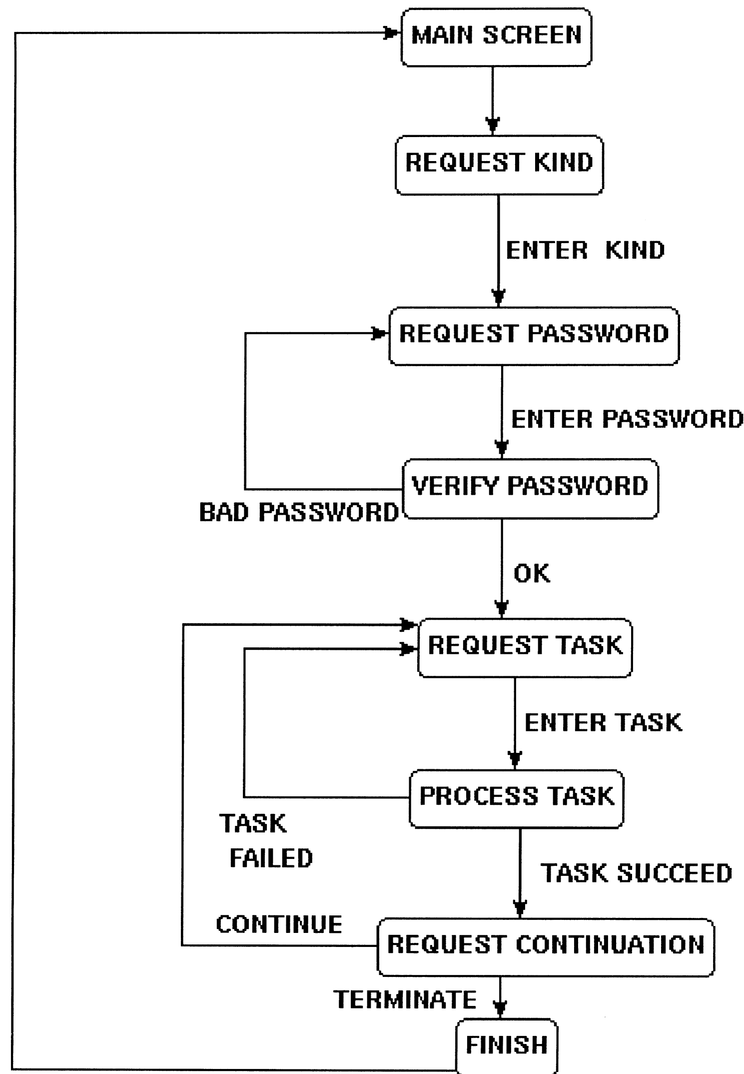

Fig. 3. Student registration system control.

ways maps to a distinct table. The primary keys for all three related classes and any link attributes become attributes of the association table. There is more than one approach to map generalization to tables. The superclass and the subclass each map to a table. The identity of an object across a generalization is preserved through the use of a shared ID. This approach is logically clean and extensible. However it involves many tables, and superclass to subclass navigation may be slow. Another approach eliminates the superclass table and replicates all the superclass attributes in each subclass table. This approach is used if a subclass has many attributes and the superclass has few attributes.

Mapping the functional and dynamic model to functions: Based on the dynamic model and functional model, we implemented the functions of HTA. Each state diagram is implemented by one or more fucntions, and each data flow diagram is implemented by one function.

Implementation: For the implementation of the HTA, Ingres Relational Database and the C programming language were used. Since the database used was Ingres, EQUEL code was embedded in the C code for accessing Ingres information.

Documentation: Documentation in style of Rajlich et al. (1994) was carried out for the code developed. Table 1 shows the annotations for the "Student_Menu" function. Similar annotations were done for all the functions in the program.

4. Homogeneous architecture

The second part of the case study consisted of developing an HOA design of the Student Registration System. The design consists of developing dependencies among the classes, and of converting the actions and activities of the dynamic model and the processes of the

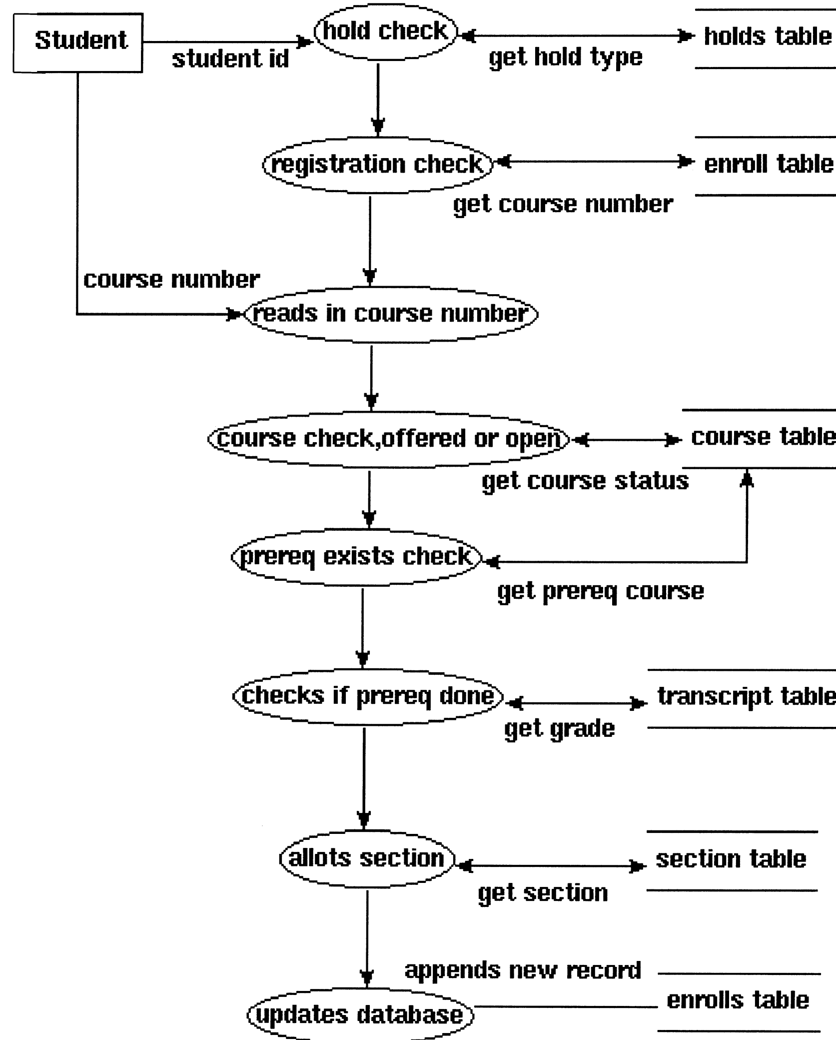

Fig. 4. Data flow diagram for initial registration.

functional model into class methods. We closely followed design method of Rumbaugh et al. (1991) and used OMTool provided by the authors. Hence the resulting class encapsulations are the typical encapsulations which result from an OMT model.

The class methods were selected in the following manner: If an operation involved data of a class A, it was implemented by a method of class A. For example, checking if a course is open involves data of class "Course". Therefore the function "open()" which implements this activity, is a method of the class "Course".

For each use case of the registration system, a method is needed to implement the flow of control. These methods become members of the classes initiating the respective flow of control. The use cases available to students are initiated by the class "Student". For example, the use case "Add Course" is implemented by a method "Add_Course()" which belongs to the class "Student". Similar is the case for administrative use cases.

The OMTool we used produces the specification of classes, attributes, inheritance hierarchy, and associations. Each object in the object diagram forms a distinct class. Each ternary association also forms a distinct class. There is thus a class named "Check_Status" for the association "check status".

Implementation: The rest of the code was written based on the dynamic model and the functional model developed earlier.

Table 1
Annotations for the function Student_Menu

Domain:
This function reads the student identification number, verifies it, then reads the choice for operation desired.
Algorithm:

```
get Student Id
Verify Id
loop forever   {
   get choice
   perform chosen task
}
```

Table 2
Annotations for the class Student

Domain:
Maintains information of a student, the Id and the name of the student.
Representation:
Name: Pointer to the first name of a student. The name of the student is of a maximum size of 10 characters.
Id: Pointer to the Id of a student. The Id is of a fixed size of 9 characters.

Documentation: The code was documented by annotations in the style of Rajlich et al. (1994). For each class, we provided two partitions: an application domain partition, and a representation partition. Table 2 shows the annotations for class "Student". For each method, the documentation has the domain partition and the algorithm partition.

5. Comparisons

In this section we compare the resulting HOA and HTA architectures. In order to interpret the results better, it should be noted that both implementations were done by the same author (Ragunathan, 1995). The author achieved a high level of proficiency in all technologies involved in the study, i.e. C, C++, Ingres, and OMT. The study was not biased by any preference for one of these technologies over the others, as the author was equally comfortable with all of them. In the case study, HTA was created first, and only after its completion HOA was created. Hence any bias caused by the learning experience would favor HOA over HTA.

Another source of bias could have been introduced by the fact that we used a relational database for HTA, while using the flat files for HOA. The reason for this choice was unavailability of a suitable object oriented database at the time of the case study. However a careful scrutiny of the resulting design and code shows that although the use of the flat files may have influenced the size of the code, it did not influence in any way the number of the classes, number of the functions, and the size of the documentation. In Table 3, we use these parameters to measure the complexity of the resulting program. They show that the structure of HTA is simpler than the structure of HOA.

Table 3
Comparisons of the complexity of the whole program

	HTA	HOA
Number of functions	20	52
Lines of annotations	225	442
Number of classes	–	12

Table 4
Comparisons for the use case "Add a course"

	HTA	HOA
Number of functions	5	9
Lines of annotations	61	101
Number of classes	–	5

As a more detailed example, let us look at the data of a specific use case "Add Course", which adds a course for a student. There are substantial differences between HTA and HOA, see Table 4. Similar differences were obtained for all use cases of the system.

We also studied the effect of addition of four new use cases to HTA and HOA. The four use cases are: Get instructor class list, Get courses offered, Remove instructor, and Close course. We modified the OMT model first, and then projected the changes into the design and the code of both HTA and HOA. Annotations documentation was also updated in each case. As an example, consider a new use case "Get_Instructor_List". In that case, HOA needed four new functions and one existing function was modified. Four classes were affected by the change. In the HTA implementation, one new function is introduced and one existing function is modified, see Table 5.

In the table, both the "coding time" and "lines of code added" were adjusted to compensate for the influence of flat files in HOA. Ideally, HOA would use an object oriented database rather than flat files. The adjusted figures of Table 5 include anticipated effect of an object oriented database. We arrived at the figures by subtracting from HOA the numbers corresponding to the implementation of flat files, and replacing them with comparable values for a database.

The cumulative result for all four use cases is in Table 6. The size of the code of the new use cases in HTA ranged from 34% to 53% of the code for equivalent adjusted HOA implementation, with the average being 44%. The effort of adding a new use case to HTA ranged from 51% to 88% of the corresponding effort in adjusted HOA, with the average being 70%. The data indicate that the HTA implementation is easier to understand, evolve, and maintain.

Table 5
Comparison for adding new use case "Get Instructor List"

	HTA	HOA	HOA (adjusted)
Coding time in minutes	40	75	54
Lines of code added	29	94	68
Line of annotations	12	35	35
Number of functions added	1	4	4
Number of old functions reused	0	0	0
Number of old functions modified	1	1	1
Number of classes affected	–	4	4

Table 6
Comparison for adding four new use cases

	HTA	HOA	HOA (adjusted)
Coding time in minutes	135	280	192
Lines of code added	115	395	262
Lines of annotations	50	124	124
Number of functions added	4	13	13
Number of old functions reused	0	2	2
Number of old functions modified	1	1	1
Number of classes affected	–	6	6

6. Conclusions

In this case study, we present an example of a process where a homogeneous architecture based on object oriented programming in C++ is less effective than a heterogeneous architecture based on combination of imperative programming and relational database techniques. The case study uses Object Modelling Technique of Rumbaugh et al. (1991) in the first stage of the development of both architectures, and uses the design techniques from the same book for both HTA and HOA. Hence the resulting encapsulation of HOA is a direct derivative of the process described by Rumbaugh et al. (1991).

The case study found that adding new use cases to the existing program is easier in case of HTA rather than HOA. This finding conflicts with two popular beliefs: that object oriented programming is always better, and that programs utilizing one technology are easier to maintain than the programs utilizing a mixture of technologies.

We speculate that the explanation of the higher evolvability of HTA lies mainly in the fact that HTA is simpler, and there is a direct mapping from OMT model to the structure of HTA code. The mapping preserves the separation between the three models of OMT, and the separation between the individual use cases. This makes HTA easy to understand and evolve, because there is a direct and simple traceability between the model and the implementation.

On the other hand, HOA has each use case divided into several functions which belong to different classes (delocalization of use cases), and this makes the structure of the program hard to understand. Each class in the HOA has parts from all three models: object, dynamic and functional. Each class also has parts of several use cases. For example the class Student has the function member "Menu" that implements the control of the operations available to a student. The declarative statements are from the object model, while the rest of the methods in the class are coming from the different parts of the functional model. When adding a new use case, several classes have to be visited, because each use case is divided into methods of several classes. The only advantage which HOA has is an occasional reuse of already existing methods for the new use cases, but we found few such situations (see Table 6).

The question may arise: How far do the observations of this case study generalize beyond the example we presented, and its obvious limitations? In order to answer this question, please note that every case study is in fact a single data point, and the conclusion drawn from it has a character of an existential proof in mathematics. All what we can say is that there are areas where heterogeneous architectures have an advantage over homogeneous architectures, as documented by our case study.

However we believe that beyond this supported statement, it is reasonable to guess that many reactive repositories for which OMT is a natural model and addition of use cases is the primary maintenance concern would benefit from heterogeneous architectures rather than from homogenous architectures. We also are convinced that heterogeneous architectures where different parts of the system are implemented by different specialized technologies deserve attention of the research community and may have a great potential in the development of evolvable software systems.

References

Jacobson, I., 1992. Object-Oriented Software Engineering, Addison–Wesley, Reading, MA.

Lewis, J.A., Henry, S.M., Kafura, D.G., Schulman, R.S., 1991. An empirical study of the object-oriented paradigm and software reuse. In: Proceedings of OOPSLA Conference, SIGPLAN Notices 26, pp. 184–196.

Ragunathan, S., 1995. A case study of two software architectures. M.S. Thesis, Department of Computer Science, Wayne State University.

Rajlich, V., Doran, J., Gudla, R.T.S., 1994. Layered explanations of software: A methodology for program comprehension. In: Proceedings of Third IEEE Workshop on Program Comprehension. IEEE Computer Society Press, pp. 46–52.

Rajlich, V., Silva, J., 1992. Two object based decomposition methods: A case study. Software Engineering Journal (UK) 7, 35–42.

Rajlich, V., Silva, J., 1996. Orthogonal architecture for evolution. IEEE Transactions on Software Engineering 22, 153–157.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., 1991. Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ.

Shaw, M., Garland, D., 1996. Software Architecture. Prentice-Hall, Englewood Cliffs, NJ.

Vaclav Rajlich is a professor and former chair of the Department of Computer Science at Wayne State University. Before that, he was an associate professor of Computer and Communication Science at University of Michigan in Ann Arbor, and software manager at the Research Institute for Mathematical Machines in Prague, Czech Republic. His current research interests include software maintenance, evolution, and comprehension. He was a founder general chair of IEEE International Workshop on Program Comprehension, general chair of IEEE International Conference on Software Maintenance, and executive vice-president of Technical Council on Software Engineering of IEEE Computer Society.

Shivkumar Ragunathan received a B.Tech in Computer Science from the Indian Institute of Technology in Kharagpur, India in 1993 and an M.S in Computer Science from Wayne State University in 1995. His areas of specialization are Software Engineering and Databases. He currently works for Oracle Corp. in Redwood Shores, CA.