DIFFICULT ISSUES IN DESIGNING ADAPTIVE OBJECT MODEL SYSTEMS

Yun Mai and Jinmiao Li and Greg Butler
Computer Science Department, Concordia University
Montreal, Canada

Keywords: Adaptive Object Model, Reflection, Software Architecture, Repository-based Software Design.

Abstract: The adaptive object model enables a system to change its behavior at run-time without re-programming. It provides an extremely extensible architecture solution for large software systems. As a particular kind of reflective architecture, the core of the adaptive object model encapsulates changeable system properties and behaviors as meta-information. Changing the meta-information reflects changes in the domain. However, this approach leads to a more complex design compared to a traditional object-oriented design and thus its implementation is difficult for developers. This paper provides a general design model that compiles techniques proposed by existing adaptive systems and models. The core of the design model is based on a layered architecture. The paper starts from a high level view of the architecture. It then zooms in different components. Major issues in designing various components are fully discussed. General design solutions are elicited as a result of the discussions.

1 INTRODUCTION

Evolution of a software system is hard. To design a system with future changes in mind becomes an art of software design. If a system is designed in a flexible way then it is still possible to minimize changes that might undermine the original system architecture. A powerful solution to make a system extremely flexible and adaptive is to use the adaptive object model (Foote and Yoder, 1998a; Foote and Yoder, 1998b; Foote and Yoder, 2001; Revault and Yoder, 2001). The model allows an application system to be easily extended and changed at run-time without changing the existing program.

Normally, a system designed with an adaptive object model is hard to understand and develop. Many papers reinforce the concepts on adaptive object model from different perspectives with examples (Johnson, 1997; Manolescu and Johnson, 1999; Martin et al., 1998; Riehle et al., 2000). But since the adaptive object model is most suitable to be used in large systems (otherwise, the system can be easily discarded and re-built from scratch) and it is very expensive to build one, there are not many industry practices that can best demonstrate how systems are built with adaptive object model. One of the well-known successful practices on adaptive object model is the Argo framework (Devos and Tilman, 1998). The system is very large and flexible, but its design is very complex compared to a traditional object-oriented design approach.

Designing an adaptive object model system is complex and difficult. It is therefore necessary to provide a design paradigm for implementing such a system. This paper unfolds the difficult issues in adaptive software system design. By compiling techniques proposed by existing adaptive object model systems, the paper captures and highlights major design issues and provides generic design solutions. The rest of the paper is organized as follows: Section 2 is the state of art of the adaptive object model. Section 3 provides the software architecture and its key features. Section 4 discusses various design issues in the software architecture. Section 5 concludes this paper.

2 RELATED WORK

In recent years, many works have been devoted to the adaptive object model (AOM). The Reflection pattern (Buschmann et al., 1996), Accountability pattern (Fowler, 1997), and Type Object pattern (Martin et al., 1998) are fundamental technologies of the adaptive object model. The Dynamic Object Model pattern (Riehle et al., 2000) captures the basic logical architecture for the adaptive object model. Detailed logical architecture designs are presented in (Yoder et al., 2001a; Yoder et al., 2001b; Yoder and Johnson, 2002).

To allow a software application to change its behavior on the fly, a well-known solution is to build the system as a so-called "reflective architecture" (Buschmann et al., 1996; Forman and Danforth, 1998; Fowler, 1997). A reflective architecture defines a self-descriptive adaptive software system by logically splitting the system into two parts: a base level and a meta level. The base level defines the domain application logic, while the meta level encapsulates and represents system structure and behavior information by defining metaobjects. Changes to metaobjects kept in the meta level affect subsequent base level behavior (Buschmann et al., 1996). In addition to splitting a system into two levels, an interface, called the Metaobject Protocol (MOP), has to be defined to allow clients to specify and manipulate changes to the system. Figure 1 shows the general structure of a reflective architecture.

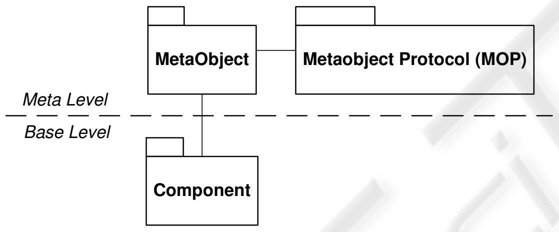

Figure 1: Reflective Architecture

The adaptive object model is a particular kind of reflective architecture. The core of the adaptive object model is the Type Object pattern (Martin et al., 1998). Type Objects explicitly represent type information as instances of a class, namely type class or meta class. Therefore, the type information can be created and modified dynamically at run-time, just like any manipulation of objects in a traditional object-oriented system. All type information are encapsulated in the meta level so that changes to the meta information reflect changes to the system behavior. In addition, attributes, relationships, and behavior of a class are normally represented separately from the class itself. These approaches can be achieved by applying some patterns such as the Property pattern (Riehle, 1997) and Strategy pattern (Gamma et al., 1995). The Type Object pattern, Property pattern, Strategy pattern, and Value Holder pattern (Foote and Yoder, 1998a; Woolf, 1994) together form a Type Square (Balaguer and Yoder, 2001; Yoder et al., 2001a), which is the core architecture of the adaptive object model (Johnson, 1997; Yoder et al., 2001a; Yoder et al., 2001b; Yoder and Johnson, 2002). Figure 2 shows the core architecture of the adaptive object model. In addition to the software patterns presented above, other patterns may also be used to build a real adaptive system. They are specific to system requirements and design implementation. Examples of such patterns include Organization Hierarchy (Fowler, 1997), History (Johnson and Oakes, 1998) (also Historic Mapping (Fowler, 1997)), Rule Object (Arsanjani, 1998; Arsanjani, 2000; Arsanjani, 2001), Serializer (Riehle and et al., 1998)etc.

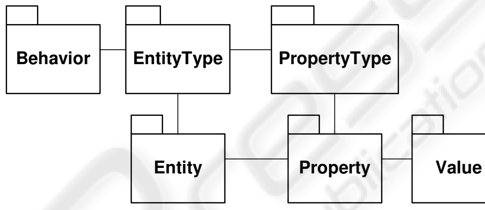

Figure 2: Core Architecture of the Adaptive Object Model

There are some successful industry experiences in using the adaptive object model including the well-known Argo framework and Objectiva framework. Argo is a framework that is used to develop applications for a semi-government organization managing several hundred public schools. The framework behavior is driven by a repository. New applications can be developed through modeling and configuration rather than through coding (Devos and Tilman, 1998). Objectiva is a framework that is used to develop applications for telecommunication billing. It aims to develop a single billing system to handle any kind of telecommunication service (Anderson and Johnson, 1998).

3 SOFTWARE ARCHITECTURE

There are two key architecture features in a system designed with the adaptive object model: metadata driven and layered architecture.

3.1 Metadata Driven

An adaptive system is a system that could adapt itself to requirement changes and system changes on the fly. Such a system is self-describable. It maintains information about system itself and uses this information to remain changeable and extensible. The foundation of the adaptive object model is the Type Object pattern (Martin et al., 1998). As described in Section 2, object types are represented as objects (named type objects) and processed like any objects. By changing the type objects at run-time, system behaviors and states are changed accordingly.

An adaptive object model system is an open metadata driven software system. The system contains a repository for different information (Devos and Tilman, 1998; Forman and Danforth, 1998):

- Domain objects
  These objects are about the knowledge of an application domain. For example, in a university management system, domain objects can be objects representing students, professors, administration, faculties, etc.
- Meta-information
  These are information about the object model that describes the domain object types and their relationships. They can be information about the application structures, business models, user views, etc.

The repository contains all information about an application domain, the domain model, and the model description itself. The repository is accessible at runtime so by changing the meta-information, the system becomes adaptive.

3.2 Layered Architecture

Layering is a common design technique to break down a complex system. It reflects the "divide and conquer" principle and helps to minimize subsystem dependencies. A lower layer provides services to the layer above it. A layered system can be easily understood in isolation and be substituted easily. An adaptive object model system can be broken down into three layers: Front Room, Presentation Server, and Back Room. Figure 3 shows its layered architecture.

- Front Room is a subsystem that provides services to the end user. It includes a set of presentation tools such as data access tools, query management, end-user applications, tools for configuration and administration, etc.
- Presentation Server is a subsystem that maintains a repository for object model and domain objects. It contains some managers that handle object caching, query, transaction and lock. Presentation Server interacts with Back Room subsystem to read/write data from/to different data stores.
- Back Room is a subsystem interacting with data sources. There are two abstraction layers in this subsystem. Data Mapping Engine maps the in-memory data (such as in-memory objects) into an abstract representation (logical data representation). Persistency Engine then maps the logical data representation to its physical storage (physical data representation). Data Mapping Engine has no knowledge of any physical data storage detail, while Persistency Engine has no knowledge of the in-memory data format.

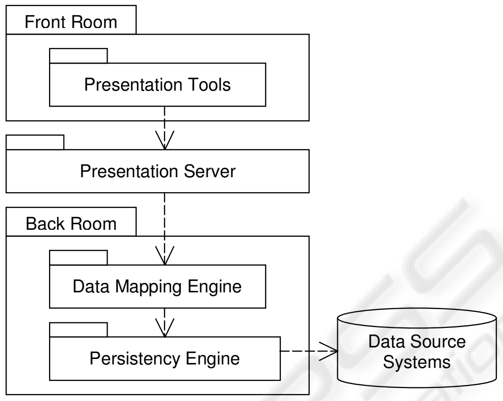

Figure 3: Layered Architecture for AOM

4 MAJOR DESIGN ISSUES OF THE AOM ARCHITECTURE

In this section, we follow the software architecture structure presented above and further discuss major issues that may be encountered in each part of the layered architecture design.

4.1 Front Room

Front Room refers to some tools and applications that provide interactive interfaces to end users. First of all, there are end-user applications that provide main functions to the end user for daily processing. In addition, administration and configuration tools have to be provided to allow the authorized high-level end user to monitor and change objects (including meta objects) in the repository. There are some issues related to the object manipulation in the Front Room.

Issue 1: User Authorization Issue

Since a change of an object in the repository can cause an end-user application to change its behavior, it must be careful to provide authentication rule for only authorized users to do so. Roughly, we divide end users into two groups: operator only runs and uses the end-user applications, while administrator not only is able to run end-user applications but also can use configuration and administration tools to browse objects, add new objects, and change existing objects. Figure 4 shows the use case diagram.

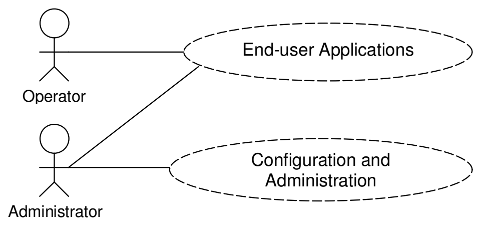

Figure 4: Use Case Diagram for the AOM System

Issue 2: Object Configuration Issue

A typical object configuration scenario in an adaptive object model system is: the user creates some criteria to search for an object, upon receival of a set of matched objects, the user picks one and displays its properties for browsing or configuration. The more adaptive a system is, the more objects have to be stored for configuration. We want to store as much information as possible for configuration. But to provide customized forms for every object to be browsed and configured is a daunting task.

To minimize the workload in creating object browsing and configuration forms, factoring out common objects and common properties is very important. It is generally a good practice to provide a common object query window and a common object browsing window, and for each object, a separate window can be created to show object properties. Creation of object property windows should take advantage of the Type Object pattern so that all object properties can be retrieved and displayed via object introspection. In addition, rules and constraints should be defined to ensure object integrity. Some adaptive systems such as Argo (Devos and Tilman, 1998) and Objectiva (Anderson and Johnson, 1998) provide good examples in designing configuration and administration tools.

Issue 3: End-User Programming

The adaptive object model differs from traditional object model in that the system behavior can be changed at run-time. This is achieved by providing a simple domain-specific language (such as scripts), so sophisticated users can modify existing object behavior or add simple behavior for newly created types without re-deployment. The domain-specific language can be used to specify process rules, class behavior, and various constraints. A specialized compiler should be provided to parse the scripts into an abstract syntax tree or rule objects (Arsanjani, 2000), which will be stored and interpreted for run-time execution.

4.2 Presentation Server

Presentation Server is the middle tier in the adaptive object model software architecture. It bridges the Front Room and the Back Room. Presentation Server provides a data repository that manages the in-memory information. As a data container, there are some important design issues about its capacity, operation performance, data load and update, etc.

Issue 1: Performance Issue

As stated before, Presentation Server is essentially an in-memory repository that contains domain objects that describe the application domain and the meta-objects that specify the meta-information. On one hand, all information about the application domain and the system itself need to be stored as objects in the repository. On the other hand, the repository is an in-memory data container and will have a limited capacity. It is impossible to store all objects in memory in most cases. Therefore, some objects have to be loaded on demand. If the user issues an operation and most related objects are not in memory, the system has to load the meta-objects, and then the domain objects. It is inevitable that the user will perceive a significant delay.

To solve this problem, we need a Cache Manager. The Cache Manager will implement a certain number of caching strategies. So at run-time, depending on the operation context, a caching strategy is chosen and objects are loaded implicitly on demand under unawareness of the end user. When an object is requested, the cache (in-memory repository) is checked first, if the object exists it is returned to the application; otherwise, the Cache Manager is responsible for loading it from the persistent storage (by sending requests to the Back Room). The Cache Manager decides what objects to be pre-stored in the repository according to a caching strategy or a combination of multiple caching strategies. With fine tuned caching strategies, the performance for data access in the adaptive system is improved.

Another performance issue is when to create objects in the in-memory repository. If objects are be created/loaded on demand, we will face the same performance problem since creating/loading a large number of objects will be expensive. A solution is to bulk load objects into the repository initially. A caching strategy will determine which objects to be loaded at boot and a bootstrapping process will interact with the Cache Manager to bulk load objects when the Presentation Server is up.

Issue 2: Concurrence Access Issue

Since objects are cached in the in-memory repository (and stored in some persistent storages too), it is risky that multiple concurrent updates could be made to the same properties of some objects in a distributed environment. So the issue is, how can we control accesses to the objects so that they will not be left in inconsistent states if multiple manipulations occur?

One straightforward and commonly used solution is to use transactions and locks. All data operations must be performed within the context of a repository transaction that keeps track of local changes to one or more objects. Once committed, a transaction is translated into underlying database operations and transactions (Devos and Tilman, 1998). Furthermore, to avoid any accidental corruption of data integrity, a locking mechanism is necessary. The Locking Manager is responsible for ensuring that object properties are correctly locked in a transaction.

Issue 3: Data Consistency Issue

Another problem with object manipulation is object inconsistency. When an object configuration changes, all objects depending on this object may need to be changed accordingly. Some data structures such as Direct Acyclic Graph (DAG) (Shaffer, 1997) can be used to manage the object dependency relationships. Should all dependent objects (both in the in-memory repository and in physical data stores) change immediately, or in-memory objects change first while the others change only when they are loaded? When an object is not referenced any more, should it be destroyed and when? There are many small issues relating to data consistency. The solution somewhat depends on the system requirements (e.g. performance requirements and organization policies). A component named Consistency Manager can be designed to encapsulate the solution for these issues.

Issue 4: Version Control Issue

Object manipulation also brings up an interesting topic on object history management. When an object configuration changes, should we maintain its history? This can be up to a system requirement. But, a good practice is to keep the versions of the object because they are always useful and desirable. For example, if an organization changes one of its statistics forms to a new format since May 1, 2003, should all forms (with data) before this date be changed to adapt to the new form format? Normally, people would like to display the old data with the old form, while display the new data with the new form. In such case, it is important to keep both the new and old form formats in order to interpret different data. A Version Control Manager component is used to control object versioning. It is responsible for associating meta-data version information with the corresponding base-level versioned data, and choosing an appropriate meta-data version to correctly interpret the associated versioned data.

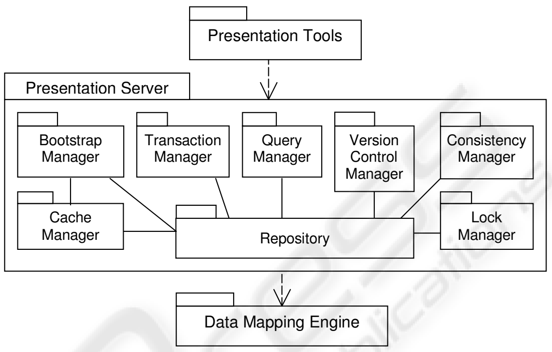

Figure 5: Presentation Server Architecture

Design Discussion

Figure 5 shows the high-level design of the Presentation Server subsystem. The core of this subsystem is the Repository package. It handles any access to the in-memory repository. The data contained in the repository are partitioned into type objects and normal objects constructed according to Figure 2. There are some other auxiliary packages such as Transaction, Query, Caching, Locking, Bootstrapping, Version Control, and Data Consistency, that interact with the Repository to help control access to and populate the repository. Each auxiliary package can define a manager that provides the main functionality for that package. The behaviors are illustrated in Figure 6 for executing a query and Figure 7 for executing a configuration command.

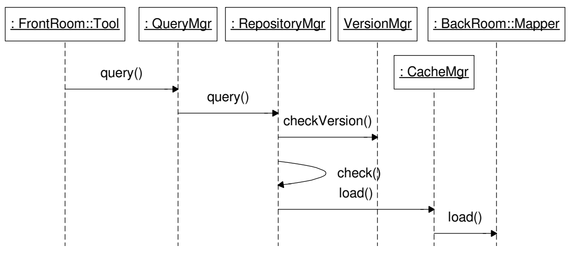

Figure 6: Sequence Diagram for User Query

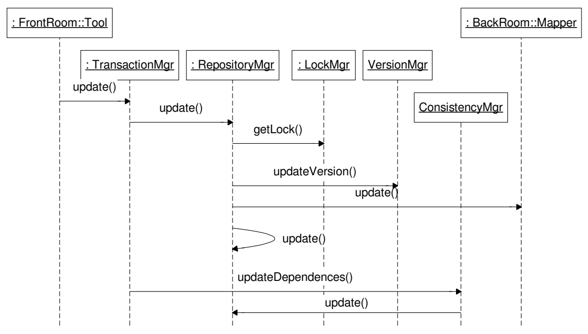

Figure 7: Sequence Diagram for User Configuration

4.3 Back Room

An adaptive object model system is a kind of repository-based system, that is, the application's behavior is driven by the repository. Therefore, we need a persistency mechanism to protect the repository information from loss. Back Room is such a subsystem to handle data persistency in the AOM architecture.

Issue 1: Model Persistency Issue

A traditional system only stores application data (domain information) into persistency. In an AOM system, only store domain information are not enough. To avoid any loss of the meta-information, we also need to store them to a persistency media. In other words, we not only store application information (domain objects) as persistency, but also store meta-objects as persistency.

Issue 2: Data Storage Issue

As discussed above, application information such as objects, code-based rules or constraints, different formats of domain information, should be stored in the persistent storage. Different information may require different data storage carriers. Typical data stores include object-oriented database systems, relational database systems, XML (Extensible Markup Language) documents, flat files, and so forth. For an object-oriented software system, object-oriented database is a straightforward solution for object storage. XML file format is an alternative solution because there exist many ready-to-use XML tools (e.g. SAX and DOM parser) that simplify data process and speed up system development. But for relational database, an object-relational mapping mechanism should be provided to handle the data transformation between in-memory objects and relational data. The following will give a more detailed discussion on the object-relational mapping.

Issue 3: Persistency Access Issue

Since there are different data storage formats, access to persistent storage varies greatly. The problem is, if such chunk of connectivity code has to be repeated wherever a persistency connection is required, it tightly couples the application and the data sources so that migration from one type of data source to another is difficult.

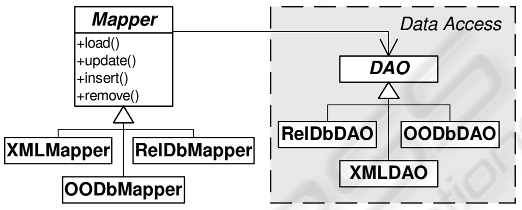

Figure 8: Persistency Access with Data Access Object (DAO) Pattern

Data Access Object (DAO) (Sun Microsystems Inc., 2003) aims to solve persistency access problem. A DAO encapsulates all accesses to a data source. In an AOM system, we can create a DAO class and subclass it with concrete DAOs that handle different data storage types. One concrete DAO is responsible for handling any access to objects stored in that corresponding media type. Figure 8 (grey area) shows an example.

Issue 4: Persistency Mapping Issue

If objects are stored to the same persistency media, they will share the same DAO. But, how can the DAO tell the differences of the objects it is going to handle?

A DAO cannot tell the differences of the objects. All objects must be processed before they are forwarded to a DAO. A mapping mechanism must be provided in order for a DAO to correctly process the objects. Figure 9 shows an example of a mapping engine.

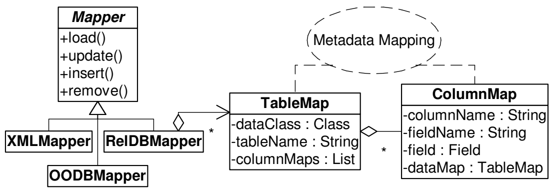

Figure 9: Persistency Mapping with MetaMapping Pattern

Our goal is to map objects in storage to objects in memory. As shown in Figure 9, by applying the

MetaData Mapping pattern (Fowler, 2002), relational database mapper defines a Table Map class which maps each data class to its table name, and a Column Map class to maps data class field types to table column names. By using the information provided by Table Map and Column Map, the application is easy to flatten an object and store it into relational database or assembly it when retrieve it from relational database. Figure 10 shows a scenario that how an object is loaded from relational database system.

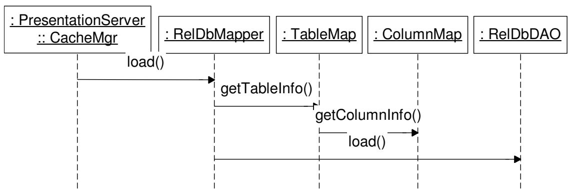

Figure 10: Sequence Diagram for Data Load from Relational Database

Design Discussion

Figure 11 shows a common high-level design of the Back Room subsystem. Back Room contains two levels of data abstraction. The top level, Data Mapping Engine, is the one for logical data mapping. The bottom level, Persistency Engine, is for physical data access. Data Mapping Engine is responsible for mapping in-memory information to logical data representation. The repository not only contains objects, it also contains documents, images, videos, etc. According to the information stored in the repository, different data mapping engines should be provided. In general, we can define two kinds of data mapping engines to handle two major data catalogs.

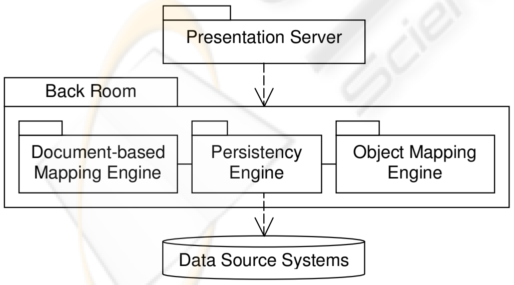

Figure 11: Back Room Architecture

Firstly, an Object Mapping Engine can be defined to map in-memory objects and their properties to their logical data abstraction. These in-memory objects may includes user-defined domain objects, objects in the object model and meta model, rule objects representing process rules, class behavior, and various constraints, as well as system utility objects.

Secondly, for documents processing, a separate mapping engine can be defined to handle document specific mapping. Document processing can be seen in many workflow systems, we will not further discuss them in this paper.

Finally, since a physical data access process is data storage type specific, we can define different media-dependent components in the Persistency Engine. Persistency Engine consists of a hierarchy of DAO objects which handle data storage type specific data load and data store.

5 CONCLUSION

To change a system is hard. The adaptive object model makes it easy to adapt a system to new changes. Although people complain it is hard to understand the adaptive object model and to develop a system with it, we still consider it worthwhile to regard it as a potential architecture and to invest in it, because the payback in terms of flexibility and ease of evolution has great value. In an adaptive object model system, everything is an object including the system aspects that describe the business model and that describe the model itself. Since objects can be created and modified at run-time, it is easy to change the system behavior and extend a system using a common object browser and configuration tool platform. Thus, no programming practice is required to make such changes. The drawback of using the adaptive object model is that it is hard to debug the system behavior since object interaction is implicit due to the reason that the interaction is encapsulated into object configuration. Still, such a system can be hard to maintain if the model is not clear or the components of the systems design are not clear. Hence, the need for a software architecture and design guidelines is imperative.

The main contribution of this paper is that it addresses the complexity and difficulty of developing an adaptive software system. By compiling techniques proposed by existing adaptive object model systems, the paper captures and highlights major design issues and provides generic design solutions. One difference between the solution proposed in the paper and other papers is that we define a layered architecture for the adaptive system. Layering de-couples interactions between subsystems so that it is easy to plug in or to replace a subsystem. The architecture is completely metadata-driven. We discussed various components in the system and difficult issues that have to be considered when designing a system with the adaptive object model. The paper unfolds difficult design issues towards an implementation of an adaptive software system with the adaptive object model. The guidelines suggested in this paper will help software designers to address major design problems inherent in the adaptive system design. Using adaptive object model may introduce certain performance penalty. We will consider various means to improve it in our future work.

REFERENCES

Anderson, F. and Johnson, R. (1998). Objectiva Architecture. *UIUC'98 MetaData Pattern Mining Workshop, Urbana, IL.*

Arsanjani, A. (1998). Meta-Modeling and Grammar-oriented Object Design. *OOPSLA 98 Workshop on Metadata and Active Object Model.*

Arsanjani, A. (2000). Rule object: A Pattern language for Adaptive and Scalable Rusiness Rule Construction (Part 1: Rule Object). *Proceeding of PLoP 2000.*

Arsanjani, A. (2001). using Grammar-oriented object Design to Seamlessly Map Business Models to Component-based Software Architectures. *Proceeding of the International Association of Science and Technology for Development. Pittsburgh, PA, USA.*

Balaguer, F. and Yoder, J. W. (2001). Adaptive Object-Model Architecture. *OOPSLA 2001 Adaptive Object-Model Tutorial. Available on the web at http://www.adaptiveobjectmodel.com/.*

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

Devos, M. and Tilman, M. (1998). A repository-based framework for evolutionary software development (Argo Belgium School System). *UIUC'98 MetaData Pattern Mining Workshop.*

Foote, B. and Yoder, J. W. (1998a). Metadata and Active Object-Models. *Fifth Conference on Patterns Languages of Programs (PLoP '98) Monticello, Illinois.*

Foote, B. and Yoder, J. W. (1998b). Metadata and Active Object-Models. *Workshop Position Paper; OOPSLA '98.*

Foote, B. and Yoder, J. W. (2001). Adaptive Object-Models. *Smalltalk Solutions 2001 AOM Workshop Presentation, Rosemont, IL, USA.*

Forman, I. R. and Danforth, S. H. (1998). *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming.* Addison-Wesley.

Fowler, M. (1997). *Analysis Pattern: Reusable Object Models.* Addison-Wesley.

Fowler, M. (2002). Patterns of Enterprise Application Architecture. *OOPSLA 2002.*

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

Johnson, R. (1997). Dynamic Object Model. *OOPSLA '97.*

Johnson, R. and Oakes, J. (1998). User-Defined Product Framework. *UIUC'98 MetaData Pattern Mining Workshop.*

Manolescu, D. and Johnson, R. (1999). Dynamic Object Model and Adaptive Workflow. *OOPSLA'99 Metadata and Active Object-Model Pattern Mining Workshop, Denver, Colorado.*

Martin, R. C., Riehle, D., Buschmann, F., and Vlissides, J. (1998). *Pattern Languages of Program Design, 3.* Addison-Wesley.

Revault, N. and Yoder, J. W. (2001). Adaptive Object-Models and Metamodeling Techniques. *Workshop Results; ECOOP 2001 Budapest, Hungary.*

Riehle, D. (1997). A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. *Ubilab Technical Report 97-1-1. Zurich, Switzerlang: Union Bank of Switzerland.*

Riehle, D. and et al., W. S. (1998). Serializer. *Pattern languages of Program Design 3. Addison-Wesley, Chapter17*, pages 293–312.

Riehle, D., Tilman, M., and Johnson, R. (2000). Dynamic Object Model. *PLoP 2000.*

Shaffer, C. A. (1997). *A Practical Introduction to Data Structures and Algorithm Analysis.* Prentice Hall.

Sun Microsystems Inc. (2003). Core J2EE Pattern. *http://java.sun.com/blueprints/corej2eepatterns/.*

Woolf, B. (1994). Understanding and Using ValueModels. *Available on the web at http://www.ksccary.com/article6.htm.*

Yoder, J. W., Balaguer, F., and Johnson, R. (2001a). Adaptive Object Models for Implementing Business Rules. *Position Paper for Third Workshop on Best-Practices for Business Rules Design and Implementation, OOPSLA.*

Yoder, J. W., Balaguer, F., and Johnson, R. (2001b). Architecture and Design of Adaptive Object Models. *Intriguing Technology Presentation at the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), ACM SIGPLAN Notices, ACM Press.*

Yoder, J. W. and Johnson, R. (2002). The Adaptive Object Model Architectural Style. *The Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02).*