A Framework for Component Design using MVC Design Pattern

ABHIK SENGUPTA[1]
SABNAM SENGUPTA[2]
SWAPAN BHATTACHARYA[3]

[1]Cognizant Technology Solutions, Plot GN-34/3, Sector V,
Salt Lake Electronic Complex, Kolkata - 700091, India
abhik.sengupta@cognizant.com
[2]Dept. of Information Technology,
B. P. Poddar Institute of Management and Technology,
137, V.I.P Road, Kolkata - 700052, India
sabnam_sg@yahoo.com
[3]National Institute of Technology, Durgapur - 713209, India
bswapan2000@yahoo.co.in

Abstract. In this paper, we propose a framework where functional requirements are traced from use case model to component model via analysis and design models. Here, components of the component models are derived by grouping and packaging design classes based on the type of analysis classes they are derived from. As there are three different types of analysis classes: boundary, controller and entity, the design classes derived from the corresponding analysis classes get classified at the first iteration. The components thus derived using this approach form the components of Model View Controller Architecture; different components having design classes of similar functionalities. This framework can be used to verify and ensure that use case flow of events is traced in analysis model and then to component model via design models. The architecture with the components designed using this framework also ensures separation of concerns, roles among the components to achieve high cohesion and low coupling.

Keywords: Model-View-Controller design pattern, Requirement Traceability, Use case model, analysis model, design model, component model, Component based architecture, UML Component diagram, XML.

(Received July 17, 2008 / Accepted November 05, 2008)

1 Introduction

Component-Based development, realizing the intuitive and attractive idea of rapidly obtaining complex software systems by the assembly of simpler components, has long captivated the industrial practitioners with the promise of cheaper products with higher reliability and maintainability. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Reusability, whose benefits include both the reduction of costs and time-to-market of software products, is a key issue in software engineering. Component-based software development has emerged to increase the reusability and interoperability of pieces of software. Component-based development aims at constructing software artifacts by assembling prefabricated, configurable and independently evolving building blocks, the so-called components.
However, it is only via a rigorous design discipline and by adopting standard modeling notations as well as strict documentation and design rules that components independently built can effectively interact. This is the basic notion of Design-by-Contract [22], discipline originally conceived for Object-Oriented systems, but even better

suited for Component Based development; indeed Objects and Components, though differing concepts, share many aspects. In recent years, the focus of software development has progressively shifted upward, in the direction of the abstract level of architecture specification. High-level and standardized models must be adopted, in such a way that the consistency (compatibility and the interoperability) among components can be verified as early as possible. The widespread adoption of the Unified Modeling Language (UML) evidences this trend, and its flexibility to specialized yet standard-compatible extensions, where necessary, provides a valuable tool for pursuing this direction. In particular, Cheesman and Daniels [11] describe how UML can be specialized for modeling within a Component Based paradigm embracing the basic principles of the Design-by-Contract approach [22]. Very recently, the OMG Model Driven Architecture (MDA) approach to development pursues a complete separation between the base platform independent model of an application, and the descriptions of one or more platform-specific models, describing how the base is implemented on each of the supported platforms [6]. The base model in MDA is specified in UML.

Model-view-controller (MVC) is an architectural pattern used in software engineering. In complex computer applications that present lots of data to the user, one often wishes to separate data (model) and user interface (view) concerns, so that changes to the user interface do not affect the data handling, and that the data can be reorganized without changing the user interface. The model-view-controller solves this problem by decoupling data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

This work focuses in that direction of designing and packaging components with the design classes. These design classes trace the analysis model that is derived from the use case model. As there can be three types of analysis classes: boundary, controller and entity, the design classes also get categorized accordingly. The components, too, being derived by packaging design classes of similar functionalities get their roles defined at a very early stage. There can be three types of roles:

Model: Domain specific representation of the information, i.e., the data models,

View: Components, through which the users interact with the system,

Controller: Components that get invoked directly with the user interaction and they invoke the appropriate models based on the user input.

These components build Model-View-Controller (MVC) architecture, which is essentially component-based architecture.

2 REVIEW OF RELATED WORKS

Lots of research works are going on in the field of Component based architecture. In most of these works, special interest has been given recently to the reconfiguration and migration of components in component-based system. In [19], Wallnau K et al have discussed about the relationship of software architecture to software component technology.

Some design methodologies addressing component-based development have been proposed recently. Most of them are based on the UML [5], c.f. [14].

Cheesman and Daniels [11] describe how UML can be specialized for modeling within a Component Based paradigm embracing the basic principles of the Design-by-Contract approach [22]. Amber [29] is an abstract component based modeling language combines a model-based approach [29] with a UML-based approach [14]. This combined approach aims at profiting from the advantages of both approaches.

Catalysis [15]is another complex software development process based on UML. Similarly to the Unified Process [17], Catalysis is much like a process template, which can be tailored according to a particular development project. Catalysis being flexible and scalable, it is popular among software developers. A major benefit of Catalysis is its explicit use of components.

However, being a broad software development process, Catalysis is not completely component-oriented. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. The differences of Component Based Approach to the Object Oriented approach are discussed in [27].

Component-based software development (CBSD) focuses on building large software systems by integrating previously existing software components. By enhancing the flexibility and maintainability of systems, this approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the spiraling maintenance burden associated with the support and upgrade of large systems. CBSD shifts the development emphasis from programming software to composing software systems [12]. According to Bachmann et al. [7], in a CB system, components and frameworks should have certified properties; and these certified properties should provide the basis for predicting properties relative to the whole system built out of those components. In recent years, the focus of software development has progressively shifted upward, in the di-

rection of the abstract level of architecture specification. High-level and standardized models must be adopted, in such a way that the compatibility and the interoperability among components can be verified as early as possible. The widespread adoption of the UML evidences this trend, and its flexibility to specialized yet standard-compatible extensions, where necessary, provides a valuable tool for pursuing this direction [18].

At the foundation of the Component-Based approach is the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once, rather than many times, and that common systems should be assembled through reuse rather than rewritten over and over. CBSD embodies the “buy, don’t build" philosophy espoused by Fred Brooks [8]. CBSD is also referred to as component-based software engineering (CBSE) [9], [10].

Component-based systems encompass both commercial-off-the-shelf (COTS) products and components acquired through other means, such as non-developmental items (NDIs).

Because individual components are written to meet different requirements, and are based on differing assumptions about their context, components often must be adapted when used in a new system.

Components must be adapted based on rules that ensure conflicts among components are minimized. The degree to which a component’s internal structure is accessible suggests different approaches to adaptation [28].

One recent trend is toward a “product line" approach that is based on a reusable set of components that appear in a range of software products. This approach assumes that similar systems have similar software architecture and that a majority of the required functionality is the same from one product to the next. The common functionality can therefore be provided by the same set of components, thus simplifying the development and maintenance life cycle. Results of implementing this approach can be seen in two different efforts [21], [1].

While there are several efforts focusing on component qualification, there is little agreement on which quality attributes or measures of a component are critical to its use in a component-based system. A useful work that begins to address this issue is “SAAM: A Method for Analyzing the Properties of Software Architecture" [16]. Another technique addresses the complexity of component selection and provides a decision framework that supports multi-variable component selection analysis [20]. Other approaches, such as the qualification process defined by the US Air Force PRISM program, emphasize “fitness for use" within specific application domains, as well as the primacy of integratebility of components [4]. Another effort is Product Line Asset Support [3].

There are a handful of systems where COTS has been implemented successfully, e.g., Deep Space Network Program at the NASA Jet Propulsion Laboratory [25], Lewis Mission at NASA’s Goddard Space Center [2], Air Force Space and Missile System Center’s telemetry, tracking, and control (TT & C) system called the Center for Research Support (CERES) [13].

As in recent time the focus in the software industry has moved into reusability of components and creation of repository of components, it would be effective if we can design a component model where the roles of different components are very well defined that would make the components easily replaceable and modifiable as the components become highly cohesive and less coupled with each other. With that aim we have proposed a XML based framework for designing components from requirements in [24]. In [23] and [26], we have proposed formal approaches using Z notation for formalization of functional requirements and some of the widely used UML diagrams in analysis and design phases of software development. In [24], we have proposed a methodology for deriving components from requirements. This paper is an extension of that approach. In this paper, we propose architecture, based on Model-View-Controller design pattern in designing a component that trace functional requirements.

3 SCOPE OF WORK

A software component is a physical, replaceable part of a software system that packages implementation and provides realization of a set of interfaces. When we are considering software architecture based on components, a component should have a specification, it should have an implementation, it should conform to some standards, it has to be package able into modules, and it should be deployable. A component specification is usually complete; it contains all the information that a client of the component needs to know. A component specification makes it easier to buy, sell, and replace components if the component fulfils its contract, it should function correctly in the system. From a client’s perspective or user’s perspective, there is no need to explore into a lower level of detail before using the component.

But, from an architectural perspective it is extremely important how these components are built so that there is a very clear separation of concerns. The architecture based on these components can achieve a very high cohesion and very low coupling. With that vision, we here propose to derive component models that trace to

the use case models that handle functional requirements specified in the Software Requirement Specification (SRS) Document. Analysis classes in the analysis model realize use case models and analysis classes are traced to design classes.
As analysis classes are classified as: boundary, control and entity classes, the design classes also get classified accordingly.
Packaging design classes that play similar kind of roles in the architecture derives the components of component model. This approach is illustrated in Figure 1.
When different components are assembled, they are assembled through the interfaces they implement. Interfaces are specifications of services provided by classes or components. Interfaces are most closely associated with components; a component without an interface may be technically well formed, but suspect. Functional Specification of these components and consistency verification among them are very important to ensure cost effectiveness at a very early stage of deployment.
In this paper, we propose a component based architectural framework that follows Model-View-Controller design pattern where roles of the components are well defined; ensuring separation of concerns. Packaging design classes of similar roles; ensuring high cohesion and low coupling build these components. We propose several XML schemas for different models used in different phases of software development.

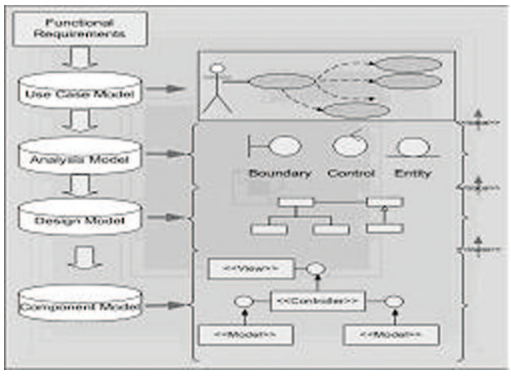

Figure 1: The Component Design Model

4 FUNCTIONAL REQUIREMENTS TO COMPONENTS

Traditional object-oriented software development aims at providing reusability of object type definitions (classes), at design and implementation levels. In contrast, component based development aims at providing reusability of components at deployment level. In this way, components represent pieces of functionality that are ready to be installed and executed in multiple environments. In this paper, we propose to address a design issue in component-based development, i.e., separation of concerns, roles of components to achieve low coupling and high cohesion among the components. For that, we propose some restrictions, following best practices that are adhered to by most designers in any case, in designing these components. The restrictions are:
1. Each use case of use case model has to be traced in
a. One or more boundary classes
b. One or more entity classes
c. One or more control classes
In the analysis model
2. An analysis class has to be traced in one or more design classes.
3. If a design class in design model traces more than one analysis classes, then the type of the analysis classes has to be the same, i.e., either one of boundary, control and entity.
4. When a design class or a set of classes are grouped and packaged to build components, the design classes that trace same type of activity classes have to be grouped.
For example, if there are three design classes, all of them tracing “boundary" type analysis classes, they can build a component. But, if they trace different types of design classes they may not be grouped to build a component.
The restriction 4 ensures that each component plays a unique role in the architecture.
a. They can act as “interfaces" through which the users interact with the system, or, (View)
b. They can act as components that invoke different data models based on user input, or, (Controller)
c. They can act as the data models themselves. (Model)
This builds MVC architecture.
In the following sections, we propose XML schemas for different models used in software development. These XML schemas conform to OMG’s XMI standard. Java programs that use XML parser are used to verify the restrictions proposed.

4.1 Use Case Models to Analysis Models

UML use case diagrams have become the de-facto standard for defining and capturing functional requirements. In Unified software development process use case models consist of UML use case diagrams. A use case diagram is composed of use cases, their actors, their relationships and their flow of events (also known as Activity Flows). In the analysis model, a use case is realized in collaboration and collaborations are mapped to

the analysis classes. There can be three types of analysis classes: Boundary, Control and Entity. Boundary classes in general are used to model interaction between the system and its actors. Entity classes in general are used to model information that is long-lived and often persistent. Control classes are generally used to represent coordination, sequencing, transactions, and control of other objects. And it is often used to encapsulate control related to a specific use cases. Here, in order to trace functional requirements into component design, we propose a restriction: A use case has to be traced in one or many analysis classes. This translation is diagrammatically represented in Figure 2.

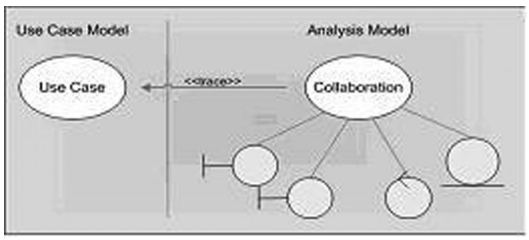

Figure 2: From Use Case to Analysis Model

For this tracing, we propose XML schemas representing use case model and analysis model as shown in Schema 1 and 2 respectively.

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="URI" xmlns:
xmi="http://www.omg.org/XMI"
xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:p="URI">
<xsd:import namespace=http://www.omg.org/XMI schema Location="xmi20.xsd"/>
<xsd:complexType name="UseCaseModel">
<xsd:sequence>
<xsd:complexType name="UseCase">
<xsd:element name="ucId" type="xsd:integer"/>
<xsd:element name="ucName" type="xsd:string"/>
<xsd:complexType name="Events">
<xsd:complexType name="Event">
<xsd:attribute name="eventId" type="xsd:integer"
use="required"/>
<xsd:element name="eventDesc" type="xsd:string"/>
<xsd:complexType name="tracedByAnalysisClasses" >
<xsd:element name="analysisClass" type="xsd:string" />
</xsd:complexType>
</xsd:complexType>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>

Schema 1: XML Schema representing Use Case Models

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="URI" xmlns:
xmi="http://www.omg.org/XMI"
xmlns:xsd=http://www.w3.org/2001/XMLSchema xmlns:p="URI">
<xsd:import namespace=http://www.omg.org/XMI schemaLocation="xmi20.xsd"/>
<xsd:complexType name="AnalysisModel">
<xsd:sequence>
<xsd:complexType name="AnalysisClass">
<xsd:attribute name="type" type="xsd:string"/>
<xsd:element name="name" type="xsd:string"/>
<xsd:complexType name="EventsDealt">
<xsd:sequence>
<xsd:complexType name="Event">
<xsd:attribute name="id" type="xsd:integer" use="required"/>
</xsd:sequence>
</xsd:complexType>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>

Schema 2: XML Schema representing Analysis Models

These XML schemas follow the XMI standards.
A Java program that uses an XML parser is used to verify if all the events in activity flows of use case model has got mapped to the analysis classes in the Analysis model. The same program is also used to ensure that the type of analysis classes can be either one of boundary, control and entity.

4.2 Analysis Models to Design Models

The analysis classes derived in the analysis models get traced to the design classes of the Design Model. This is shown in Figure 3.

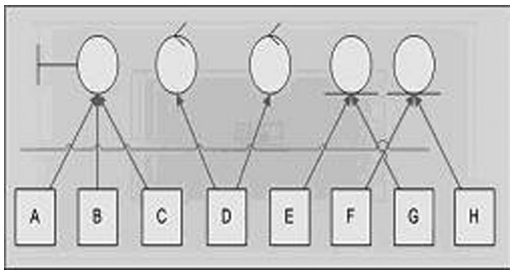

Figure 3: From Analysis to Design Model

Here we propose a XML schema representing a de-

sign model in schema 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="URI" xmlns:xmi=
"http://www.omg.org/XMI"
xmlns:xsd=http://www.w3.org/2001/
XMLSchema xmlns:p="URI">
<xsd:import namespace=http://www.omg.org/XMI schemaLo-
cation="xmi20.xsd"/>
<xsd:complexType name="DesignModel">
<xsd:sequence>
<xsd:complexType name="DesignClass">
<xsd:element name="name" type="xsd:string"/>
<xsd:complexType name="TracedAnalysisClasses">
<xsd:element name="analysisClass" type="xsd:string"/>
</xsd:complexType>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType></xsd:schema>
```

Schema 3: XML Schema representing a Design Model

Similarly a Java program that uses an XML parser is used to verify that each analysis class in Analysis model gets traced to one or more design classes. The same Java program is also used to ensure that if a design class in design model traces more than one analysis classes, then the type of the analysis classes has to be the same, i.e., either one of boundary, control and entity.

4.3 Design Models to Component Models

The Design classes that trace the activity classes are packaged to build components. This is shown in Figure 4. Here, we propose a XML schema that represents

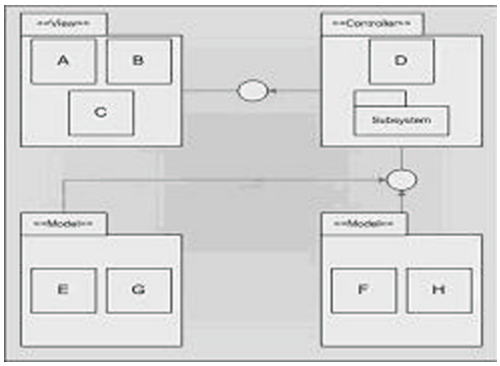

Figure 4: The Component Model from the Design Classes

the component model as shown in schema 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="URI" xmlns:
xmi="http://www.omg.org/XMI"
xmlns:xsd=http://www.w3.org/2001/
XMLSchema xmlns:p="URI">
<xsd:import namespace=http://www.omg.org/
XMI schemaLocation="xmi20.xsd"/>
<xsd:complexType name="ComponentModel">
<xsd:sequence>
<xsd:complexType name="Component">
<xsd:attribute name="id" type="xsd:integer" use="required"/>
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="type" type="xsd:string"/>
<xsd:complexType name="DesignClasses">
<xsd:complexType name="class"> <xsd:attribute name="name"
type="xsd:string" use="required">
</xsd:complexType>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Schema 4 : XML Schema representing a Component Model

A Java program that uses a XML parser is used to verify that the design classes that build a component trace the same type of analysis class in the analysis model; as stated in restriction 4. In the following section we explain our approach with the help of a case study.

5 CASE STUDY

We have considered a simple example of a Library System where a member can register, cancel membership, issue and return books from the library. The use case diagram is shown as in Figure 5.

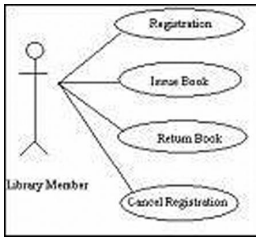

Figure 5: The Component Model from the Design Classes

The Flow of events for "Registration" is:

1. Person details are entered.
2. Checking is made whether an existing member or not.
3. If not an existing member, membership is created and a member ID is generated.

The Flow of events for "Issue Book" is:

1. Member ID is entered and validated
2. Every Member has a maximum allowable limit for is-

suing books, which depends on member category. Check whether member is allowed for issuing books
3. If issue is allowed accept Book ID and validate it.
4. Check if the book is already issued to the member and needs re-issue.
a. If re-issue request then check if there is any demand pending
b. If yes, re-issue request rejected.
c. If no, the book is re-issued.
5. If request is for issue
a. Check for availability of book
b. If available, issue the book
6. Otherwise, place demand on hold.
The Flow of events for “Return Book" is:
1. Member ID is entered and validated.
2. Book ID is entered and validated.
3. Checking is made whether that book was issued to that member or not.
4. Book data is updated.
5. Member data is updated.
The flow of events for “Cancel Membership" is
1. Member ID is entered and validated
2. Check if any book is already issued to the member or not.
3. If no book is issued to the member, the membership is cancelled.
The part of the XML file representing this use case model is shown in schema 5.

```
<?xml version=“1.0"?>
<UseCaseModel>
<UseCase>
<ucId> 01 <ucId>
<ucName> Register </ucName>
<Events>
<Event eventId = ‘1.1’>
<eventDesc>
Person details are entered.
</eventDesc>
<tracedByAnalysisClasses>
<analysisClass>
Authorization Interface
</analysisClass>
</tracedByAnalysisClasses>
</Event>
<Event EventId = ‘1.2’>
<eventDesc>
Checking is made whether an existing member or not.
</eventDesc>
<tracedByAnalysisClasses>
<analysisClass>
Authentication
</analysisClass>
<analysisClass>Member
</analysisClass>
</tracedByAnalysisClasses>
</Event>
</Events>
</UseCase>
<UseCase>
<ucId> 01 <ucId>
<ucName> Issue Book </ucName>
<Events>
<Event EventId = ‘2.1’>
<eventDesc></eventDesc>
<tracedByAnalysisClass>
</tracedByAnalysisClass>
</Event>
<Event EventId = ‘2.2’>
<eventDesc></eventDesc>
<tracedByAnalysisClass>
</tracedByAnalysisClass>
</Event>
.....
</Events>
</UseCase>
.....
</UseCaseModel>
```

Schema 5: XML Document representing Use Case Model of Library System

The use cases of the Use Case Model are traces to the Analysis classes that build the analysis model as shown in Figure 6.
The part of the XML file that represents the analysis model is shown in schema 6.

```
<?xml version=“1.0"?>
<AnalysisModel>
<AnalysisClass type=‘boundary’>
<name> Authorization Interface </name>
<EventsDealt>
<Event id=‘1.1’ />
</EventsDealt>
</AnalysisClass>
<AnalysisClass type=‘control’>
<name> Authentication </name>
<EventsDealt>
<Event id=‘1.1’ />
<Event id=‘4.3 />
</EventsDealt>
</AnalysisClass>
.....
</AnalysisModel>
```

Schema 6: XML Document representing Analysis Model of a Library System

The Analysis classes are traced to the design classes. The process is depicted in Figure 7.

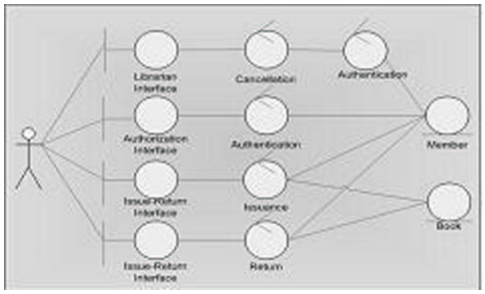

Figure 6: Analysis Model of LMS

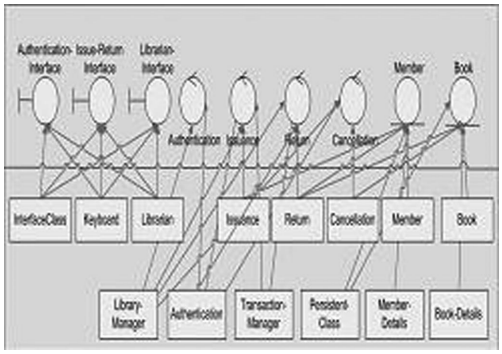

Figure 7: Analysis Classes traced to Design Classes

From this, we can derive the design model of the Library System as shown in Figure 8.
The part of the design model is represented in XML as shown in schema 7.

<?xml version="1.0"?>
<DesignModel>
<DesignClass>
<name>Display</name>
<TracedAnalysisClasses>
<analysisClass> Authorization-Interface
</analysisClass>
<analysisClass> Issue-Return-Interface
</analysisClass>
</ TracedAnalysisClasses >
</DesignClass>
<DesignClass>
<name>Transaction-Management</name>
<TracedAnalysisClasses>
<analysisClass>Issuance</analysisClass>
<analysisClass>Return</analysisClass>
</ TracedAnalysisClasses>
</DesignClass>
</DesignModel>

Schema 7: XML Document representing Design Model of a Library System

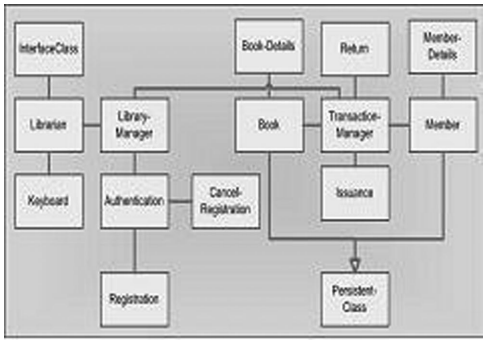

Figure 8: The Design Model of LMS

We now propose to package classes into components to generate the Component Model of the Library System as shown in Figure 9.
It is a Component-Based Model-View-Controller Architecture. Part of a Component Model of the Library System is represented in XML format in schema 8.

<?xml version="1.0"?>
<ComponentModel>
<Component id='01'>
<name>Library-Interface </name>
<type>View</type>
<DesignClasses>
<class name="Display"/>
<class name="Keyboard"/>
<class name="Librarian"/>
</DesignClasses>
</Component>
<Component id='04'>
<name>Transaction-Management</name>
<type>6 Controller </type>
<DesignClasses>
<class name='Transaction-Manager"/>
</DesignClasses>
</Component>
</ComponentModel>

Schema 8: XML Document representing Component Model of a Library System

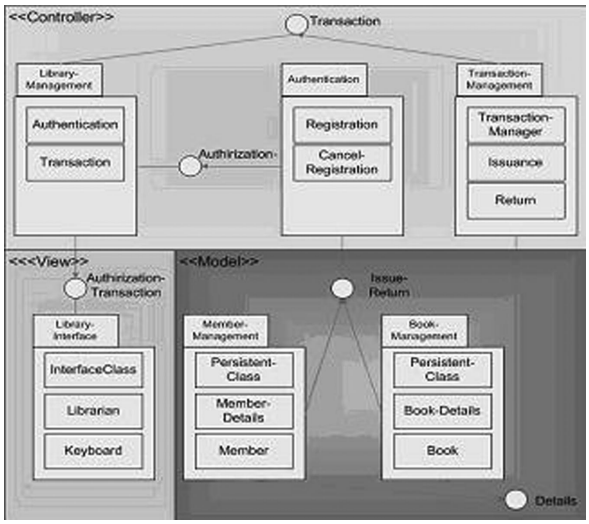

Figure 9: The Component Model of LMS

6 CONCLUSION

In this paper, we propose a framework for designing a system based on Component Based Architecture. Here, we propose to achieve separation of concerns among these components using Model-View-Controller design pattern. This would enable us building components that are highly cohesive and less coupled with other components making them easily replaceable and modifiable. Reusability of such components will be high as the roles of the components are going to be well defined and non-overlapping.

References

[1] Air force/stars demonstration project home page. *Available at http://www.stsc.hill.af.mil/crosstalk/1994/08/xt94d08e.asp*, 1995.

[2] Mechanisms/incentives for reuse and cots use. *Available at www.sei.cmu.edu/str/descriptions/cbsd.html*, 1996.

[3] "plas". *Available at http://plas.fit.qut.edu.au*, 1996.

[4] Portable, reusable, integrated software modules (prism) program. *Available at http://stinet.dtic.mil/oai/oai?& verb=getRecord& metadataPrefix=html& identifier=ADA284567*, 1996.

[5] Object management group. *UML Revision Task Force: OMG UML v. 1.3: Revisions and Recommendations*, 1999.

[6] Model driven architecture, a technical perspective. *doc. n. ab/2001-01-01, n. ormsc*, July 26-29 2001.

[7] Bachman Felix, B. L., Charles, B., Santiago, C.-D., and Fred, L. Technical concepts of component-based software engineering. *Technical Report CMU/SEI-2000-TR-008, Available at www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf*, May 2000.

[8] Brooks, F. P. J. No silver bullet: Essence and accidents of software engineering. *Computer Vol. 20, Issue 4*, April 1987.

[9] Brown, A. W. Preface: Foundations for component-based software engineering. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, Los Alamitos, CA, IEEE Computer Society Press*, pages pp. 8–10, 1996.

[10] Brown, A. W. and Wallnan, K. C. Engineering of component-based systems. *2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS' 96), Montreal, CANADA*, pages pp. 414–419, October 21-25 1996.

[11] Cheesman, J. and Daniels, J. Uml components, a simple process for specifying component-based software. *Addison Wesley*, 2001.

[12] Clements, P. C. From subroutines to subsystems: Component-based software development. *Component-Based Software Engineering: Putting Pieces Together, Addison-Wesley Longman Publishing Co., Inc*, pages pp 189–198, 1996.

[13] Col., R. D. M. L. Lessons learned in the development and integration of a cots-based satellite tt & c system. *33rd Space Congress. Cocoa Beach, FL*, April 23-26 1996.

[14] C.R.G. de Farias, L. F. P. and van Sinderen, M. A component-based groupware development methodology. *4th Int. Enterprise Distributed Object,Computing Conference (EDOC'00) Makuhari, Japan*, pages pp. 204–213, September 25-28 2000.

[15] D'Souza, D. F. and Wills, A. C. Objects, components and frameworks with uml: the catalysis approach. *Addison Wesley, USA*, 1999.

[16] et al., G. A. Saam: A method for analyzing the properties of software architecture. *16th International Conference on Software Engineering (ICSE'94), Sorrento, Italy*, pages pp 81–90, May 16-21 1994.

[17] I. Jacobson, G. B. and Rumbaugh, J. The unified software development process. *Addison Wesley, USA*, 1999.

[18] Ivar Jacobson, G. B. and Rumbaugh, J. The unified software development process. *Addison Wesley, USA*, 1999.

[19] K. Wallnau, S. H., J. Stafford and Klein, M. On the relationship of software architecture to software component technology. *6th International Workshop on Component-Oriented Programming (WCOP'06), Budapest, Hungary*, June 19 2001.

[20] Kontio, J. A case study in applying a systematic method for cots selection. *18th International Conference on Software Engineering (ICSE'96), pages 201-209, Berlin, Germany*, pages pp 201–209, March 25-30 1996.

[21] Lettes, J. A. and Wilson, J. Army stars demonstration project experience report (stars-vca011/003/02)". *Manassas, VA: Loral Defense Systems-East*, 1996.

[22] Meyer, B. Applying design by contract. *Computer, 25 (10)*, pages pp 40–52, October 1992.

[23] Sabnam Sengupta, A. K. and Bhattacharya, S. Requirement traceability in software development process: An empirical approach. *19th IEEE/IFIP Symposium for Rapid Systems Prototyping RSP 2008, Monterey, CA, USA*, pages pp 105–111, June 2-5 2008.

[24] Sabnam Sengupta, A. S. and Bhattacharya, S. Requirements to components: A model-view-controller architecture. *14th Monterey Workshop, Monterey, CA, USA, 2007*, pages pp 167–184, September 10-13 2007.

[25] Seaman, C., Morisio, M., Basili, V., Parra, A., Kraft, S., and Condon, S. Cots-based software development: processes and open issues. *Journal of Systems and Software, Vol. 61, Issue 3*, page pp 189, April 2002.

[26] Sengupta, S. and Bhattacharya, S. Formalization of functional requirements of software development process. *Journal of Foundations of Computing and Decision Sciences (FCDS), Institute of Computing Science, Poznan University of Technology, Poland Vol 33, issue 1*, pages pp 83–115, 2008.

[27] Szyperski, C. Component software: Beyond object-oriented programming. *Second Edition Addison-Wesley / ACM Press*, 2002.

[28] Valetto, G. and Kaiser, G. Enveloping sophisticated tools into computer-aided software engineering environments. *7th IEEE International Workshop on CASE Los Alamitos, CA: IEEE Computer Society Press, Toronto, Ontario, Canada*, pages pp 40–48, July 10-14 1995.

[29] van Sinderen, D. Q. M. and Pires, L. F. A model-based approach to service creations. *7th International Workshop of Future Trends in Distributed Computing FTDCS'99, Cape Town, South Africa*, pages pp. 102–110, December 20-22 1999.