Using Design Patterns for Refactoring Real-World Models

Hamed Yaghoubi Shahir, Ehsan Kouroshfar, Raman Ramsin
Department of Computer Engineering,
Sharif University of Technology
Tehran, Iran
yaghoubi@ieee.org, kouroshfar@ce.sharif.edu, ramsin@sharif.edu

Abstract— **Many software development methodologies are based on modeling the real world. In some of these methodologies, real-world models are gradually transformed into software models, while in others, the real world is only considered as a preliminary source of insight into the physical business domain. Real-world modeling was pushed to the sidelines due to anomalies in real-world modeling approaches; however, with the advent of the Model-Driven Architecture (MDA), real-world conceptual modeling is likely to regain its importance.**
We propose a method for using Design Patterns in the context of model transformation, where real-world models are refactored through application of these patterns. Although the patterns are not applied in their original contexts, we show through examples that they are equally applicable to real-world models.

Keywords-Design Patterns; Real-World Modeling; Model Transformation; Model Driven Architecture

I. INTRODUCTION

Real-world modeling has been defined as “a technique to model the real world as it is; it identifies a class corresponding to an entity in the real world that can be observed” [9]. Many object-oriented methodologies begin modeling activities from the real world, and many methodologists agree that object-oriented concepts provide the possibility to consider the real world as the physical manifestation of the problem domain. In some object-oriented methodologies, real-world modeling is the basic modeling activity; examples include: *Shlaer-Mellor*, *Object Modeling Technique* (OMT), *Catalysis*, and *Feature Driven Development* (FDD) [14].

Real-world modeling has its own advantages and disadvantages. Since real-world models are elicited from the physical domain, they are tangible to both users and developers. Real-world models can therefore provide a more effective means of communication among developers, domain experts and stakeholders; requirements elicitation is thereby facilitated. The potential for effective domain-driven development is therefore enhanced, since real-world models show all the traits associated with effective domain models [5].

Despite the above advantages, various anomalies have been observed in real-world modeling, due to which it has declined in importance in recent years. The main disadvantages are as follows [9]:

- Actors in the real world become classes in the system; this may produce redundant or god classes.
- Irrelevant classes are introduced.
- Irrelevant operations and associations are introduced.

The above problems violate encapsulation, and as a result, cohesion and coupling are compromised.

We use a pattern-based model transformation approach originally introduced in [15], and propose the use of design patterns in this approach. The approach benefits from the advantages of real-world modeling while avoiding its anomalies and disadvantages.

The rest of the paper is structured as follows: Section 2 discusses the background and motivations for this research. In Section 3, we describe our transformation approach based on real-world versions of software design patterns. Section 4 provides a case study of applying selected design patterns to real-world models. The last section presents the conclusions, as well as suggestions for furthering this research.

II. BACKGROUND AND MOTIVATIONS: PATTERN-BASED MODEL TRANSFORMATION

Before introducing the details of the method proposed, a brief look at the research area would be in order. Many approaches have been proposed for pattern-based model transformation. Judson *et al.* have proposed a design-pattern-based model transformation approach at the metamodel level in [10]. They also provide some case studies on model transformation using different types of design patterns. France *et al.* have proposed a metamodeling approach to pattern-based model refactoring [7], and also a role-based metamodeling approach to specifying design patterns [11]. Wang *et al.* have provided a simple UML profile for design patterns so as to represent design patterns in UML models, and have proposed and implemented a model transformation approach based on these models [17]. Dong *et al.* have proposed a model transformation approach for design pattern evolutions [4], and Kim *et al.* have presented a framework for pattern-based model evolution approaches [12]. Ramsin has proposed an approach for using *Reengineering* and *Refactoring* patterns for transforming real-world models into software models [15].

Although many approaches have been proposed for using software patterns in model refactoring and model transformation, most of these approaches apply the patterns during later stages of development. Motivated by the merits of real-world domain modeling, we extend the approach introduced in [15] and propose a design-pattern-based approach for refactoring real-world models. To this aim, we provide real-world-modeling counterparts for a number of GoF design patterns [8], thus using them in a context which is different from the one they were originally intended for. This approach is based on the observation that design patterns possess a rationale and tangibility that appeals to the human mind, thus even transcending their usage in software development. This is mainly because software patterns are proven *human-devised* solutions to *common* problems. It is therefore quite natural to find many "software patterns" being used as problem-solution pairs in various older contexts, long before their usefulness in software development was recognized; examples include business organizations and social structures. Since real-world domain models and organizational structures are manifest in conceptual system models, applying design patterns can improve model quality; this is analogous to the approach adopted in [13], where similar patterns are used for improving business organizations.

III. Pattern-Based Approach to Transforming Real-World Models

The transformation approach adopted herein was originally introduced in [15]. In this section, we will briefly introduce the approach in order to delineate the context for using design patterns in model transformation. In the transformation approach of [15], real-world domain modeling is conducted iteratively and in a top-down fashion. Human workers, systems and data-stores of the problem domain are modeled as collaborating objects in a *Context Object Model*. A notation similar to UML collaboration diagrams is used for representing the model, except that the links are adorned with data/control flows rather than sequence numbers (Figure 8). Responsibilities of the context objects are defined as *features*, much in the fashion of the FDD methodology [14]. The resulting functional models comprise the main bulk of the *Context Model*. The system is then introduced as an object into the Context Model, and responsibilities are assigned to the system through redistribution and/or duplication of the features. The next step focuses on the design of the system as an extension to the problem domain: The *system* object produced during the previous step is opened up, and the system is designed as an extension to the organization, using the same types of elements already present in the problem domain; *System Object Models* are thereby built.

The *System Model* thus built is then converted to the *Software Model* through using patterns to iteratively redistribute features among objects. The objective is to enhance encapsulation, increase cohesion and reduce coupling, and also to introduce architecture. The redistribution procedure is devised in such a way as to resolve the problems typically afflicting analysis approaches which are based on object-oriented real-world modeling [9]. This marks the transition from the problem-domain-based system to the software system, signifying the transition to solution domain. The resulting *Software Object Models* comprise the functional component of the *Software Model*. The relationships between the models is shown in Figure 1.

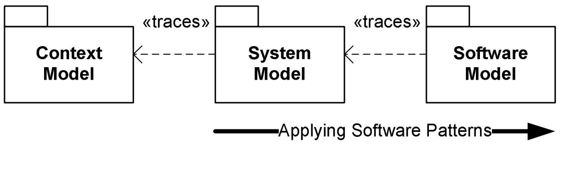

Figure 1. Relationships among the models produced

As prescribed in [15], three reengineering patterns [3] and seven refactoring patterns [6] are of utmost use in the starting iterations. These include: Move behavior close to data, Eliminate navigation code, Split up god classes, Move method, Move field, Extract class, Inline class, Hide delegate, Remove middleman, and Encapsulate field. Design patterns ([1], [8]) can be used in later iterations to help introduce specific architectures and mechanisms. We have used this transformation approach to apply five GoF design patterns [8] in a model refactoring context.

A. Design Patterns Adapted for Use in Real-World Model Transformation

New versions for five GoF design patterns [8] are provided herein, redefined so that they can be applied to real-world models.

- **Mediator**
- *Context*: Organizational units where complex interactions occur among active elements (thus increasing dependencies).
- *Problem*: How can we deal with elements of the problem domain (objects/entities/clerks) which interact and communicate in a complex fashion?
- *Solution*: A mediator (manager) element promotes loose coupling by keeping interacting elements from referring to one another directly. The elements interact with the mediator instead of with each other.
- *Application example*: Communicating clerks in different departments. *Solution:* Assign a mediator acting as a coordinator (Figure 2).

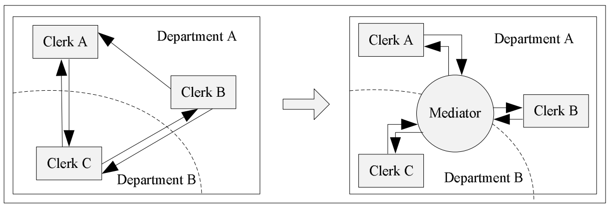

Figure 2. Example of applying Mediator

- **Façade**
- *Context*: Organizational structures where highly coupled units (departments/sections/groups) interact, or where clients need to interact with internal units of the organization.
- *Problem*: How can we decouple organizational units (departments/sections) from each other, and how can we decouple the client from internal units?
- *Solution*: An intermediary resides between the units with the ultimate aim of reducing interdependencies.
- *Application example*: Departments interacting with clients and with each other. *Solution:* Assign facades to the departments and a mediator to manage their communications (Figure 3).

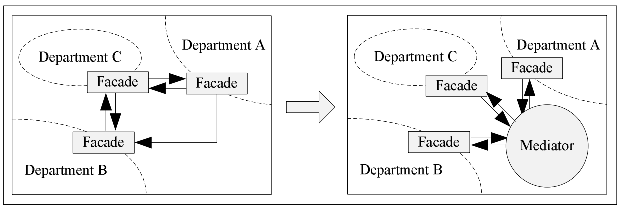

Figure 3. Example of applying Facade and Mediator

- **Visitor**
- *Context*: Organizational units where there is a need for special services that are implemented differently for each and every element (clerk/object/entity) in the unit, and where the service provider should have the knowledge of how to provide a specific service to a specific client.
- *Problem*: Services are required which cannot be performed by the clients (organizational elements).
- *Solution*: A specialized service-provider/consultant is put in charge of serving the clients by visiting them and providing the specific service required.
- *Application example*: There is a need for a specific service in many departments for different internal clients, and the clients lack the expertise or the resources required to perform the service themselves; furthermore, the exact nature of the service depends on the client. *Solution:* A service department is established to visit the clients and provide the service (Figure 4).

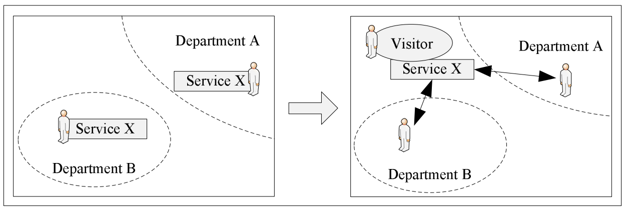

Figure 4. Example of applying Visitor

- **Proxy**
- *Context*: Organizational units where there are service-providers that receive many requests, and for some reason (security, efficiency, etc.) cannot or should not respond to the requests directly.
- *Problem*: How can some tasks be performed/monitored/controlled transparently before being passed on to the actual service-provider?
- *Solution*: Assign a proxy as a middleman which can control interactions with actual service-providers, and which can help them with their responsibilities.
- *Application example*: There is an external service-provider who can provide better service than that which is provided internally. *Solution:* Consider the external service-provider as an external visitor, and assign an internal unit to act as its proxy (Figure 5).

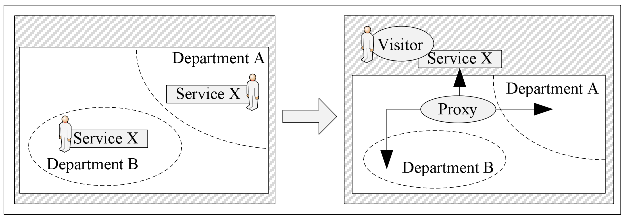

Figure 5. Example of applying Visitor and Proxy

- **Observer**
- *Context*: Where there is a need for monitoring changes in order to ensure consistency, or to enforce certain rules.
- *Problem*: How can we implement task monitoring or rule enforcement in an organizational unit?
- *Solution*: Assign observers to ensure consistency between information sources and information consumers, and also to monitor information change and make sure that certain rules are upheld.
- *Application example*: An organizational element/unit needs to keep track of changes occurring in some other unit. *Solution:* The interested element/unit becomes an observer of the unit it needs to keep informed about (observee); this means that the observer registers itself with the observee so that any changes can be sent to it upon occurrence (Figure 6).

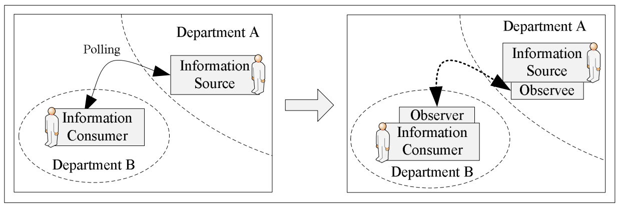

Figure 6. Example of applying Observer

In cases where the observee resides outside the system – e.g., when the system needs to update itself with external information – an internal proxy is assigned to the external source (Figure 7).

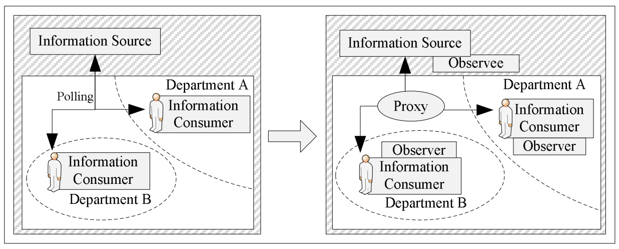

Figure 7. Example of applying Observer and Proxy

B. Suitable Types of Design Patterns for Transformation Purposes

The real-world model transformation context involves situations where there are interactions among real-world components, and such interactions need the implementation of appropriate behaviors on suitable structures. *Structural* and *Behavioral* design patterns are therefore better suited to the context, as *Creational* design patterns can only address a very limited range of context-relevant problems.

IV. CASE STUDY

A case study is provided in this section to demonstrate the applicability of the proposed design-pattern-based approach to real-world model transformation. The Estate Agency System used in the case study has been adapted from [15]. In the current system, potential *buyers* and *sellers* visit the premises of the agency in person or contact agency *clerks* by phone in order to obtain information about the properties on the market, put new properties up for sale, request viewings, or make an offer on a property. Information about properties, customers and transactions is stored in a database and maintained via an existing computer-based *record management system*. The aim is to develop an online estate-agency system that provides property search facilities online, and allows registered customers to put properties up for sale, request viewings, make offers, negotiate deals, and seek professional advice. The present computer-based record management system is to be considered as an external data storage system, interfaced in order to provide database management facilities to the online system. A subset of the requirements is listed below:

- The system should maintain information on sellers and buyers, properties, and transactions,
- Customers need to search in the properties that are registered in the system,
- Sellers and buyers need to pay via credit card, but the system should use secure payment providers,
- The system should provide online property search facilities to its users, using an external search engine for providing these facilities,
- There may be a need to automatically update property prices through consulting external sources.

This section contains the results of applying design patterns to transform the system model of the estate agency to its refactored counterpart. We will describe the transformation steps which are applied to the *before* state of the case study (Figure 8) in order to obtain the *after* state (Figure 9):

Step 1 - Applying *Mediator* and *Façade* at the boundary of the system: Since the interactions between outside actors and inside elements are complex, we first introduce a *Mediator* at the boundary of the system. In order to decouple the system's elements from outside actors, a *Facade* is introduced (named after the system); it is then merged with the mediator.

Step 2 - Applying *Visitor* and *Proxy* to accommodate external service-providers: Secure Payment and Search facilities are outside the scope of the system. This means that external service-providers are to be used. We have therefore inserted an external *Visitor* to provide each of the required services, with corresponding *Proxies* placed inside the system.

Step 3 - Applying *Observer* and *Proxy* to provide automatic price updates: Since the system needs to automatically update the price of the properties through consulting external sources, an *Observer* is inserted to provide the required information. Since the Price Observed resides outside the system, a corresponding internal *Proxy* is assigned to it.

Step 4 - Applying *Proxy* to manage record repositories: A *Proxy* acting as a repository manager is assigned to each of the three record repositories.

While the aim of applying these patterns is to introduce useful architectures, reduce coupling and promote cohesion, it is important to note that we have applied *Design* patterns in the *Requirements Engineering* and *Analysis* phases. By applying the design-pattern-based transformation approach, the initial Real-World Model is refactored to better address the requirements. The following advantages can be observed in the resulting model:

- The produced model can be easily understood by the stakeholders, since it is directly transformed from the real world, and is composed of the same concepts.
- We can use a specific proposed design pattern for applying common *Requirements Engineering* decisions in different projects. For example, as stated in the case study, the use of the *Visitor* pattern is recommended where an external service is required. By formatting such decisions as patterns, they can be reused in similar situations.
- The proposed approach can inspire new or enhanced business processes within the organization, thus promoting *Business Process Reengineering* (BPR). The use of *Façade* and *Proxy* patterns in our case study is a typical example.
- Building solution-domain and implementation-domain classes can be facilitated by using the proposed patterns during initial stages of development. For example, applying real-world versions of *Observer* and *Proxy* during analysis may eliminate the need for such decisions in later phases.

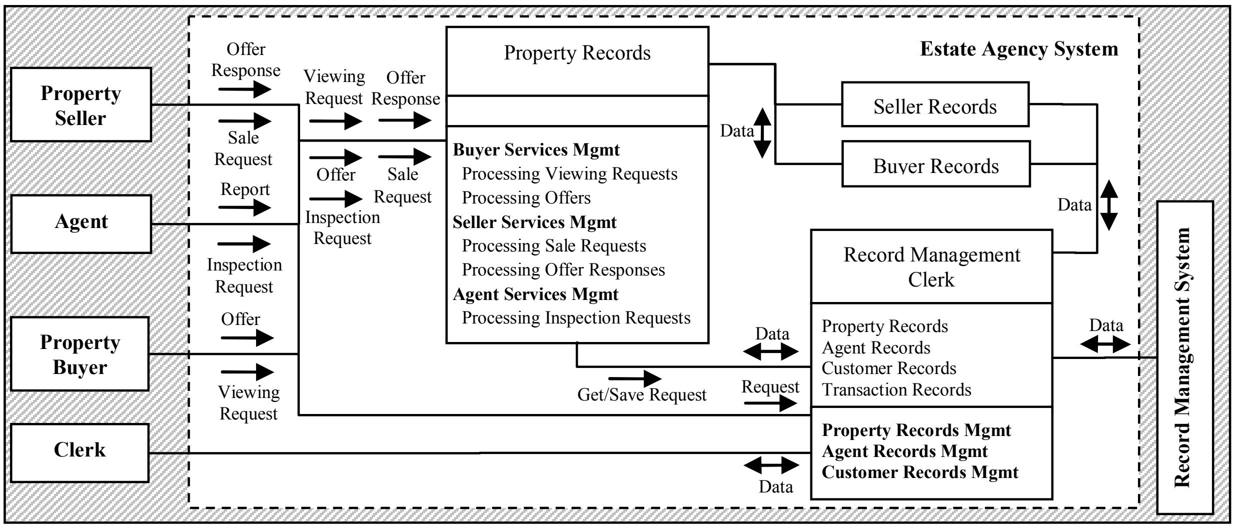

Figure 8. The real-world model before the application of Design Patterns [15]

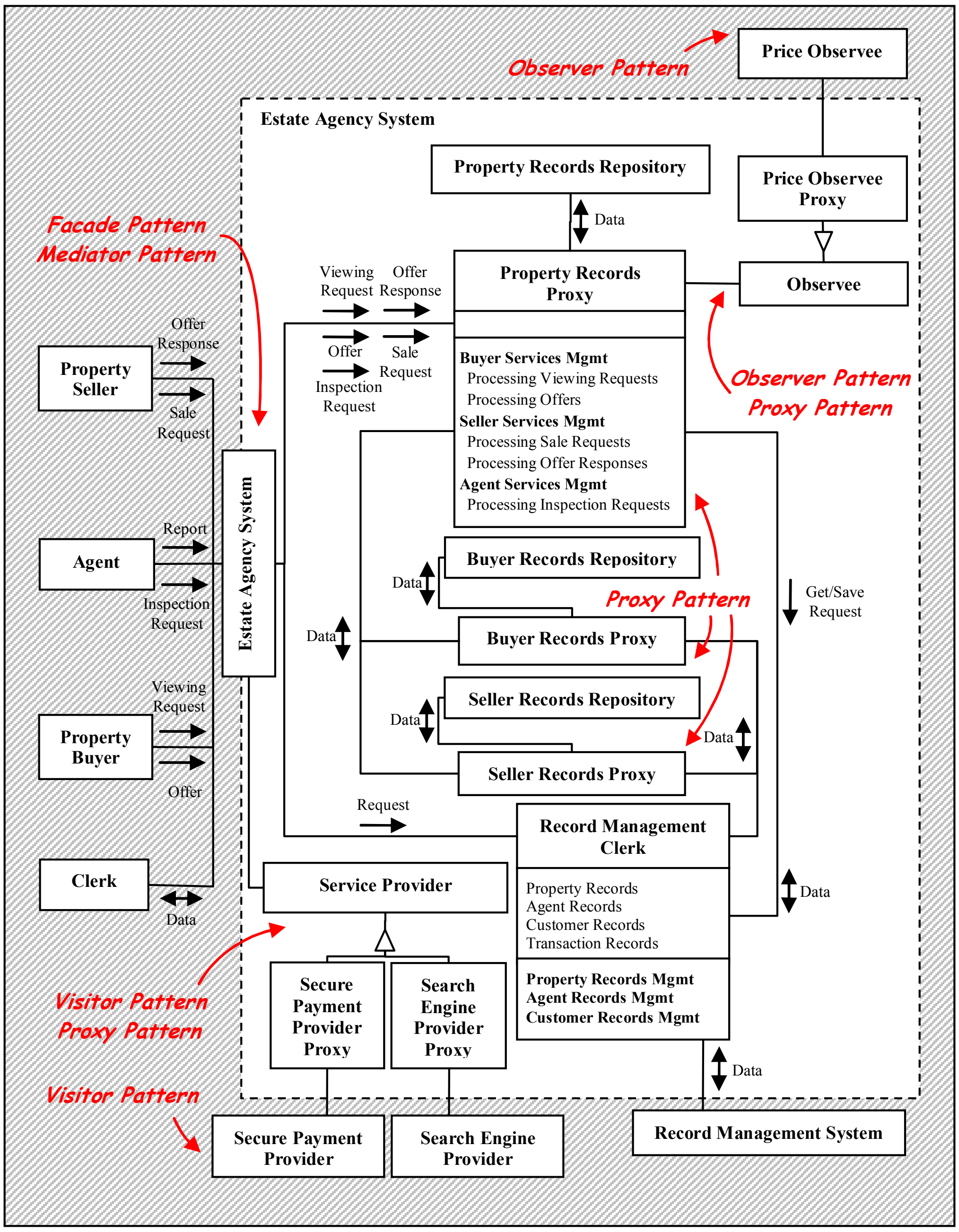

Figure 9. The result of applying design patterns to the real-world model

V. CONCLUSIONS AND FUTURE WORK

In software engineering, real-world modeling has typically been used during systems analysis as a prelude to defining the internal structure of software systems. We have proposed a design-pattern-based approach for transforming/refactoring real-world models. The pattern-based real-world model transformation approach proposed in [15] has been used as a framework for applying design patterns.

We have proposed new definitions for five prominent design patterns, thus explaining their application in the new context. The approach has been demonstrated through a case study, where the five patterns have been applied to a real-world model to ultimately produce an improved model of the software system. The resulting models are richer as to architectural features, are more intelligible to domain experts and end users, and lend themselves better to the application of patterns in later phases of development.

The proposed design-pattern-based refactoring approach is applicable as a pattern-based model transformation approach in a *Model-Driven Architecture* (MDA) context. This research can be furthered through defining the design patterns in the *Epsilon Wizard Language* (EWL), and providing complete tool support for the approach. The applicability of the approach can then be empirically assessed through applying it in the context of industrial-scale *Model-Driven Development* (MDD) projects. Another strand of research can focus on exploring the applicability of other types of patterns in a real-world model transformation context; Architectural [1] and Organizational [13] patterns seem to be especially promising in this regard.

ACKNOWLEDGEMENT

We wish to thank the ITRC Research Center for sponsoring this research.

REFERENCES

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., Pattern Oriented Software Architecture: A System of Patterns, Wiley, New York, 1996.

[2] Czarnecki K., and Helsen, S., "Classification of Model Transformation Approaches". In Proceedings of the OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[3] Demeyer, S., Ducasse, S., and Nierstrasz, O., Object-Oriented Reengineering Patterns, Morgan-Kauffman, San Francisco, 2003.

[4] Dong, J., Yang, S., and Zhang, K., "A Model Transformation Approach for Design Pattern Evolutions", In Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), pp. 80-92, 2006.

[5] Evans, E., Domain-Driven Design: Tacking Complexity in the Heart of Software. Addison-Wesley Longman Publishing Co., 2003.

[6] Fowler, M., Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[7] France, R., Ghosh, S., Song, E., and Kim, D., "A Metamodeling Approach to Pattern-Based Model Refactoring", IEEE Software, Vol.20, No.5, pp. 52-58, 2003.

[8] Gamma, E., Helm, R., Johnson, R., And Vlissides, J., Design Patterns: Elements of Reusable Object-oriented Software, Addison Wesley, 1995.

[9] Isoda, S., "Object-Oriented Real-World Modeling Revisited", Journal of Systems and Software, 59, 2 (November), pp. 153-162, 2001.

[10] Judson, Sh. R., France, R. B., and Carver, D. L., "Specifying Model Transformations at the Metamodel Level", In Proceedings of the Workshop in Software Model Engineering (WiSME'03), San Francisco, CA, USA, October 2003.

[11] Kim, D., France, R., Ghosh, S., and Song, E., "A Role-Based Metamodeling Approach to Specifying Design Patterns", In Proceedings of the 27th Annual International Conference on Computer Software and Applications (COMPSAC'03), 2003.

[12] Kim, S. and Carrington, D. "A Pattern based Model Evolution Approach", In Proceedings of the XIII Asia Pacific Software Engineering Conference (APSEC'06), 2006.

[13] O'Shaughnessy, J., Patterns of Business Organization, George Allen & Unwin Ltd, 1976.

[14] Ramsin, R., and Paige, R. F., "Process-Centered Review of Object-Oriented Software Development Methodologies", ACM Computing Surveys, February 2008, pp. 3:1-89.

[15] Ramsin, R., The Engineering of an Object-Oriented Software Development Methodology, Ph.D. Thesis, University of York, April 2006. Available at: http://www.cs.york.ac.uk/ftpdir/reports/YCST-2006-12.pdf.

[16] Sendall, Sh., and Kozaczynski, W., "Model Transformation: The Heart and Soul of Model-Driven Software Development", IEEE Software, Vol. 20, No.5, pp. 42-45, 2003.

[17] Wang, X., Wu, Q., Wang, H., and Shi, D., "Research and Implementation of Design Pattern-Oriented Model Transformation", In Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), 2007.