Supporting Layered Architecture Specifications: A Domain Modeling Approach

Jenny Abramov and Arnon Sturm

Department of Information Systems Engineering,
and Deutsche Telekom Laboratories
Ben-Gurion University of the Negev
Beer Sheva 84105, Israel
{jennyab,sturm}@bgu.ac.il

Abstract. Software architectural patterns help manage complexity through abstraction and separation of concerns. The most commonly used architectural patterns are layered architectures, which benefit from modularity and reuse of layers. However, they lack in supporting changes, as there is a need to do a substantial amount of rework on the layers in order to incorporate changes. Furthermore, the comprehension of specifications which are based on a layered architecture can be difficult. In order to address the aforementioned limitations, we adopt a domain engineering approach called Application-based Domain Modeling (ADOM). Using ADOM, we refer to each layer as a separate domain model, whose elements are used to classify the application model elements. Consequently, the application model is represented in a unified form, which incorporates information from all of the layers. This allows performing changes in the model, without creating cascades of changes among the layers' models in order to synchronize them.

Keywords: Layered architecture, ADOM, UML, Domain modeling.

1 Introduction

As software systems are complex, they must be built on a solid foundation, namely their architectural design. A major mean for designing software architectures is architectural patterns, which are used to enhance modularity by helping in managing complexity through abstraction and separation of concerns. Architectural patterns specify the structure of a system by providing a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them [3].

Among the various software architectural patterns the most common one is the layered architecture [5]. The notion of layered systems has become popular since it was first introduced at 1968 by Dijkstra for the "THE system" [4] operating system. It was also presented in the OSI seven layer network architecture [18] and in the area of artificial intelligence [2]. A key principle of a layered architecture [5] is to factor out responsibilities into separate cohesive units and define the dependencies between them. Layers are meant to be loosely coupled, with dependencies in only one direction such that the

I. Bider et al. (Eds.): BPMDS 2010 and EMMSAD 2010, LNBIP 50, pp. 195–207, 2010.

lower layers provide low-level general services, and the higher layers are more application specific. Collaboration and coupling is from higher to lower layers. Thus, the layered architectural style introduces a hierarchical subsystem structure. The hierarchical structure considered as a good structure to deal with complex systems in a scalable way. Moreover, other architectural styles, as the component-based architecture, arrange their concepts in layers [17]. The functionality and the amount of layers vary across applications. However, there are some fairly standard layers. For instance, the typical architecture style for user-oriented systems is the standard three layers architecture, which includes the presentation, the business logic, and the persistence layers.

A layered architecture can help mitigate complexity and improve understandability by grouping functionality into different areas of concerns. It supports design based on increasing levels of abstraction, which enables the designer to partition a complex problem; and it improves the maintainability and extensibility of the application by minimizing dependencies, allowing exchange of layers, and isolating technology upgrades. However, the standard layered architectures lack in supporting changes that have to be propagated through multiple layers, since those changes cause a cascade of changes on many layers in order to incorporate apparently local changes. For example, in many user-oriented systems, such as information systems, many entities require representation in several layers. In that case, a change in the type of an attribute will require from the developer to find all the related entities in all layers in order to change the type of the corresponding attributes. Since software design generally evolves over time, it is important to find a way to overcome those limitations. We believe that the aforementioned reasons also hinder the comprehension and the ease of construction of large and complex systems.

Software systems are usually specified via models. There are two major approaches for utilizing architectural patterns though models: Architecture Description Languages (ADLs) which aim at formally representing software architectures and the Unified Modeling Language (UML) which is a generic modeling language that can also be used to describe software architectures. These approaches provide methods and tools for representing and analyzing architectural designs; however, since these are general-purpose approaches, it is difficult to address the aforementioned specific limitations of layered-based applications. Motivated by the popularity of layered architectures and the limitations inherent to general-purpose approaches, this paper presents an idea of a modeling approach specifically for modeling layered architectures.

In this paper we adopt a domain engineering approach called Application-Based Domain Modeling (ADOM), which enables specifying and modeling domain artifacts that capture the common knowledge and the allowed variability in specific areas, guiding the development of particular applications in the domain, and verifying the correctness and completeness of applications with respect to their relevant domains. Referring to the layered architecture, each layer is represented as a domain model and application model elements are classified by the layers' (domain) models elements. In that case, the designer needs to manage a unified application model without losing the architectural information. Thus, when dealing with a change, the designer makes the required change on the unified model without being required taking care of cascading changes among layers. We term the resulted model a Multi-Classified Model (MCM).

The structure of this paper is as follows. Related work is presented in Section 2. Section 3 introduces the ADOM approach whereas Section 4 elaborates on the proposed

approach, the Multi-Classified Model (MCM). Section 5 summarizes the strengths of the MCM and finally, Section 6 concludes and discusses future plans.

2 Related Work

As software architecture and the practice of architectural design have been recognized as significant issues in complex software systems engineering. Researchers and practitioners have been proposing numerous[1] formal notations for representing and analyzing architectural designs. These methods were meant to increase comprehension of architectural designs, to improve the ability to analyze consistency and completeness, and to support changes of the design during the development lifecycle. Architectural design approaches can be classified into three research areas: (1) ADL-related approaches, (2) UML-based approaches, and (3) the combined approaches.

Most ADLs support formal architecture representation including topological constraints from a structural point of view. They vary in the level of abstraction they support, their terminology, the information they specify and the analysis capabilities they provide. Each ADL provides certain distinct capabilities. Most first generation ADLs were developed to support some specific software architecture aspects. For instance, Rapide [6] is a language for modeling architectures of distributed systems. It is an event-based architecture definition language. Wright et al. [19] formalize the semantics of architectural connections. Darwin [8] is designed for dynamic architecture using π-calculus to formalize architectural semantics. Second generation ADLs identify and process fundamental concepts common to first generation ADLs in order to allow architectural interchange. For example, ACME [6] was developed to provide a framework to integrate different ADLs by supporting mapping of architectural specifications from one ADL to another.

Since ADL users were required to learn the specific notation of each ADL, and ADLs were not integrated in any development process, ADLs have not come into extensive use in the industry. Hence, the usage of a standard language such as UML for describing architectural design might make it easier to understand, mitigate the effort of preserving the architecture consistent during development phases, and it would be supported by existing and fairly standard tools. Furthermore, as UML has become standard general modeling language for software development, representing the architecture with UML will allow integrating it with the rest of software artifacts.

However, UML is not designed, syntactically or semantically, to represent software architecture elements and therefore does not support some architecture concepts, such as the lack of supporting connectors and architectural styles [19]. For that reason, many studies suggested extending the vocabulary and semantics of UML to apply its modeling capabilities to the concepts of architecture design. It is important to mention that UML 2.0 embraces much more constructs that are important to architecture description than UML1.x, such as components and connectors. Medvidovic et al. [10] identified three approaches for modeling software architectures using UML: (1) using UML "as is" [8]; (2) Extending UML in a heavyweight way [13] by adding new modeling elements or replacing existing semantics via direct modification of the

[1] According to Malavolta et al. [9] there are more than 50 ADLs proposed in academia and industry.

UML metamodel; and (3) Extending UML in a lightweight way [10][20] by using the extension mechanism of UML for defining new modeling elements. The adaptations are defined using stereotypes, which are grouped in a profile.

In addition to ADLs and UML approaches there are some combined approaches. These approaches seek to combine ADLs with UML notations. For example, Roh et al. [16] suggest a layered language which uses UML, generic ADL comprising of domain-independent elements, and a domain-specific ADL.

There are some important architecture specific problems that can be minimized or avoided by concentrating on a specific architectural style, such as the layered architectures. However, all of the approaches mentioned above do not deal with the disadvantages of some specific architectural style but concentrate on a general representation of software architectures and therefore cannot take advantage of the unique properties of some specific architectural style.

In this paper we introduce our approach to support structural layered architectures' specification. We use as an example the standard three layered architecture, which is the most basic and common structure in user-oriented systems [1][12].

3 Application-Based Domain Modeling (ADOM)

The Application-based Domain Modeling (ADOM) [14][15] is rooted in the domain engineering discipline, which is concerned with building reusable assets on one hand and representing and managing knowledge in specific domains on the other hand. ADOM supports the representation of reference (domain) models, construction of enterprise-specific models, and validation of the enterprise-specific models against the relevant reference models. The architecture of ADOM is based on three layers:

(1) The language layer comprised of metamodels and specifications of the modeling languages. In this paper we use UML 2.0 class diagrams as the modeling language, since the focus of this paper is on the structural aspect of the architecture.

(2) The domain layer holds the building elements of the domain and the relations among them. It consists of specifications of various domains; these specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain. The structure and the behavior of the domain layer are modeled using a modeling language that is defined in the language layer. In this paper we refer to the structure of each layer as a domain model.

(3) The application layer consists of domain-specific applications, including their structure and behavior. The application layer is modeled using the knowledge and constraints presented in the domain layer and the modeling constructs specified in the language layer. An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain model should be applied in any application model of that domain. In order to achieve this goal, any element in the application model is classified according to the elements declared in the domain model using UML built-in stereotype and tagged values mechanisms. In this paper the application model elements are multi-classified by the layers' (domain) model elements.

For describing variability and commonality, ADOM uses multiplicity stereotypes which can be associated to all UML elements, including classes, attributes, methods, associations, and more. The multiplicity stereotypes in the domain model aim to represent how many times a model element of this type can appear in an application model. This stereotype has two associated tagged values, min and max, which define the lowest and the upper most multiplicity boundaries, respectively. For clarity purposes, four commonly used multiplicity groups were defined: <<optional many>>, <<optional single>>, <<mandatory many>>, and <<mandatory single>>. The relations between a generic (domain) element and its specific (application) counterparts are maintained by the UML stereotypes mechanism: each one of the elements that appears in the domain model can serve as a stereotype of an application element of the same type (e.g., a class that appears in a domain model may serve as a classifier of classes in an application model). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain model. Some optional generic elements may be omitted and may not be included in the application model and some new specific elements may be inserted to the specific application model, these are termed application-specific elements and are not stereotyped in the application model.

ADOM also provides a powerful verification mechanism that prevents application developers from violating domain constraints while (re)using the domain artifacts in the context of a particular application. This mechanism also handles application-specific elements that can be added in various places in the application model in order to fulfill particular application requirements.

4 The Multi-Classified Model Approach

In this paper we propose the Multi-Classified Model (MCM) approach for specifying layer-architecture-based applications. Working with MCM advocates the following steps: (1) Modeling a layered application; (2) Verifying the application against the predefined layers (as domains); and (3) Transforming the model into a layered application. In this paper, we focus on the first step and partially on the second step. In this paper, the MCM approach uses ADOM [14][15] with UML 2.0 class diagrams as the modeling language in order to describe the static structure of a system in terms of classes and their relationships.

As this paper focuses on the application designer activities, we assume that a software architect has already provided the following artifacts: (1) a set of domain models that represent the layers; (2) the dependency among the layers (domains); and (3) the dominant domain, which is used for creating the initial skeleton of the application, as it represents the core of the application domain.

Having the domain and architectural knowledge, the designer should preserve the following steps while specifying the (layered) application model:

1. Automatically generate an initial application model based on the dominant domain model, as common in ADOM.
2. Manually modify and refine the application model, based on the application specifications.

3. Manually classify the model according to the various layer (domain) models, which were provided by the software architect.
4. Automatically verify the application model with respect to the layer (domain) models. This is done using the ADOM validation procedure for each model separately.

Note that the various steps can be done iteratively.

In order to demonstrate the MCM approach we use a *ticket selling system* designed as a standard three layer architecture application. The tickets selling system allows registered users to buy and sell tickets to events. As a modeling infrastructure, for the *ticket selling system*, a designer has three domain models representing the three layers[2]: (1) the Web Interface (WI) domain, which represents the presentation layer, is depicted in Fig. 1; (2) the Registration (R) domain, which represents the business logic layer, is depicted in Fig. 2; and (3) the Relational Database (RD) domain, which represents the persistent layer, is depicted in Fig. 3. The dependencies among the domains are defined as follows: WI depends on R and R depends on RD, which correspond to three layered architecture. The dominant domain is the Registration (R) domain since it is the core asset for applications in this domain.

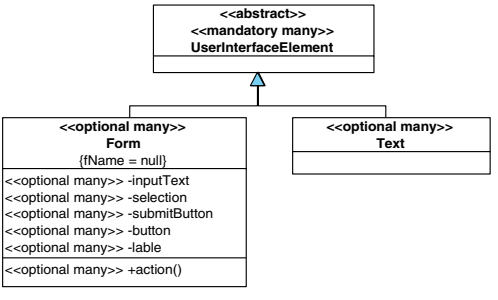

Fig. 1. The Web interface domain model

The domain models presented in Fig. 1, Fig. 2. and Fig. 3. state that all application models that are derived from them should follow the rules (among others) of:

- At least one *Form* or *Text* page should be specified in the application model, as specified within the *Web Interface* (WI) domain in Fig. 1.
- At least one *Client*, one *Provider*, one *Product*, one *ProductItem*, and one *Registration*, should be specified in the application model while preserving the structure as specified within the *Registration* (R) domain in Fig. 2.
- At least one database *Table* should be specified in the application model as specified within the *Relational Database* (RD) domain in Fig. 3.

In addition, the applications that follow these domains should specify the attributes and the operations of each class following the specifications of the domains. For example, each application has to have at least one *ProductItem*, which exhibits zero or more *detail* attribute, possibly one Boolean operation *checkAvailability* of the product, possibly

[2] Note that these three domains models are partial and we kept them small in size for the purpose of presenting the MCM-based approach principles.

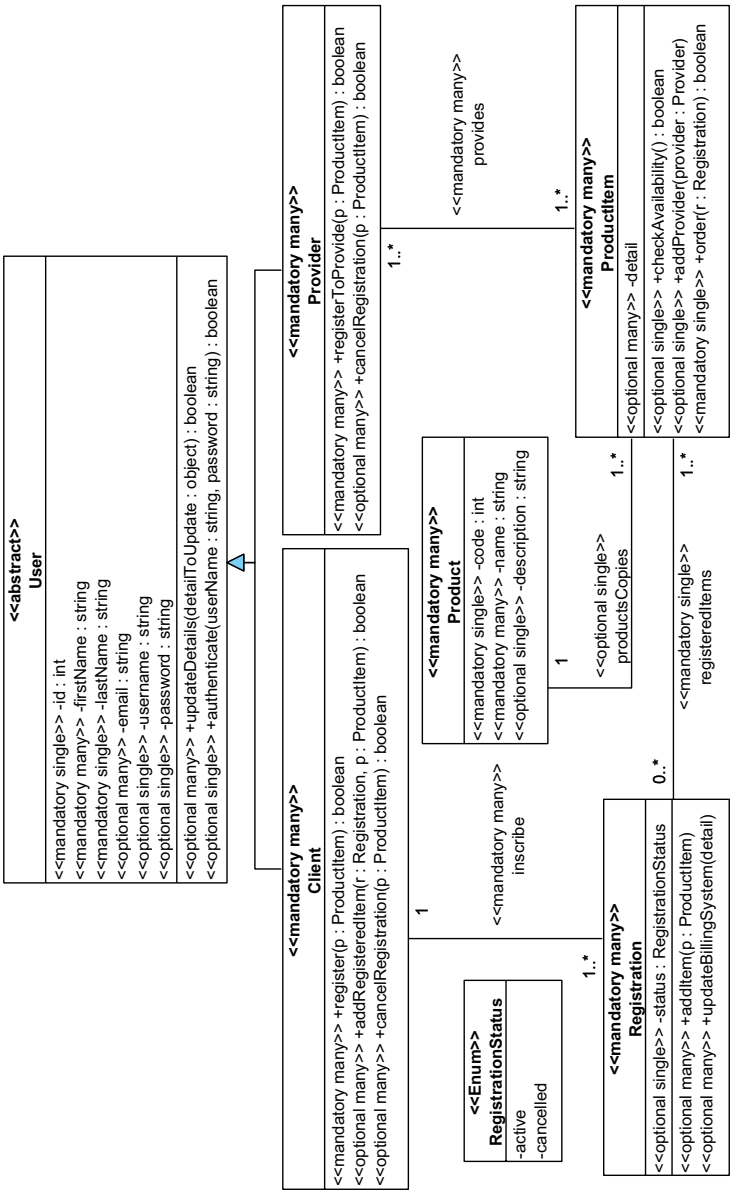

Fig. 2. The registration system domain model

one *addProvider* operation, and exactly one Boolean *order* operation. As common in ADOM, the relations between a layer (domain) element and its application counterparts are maintained by UML stereotypes, such that a domain element serves as a stereotype of an application element.

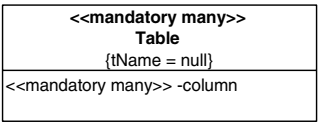

Fig. 3. The relational database domain model

Since there is a many-to-many relationship between the application elements in the various layers, tagged values associated with the domain elements stereotype can be used as parameters in the application model. For example, in the *Relational Database* (RD) domain, in Fig. 3, the *tName* tagged value associated with *Table* allows to specify in the application model the table name, to which the application element is associated. Tagged values associated with attributes or operations in the domain model are not presented due to visibility reasons. For instance in the *Relational Database* (RD) domain, in Fig. 3, a *key* can be associated with table *column*, this tagged value can be used to indicate the key type of a column in the database table.

Having domain models representing the various layers and the modeling infrastructure, the designer can generate the initial application model, modify and refines it, and finally classify it following the guidelines provided by the layer (domain) models. As in all layered architectures, each application element is associated with at least one layer (domain). For each such layer, an application element can either be classified by a specific layer (domain) element, or otherwise, be an application-specific element in that layer, and to be classified by the layer (domain) itself.

Fig. 4 depicts the resultant model of the *ticket selling system*. The model, in Fig. 4, also shows that this application is structured by the three layered architecture which is defined by three domains: Web Interface (WI), Registration (R), and Relational Database (RD). Each class is classified according to its requirements to the appropriate layers defined by the relevant domain models. From this representation we can easily understand the classification of each element with respect to the layers it belongs to. Any element which is application-specific is classified by the domain, but not by domain elements. In Fig. 4, the *Seller*, whose details are generalized from *User*, is classified as *<<WI.Form fName=login>>*, *<<WI.Form fName=myAccount>>*, *<<R.Provider>>*, *<<RD.Table tName=sellers>>*, and *<<RD.Table tName=orders>>*. This means that the *Seller* class has impact on all layers, the *Seller* class is represented in two forms: *login* and *myAccount*, it is follows the *Provider* class from the *Registration* (*R*) domain and maintains persistent data in two database tables: *sellers* and *orders*.

The attributes and the operations of each class are also classified according to the attributes and operations of the assigned domain classes, i.e. the *id* attribute in the *User* class is classified *<<User.id>>*. Similar to class classification, the attributes and operations that are application-specific elements are classified by the layer (domain) name they belong to. For example, the *orderNum* attribute in the *Order* class is classified *<<R>>* meaning that it is an application specific element that belongs to the registration layer.

For clarity purposes, we recommend presenting only the attributes and operations classification of the dominant domain layer, in the case of the *ticket selling system* it is the *Registration* (i.e., the business logic) layer.

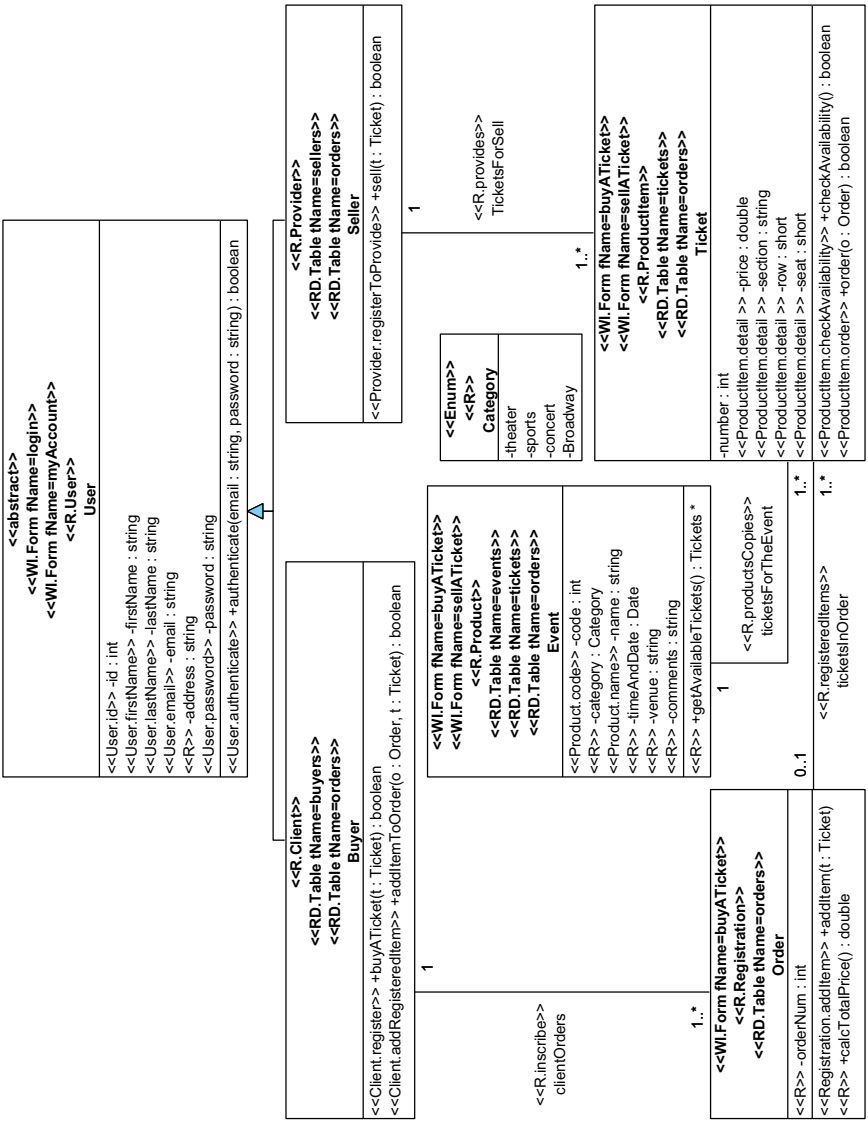

Fig. 4. The MCM tickets selling system application model

Fig. 5 shows the hidden classifications of the *Ticket* class. It specifies that the *number* attribute is a column in the *tickets* and *orders* tables and it also serves as a primary key in the *tickets* table. In addition, the *price*, *section*, *row*, and *seat* attributes are columns in the *tickets* table and they are presented to the user as a label in the *buyATicket* form and as an input text in the *sellATicket* form. As already mentioned, in addition to the classification, we defined tagged values to be used as parameters in the application model. For example in Fig. 5, the *fName* that is associated with the

form (defined in the *Web Interface* (WI) domain, Fig. 1) indicates the form name, or the *key* that is associated with the table column (defined in the *Relational Database* (RD) domain, though is hidden in Fig. 3.) indicates the key type of this attribute in the database table. The motivation for that enhancement was that there is a many-to-many relationship between the elements in the various layers. In the case of the *ticket selling system*, the *buyATicket* form gathers elements from various classes: *Ticket*, *Event*, and *Order*, and each one of those classes elements can refers to several *Form*s or *Table*s.

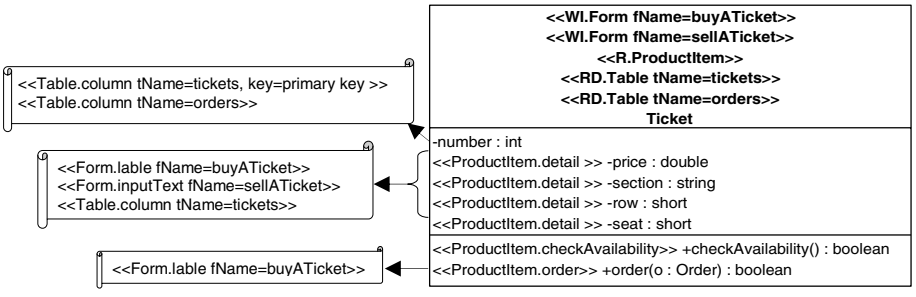

Fig. 5. The hidden classifications of the Ticket class are presented on the left

Having defined the application model, the developer should perform verification of the MCM-based application model with respect to the domain (reference) model. The verification of an application model is done separately for each domain. As described by the ADOM approach [15], the algorithm is performed in three steps: element reduction, element unification, and model matching. In the following we briefly present this algorithm.

In the ***element reduction*** step, classes that are not stereotyped by elements of the same domain model are neglected from each model separately. In the example of the *tickets selling application*, with respect to the *RD* domain the *User* and *Order* classes are omitted, with respect to the *R* domain the *Category* class is omitted, and with respect to the *WI* domain the *Buyer* and *Seller* classes are omitted.

During the ***element unification*** step, classes having the same domain stereotype are unified, leaving only one class in the resultant model. The multiplicity of that class denotes the number of distinct classes in the application model having the same stereotype. In the example of the *tickets selling application*, the resultant model of the *RD* domain consists of the *Table* class with multiplicity of 10. The resultant model of the *R* domain consists of 6 classes: *User*, *Provider*, *Client*, *Registration*, *Product*, and *ProductItem* all with multiplicity of 1. Finally, the resultant model of the *WI* domain consists of the *Form* class with multiplicity of 7.

In the ***model matching*** step, the resultant models of the previous step are matched against their corresponding domain models in order to verify the multiplicity of the elements. In addition the application model structure for each layer is verified with respect to the appropriate domain models. In the example of the tickets selling application the model adheres with the domain models which represent the layers of the system (i.e., the RD, R, and WI domains).

5 Summary

In this paper we have presented the Multi-Classified Model (MCM) approach for specifying layered-architecture based systems. The MCM approach extends a domain engineering approach called Application-based Domain Modeling (ADOM). Utilizing ADOM, MCM refers to each layer as a separate domain model, whose elements are used to classify the application model elements. Consequently, the application model is represented in a single model, which incorporates information from all of the layers. In this paper we have elaborated the first step of the approach, which deals with modeling a layered application from the structural point of view. Furthermore, we have briefly reviewed the second step, which deals with verifying the application against the predefined layers. We have illustrated the approach through a *ticket selling system* designed as a standard three layered architecture application.

We believe that the proposed Multi-Classified Model approach enhances the following:

1. **Evolution and Maintainability:** Since MCM represents the application model in a unified form, which incorporates information from all of the layers it can be easily modified as new requirements will not cause cascades of modification. In that case, the designer needs to manage a unified application model without losing the architectural information. Moreover, as it is easy to change the application model, the approach will prevent 'software decay' through time.
2. **Comprehension:** Software architectures enhance the ability to comprehend large systems by abstraction and separation of concerns. However, traditional modeling techniques, such as UML, do not support them very well. For example, in order to present the three layer architecture using UML the designer will have to describe three different packages and the relationships among them. The Multi-Classified Model approach simplifies the application model of a system with the layered architecture due to the use of a unified model for the application specification. Thus, the model is much more compact. In addition, as application model elements, such as classes, are classified according to their requirements to the appropriate layers (domains), we can easily understand the classification of each element with respect to the layers it belongs to. Thus, it seems plausible that the general understanding and readability of the application model will increase with respect to the traditional modeling techniques representation.
3. **Construction:** The defined domain models provide a partial blueprint for the development of the application model. The Multi-Classified Model approach typically documents abstraction boundaries between parts of an application, clearly defining which element belong to which layer, and constraining what parts of a system may rely on services provided by other parts. That helps to construct a good structure which is based on proven domain models.
4. **Verification:** Since the Multi-Classified Model approach uses ADOM, it provides verification to the application model according to the constraints imposed by the domain models which represent the layers. This cause the models to be valid with respect to best practices specified within the domain layer of ADOM. Furthermore, the Multi-Classified Model approach enforces the developers to preserve all the rules that were defined in the domain layers during the development.

It is worthwhile to mention that we have constructed a *student-courses registration system* under the same modeling infrastructure, showing that the approach can be used in different settings as well.

In the future we plan to formulate the Multi-Classified Model approach to support automatic generation of application code with the use of transformation rules that will be defined for each domain. This will support the third step of the proposed approach. In addition, we plan to extend the Multi-Classified Model approach to deal with dynamic aspects of the architecture as well. Furthermore, we intend to perform empirical evaluations in order to examine the benefits of the MCM approach, namely, model changeability, model comprehension, and model construction.

References

[1] Allen, R.J., Douence, R., Garlan, D.: Specifying Dynamism in Software Architectures. In: Proceedings of the Workshop Foundations of Component-Based Systems, pp. 11–22 (1997)

[2] Brooks, R.A.: A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation 2(1), 14–23 (1986)

[3] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Inc., Chichester (1996)

[4] Dijkstra, E.W.: The structure of the "THE"-multiprogramming system. Communications of the ACM 11(5), 341–346 (1968)

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

[6] Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural Description of Component-Based Systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–67. Cambridge University Press, New York (2000)

[7] Luckham, D.C., Vera, J.: An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering 21(9), 717–734 (1995)

[8] Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)

[9] Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies. IEEE Transactions on Software Engineering 36(1), 119–140 (2010)

[10] Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology 11(1), 2–57 (2002)

[11] Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A Language and Environment for Architecture-Based Software Development and Evolution. In: Proceedings of the 21st International Conference on Software Engineering.ICSE 1999, pp. 44–53. ACM, New York (1999)

[12] Pastor, O., Molina, J.C.: Model-Driven Architecture in Practice: a Software Production Environment Based on Conceptual Modeling. Springer-Verlag New York, Inc. (2007)

[13] Pérez-Martínez, J.E.: Heavyweight Extensions to the UML Metamodel to Describe the C3 Architectural Style. ACM SIGSOFT Software Engineering Notes 28(3), 5 (2003)

[14] Reinhartz-Berger, I., Sturm, A.: Enhancing UML Models: A Domain Analysis Approach. Journal of Database Management 19(1), 74–94 (2007)

[15] Reinhartz-Berger, I., Sturm, A.: Utilizing Domain Models for Application Design and Validation. Information and Software Technology 51(8), 1275–1289 (2009)
[16] Roh, S., Kim, K., Jeon, T.: Architecture Modeling Language based on UML2.0. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC), pp. 663–669. IEEE Computer Society, Los Alamitos (2004)
[17] Szyperski, C.: Component Software: Beyond Object-Oriented Programming., 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2002)
[18] Tanenbaum, A.S.: Computer Networks. Prentice Hall PTR, Englewood Cliffs (1985)
[19] Weigert, T., Garlan, D., Knapman, j., Møller-Pedersen, B., Selic, B.: Modeling of Architectures with UML (Panel). In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 556–569. Springer, Heidelberg (2000)
[20] Zarras, A., Issarny, V., Kloukinas, C., Kguyen, V.K.: Towards a Base UML Profile for Architecture Description. In: Proceedings of ICSE 2001 Workshop on Describing Software Architecture with UML, pp. 22–26. IEEE Computer Society, Los Alamitos (2001)