JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 7, September-August 2008

An Extended Component Model and its Evaluation for Reliability & Quality

R. Senthil, D. S. Kushwaha, and **A. K. Misra**
CSED, MNNIT, Allahabad, India

Abstract

The main focus of this research is on Component Based Software Engineering (CBSE) and reliable generic connectors for the software components. An attempt has been made to describe n-tier architecture; in particular, data access architecture in a component based application. Having established this, an attempt has also been made to evaluate it against the external and internal quality factors. The work tries to establish that enhanced component model (ECM) is a reliable model. It also makes an attempt to express how data access objects (DAO) in the DAO layer interacts with the business-tier and data source in achieving reliable, reusable, robust and scalable component model by implementing Data Adapter interface. To establish these results, a case study has been carried out and it is found that the application so developed is scalable and robust as one can migrate from one data source to another using this quality model.

Keywords: software components, composition, pattern, encapsulation, component-based software engineering, interfaces.

1 INTRODUCTION

Component based Software Engineering focuses on software components and generic connectors that compose them into one unit. A component model needs to define the following things: (i) what components are, i.e. their syntax and semantics; and (ii) how to compose these components, i.e. the semantics of their composition [Lau 05]. A component[Souza99] is a coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the services it provides, (c) has explicit and well-specified interfaces for services it expects from others, and (d) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves. Objects or port-connector type architectural units are used as components in the current component models.The composition mechanisms in such models have method-calls and port-to-port connections. Nevertheless, these models do not define a proper theory for composition. It is trusted that encapsulation and compositionality are key concepts for such a theory.

There are various industrial component models [Lau 05] namely: CORBA CM, MS COM, .NET and Enterprise Java Beans [Micro01]. CORBA CM is a flat model and it does not support hierarchical composition of components. In current software component models, components are typically objects as in object oriented languages, and port-connector type architectural units, with method calls and ADL (architecture description languages [Garla96]) connectors as composition mechanisms respectively.

These component models does not support these standard semantics. Many researchers have proposed different definitions for components and few have been enumerated here.

According to Szyperski [Szype02]:"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Component is defined by Meyer [Russe03] as:

"A component is a software element (modular unit) satisfying the following conditions:

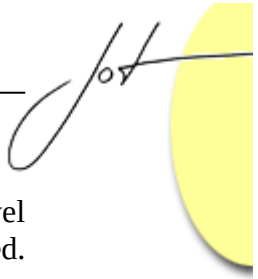
1. It can be used by other software elements, its 'clients'.
2. It possesses an official usage description, which is sufficient for a client author to use.
3. It is not tied to any fixed set of clients."

In both the above definitions, the component composition is not fairly defined in the above definitions, the definition for a component model given in Heineman and Councill [Heine01] looks more appropriate:

"A component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

Nevertheless, there is a commonly accepted abstract view of what a component is, viz. a software unit that contains (i) code for performing services, and (ii) an interface for accessing these services (Fig. 1(a)). To provide its services, a component may require some services.

Though the concept of encapsulation is defined in many component models, component composition is a challenging concept in developing a component model. Connectors are used to compose two components and also play a vital role to take advantage of reusability. Before the enhanced component model is specified using DAO (Data Access Object) with Data Adapter interface in data-tier architecture [Senth07], the value of the proposed work is clarified by specifying the advantages of connectors. Connectors [Baele01] help to localize information about interactions of components in a system such that information is no longer spread over all communicating components and is easier to change and maintain. It also provides natural support for components with incompatible packaging (a connector that mediates a communication among components can accommodate incompatibilities of their interfaces). This is also needed to support dynamic changes in system connectivity (relations among components are not hard-coded



in the code of the components) and software component adaptations. It enables high-level intentions of time, reliability, ordering, performance etc. to be specified and checked. Hence the component based software engineering demands requirements like reusability, compatibility and flexibility that are satisfied when connectors are used to compose components. Using these concepts, an attempt has been made to present a software component model in which the Data Access architecture has the Data Access Object implemented with Data Adapter interface.

An enhanced software component model is proposed for a n-tier architecture that enables to migrate from one data source to another. The UML's definition of component is broad enough [Parris99] to address classic object models, such as COM+, CORBA, and Enterprise Java Beans, as well as alternative object models, perhaps involving dynamic Web pages, database tables, and executables using proprietary communication mechanisms. The formal model for a software component and its use in the n-tier architecture is discussed briefly under related work followed by the proposed Enhanced Component Model (ECM). Having established this, an attempt has also been made to evaluate it against the external and internal quality factors which are described in the next section.

2 SOFTWARE COMPONENTS AND INTERFACES

Overview

The component interfaces play an important role primarily because of the following reasons:

1. The user view on a component is usually referred as a black-box view. An access point of a component is referred as a component interface. Therefore, a component is encapsulated.
2. Compositionality requires that every component (composed and primitive) may be a subcomponent of an arbitrary composite component except itself.

Components and their interfaces are represented in general UML meta-model [Ivers07].

A component is shown as a rectangle with a keyword <<component>>. Optionally, in the right hand corner a component icon can be displayed. A component icon is a rectangle with two smaller rectangles jutting out from the left-hand side. This symbol is a visual stereotype and has the component name. Components can be labeled with a stereotype. There are a number of standard stereotypes, ex: <<entity>>, <<subsystem>>. A component defines its behavior in terms of provided and required interfaces.

Example: A Grep subtype of Filter is defined. The component instance is shown as an anonymous instance of the Grep subtype, and its structure and behavior matches that of the Grep subtype in figure 1.

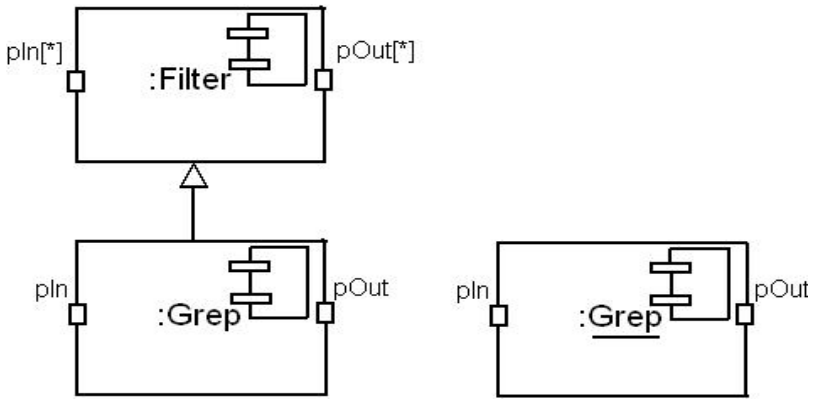


Fig. 1 omponent &Connector Types

An interface is the definition of a collection of one or more operations. It provides only the operations but not the implementation. Implementation is normally provided by a class/ component. In complex systems, the physical implementation is provided by a group of classes rather than a single class.

Interface may be shown using a rectangle symbol with a keyword <<interface>> preceding the name Rectangle can be expanded to show details for displaying the full signature. Interfaces can be provided or required. The services in a Component can be accessed with the help of interfaces.

A provided interface characterizes services that the component offers to its environment. It is modeled using a ball, labeled with the name, attached by a solid line to the component. A required interface characterizes services that the component expects from its environment. It is modeled using a socket, labeled with the name, attached by a solid line to the component which is shown in figure 2.

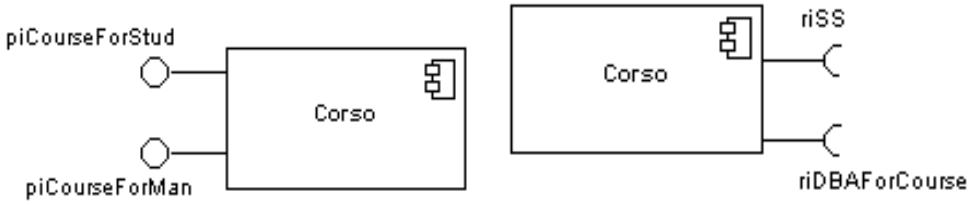


Fig. 2 Component with their provided and required interface

Composition Operators

A component can be composed with another component using composition operator. The composition operator that we use here is connector. A Connector [Baele01] is a communication connection between two components, linking a required interface of a component to a provided interface of another component. Connectors can be developed by composing a number of simple connectors and components (a kind of middleware-like component) into a more complex high-level connector. In analogy with components, a distinction can be made between connector blueprints and instances, as well as between

connector specification, design and implementation. The connection between interfaces of components is the responsibility of the component system. The set of properties attached to a connector depends on the application domain of the component system.

3 EXTERNAL & INTERNAL QUALITY FACTORS

ISO/IEC 9126-1:2001 defines a model for software product quality that categorizes software quality attributes into six characteristics: functionality, usability, efficiency, maintainability and portability. These characteristics are further divided into sub-characteristics. The quality model [Manty04] for external and internal quality is illustrated in figure 3. This model is interpreted for CBSE under the six major properties as:

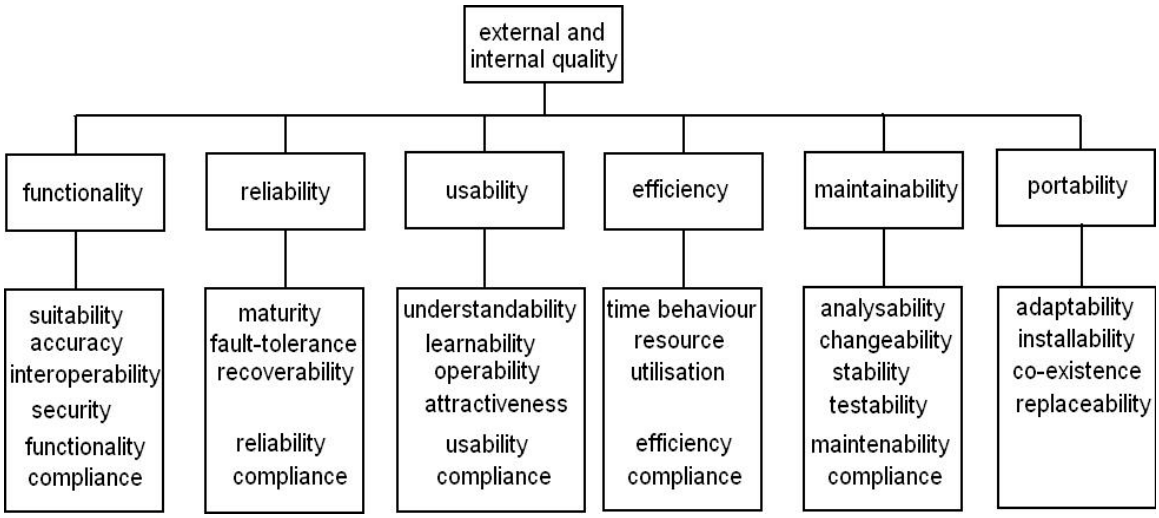


Fig. 3 Quality Model for external and internal quality

- **Functionality:** This characteristic express the ability of a component to provide the required services, when used under specified conditions;
- **Reliability:** This characteristics express the ability of the component to maintain a specified level of performance, when used under specified conditions;
- **Usability:** This characteristic express the ability of a component to be understood, learned, used, configured, and executed, when used under specified conditions;
- **Efficiency :** This characteristic express the ability of a component to provide appropriate performance, relative to the amount of resources used;
- **Maintainability:** This characteristic describes the ability of a component to be modified; later and
- **Portability:** This characteristic is defined as the ability of a component to be transferred from one environment to another.

4 RELATED WORK

Various researches have been carried out on Component Model and its interfaces by [Micro01], [Bottc03], [Merz 06] and [Matid04]. Most of these focus on viewing persistent information and facilitates implementations of data stores into application servers but have not paid much attention to the increase in code efficiency and reduction in complexity of development cycle. A brief description of the existing model is given in the following section.

The EJB Component Model

An Enterprise Java Bean [Micro01] is a deployable component controlled by an application server's container. An Entity Bean(EB) is a persistent, distributed object and consists among other things of a Home Object, an EJB Object, and an entity bean instance. As an entity bean is also a component, the client will never call any of its methods directly, only indirectly through the EJB object, which itself manages the service interactions and delegates the call to the associated entity bean instance.

There are two possible ways to persist an EB: either using container managed persistence (CMP) or bean managed persistence (BMP). Using CMP, the persistent object description is declared in the bean's deployment descriptor. The container then generates the backend-specific code at deployment time. Using BMP, persistence management code (for example, using JDBC) has to be written in the bean manually. Figure 4 shows the principal constituents of an entity bean (EB) in its interaction with a client.

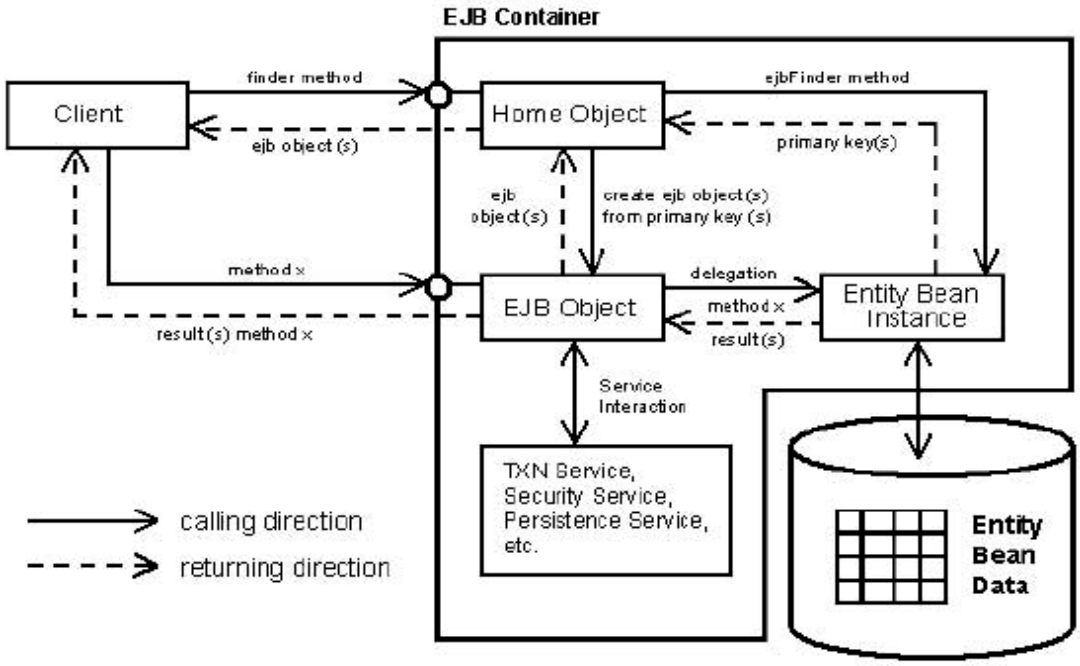


Fig. 4 Exemplification of the EJB 2.0 component model

An Entity Bean[Bottc03] can only be tested inside its operational environment, namely the application server's container. This alone increases the amount of work in deployment and has a complicated debugging processes, besides the overhead needed for coding a component. This work have not paid much attention to minimizing complexity in the development cycle.

JDO Component Model

JDO (Java Data Objects) [Russe03] API provides a standard approach for achieving object persistence in Java technology. The high-level JDO API is designed to provide a transparent interface for developers to store data, without having to learn a new data access language (such as SQL) for each type of persistent data storage. JDO[Merz 06] can be implemented using a low-level API (such as JDBC) to store data. It enables developers to write Java code that transparently accesses the underlying data store, without using database-specific code.

The two main objectives of the JDO architecture, which is shown in Figure 4, are to provide Java application developers a transparent Java technology-centric view of persistent information and to enable pluggable implementations of data stores into application servers.

JDO Class Types

There are three types of classes in JDO:

- Persistence-capable.
- Persistence-aware.
- Normal.

The JDO Programming Model

JDO defines two types of interfaces: the JDO API (in the javax.jdo package) and the JDO

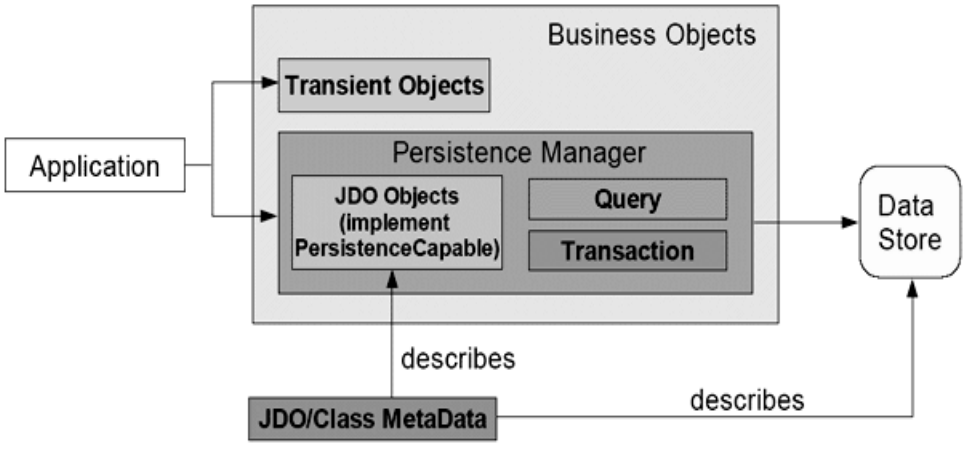


Fig. 3 JDO Architecture

service provider interface (SPI) (in the javax.jdo.spi package). The JDO API is for application developers, and the JDO SPI is for container providers and JDO vendors.

An application has two primary interfaces to JDO:

PersistenceManagerFactory represents the point of access that application developers use to obtain an instance of PersistenceManager. Instances of this interface can be configured and serialized for later use. However, once the first PersistenceManager is obtained from the PersistenceManagerFactory, the factory can no longer be configured. One could use the following code to obtain a PersistenceManagerFactory:

```
   // set some properties for the JDO implementation and data
store
      Properties props = new Properties();
      props.put(...);
      // get a PersistenceManagerFactory
      PersistenceManagerFactory pmf=
  JDOHelper.getPersistenceManagerFactory(props);
```

PersistenceManager is the primary interface for JDO-aware application components. It provides methods to make an object persistent, as well as retrieving persistent objects and removing them from persistent storage.You can use the following code to obtain a PersistenceManager:

```
      PersistenceManager pm = pmf.getPersistenceManager();
```

Once a PersistenceManager is obtained, an application can perform tasks such as making an object persistent, retrieving an object from persistence, deleting an object from persistence, updating an object, and so on.

The benefits of using JDO are the portability, transparent database access, Ease of use, High performance, Integration with EJB. Applications can take advantage of EJB features such as remote message processing, automatic distributed transaction coordination, and security throughout the enterprise.

The disadvantage of the JDO API is that it is not always consistent. Also, specification development progresses slowly. The JDO code is unsafe and difficult to maintain by hand. These disadvantages are overcome with the help of DAO implementing Data Adapter interface. The DAO is consistent in data interface.

DAO Component Model

Many real-world enterprise applications need to use persistent data at some point. For many applications, persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on separate systems

The DAO (Data Access Object) pattern [Matid04] defines the following classes:

- *Client* - represents the data client, which requires access to the data source to obtain, modify and store data, in our case this is the business object.
- *AbstractDAO* - represents the interface which the *ConcreteDAO* implements, providing such an interface assures that the *Client*, by programming to the

interface, remains intact when the *ConcreteDAO* is replaced with another implementation which implements this interface.

- *ConcreteDAO* - represents the key object of this pattern, it abstracts the underlying data access implementation for the Client, providing transparent access to the data source.
- *DataSource* - represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system or another system (legacy/mainframe).
- *ValueObject (TransferObject)* - represents a data carrier object, which encapsulates all data read or transferred to the data source, rather then populating the client with data directly from the *ConcreteDAO,* This object is being used to achieve low coupling between the client and the data source. The Class diagram for the DAO pattern given in [Matid04] is in the figure 4.

In figure 4, only Data Access Object is being used to abstract and encapsulate all access to the data source. The DAO [Matid04] implements the access mechanism required to work with the data source. The DAO completely hides the data access implementation details from its clients.

Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the business component and the data source. This is depicted in the existing software component model.

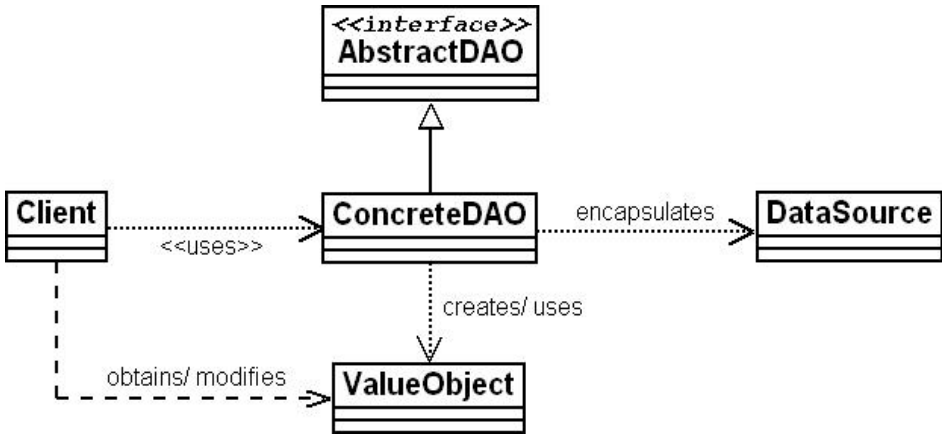


Fig. 4 DAO Pattern – Class Diagram

The performance of the component based software is arrived at by doing Unit test [Cheon02] for the Data Access Object at a test instant t0. Figure 5 and Figure 6 shows how the Data Access Object acts as a connector between the business logic tier and the data tier of the n-tier architecture. The code for the above two models is tested with the help of JUnit framework [Frame07]. Unit testing is being done using this framework in order to check for code complexity and the resulting efficiency of the code. For the same

problem written in object-oriented language carries lower cognitive complexity [Kushwa06] as compared to the program written in a procedural language. Since the case study has been implemented in Java [Micro05] which is an object-oriented language, the complexity of the code is reduced. It is hypothesized that an overly complex code (i.e. an unstructured code with low cohesion) will be difficult to maintain and is likely to be unreliable. In order to create software, our design decisions, cognition, metacognition, learning process and problem comprehensibility should be able to guide us to create software such that overall complexity [Kushw06] is reduced. The most efficient way to deal with developing reliable software for large systems is by creating smaller modules. This is accomplished in the proposed model by having Data Access Object implementing Data Adapter interface in the following section.

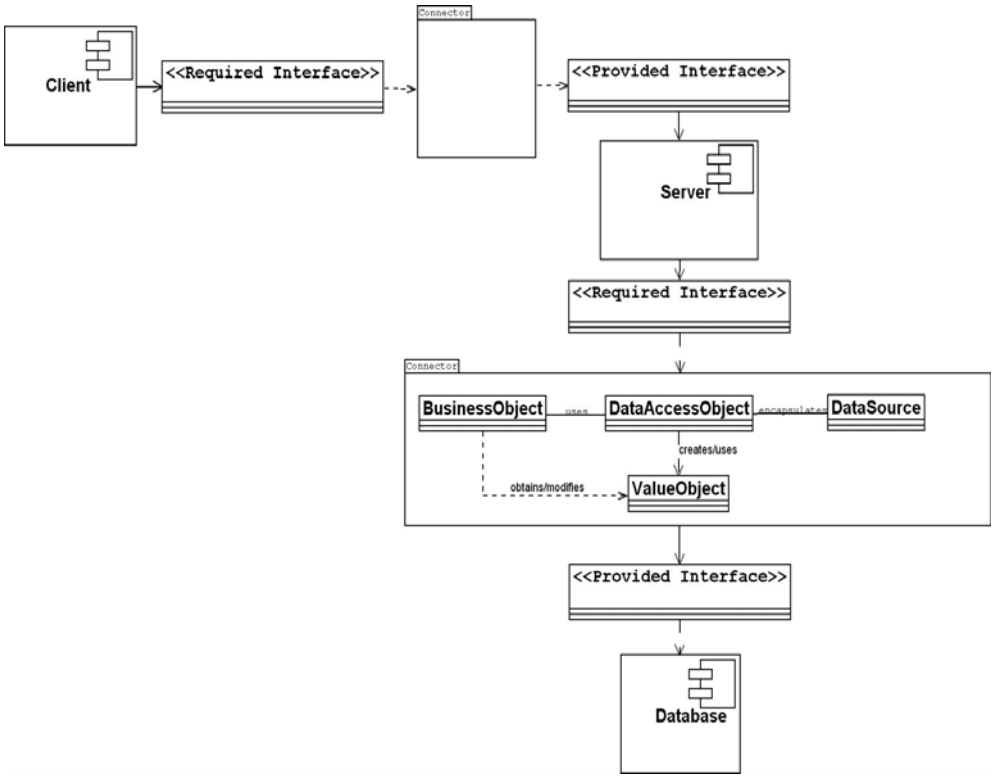
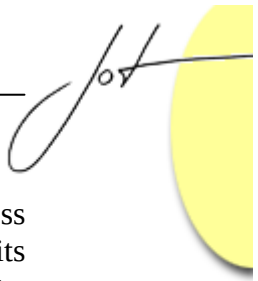


Fig. 5 Existing Software Component Model

5 ENHANCED COMPONENT MODEL (ECM)

The proposed ECM model [Senth07] tries to develop a software component model especially the data access architecture using UML. The different UML notations [Ivers07] used to land at the model have been introduced in the earlier sections.

The DAO [Matid04] implements the access mechanism required to work with the data source by implementing Data Adapter Interface. The data source could be a persistent store like an RDBMS, a repository like an LDAP database. Usage of DAO with



Data Adapter interface leads to portability among several database vendors. The business component that relies on the DAO uses the simpler interface exposed by the DAO for its clients and it completely hides the data access implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, it is found that this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the business component and the data source and hence all DAO classes has been declared as a generic connector package. This is depicted in Figure 6.

Data Adapter interface definition:

The Data Adapter interface(DAI) which is implemented by the DAO, has the following methods in the proposed Enhanced Component Model:

```
public interface DataAdapter {
 // Returns the name of this data adapter
 public String getName();
 // Returns the type of data handled by this data adapter
 public String getType();
 // Returns a list of all operations supported by this data
adapter
 public IOOperation [] getSupportedOperations();
 // For initialization, like opening files or database
connections.
 public void init();
}
```

In the above interface, the first method getName() returns the name of the database which is currently being used by the class implementing this interface. The return type for this method is of type String. The second method getType() returns the type of data handled by the class implementing this interface. When this interface is implemented by the class in which DAO pattern [Matid04] is used, the code complexity and code efficiency of the application is verified and finally, one is able to arrive at the performance of the component based software that is engineered to be efficient and less complex. This model is shown in figure 6. The models in figure 6 and figure 5 are evaluated for the performance in terms of efficiency of the code and complexity of the code. Also the dependencies between objects in the data access object layer of the proposed model is less i.e. there exists loose coupling between objects by introducing DAO layer. It is also found that it is possible to migrate across several data sources. So the application developed using this model is scalable and robust. The application code is reusable as a result of which there is a reduced time to market.

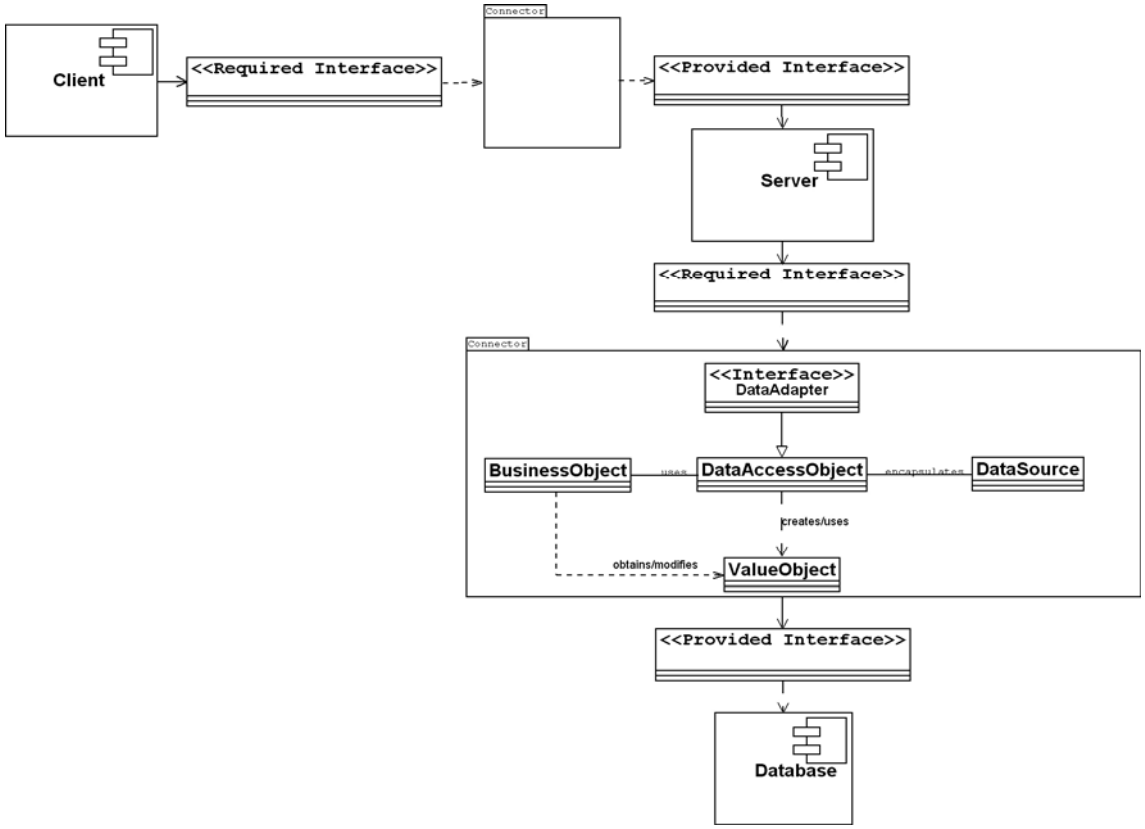


Fig. 6 Enhanced Component Model (ECM)

Performance of the Enhanced Component Model (ECM)

Complex software application solutions have several data sources. A case study has been carried out on course registration manager which has extended the classes involved in data access with Data Access Object [Matid04] in order to corroborate the proposed model. The classes for data access in course registration manager extend Data Access Object and implements Data Adapter interface. The performance of the component based software is instituted by doing Unit test for the Data Access Object at a test instant t_0. The number of success and failure cases is noted at instance t_0. At another test instance t_1, Unit test is performed on the Data Access Object that implements Data Adapter. The performance of the software is assessed in terms of code efficiency. The code is tested for its efficiency by doing Unit testing. Unit testing is done with the junit [Frame07] framework. A sample code has been taken to perform unit testing [Cheon02]. Ant [Kushwa06], a Java-based build tool has been used to compile the java classes and perform unit testing. The java source code is compiled and assembled using Ant. Ant integrates tightly with the JUnit [Frame07] test framework for XP-style unit testing.

Ant uses XML files called build files to describe how to build, test, and deploy an application. Using XML enables developers to edit files directly, or in any XML editor, and facilitates parsing the build file at run time.

The source files are located in 'src' folder of the sample application for the model in figure 5. Now, on entering the command ant from the root folder where the application resides, the java source files Simple.java, SimpleDAO.java and SimpleDAOTestCase.java gets compiled and class files are generated for execution.

To test this code, the Hypersonic SQL database has been employed. Then, the command ant test [Kushwa06] has been used that results in the generation of unit test report. It is observed that at test instance t_0, the time taken for the execution of test cases in the DAO is 0.109 seconds. At a later test instance t_1, the time taken for the execution of test cases for the DAO that implements DataAdapter interface is 0.094 seconds. As the same code was tested with mySQL and postGRESQL databases, it is found that the test report yielded the results that are in table 1. It is observed from table 1 "As the time taken for the execution of test cases decreases, the code efficiency increases". It is also found that it is possible to migrate across several data sources. So the application developed using this model is scalable and robust. The application code is reusable as a result of which there is a reduced time to market. Thus it can be found that the code efficiency of the component-based software is inversely proportional to the execution time for the test cases, with the introduction of DataAdapter interface in Data Access Object.

Database	Timetaken for test cases(in seconds)	
	DAO	DAO implements DataAdapter interface
Hypersonic SQL	0.109	0.094
postGRESQL	0.422	0.19
mySQL	0.116	0.026

Table1: Results comparing proposed model with existing one

Also it is found from the results that implementing DataAdapter interface with Data Access Object in these three sample databases leads to reduction in execution time which is due to consistency of our proposed model. The Chart in figure 7 shows an increase in efficiency of the code through the implementation of DataAdapter interface with Data Access Object.

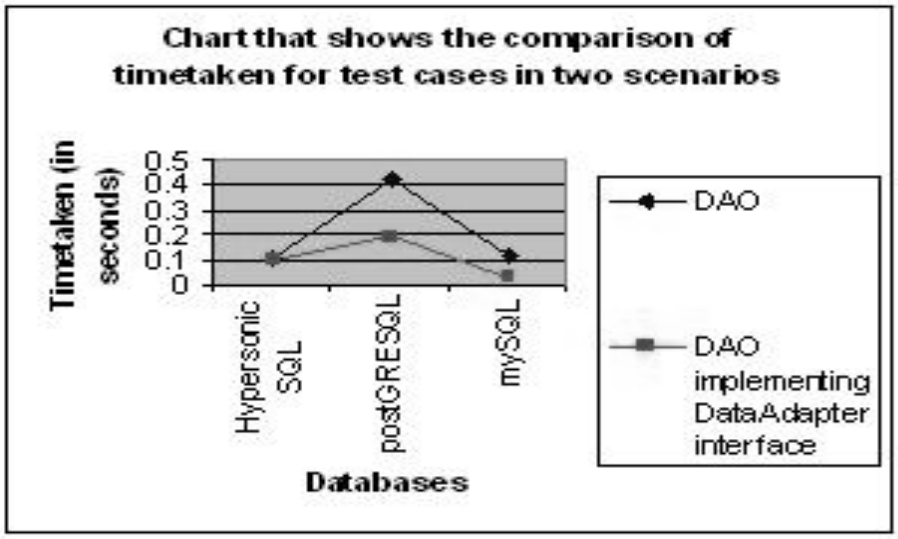


Fig. 7 Graph Showing the Performance of ECM Model

When the unit test is carried out on all the DAOs in the DAO layer, it has been able to ascertain that there is an increase in code efficiency and reduction in code complexity as the Data Adapter interface is implemented by DAOs. It is inferred that there will be data consistency because there is a reduction in execution time of test cases for all DAOs implementing the interface under discussion.

The next section has carried out a case study to establish the performance of the proposed model.

6 CASE STUDY

In order to validate the proposed model, a case study has been performed on Course Registration Manager. The requirement scenario for this case study is taken from the Dean Academics of a Technical Educational Institution. The case study comprises of a user interface component, meant for course management. The JSP component is used for this purpose.

The components in the middle-tier include courses, register, membership, class and schedule.These are considered as business objects in the implementation of the design. The components that lie between middle-tier and database-tier are CoursesDAO, RegisterDAO, MembershipDAO and ClassDAO. The Data Access Objects are the core j2ee patterns [Alur] which are used to encapsulate the client and middle-tier from that of the database. It means that it is possible to migrate from one database to another. The figure 6 above is the general component model as it is possible to migrate from one database to another database because DataAdapter interface is separated from implementation. This model is mapped to this case study “Course Registration Manager” which is shown in figure 8.

Validation of the Results

The classes for data access to view courses and add new courses in course registration manager are BaseDAO, CoursesDAO, CoursesDAOTestCase and Course. The validation considers two different instances of time t0 and t1. At t0, the CoursesDAO class extends BaseDAO. The performance of the component based software is arrived at by doing Unit test[Cheon02] for the CoursesDAO class that extends BaseDAO at a test instant t0. The number of runs, number of success and failure cases is noted at instance t0. At another test instance t1, Unit test is performed for the class that extends Data Access Object and implements Data Adapter interface.

The number of success and failure cases is noted at this instant. Unit testing is done with the junit [Frame07] framework. After configuring the system by giving the proper classpath for these testing tools, the code is tested for its efficiency using Unit testing as discussed in the previous section.

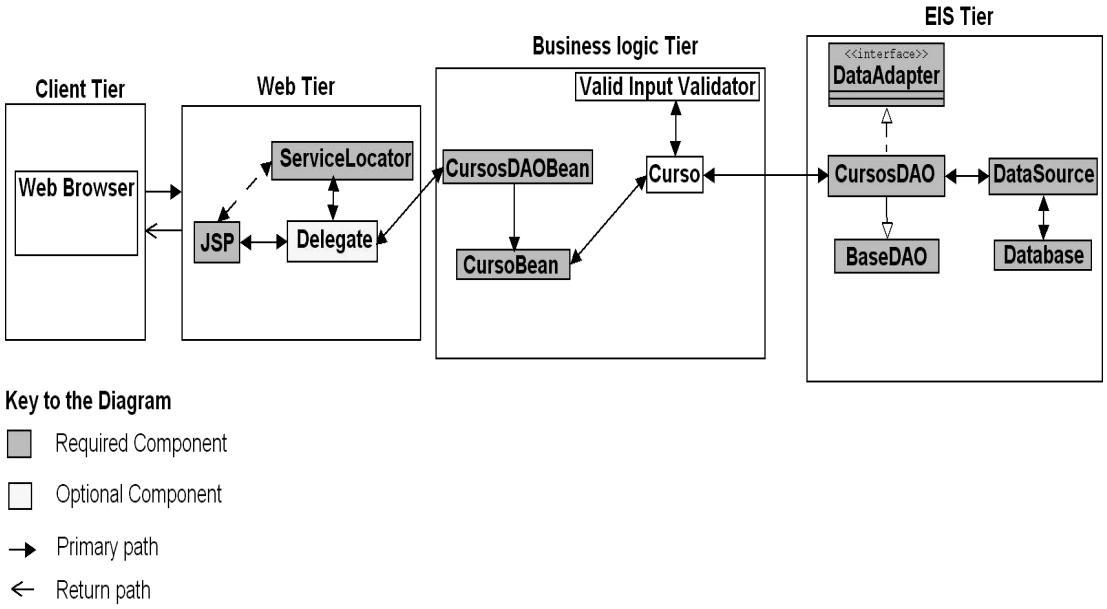


Fig. 8 Architecture Map of Course Registration Manager

Ant uses XML files called build files to describe how to build, test, and deploy an application. Using XML enables developers to edit files directly, or in any XML editor, and facilitates parsing the build file at run time. The XML file build.xml below is used to build and test the code for the database My SQL Server. The source files are located in 'src' folder of the sample application. Now, on entering the command ant, the java source files Course.java, CoursesDAO.java and CoursesDAOTestCase.java gets compiled and class files are generated for execution.

In the next step, the command ant test [Kushwa06] results in the generation of unit test report. At test instance $t_{0,}$ the time taken for the execution of the test cases is 2.312 sec. At time t_1, the class CoursesDAO that implements DataAdapter interface is tested for

code efficiency and the time taken for the execution of the test cases is 0.734 sec. When the same code was tested with hypersonic SQL and postGRESQL databases, the results were generated as in table 2. The Unit test performed on CoursesDAO in the two cases that has been undertaken and finally results could be arrived as in Table 2. It can be seen that the execution time decreases substantially when an application is implemented where DAO implements DAI. The chart in figure 9 shows the performance of the ECM Model.

Database	Timetaken for test cases(in seconds)	
	DAO	DAO implements DataAdapter interface
Hypersonic SQL	2.641	1.327
postGRESQL	2.837	0.553
mySQL	2.312	0.734

Table 2 Results that compares the proposed ECM with existing one for the case study

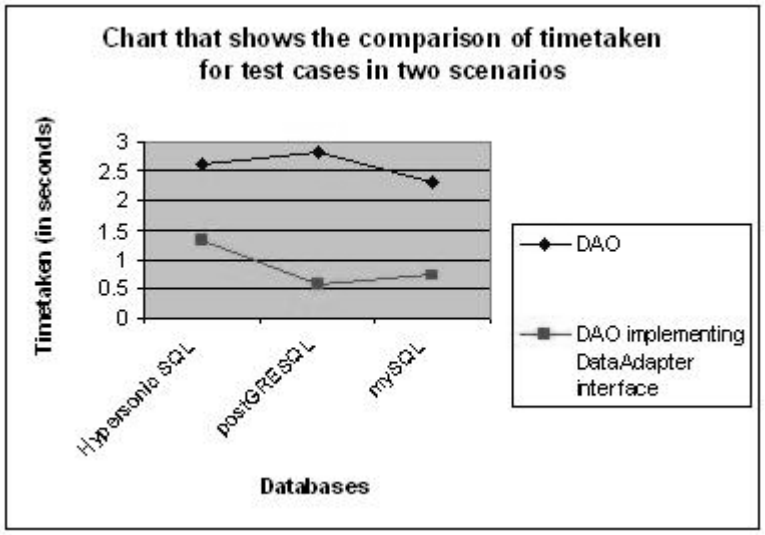
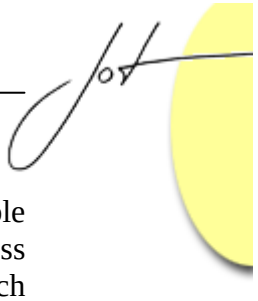


Fig. 9 Graph Showing the Performance of ECM Model

The Unit test conducted on RegisterDAO, MembershipDAO and ClassDAO results in the same effect like increase in code efficiency, reduction in code complexity and consistency in data interface.

7 VALIDATION OF ECM FOR EXTERNAL AND INTERNAL QUALITY FACTORS

In the ECM Model, a layer of DAOs along with the Data Adapter interface makes it easier for an application to migrate to a different database implementation. The business objects have no knowledge of the underlying data implementation. Thus, the migration involves changes only to the DAO layer. Hence we can infer that the application developed using the ECM Model is portable across several database vendors. The



external quality of the component model is apparent. Database migration is possible without any changes in business logic. Alterations are needed only in the data access logic of the DAO layer. Therefore, the DAO layer of the ECM model is adaptable which projects the component model's internal quality.

Since the DAO code that implements Data Adapter interface can be unit tested on the client, the ECM model is testable and thus it is evident that the internal quality of testability adheres to the component model. The use of data access objects at the database level in the component model allows multiple data sources to utilize the same logic without any re-coding. Therefore, the ECM Model is compliant with multiple data sources which reveals that the code developed out of the model is maintainable. Hence the internal qualities of maintainability namely testability and compliance holds good for the component model under discussion.

Efficiency is an external quality of component [Cheon02]. Time behaviour and resource utilization are internal qualities of component. As the execution time of test cases in DAO that implements Data Adapter interface decreases with respect to the one with only DAO, the code efficiency increases and thus the code complexity is reduced. With the implementation of Data Adapter interface with DAO class, all DAO methods must relinquish control of acquired database resources like connection, statements, and result sets. This control is accomplished even by a novice programmer. Hence, we infer that the ECM model is efficient.

The ability of the software(developed from ECM model), systems and business processes to work together to accomplish a common task such as accessing data from several database vendors is called operability / portability of the system. As it is possible to migrate across several data sources, the model is portable. When a component uses a vendor-specific API, it is locked into that vendor's product line. Since the ECM model has the DAO layer which provides a layer of indirection that isolates vendor-specific code in a class or several classes, where it can easily be replaced if necessary or desirable, it is usable and hence the quality of usability holds true.

With data persistence as the key element of the ECM Model, it is found that there is a lot of maturity in the component model. For a system developed using ECM Model, it works with performance degradation when the system fails. Also the DAO layer of the Model is compliant with several database vendors. Hence the Model is reliable.

All the required services from the business-tier such as creation, retrieval, updation and deletion operations are provided by the services in the data-access logic of the DAO layer which, in turn, does all the operations accomplished with that of the database. Hence the DAO layer of the ECM model has all the services that can be provided to the business-tier with the help of database. Therefore the internal quality namely suitability holds true.

Usage of ECM Model to retrieve records across several database vendors leads to data accuracy. There is an improved functionality in the ECM model in which a DAO is used to provide CRUD-type functionality for abstracting a database from business logic. CRUD is an acronym commonly used in the database world that stands for: (1) Creating

new records in the database (2) Reading records from the database (3) Updating records in the database (4) Deleting records in the database. Apart from this, it is possible to know the name of the database and the type of data supported by that database.

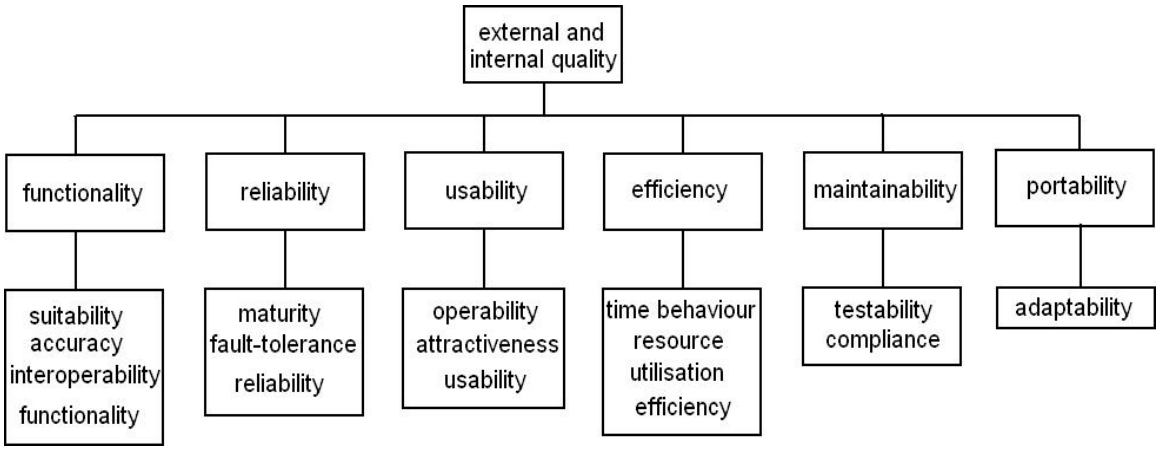


Fig. 10 External and Internal Qualities for ECM Model

Since the ECM model enables to migrate across several database vendors, it is established that the model is compliant. Thus, there is an improved functionality which is an external quality of component model. The quality attributes that are applicable to ECM Model were identified from figure 1. They are given in figure 10.

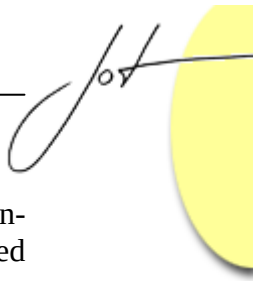
8 RESULTS, CONCLUSION AND FUTURE WORK

This research makes an attempt to establish and validate ECM. An attempt has also been made to evaluate ECM model against the external and internal quality attributes. To establish the same, a case study named "Course Registration Manager" was undertaken.

Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components.

The functional requirements of the ECM model are validated with the help of execution time of the test cases and thereby it is concluded that code efficiency is increased and the code complexity is reduced by mapping the ECM model onto the course registration manager. The analysis is provided on an abstract level with no focus on concrete component model characteristics. The component composition is being achieved through the usage of Core J2EE Design Patterns Data Access Object implementing Data Adapter interface in this paper. Most of the requirements needed for a component model are met in the ECM model.

The non-functional requirements (quality characteristics) of the ECM model were evaluated against the external and internal quality factors as put forward by ISO/ CE2 9126-1:2001 in this work and it is found that the proposed ECM satisfies most of the attributes of the Quality factors.



In future, an attempt shall also be made so that few functional and all of the non-functional requirements for a component model shall be satisfied through aspect-oriented component based software engineering.

REFERENCES

[Garla96] M.Shawand D.Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[Parris99] Allen Parrish, Component Based Software Engineering: A Broad Based Model is needed, Brandon Dixon, David Hale in *International Workshop on Component- Based Software Engineering proceedings*, May 17th-18th 1999, pp. 43-46.

[Souza99] Desmond F. D' Souza, Alan Cameron Wills, *Objects, Components and Frameworks with UML*, The Catalysis Approach, Addison-Wesley, 1999.

[Baele01] Stefan Van Baelen, Software Development Process for Real-Time Embedded Software Systems, *Information Technology for European Advancement*, Dec. 2001.

[Heine01] G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[Micro01] Sun Microsystems: Enterprise JavaBeans Specification, Version 2.0. Palo Alto, 2001.

[Cheon02] Yoonsik Cheon and Gary T. Leavens, A Simple and Practical Approach toUnit Testing: The JML and JUnit Way, In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Lecture Notes In Computer Science; Vol. 2374, 2002, pp. 231 - 255.

[Szype02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

[Bottc03] Stefan Böttcher and Robert Hoeppe, Do We Need Components for Persistent Data Storage in the Java 2 Enterprise Environment?, *NetObjectDays 2002*, Vol. 2591, 2003, pp. 152–165.

[Green03] Greenwich, *Java Development with Ant*, Manning, 2003.

[Meyer03] B. Meyer. The grand challenge of trusted components. In *Proc. ICSE 2003*, IEEE, 2003, pp. 660–667.

[Russe03] C.Russell. Java Data Objects(JDO) Specification JSR-12. Sun Microsystems, 2003.

[Manty04] Annukka Mantyniemi, Minna Pikkarainen and Anne Taulavuori, A Framework for Off-The-Shelf. Software Component Development and Maintenance Processes. *Espoo 2004. VTT Publications*. 525, 2004, pp. 88-89.

[Matid04] Danijel Matid,Dino Butorac and Hrvoje Kegalj, Data Access Architecture in Object Oriented Applications Using Design Patterns, IEEE MELECON 2004, May12-15, 2004, Dubrovnik, Croatia.

[KiuL05] Kung-KiuLau, MarioOrnaghi and ZhengWang, A Software Component Model and its Preliminary Formalisation, *FMCO 2005*, Oct. 2005, pp. 1-21.

[Lau 05] K.K.Lau and Z.Wang. A taxonomy of software component models. In *Proc. 31st Euromicro Conference*, IEEE Computer Society Press, Sep. 2005, pp. 88–95.

[Micro05] Sun Microsystems, Java, Version 2.0, 2005.

[Kushw06] D.S.Kushwaha and Misra A. K, "Cognitive Software Development Process and Associated Metrics -- A Framework", In *Proceeding of the 5th IEEE International Conference on Cognitive Informatics (ICCI'06)*, 2006.

[Kuhwa06] D.S.Kushwaha, R. K. Singh and A. K. Misra, CICM: A Robust Method to Measure Cognitive Complexity of Procedural and Object-Oriented Programs, *WSEAS Transactions on Computers*, Vol. 5, no. 10, Oct 2006, pp.2348-2355.

[Merz 06] Matthias Merz and Markus Aleksy, Using JDO Secure to Introduce Role-Based Permissions to Java Data Objects-Based Applications, *DEXA*, Lecture Notes in Computer Science, Vol. 4080, 2006, pp. 449-458.

[Frame07] JUnit Framework, http://junit.sourceforge.net/, a Testing Tool, March 2007.

[Ivers07] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, Jaime Rodrigo, Oviedo Silva, “Documenting Component and Connector Views with UML 2.0, Technical report, Carnegie Mellon University, 2007.

[Senth07] R. Senthil, D. S. Kushwaha, A. K. Misra, An Improved Component Model for Component Based Software Engineering, ACM SIGSOFT, Software Engineering Notes, Vol. 32, no. 4, Jul 2007 (To Appear).

[AlurWeb] Deepak Alur, John Crupi, Dan Malks, *Core J2EE Design Pattern: Best Practices and Design Strategies*, Java 2 Platform Enterprise Edition Series. Sun Microsystems.

About the authors

Dr. D.S.Kushwaha received his Doctorate Degree In Computer Science & Engineering from Motilal Nehru National Institute of Technology, Allahabad, India in the year 2007 under the guidance of Dr.A.K.Misra. He is presently working with the same Institute as Assistant Professor in the department of Computer Science & Engineering. His research interests include areas in Software Engineering, Data Mining, Cognitive Sciences, Object Oriented Technologies and Data Structures. He can be reached at dharkush@yahoo.com

Dr.A.K.Misra received his Doctorate Degree In Computer Science & Engineering from Motilal Nehru National Institute of Technology, Allahabad, India in the year 1990. He is presently working with the same Institute as Professor in the department of Computer Science & Engineering. His research interests include areas in Software Engineering, Programming Methodology, Artificial Intelligence, Data Mining, Cognitive Sciences, Object Oriented Technologies and Data Structures. He has over 60 International publications in various Conferences & Journals. He can be reached at arun_kmisra@hotmail.com

R. Senthil, CSED, MNNIT, Allahabad, India can be reached at sentil77@gmail.com