Research on the Adaptive Object-Model Architecture Style

YAO Hai-qiong, NI Gui-qiang

(1. Institute of Communications Engineering, PLA University of Science and Technology Nanjing 210007 China; 2. Institute of Command Automation, PLA University of Science and Technology Nanjing 210007 China)

Abstract The rapidly changing requirements and business rules stimulate software developers to make their applications more dynamic, configurable, and adaptable. An effective way to meet such requirements is to apply an adaptive object-model (AOM). The AOM architecture style is composed of metamodel, model engine and tools. Firstly, two small patterns for building up metamodel are analyzed in detail. Then model engine for interpreting metamodel and tools for end-uses to define and configure object models are discussed. Finally, a novel platform—applicationware—is proposed.

Key words adaptive object-model; metadata; metamodel; patterns; applicationware

It is a general rule that change is permanent. However, software development that specifies system requirement fully in advance can seldom afford changing and always deals with it as an afterthought. It requires a novel approach to give support to an evolutionary software development process from a new perspective. An effective way to meet such requirements is to use an adaptive object-model (AOM), which is presented by Joseph W. et al. With the real power of AOM, that is, description-driven, end-user programming and interpreting at runtime, software developers can make their applications more dynamic, configurable, and adaptable.

The AOM is based on two ideas: description-driven and runtime reflection, to cope with changing user requirements and to be configured as wanted to\(^\text{1,2}\). The variable part of domain objects, including attributes, behaviors, relationships and business rules, is pushed into description. AOMs represent the description as configuration data, or metadata\(^\text{3}\). Metamodel is an object model for describing metadata. Modifying the configuration data changes the object model. Furthermore, changing the object model leads to changes of system behavior. Runtime reflection customizes the system semantics at runtime, with which AOMs interpret the descriptive information at run-time. The main consequence of developing a system with the AOM is that the immediate effect on the system will be made if changes to an object model come up.

End-user programming empowers users and domain experts to customize the system through modeling and configuring rather than coding\(^\text{1}\). This can reduce time-to-market and smooth the communication between users and developers.

The AOM brings innovation to the development process. The developers do not generate end-user applications. Alternatively, they provide users with tools to model and configure object model, as well as build a machine to execute the model, which makes it possible to develop and modify software quickly. But this novel process of development is hard to apply because of a poor understanding of this technology.

In this paper, by analyzing the AOM architecture style in depth, centered on how to describe, build and interpret object model, we propose a novel platform based on AOM, applicationware.

1 Architectural Style of AOMs

The object-oriented community is trying to seek solutions to the challenges associated with building software for dynamic environments. Most
object-oriented systems employ a static object model. In such a model, the architect defines the object model in the design stage and the programmers translate it into codes in the implementation stage. Apparently, the static model is fixed and cannot be changed at runtime. When a new class is added into the system, or the business rule is changed, the programmers have to re-deploy the new system, repeating the cycles of build-compile-release. In contrast, the AOM provides an alternative to traditional object-oriented method, which stores its object model as metamodel and interprets it at runtime\cite{3, 4}. It is a model based on instances, classified as instance-level, while traditional object-oriented programming is ranked as class-level. Since the AOM pushes the complexity into the metadata, which is stored in a database or XML files, the object model enables the system to adapt quickly to the changing business needs by simply changing metadata rather than coding. It is exciting that the era where business rules are buried in code will come to an end. The AOM consists of metamodel, model engine and tools.

1.1 Metamodel

Metamodel is made up of two small patterns to define attributes and behaviors.

TypeSquare

TypeSquare pattern is to descript attributes, illustrated in Fig.1\cite{5, 6}. The upper side is an entity pattern, and the lower one is a property pattern.

An entity pattern is presented to create new classes dynamically at runtime by instantiation. Instead of traditional hierarchy, all properties that are shared by a specific type of objects go into the class EntityType. All properties varied within objects go into the class Entity. When a new object is added, two instantiations must be imposed. The first instantiation is carried on EntityType whose instances represent a type of object, that is, class Entity. The second one is employed on Entity whose instances represent a specific object\cite{6}. Consequently, new classes and objects can be created dynamically and the number of classes is decreased.

A flexible object model needs a scheme that does not require code changes. Property pattern is possible to be a reasonable way to add or remove attributes on the fly, which implements attributes in a different manner. With an instance variable, the property pattern holds a collection of attributes\cite{5}. Each attribute is associated with a unique key. Users use these keys to access, alter, modify or remove attributes in Entity at runtime, as illustrated in left side of Fig.1.

Entity has attributes that can be defined using Property at runtime. As showed in lower side of Fig.1, PropertyType defines the primary type, based on which composite type can be constructed. As a result, a new attribute required can be created dynamically. Just as the Entity, two instantiations are needed to get concrete property value of an object. A new type of property is created through instantiating PropertyType for the first time. Next, specific property value is gained through instantiating the class Property.

However, there are still some problems need to be solved. It is necessary to check access to a property. PropertyType can check whether a given value for a Property is valid. So it is an applicable solution to link every Property to a PropertyType object that can carry out these checks. Also, it is required to define whether a certain type of Property is acceptable for an Entity. Thus, Property is used again and a collection of PropertyType objects for EntityType is defined so that Entity instance can check with its EntityType whether a specific property is acceptable or not. In addition, consistency can not be ignored. Namely, all EntityTypes need to ensure that only legal types for their properties are allowed. TypeSquare pattern, as illustrated in Fig.1, shows these factors mentioned above.
Strategy
With an object-oriented approach, methods define the behavior of objects. Programmers define methods of objects through coding. Consequently, changing business rules leads to write new codes. Also, different entities can rarely share the same method. To customize the behavior of system dynamically, it is critical of a new scheme to control object’s behavior. A strategy is an object that represents an algorithm. The strategy pattern provides a standard interface for a family of algorithms shared by all entities. Users can utilize any algorithm of a given family to define the object’s behavior through configuring rather than coding. These Strategies can be plugged into the appropriate Entities during the instantiation.

1.2 Model Engine
The metadata is based on the knowledge of domain experts rather than developers. In order to interpret the metadata, model engine should be developed for instantiating and manipulating these entities according to their rules in the application. Builder and Interpreter pattern are commonly used for building the structures from the metamodel and interpreting the results. There are two places to interpret the metadata for describing the business rules and object model. The first place is where the objects are constructed, e.g. when the object-model is instantiated into concrete entities, attributes and relationships. The second place is for the actual behavior associated with the entities during the interpretation of the business rules at runtime. Thus, new types of objects are created with their respective attributes and meaningful operations applied on them. And the AOM allows runtime modification of types and hence supports flexibility and fast evolution of applications. However, it brings about difficulties to build and understand these adaptive systems. There are two co-existing object models, user-model and metamodel, as well as two co-existing object systems, that is the model engine and the AOM to be interpreted. The model engine is an interpreter written in an object-oriented programming language. So it is critical to design and develop model engine delicately.

1.3 Tools
One of the main reasons to design an AOM is to enable users and domain experts to change the behavior of the system by defining and configuring entities, rules and relationships. The goal is to allow users to extend the system without programming. User can directly access to the object model. AOM architecture pushes complexity into the configuration data and pushes configuration decision out onto the users. Accordingly, generic tools may be inadequate to deal with an AOM. It results in the development of supporting tools that let users and domain experts handle new products without programming and make changes to their business models at runtime.

End-users can define key concepts for their application domain with configuration tools, which might be interpreted or even compiled at runtime. The AOM defines the objects, their states, the events, and the conditions under which the objects change state. The tools are useful to assist with manipulating the object model. User can “program” without programming. But power comes along with price. The more power and flexibility the users are entitled, the more chances they have to make mistakes. For this reason, the architecture should be undertaken with a solid infrastructure of designing, programming and developing configuration tools. And exposed a full-featured programming language, users have to become specialized programmers beyond their pay-grade. Instead, the key here is to expose only those concepts and rules with which they need to operate.

Moreover, non-programmers are willing to “program” in a way that they could understand. The concepts that the object model exposes should make sense in terms of business notions, in that users are familiar with the business, but unfamiliar with programming. Gradually, the AOM can evolve into a domain-specific language.

2 Our Work
2.1 What Is Applicationware?
We have been confused by some issues since we developed applications in the field of network management. The key problems are presented as
follows: 1) requirements changing within applications domain; 2) rapid changing of business rules; 3) duplicated development; 4) multiple, competing middleware systems; and 5) difficulty in communicating between users and developers. We have being strived to seek solutions for years. Now we propose a novel platform—applicationware—in our research subject, Network Management Integrated Supported System for Information.

Applicationware is characterized as business-driven, adaptable and platform independent, as well as provides a suit of tools for users to customize the application in a domain. It represents the abstraction level higher than middleware and hides differentiation among multiple middlewares. A new development process comes into being along with applicationware.

The domain experts build business model with tools of applicationware. Then end-user application is generated automatically after business model interpreted by the built-in machine of applicationware. As we know, we concentrate on technology when developing middleware-based applications; in contrast, we focus on the nature of business logic when developing based on applicationware.

Generating applicationware, we partly employ AOM technique to make it oriented to the business logic and adaptable. Applicationware is made up of three parts: 1) business descriptor, 2) business rule engine, and 3) business processor.

Business model is a logic mapping from business requirements in real word. According to different business requirements, business descriptor creates business models with metadata. As business requirements changing, end-users or domain experts can modify the business model with configuration tools. So users and domain experts handle new products rather than communicate with programmers, by whom the code is modified to meet the changing requirements.

Business rule engine acts as a rule interpreter and serves to integrate with legacy applications.

Business processor can be treated as a model engine in order to interpret metamodel and generate the end-user application.

2.2 Issues on Development of Applicationware

But it is not easy to develop an applicationware for a general domain. The main issues that should be considered are how to reuse at the level of domain, how to deal with business rules and relationships between objects, and how to supply the tools preferable to domain experts.

Reusability on domain framework level is higher than on program level for avoiding duplicated development in the same domain. But it is hard to realize reuse on the domain framework, since it requires skilled architects who are familiar with the domain enough to abstract generic information.

As much as we know, business rules in the real world are complex and diverse. We must make efforts to describe and configure business rules. It is probably involved in creating a rule language and designing an interpreter for it delicately.

Configuration and administration tools are provided for domain experts to build and modify business model. When developing them, we should take into account business models and context, with which domain experts are conversant. Consequently, it may evolve into a domain-specific language.

3 Summary

The AOM innovate the development process. Users and domain experts can build and modify business model with configuration and administration
tools. Then the end-user application is generated automatically after business model interpreted by the built-in model engine. Due to user customizing and interpreting at runtime, the AOM is suitable for developing dynamic, configurable and adaptable applications. This architectural style can be very useful in systems, which specifically emphasize flexibility and need to be dynamically configurable. Otherwise, advantages are followed with disadvantages as a rule. There could be a higher initial cost associated with developing an AOM. And it is hard to understand and maintain, primarily due to the many levels of abstraction. Recently, related techniques to AOM, including generative programming, metamodeling, model-driven development and agile development, have been done towards flexible and adaptable systems.

After exploring AOM architectural style and the new development process derived from it, we propose a novel platform—applicationware. It is related to several techniques, for instance, business modeling, domain analysis, model-driven development and agile development. Applicationware is presented to adapt to rapid changes and individual requirements. However, applicationware is just a period achievement of our ongoing efforts of research. Currently, our work emphasize on the business descriptor to describe the attributes of object and simple operations. Future work will be focused on business rule engine for configuring and generating complex business rules. And we will make further elaboration of business processor and configuration and administration tools.

References

Brief Introduction to Author(s)
YAO Hai-qiong (姚海琼) is an M.S. candidate at the Institute of Communications Engineering, PLA University of Science and Technology. She received her B.S. degree in computer software from the Department of Computer Science & Technology, Yunnan University, in 1997. Her research interests include software engineering and network management. E-mail: lilyao@vip.sina.com

NI Gui-qiang (倪柱强) is an associate professor and M.S. advisor in the Department of Computer Engineering at the Institute of Command Automation, PLA University of Science and Technology. His research interests are centered on software engineering, network management and distributed computing. E-mail: ngq1966@vip.sina.com