Smell Detection in UML Designs which Utilize Pattern Languages

B. Zamani and G. Butler

***Abstract*—Smell detection is the idea of improving the quality of software by finding and fixing the problems (bad smells) in the source code. The same idea is applicable at the design level. Early detection of the problems in UML design models helps designers produce high quality software.**

In this paper, we present a process called Sign/Criteria/Repair (SCR) for detecting and fixing the smells in the application of a pattern language in a UML design. We investigate how the SCR process can be implemented in three different environments, ArgoUML, Epsilon, and OCLE, and how these tools can help the designer improve a UML model.

***Index Terms*—MDD, pattern language, smell detection, quality assessment.**

I. Introduction

One of the challenges in Model Driven Development (MDD) and its approaches, e.g., Model Driven Architecture (MDA) [1], as a new paradigm in software engineering, is software quality management. Since the models are the main artifacts which drive software development in MDD, quality assessment of models is an important issue. The tool assistance for quality assurance is necessary since merely manual inspection or review is not enough [2].

Smell detection is the idea of improving the quality of software by finding and fixing the problems- called bad smells- in the source code. The same idea is applicable at the design level [3].

In UML documents, e.g., UML 2.0 Infrastructure [4], Well-Formedness Rules (WFRs) help validate the abstract syntax and help identify errors in UML models. UML uses the Object Constraint Language (OCL) [5] for expressing WFRs. However, the semantic and aesthetic checks, if described, are explained by natural language since they are contingent on the underlying domain of the model. Here is where CASE tools come into play and can help designers in finding the problems and checking the quality of the models.

Designers are interested in using patterns while building software. One benefit of using patterns is to help designers communicate their idea. The name *pattern language* comes from the fact that patterns create a vocabulary about the design if we always use the suggested pattern names. Martin Fowler's "Patterns of Enterprise Application Architecture" [6] (Patterns of EAA) can be considered as a pattern language for designing enterprise applications.

Manuscript received August 14, 2007; revised June 17, 2008.

B. Zamani is with the Department of Computer Engineering, University of Isfahan, Isfahan, I. R. Iran (e-mail: zamani@eng.ui.ac.ir).

G. Butler is with the Department of Computer Science and Software Engineering, Concordia University, Montreal QC, Canada (gregb@encs.concordia.ca).

Publisher Item Identifier S 1682-0053(09)1641

In this paper, our goal is to investigate how easy it is to do smell detection and quality assessment on models that utilize Patterns of EAA as a pattern language. We present a process called Sign/Criteria/Repair (SCR) for verifying the application of patterns in a design. SCR can be viewed as a critiquing process which helps designers find problems (bad smells) in the application of patterns in their design and follow the wizards for repairing the problems. We investigate how the SCR process can be customized for Patterns of EAA, how this process is implemented in different environments, and how these environments can help the designer in detecting and fixing the problems in a design model. As our case studies, we have selected six EAA patterns and three state-of-the-art environments, ArgoUML [7], OCLE [8], and Epsilon [9].

There exist several works related to smell detection in a design. The works in the first group focus particularly on detecting design patterns [10]-[13], while works in the second group focus on quality assessment of models [2], [14], [15]. Besides these two groups, the most comprehensive design critiquing system presented until now seems to be ArgoUML [7]. Our work is close to the first group considering the fact that we intend to detect patterns. However, our work differs from those in the sense that none of the above works has used the UML Profile [4] technique (and stereotypes) as a powerful tool for detecting model elements.

The rest of the paper is organized as follows. In Section II, we briefly introduce Patterns of EAA and discuss how it can be considered as a pattern language. Section III presents the idea of smell detection. In Section IV, the SCR process for detecting smells in using a pattern language in models is introduced. In Section V the case studies of integrating the SCR process into selected tools are described. Finally, in Section VI, we conclude the paper.

II. Patterns of EAA as a Pattern Language

The Software community has borrowed the words "pattern" and "Pattern Language" from the work of architect Christopher Alexander [16]. To quote "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

By adapting this definition, software experts have defined (discovered) hundreds of patterns as solutions to recurring problems in software design. Each pattern author has her own pattern form, which mostly consists of the following items: the name of the pattern, the problem, the solution, and the examples of pattern usage. By documenting the patterns and the relationship among them,

Table Data Gateway: An object that acts as a Gateway to a database table. One instance handles all the rows in the table.

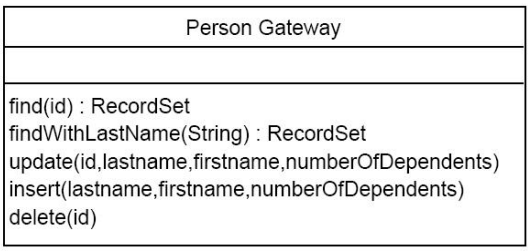

Fig. 1. The Table Data Gateway Pattern [6].

in fact pattern authors are defining a language, called Pattern Language, which could be used by the designers in developing new software systems [17]. If we consider each pattern as a recipe for a solution, a pattern language is a set of recipes for a whole system. Pattern names play a crucial role in a pattern language, because the designers can use those names as a vocabulary that helps them communicate more effectively [6].

Among many available sources of documented patterns, the most famous one is the seminal book on design patterns known as "Gang of Four" (GoF) book after its four authors [18]. The annual conference on Pattern Language of Programs (PLoP) [19], which is now in its 16th, is another source dedicated to pattern authors to present their works. Some of the patterns presented in the conference are published in PLoP Design book series [20]. There are several classifications for patterns. Each class can be considered as a family of related patterns. For instance, in the GOF book, the patterns are classified by two criteria, purpose and scope. Purpose can be creational, structural, or behavioral and reflects what a pattern does. Scope specifies whether the pattern applies to classes or objects [18]. In [20], patterns are divided into six parts, design patterns, distributed patterns, and architecture patterns, to name a few.

In this paper, we focus on the enterprise architectural patterns presented in the book "Patterns of EAA" [6]. Over forty patterns are defined in the book as solutions to recurring problems, which are applicable to web-based enterprise applications. The set of patterns introduced in the book are related to each other and can be used to describe an application as a whole. Therefore, this set can be viewed as a pattern language for the design of web-based enterprise applications.

The Patterns of EAA are decomposed into three layers, based on the idea of three-tiered architecture for client-server platforms, i.e., presentation, domain, and data source. The presentation layer is responsible for user interface, the domain layer deals with domain logic and business rules, and the data source layer is related to communicating with the database of the system.

The pattern language defined in the Patterns of EAA book helps the designer in deciding what patterns to use when designing an enterprise application. There are different alternatives for different layers of the application, and there are some recommendations. For instance, if you are using the Transaction Script pattern for the domain layer, then there are two alternatives for the data source layer, the Row Data Gateway pattern and the Table Data Gateway pattern.

Record Set: An in-memory representation of tabular data.

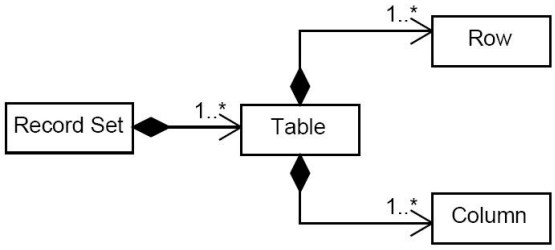

Fig. 2. The Record Set Pattern [6].

Patterns of EAA is a well-known source to be used by designers of enterprise applications, albeit, applying a pattern needs expertise and the novice designers are vulnerable to making mistakes in using patterns.

To make our case studies simple and concrete enough, we have selected six patterns: four patterns from "Data Source Architectural Patterns" including Table Data Gateway, Row Data Gateway, Active Record, and Data Mapper, as well as two patterns from "basic patterns" including Money and Record Set. In the following, we summarize the definition of the Table Data Gateway and Record Set patterns which are the target of our discussions in the coming sections.

As indicated in Fig. 1, the essence of the Table Data Gateway pattern is that it holds all the SQL commands, e.g., selects, inserts, updates, and deletes, in the form of a simple interface, for accessing a single table or view. Others call these methods for interacting with the database. Each method gets the input parameters and maps them into a SQL call which is executed against a database connection. Therefore, the developer does not need to be worried about writing SQL codes.

There are two alternatives to return the multiple data items resulted from SQL queries, a map or a Record Set. Record Set is another pattern in EAA pattern language. For people who are familiar with two-tier applications, using a Record Set is more convenient. As Fig. 2 shows, the Record Set pattern provides an in-memory structure which is exactly the same as the result of an SQL query. This brief description shows how the designer is able to utilize a pattern language in designing a system.

III. SMELL DETECTION

Code smells are symptoms that when present may indicate that source code is unhealthy and needs to be revised. Smell detection is the idea of improving the quality of software by finding and fixing the problems (bad smells) in the source code. The same idea is applicable at the design level. As it is discussed in the next section, we have selected the term "repair" to refer to the task of fixing the problems found in a model. In this context, repair could refer to fixing an error in applying a pattern in the model, adding missing items to the model, or making the design better. However, when it comes to fixing the structural problems in the source code, the term "refactoring" has more advocates in the software community, since refactoring preserves behavior.

Refactoring is a well-known technique in software engineering that improves a software design by applying a series of small behavior preserving transformations [3].

TABLE I
SOME CODE SMELLS AND CORRESPONDING REFACTORINGS IN ECLIPSE

Code Smell	Rafactoring
Two classes have common interfaces	Extract Interface
Long method	Extract Method
Need to change parameters of method	Change Method Signature
Inappropriate name for class, method, or attribute	Rename
Same method/attribute in subclasses	Pull Up

Nowadays, there is a tendency towards the tools and IDEs that are equipped with plug-ins for detecting smells and applying appropriate refactorings automatically. In the "Refactoring Home Page" [21] a catalog of common refactorings along with related tools and books can be found. For instance, Table I shows some of the common code smells and suggested refactorings that are applicable by Eclipse IDE [9] on Java code.

As MDA is becoming a dominant paradigm in software engineering, more attention is given to the design rather than the code. Detecting smells at design level helps produce high quality code. Since most designers are using CASE tools for their designs, tools that are capable of detecting smells in the design are more accepted by designers.

So far there has been a few works on implementing tools for automatic detection of GOF patterns [10]-[13], however, our focus in this paper is on detecting smells in using a pattern language which is used for designing enterprise applications.

IV. SCR SMELL DETECTION PROCESS

In this section we propose a simple three-step process for verifying the application of a pattern in a design. The process is called Sign/Criteria/Repair (SCR) and it aims to help the designer, first by detecting smells in using the pattern, and second by repairing the model. The SCR process consists of the following three steps.

1. Sign: The first and most important property of a pattern is its sign. Each pattern has a unique sign. Checking the Sign is the first step of applying the SCR process. If the Sign is present, we continue the process. There are several techniques used in detecting a particular pattern. For instance, [13] uses graph similarity matching. One approach that makes the detection of patterns less tedious, and is selected in this work, is to make use of UML stereotypes [4]. In this approach, Sign is simply indicated by a class which has corresponding stereotype.
2. Criteria: The second property of a pattern is a set of criteria that indicates sound usage of the pattern. If all the criteria are satisfied, a message will be displayed to the designer to inform his/her about using the pattern and stating that the usage of the pattern is correct. For each failed criterion, which reflects a bad smell in the design, a warning message will be reported to the designer.
3. Repair: Repair is dependent on the result of criteria evaluation. For correct usage of a pattern, no repair is needed. For problematic usage of a pattern, if there exists a wizard for fixing any of the smells, upon designer's request, the repair takes place and an appropriate message will be displayed to the designer. Otherwise, a message is shown to the designer in order to inform his/her for fixing the problem manually.

TABLE II
THE STEREOTYPES USED FOR RECOGNIZING PATTERNS

Stereotype	Base Class	Corresponding Pattern
<<tabledatagateway>>	Class	Table Data Gateway
<<rowdatagateway>>	Class	Row Data Gateway
<<activerecord>>	Class	Active Record
<<datamapper>>	Class	Data Mapper
<<recordset>>	Class	Record Set
<<money>>	Class	Money

TABLE III
THE STEREOTYPES USED FOR RECOGNIZING OPERATIONS

Stereotype	Base Class	Corresponding Operation
<<find>>	Operation	Find
<<insert>>	Operation	Insert
<<delete>>	Operation	Delete
<<update>>	Operation	Update

Although, the SCR process is not restricted to any specific class of patterns, in the following, we address the problem of applying the SCR to Patterns of EAA. As it is mentioned above, to simplify the detection of patterns of EAA, we exploit one of the powerful extension mechanisms of UML models [4] by defining stereotypes corresponding to the names of patterns.

Table II shows the stereotypes that are considered for each pattern. In addition, more stereotypes are defined to represent specific operations in a class. Table III shows the stereotypes that should be used for corresponding operations.

Let us illustrate how to apply the SCR process to detect smells in the application of one of the EAA patterns, the Table Data Gateway pattern.

1. Sign: In the UML class diagram of the design model, there should be a class with stereotype <<tabledatagateway>>. The presence of this stereotype shows the designer's intention for applying the Table Data Gateway pattern.
2. Criteria: The requirements of a sound Table Data Gateway pattern are as follows.
   a. The class needs operations for insert(), delete(), and update(), and usually consists of several find() operations. Each operation is recognized by its name or stereotype. As it is mentioned in previous section, we are using the UML extension mechanism of stereotypes as an alternate way of detecting an operation. For instance, if the name of the operation starts with "insert" or if it has stereotype <<insert>>, then we recognize it as insert() operation.
   b. All of the find operations should have Record Set as return type.
   c. The parameter list of insert operation should be subset of the parameter list of update operation.
3. Repair: According to the Criteria, there are three possible smells in using the Table Data Gateway pattern. If any of the four operations is missing or if the find operations do not have the Record Set return type, then appropriate error message will be displayed

to the user. In case there are wizards for fixing the problems, and the user decides to apply the changes based on the wizards, then the appropriate operations will be added to the class. After fixing each problem, an informative message is displayed to the designer to aware his/her of the change. If there is a mismatch between the parameters, i.e., the third criterion is failed, then a warning message will be displayed to the designer to inform his/her of bad smell in the design that needs to be corrected.

For other five patterns, there are small variations in applying the SCR process, briefly described in the following.

- For Row Data Gateway pattern, there is a gateway class which has attributes that match with columns in the database table, and there is a finder class which uses this gateway to access every record of the database. The result of the find operation in Row Data Gateway is a record instead of a table.
- Active Record pattern wraps a row in a table; therefore it has one attribute for each column of the database table in addition to all above mentioned operations.
- Data Mapper pattern is used to move data between objects and a database, therefore it has the above mentioned operations but no attributes.
- Record Set pattern contains classes for table, row, and column with containment as it is indicated in Fig. 2.
- The Money pattern has two simple requirements: the class should have two specific attributes named "amount" and "currency."

To integrate the SCR process into an IDE, we need to have facilities for describing the Sign, the Criteria, and the Repair, and be able to invoke each of these parts from the IDE. Instead of building such environment, we decided to integrate the SCR process into existing tools for doing our case studies.

V. INTEGRATING SCR INTO MODELING TOOLS

As our case studies for integrating the SCR process, we have selected three state-of-the-art tools, ArgoUML, Epsilon, and OCLE. In this section, for each tool, we provide a brief overview, implementation aspects of the SCR process, and an instant evaluation of the suitability of the tool for the SCR process.

A. ArgoUML

ArgoUML [7] is an open source UML modeling tool that supports all standard UML 1.4 diagrams. Besides features such as diagram editor and reverse engineering of compiled Java code, ArgoUML is a design critiquing tool. As the creator of ArgoUML defines: "A *design critic* is an intelligent user interface mechanism embedded in a design tool that analyzes a design in the context of decision-making and provides feedback to help the designer improve the design" [15].

Simply put, ArgoUML has predefined agents, called critics, that are constantly investigating the current model and if the conditions for triggering a critic are held, the critic will generate a ToDo item (this item is called a *critique*) in the ToDo list. A ToDo item is a short description of the problem, some guidelines about how to solve the problem, and if there exists, a wizard which helps the designer solve the problem automatically.

The critics run as asynchronous processes in parallel with the main ArgoUML tool. The critics are not intrusive, since the user can totally ignore them or disable one or all of them by the critics' configuration menu. The critics are not user defined, since they all are written in Java and are compiled as part of the tool. Furthermore, a ToDo item generated by a critic will remain in the ToDo list until the origin of the problem is vanished, either manually by the designer or by following the wizards proposed by the tool.

For implementing the SCR process and integrating it into ArgoUML, we have followed the guidelines given in the ArgoUML Cookbook [22] to perform the following steps.

1. For each pattern, write a critic class. Each critic class has a predicate method that should implement Sign and Criteria parts of the SCR process.
2. Specify the *head* and *description* of the critic in the configuration file "critics.properties."
3. Register the critic in class "Init."
4. If the critic is supported by a wizard, add a method to the critic class for initiating the wizard.
5. Write a wizard class for performing the Repair part of the SCR process. Since there is no association between the critic class and the wizard class, we need to re-evaluate all the requirements that we had in the Criteria part to see what options are required to be given to the user in the wizard class.

We have implemented both critic and wizard classes for all six selected patterns of EAA in ArgoUML. As a simple example of the Java code written in ArgoUML, the following code excerpt shows a general purpose function which checks the name or stereotype of an operation in the model.

```
public static boolean opSt(Object cls, String op) {
  boolean found = false;
  Iterator operator =
  Model.getFacade().getOperations(cls).iterator();
  while (operator.hasNext()) {
    Object o = operator.next();
    String opName = Model.getFacade().getName(o);
    if (opName.startsWith(op)) { found = true; break; }
    Iterator s =
   Model.getFacade().getStereotypes(o).iterator();
    while (s.hasNext()) {
      String sName =
      Model.getFacade().getName(s.next());
       if (sName.equals(op)) { found = true;  break; }
      }
      if (found) break;
     }
     return found;
   }
```

To evaluate, from the one hand, we believe that the two concepts of "design critiquing" and "smell detection in the design" are very similar. Hence, ArgoUML is an appropriate platform for integrating the SCR process. In addition, the ToDo items and the wizards are very interactive and user friendly. However, from the other

hand, firstly, due to the fact that ArgoUML critics are implemented in Java and adding new critics requires Java expertise and is done by modifying the source code, end-users cannot add new critics for criticizing new patterns. Secondly, due to lack of association between critic class and wizard class, duplicate evaluation of criteria is needed in the wizard class. This is painful. Thirdly, it is not possible to define critics using OCL in ArgoUML, since OCL constraints can be written at the model level only. And last but not least, an important disability of ArgoUML in applying the SCR process is dependent on the logic behind critics. The fact is that critics are triggered only when one of the Criteria is violated, hence there is no possibility to inform the user about the *correct usage of a pattern* without adding more functionality to ArgoUML

B. Epsilon

The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [9] is a platform of model management languages for tasks such as model merging and model transformation. Epsilon has a base language called Epsilon Object Language (EOL) [23], for model querying, navigation, and modification, and a driver language called Epsilon Model Connectivity (EMC) that enables managing models of different technologies. So far there are six task specific languages built in the context of Epsilon, among them Epsilon Wizard Language (EWL) is very close to the idea of the SCR process. All Epsilon languages, except EWL, are available in the Epsilon plug-in for Eclipse [9]. However, as a working environment for EWL, the Epsilon team has integrated the execution engine of EWL within ArgoUML. The result is an "ArgoUML Powered by Epsilon" tool [24]. We have selected this environment for performing our case studies with EWL.

The basic concept in EWL is *wizard* (not to be confused with ArgoUML wizards). A wizard consists of a name, a guard part, a title, and a do part. Therefore, the wizard structure in EWL is compatible with our SCR process. Both Sign and Criteria parts of the SCR correspond to the guard part of the wizard in EWL. The Repair part of the SCR corresponds to the do part of EWL.

The "ArgoUML Powered by Epsilon" has added a panel named "wizards" to the user interface of ArgoUML. Users can define the specifications of wizards in EWL language and save them to a "wizards.ewl" file located in the installation directory of ArgoUML. By running ArgoUML and selecting a model element, the guards of all wizards are evaluated. If the guard of a wizard is evaluated to true, the title is displayed to the user and the body part is executed. The body normally is responsible for fixing the problems associated with the selected model element.

We have written EWL wizards for all six selected patterns of EAA. Again as a simple example, the following code excerpt shows a general purpose function written in EWL which checks the name or stereotype of an operation in the model.

```
operation Class opSt (opName : String) : Boolean {
  return self.feature.exists (o:Operation |
o.name.startsWith(opName) or
                    o.hasStereotype(opName) );
}
```

Writing wizards in EWL, for "ArgoUML powered by Epsilon," as an offline file accessible by the end-user is a novel idea. This way the designer can add a wizard for a new pattern independent of the source code of the tool. The ability to define global variables in EWL is helpful and prevents redundant calculations of conditions in the Criteria and the Repair parts of the SCR process. Overall, EWL is a rich language in model modifications, i.e., while part of the syntax is similar to OCL, it has constructs such as high-level programming languages along with operations for applying changes on the model. However, despite the "confirm" interface which ensures that the user is confirming the changes, executing a wizard in EWL needs to be more interactive. A suggestion would be to give opportunity to the user to be able to select among a list of problems that are going to be fixed automatically.

C. OCLE

Object Constraint Language Environment (OCLE) [8] as a UML CASE tool, offers many useful features such as full OCL support at both UML meta-model level and model level, and a graphical interface for creating UML diagrams. At the meta-model level, OCLE checks the well-formedness of UML models against the WFRs specified in UML 1.5. At the model level, OCLE helps users in both static and dynamic checking.

By compiling and running constraint files, the user is able to check which of the invariants are not satisfied by the model. However, it is the user's responsibility to fix the problems in the model. A Compile-time error reflects problems concerning OCL syntax. A Runtime error means that some of the invariants in constraints are violated.

Considering the SCR process, since both the Sign and Criteria steps are model independent tasks and need to be verified against the meta-model, we have to check the Sign and Criteria using OCL in ".ocl" constraint files which are meta-model level constraints. However, due to the lack of capability for model modifications by OCL, there is no corresponding part in our constraints for the Repair step of the SCR. It is up to the user to check every invariant, and for every bad smell (failed invariant), the user is responsible for fixing the problem.

We have defined the constraints related to all six selected patterns of EAA in OCLE. The following code excerpt shows a general purpose function written in OCLE which checks the name or stereotype of an operation in the model.

```
context Class
   def: let opSt ( op : String ) : Boolean =
Operation.allInstances -> exists
 ( o:Operation | o.owner = self
   and((o.name.substring(0,op.size()-1) = op)
   or o.hasSt(op)))
```

Translating the Sign and Criteria parts of the SCR process into OCL constraints in OCLE is a straightforward and condensed form. However, using OCLE needs OCL expertise and writing constraints for critiquing patterns needs knowledge of the meta-model. Furthermore, OCLE is not meant to fix the problems in a model due to the lack of update facilities in OCL language which is not able to do

modifications in a model; therefore, Repair part of SCR is not applicable in OCLE. Finally, the tool cannot help more than displaying bad smells to the user and highlighting the problematic invariants. All the repair actions are the designer's responsibility.

VI. CONCLUSIONS

The main idea of smell detection in the application of a pattern language for a domain, e.g., the Patterns of EAA, is to detect when a pattern is used, to report whether the pattern is used wrongly, and to help the designer in repairing the bad smells (problems) that are found in the application of the pattern.

We introduced a process named Sign/Criteria/Repair (SCR) for detecting and repairing the smells in the usage of patterns. A Sign is the basic characteristic of a pattern, usually in the form of stereotypes. Criteria are the minimal requirements of the pattern. Repair is a set of steps to fix smells in the application of the pattern. Each pattern has specific Sign and Criteria. Each smell has essential Repair steps.

To evaluate the idea of the SCR and its applicability and usefulness in current modeling tools, we did experiments with three state-of-the-art tools, ArgoUML, Epsilon, and OCLE. We observed that the SCR process is able to be integrated in modeling tools and help designer in detecting bad smells early in the design process. The Repair mechanism of SCR is effective in removing the problems in a design and ensures correctness. Interactive modeling tool will speed up the design process and results in more efficiency. It is worth mentioning that the required effort and the offered help is not the same in all three tools.

As a brief comparison, using OCLE needs OCL expertise, and OCLE is not meant to fix the problems in a model due to the lack of update facilities in OCL language. Adding new critics in ArgoUML requires Java expertise, however wizards in ArgoUML are interactive and user friendly. Writing wizards in EWL, for ArgoUML powered by Epsilon, by the end-user is a novel idea; however, the user needs to learn the EWL syntax, which is not difficult, to apply update transformations on UML models.

REFERENCES

[1] *OMG Model Driven Architecture*, in URL: http://www.omg.org/mda/, accessed on May 1, 2008.

[2] R. Breu and J. Chimiak-Opoka, "Towards systematic model assessment," in *Proc. Perspectives in Conceptual Modeling: ER 2005 Workshops*, vol. 3770 of LNCS, Springer 2005, pp. 398-409.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.

[4] OMG, *Unified Modeling Language: Infrastructure, v2.0. OMG document formal/05-07-05*, 2005.

[5] OMG, *Object Constraint Language: Specification, v2.0. OMG document formal/06-05-01*, 2005.

[6] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.

[7] Tigris.org, *ArgoUML official website*, in URL: http://argouml.tigris.org/, accessed on May 1, 2008.

[8] BABES-BOLYAI University, *OCLE: Object Constraint Language Environment*, in URL: http://lci.cs.ubbcluj.ro/ocle/, accessed on May 1, 2008.

[9] Eclipse, *Epsilon*, Eclipse Foundation, in URL: http://www.eclipse.org/gmt/epsilon/, accessed on May 1, 2008.

[10] F. Bergenti and A. Poggi, "Improving UML designs using automatic design pattern detection," in *Proc. 12th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE*, pp. 336-343, 2000.

[11] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *Proc. 11th IEEE Int. Workshop on Program Comprehension, IWPC'03*, pp. 94-103, 2003.

[12] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Proc. Technology of Object-Oriented Languages, TOOLS 26*, pp. 112-124, 1998.

[13] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Soft. Eng.*, vol. 32, no. 11, pp. 896-909, Nov. 2006.

[14] W. Liu, S. Easterbrook, and J. Mylopoulos, "Rule-based detection of inconsistency in UML models," in *Proc. Workshop on Consistency Problems in UML-Based Software Development*, pp. 106-123, 2002.

[15] J. E. Robbins, *Cognitive Support Features for Software Development Tools*, Ph.D. Thesis, University of California, Irvine, 1999.

[16] C. Alexander *et al.*, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[17] S. Berczuk, "Finding solutions through pattern languages," *Computer*, vol. 27, no. 12, pp. 75-76, Dec. 1994.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

[19] hillside.net, *Pattern Languages of Programs Conference official website*, in URL: http://hillside.net/plop/, accessed on May 1, 2008.

[20] D. Manolescu, M. Voelter, and J. Noble, *Pattern Languages of Program Design 5*, Addison-Wesley, 2006.

[21] ThoughtWorks, *Refactoring Home Page*, in URL: http://www.refactoring.com/, accessed on May 1, 2008.

[22] L. Tolke, M. Klink, and M. van der Wulp, *Cookbook for Developers of ArgoUML*, University of California, 2006.

[23] D. S. Kolovos, R. F. Paige, and F. A.C. Polack, "The Epsilon object language (EOL)," in *Proc. 2nd European Conf. on Model Driven Architecture-Foundations and Applications, ECMDA-FA 2006*, vol. 4066 of LNCS, pp. 128– 142, Springer 2006.

[24] D. S. Kolovos, R. F. Paige, F. A. C. Polack, and L. M. Rose, "Update transformations in the small with the epsilon wizard language," *J. of Object Technology, JOT, Special Issue for TOOLS Europe*, vol. 6, no. 9, pp. 53-69, Oct. 2007.

Bahman Zamani received his B.Sc. from the University of Isfahan, Isfahan, Iran, in 1991, and his M.Sc. from the Sharif University of Technology, Tehran, Iran in 1997, both in Computer Engineering (Software). He obtained his Ph.D. degree in Computer Science from Concordia University, Montreal, QC, Canada in 2009.

From 1998 to 2003, he was a researcher and faculty member of the Iranian Research Organization for Science and Technology (IROST) - Isfahan Branch. Dr. Zamani joined the Department of Computer Engineering at the University of Isfahan in 2009, as an Assistant Professor. His research interests include Pattern Language Verification in Model Driven Design.

Greg Butler Greg Butler is Professor in Computer Science at Concordia University, Montreal, Canada. He joined Concordia in 1992 after nine years on the faculty of the University of Sydney, Australia. Dr. Butler has broad international experience, including periods visiting the University of Delaware, Universität Bayreuth, Universität Karlsruhe, and RMIT, Melbourne. Dr. Butler has consulted on object-oriented design, object technology, and extensible software architectures.

Dr. Butler's research goal is to construct knowledge-based scientific work environments, especially those for genomics and bioinformatics. In order to succeed at this, Dr. Butler has been investigating software technology for complex software systems - including software architectures, object-oriented design, and software reuse - to complement his experience in scientific computation and knowledge-based systems.

Dr. Butler obtained his Ph.D. from the University of Sydney in 1980 for work on computational group theory. He worked in computer algebra for 20 years developing algorithms, constructing software systems, designing languages, and investigating the integration of databases and knowledge-bases with computer algebra systems. He is a major contributor to the Cayley system for computational group theory, modern algebra, and discrete mathematics.