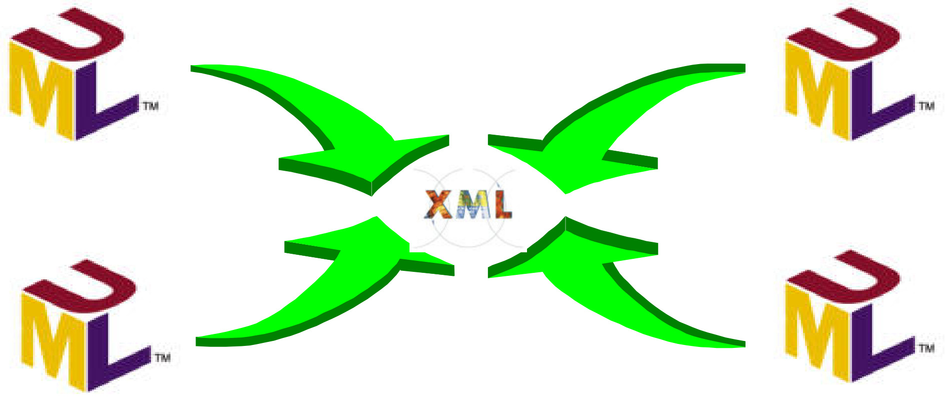
MAPPING UML DIAGRAMS TO XML



Dissertation
Submitted in the partial fulfillment of the requirement for the award of Degree of

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE & TECHNOLOGY

BY
JITENDER SINGH

UNDER THE SUPERVISION OF
PROF. PARIMALA N.

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067

dedicated to…

my younger brother Dinesh

CERTIFICATE

This is to certify that this thesis, entitled "**Mapping UML Diagrams to XML**", being submitted by Mr. Jitender Singh to **School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi** in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science & Technology**, is a record of original work done by him under the supervision of Prof. Parimala N., during the Monsoon Semester, 2003.

The results reported in this thesis have not been submitted in part or full to any other University or Institution for the award of any degree.

Jitender Singh

Digitally signed by Jitender Singh
DN: CN = Jitender Singh, C = IN, O = JNU, OU = SCSS
Reason: I am the author of this document
Date: 2004.03.17 10:55:54 +05'30'

JITENDER SINGH
(Student)

Prof. Parimala N.
(Supervisor)

Prof. Karmeshu
(Dean, SC&SS, JNU, New Delhi-110067)

ACKNOWLEDGEMENTS

"*Perhaps the most valuable result of all education is the ability to make yourself do the thing you have to do, when it ought to be done, whether you like it or not; it is the first lesson that ought to be learned; and however early a man's training begins, it is probably the last lesson that he learns thoroughly.*"

With the writing of this thesis, I would like to extend my thanks to all of the people who have helped me to be able to write it.

My first thanks go to my supervisor **Prof. Parimala N.** who helped and guided me throughout the duration of this work. Apart from her guidance in this duration, her methods of the distribution of the knowledge of the subject in the previous semester have also benefited me a lot. It has been a good test of her patience to guide me out of my rough ideas and rubbish thoughts. I am very grateful to her for providing me with important resources for the research and analysis.

I would also like to extend a special thanks to the Dean, **Prof. Karmeshu**, and other faculty members of the SC & SS, JNU, especially **Prof. PC Saxena**, **Prof. GV Singh**, **Prof. KK Bhardwaj**, **Prof. S. Balasundaram**, **Dr. DK Lobial**, **Dr. Sona Minz**, and **Dr. RC Phoha** for their able guidance, which made me equipped to carry on this work.

I would also like thank my classmates and friends for their kind help on the discussion related to my work and providing useful suggestions and resources. My special mention in this regard goes to **Hemant Gaur**, **Vikas Dahiya, Manish Bansal, Mohit Kumar, Sudesh Singh Chandel** and **Rakesh Roshan**.

I would also like to make a mention of the non-teaching staff of the school who has worked hard to provide us with a very smooth and comfortable environment for the research work.

In the last, but not the least, I would like to thank all my family members for their support over the years, which has resulted in my this contribution.

Jitender Singh
New Delhi, March 16, 2004

ABSTRACT

Unified Modeling Language (UML) is a standard object oriented design modeling language for business and technical systems and has been standardized by Object Management Group (OMG) for system specifications and design. Extensible Markup Language Schema (XML Schema) provide a means for defining the structure, content and semantics of XML documents, which has been approved as a W3C recommendation on May 2, 2001 by World Wide Web Consortium (W3C). Since UML is a standard design modeling language and XML is widely being accepted as information representation and sharing language across Internet, efforts have been initiated to map UML diagrams to XML documents. Currently most of these efforts are directed towards mapping only the static aspects of the UML diagrams. In order to build a complete system, we have to map the static as well as the dynamic aspects. In this thesis we have mapped the following diagrams of UML: Class diagram, Sequence diagram, and Collaboration diagram. We have defined the XML Schema Definition for the class diagrams, sequence diagrams, and collaboration diagrams. We have also proposed a mapping scheme for converting UML class diagrams, sequence diagrams, and collaboration diagrams to XML documents. The rules for validating the diagrams are devised and implemented. The class diagrams are validated within its own structure, while the sequence and collaboration diagrams are validated against the class diagrams used by them, apart from validating within their own structure. Finally, we have implemented a system to automate all the above functionality using Java.

CONTENTS

CHAPTER 1

INTRODUCTION

During the life cycle of a software project, voluminous data and information are usually created along the delivery processes of software products. There is always a need to share and exchange these engineering data and information among related parties involved in the project. However, because data models defined in the information management systems of various parties are usually different, it is always difficult to directly map from one data model to another for the purpose of information sharing and exchange. To address the aforementioned information-sharing problem among various parties, standardization of the data model for a software project is usually inevitable. Due to the popularity of object-oriented modeling approach in recent years, an object-oriented information model is often constructed to represent the static structure and dynamic behaviors of the information and processes in a software project and expressed by UML (Unified Modeling Language), a popular tool for object-oriented modeling. Moreover, Extensible Markup Language (XML) has become a de-facto standard for information sharing and exchange in recent years.

Therefore, there is a need to define an XML schema based upon the UML object-oriented information model to further facilitate information sharing. However, the mapping is not an easy one because the data model of an XML document is fundamentally different from that of a UML model. Especially the UML models comprise of a lot of diagrams and related data, and the structure of an XML document is hierarchical and the XML elements may be nested and repeated. Although some commercial tools provide a little bit of such mapping, but that is limited to the static aspects of a UML model and only involves class diagrams. They do not map the dynamics of the UML model like interaction diagrams, and state chart diagrams. Recently some commercial softwares have also provided features for transforming information in UML diagrams into XML documents. However, most of them can only transform simple static UML models only. That is, if the UML diagrams involve dynamics most of these tools are still incapable of making a complete transformation.

Taking motivation from this we have worked on a mapping scheme for three UML diagrams: class diagram, sequence diagram, and collaboration diagram. This mapping scheme will map these UML diagrams to an XML document.

1.1 UNIFIED MODELING LANGUAGE

The UML (Unified Modeling Language) is the successor to the wave of object-oriented analysis and design (OOA&D) methods that appeared in late '80s and '90s. It unifies the methods of Grady Booch, Rumbaugh (OMT), and Jacobson, but its reach is wider than that. It is now a modeling language and not mere method.

The Unified Modeling Language has quickly become the de-facto standard for building Object-Oriented software. It is the fusion of the modeling technologies of Grady Booch, James Rumbaugh and Ivar Jacobson. It is an Object Management Group (OMG) standard for modeling object-oriented systems.

UML is a standard language and graphical notation for creating models of business and technical systems. The OMG specification states:
"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

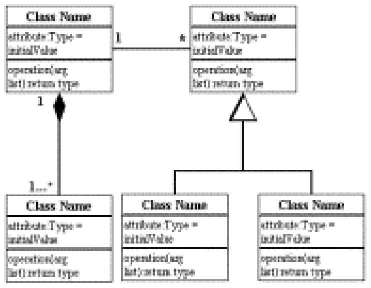
The important point to note here is that UML is a 'language' for specifying and not a method or procedure. The UML is used to define a software system; to detail the artifacts in the system, to document and construct - it is the language that the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process) - but in itself it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:
1. The User Interaction or *Use Case Model* - describes the boundary and interaction between the system and users. Corresponds in some respects to a requirement model.
2. The Logical or *Class Model* - describes the classes and objects that will make up the system.
3. The Interaction or *Collaboration Model* - describes how objects in the system will interact with each other to get work done.
4. The State or *Dynamic Model* - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflow the system will implement.
5. The *Physical Component Model* - describes the software (and sometimes hardware components) that make up the system.
6. The *Physical Deployment Model* - describes the physical architecture and the deployment of components on that hardware architecture.

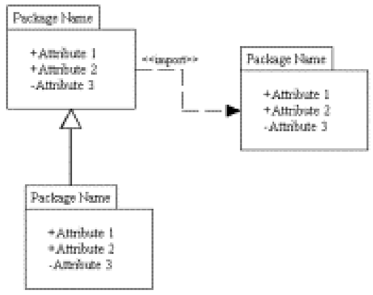
The UML also defines extension mechanisms for extending the UML to meet specialized needs (for example Business Process Modeling extensions).

1.1.1 TYPES OF UML DIAGRAMS

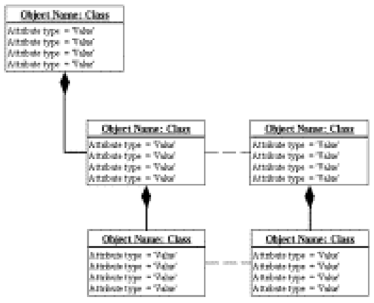
UML defines nine types of diagrams: class (package), object, use case, sequence, collaboration, statechart, activity, component, and deployment.



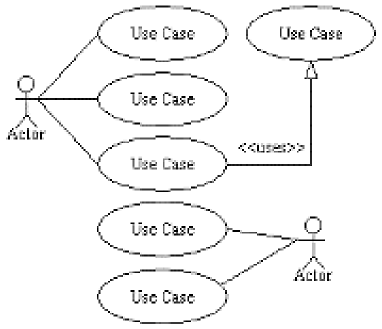
Class Diagrams
Class diagrams are the backbone of almost every object-oriented method, including UML. They describe the static structure of a system.



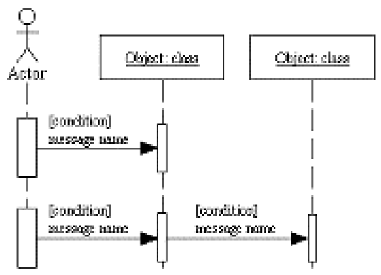
Package Diagrams
Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies between packages.



Object Diagrams
Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.

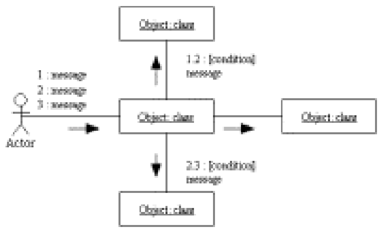


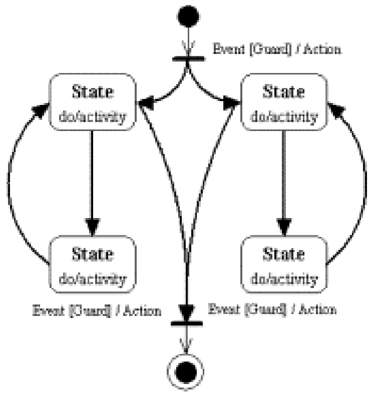
Use Case Diagrams
Use case diagrams model the functionality of system using actors and use cases.



Sequence Diagrams
Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.

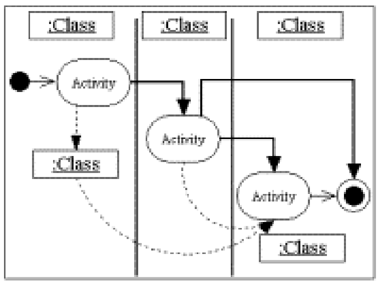
Collaboration Diagrams
Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.

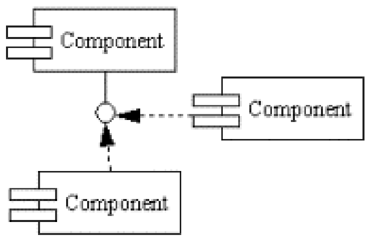




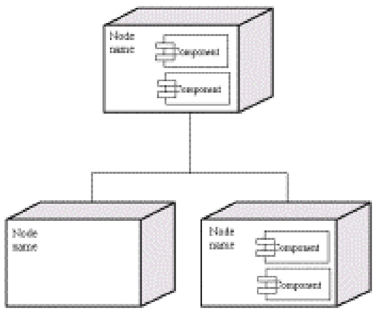
Statechart Diagrams
Statechart diagrams describe the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.

Activity Diagrams
Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.





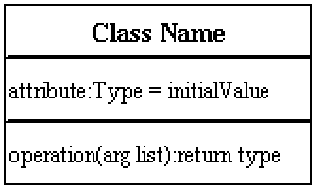
Component Diagrams
Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.



Deployment Diagrams
Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

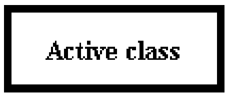
1.1.2 UML Class Diagrams

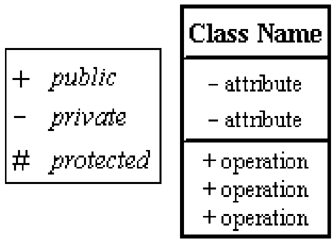
Class diagrams are the backbone of almost every object-oriented method including UML. They describe the static structure of a system. Classes represent an abstraction of entities with common characteristics. Associations represent the relationships between classes.


Class Name
attribute:Type = initialValue
operation(arg list):return type

Classes
Illustrate classes with rectangles divided into compartments. Place the name of the class in the first partition (centered, bolded, and capitalized), list the attributes in the second partition, and write operations into the third.

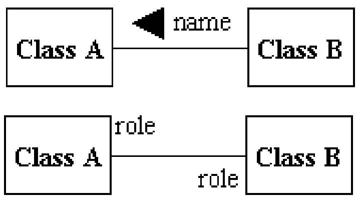
Active Class
Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border.

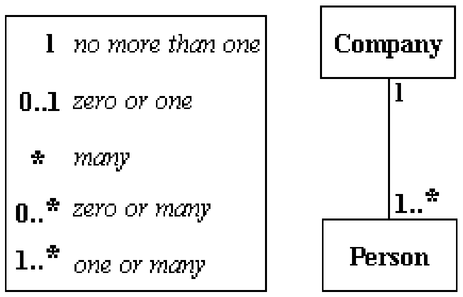




Visibility
Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.

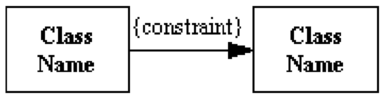
Associations
Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other. It's uncommon to name both the association and the class roles.

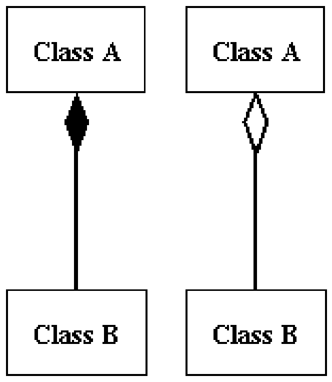




Multiplicity (Cardinality)
Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to *one* instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.

Constraint
Place constraints inside curly braces {}.

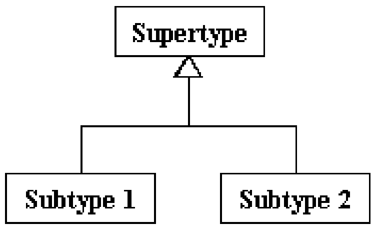




Composition and Aggregation
Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond.

Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate.

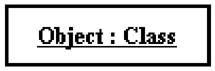
Generalization
Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.



In real life coding examples, the difference between inheritance and aggregation can be confusing. If you have an aggregation relationship, the aggregate (the whole) can access only the PUBLIC functions of the part class. On the other hand, inheritance allows the inheriting class to access both the PUBLIC and PROTECTED functions of the superclass.

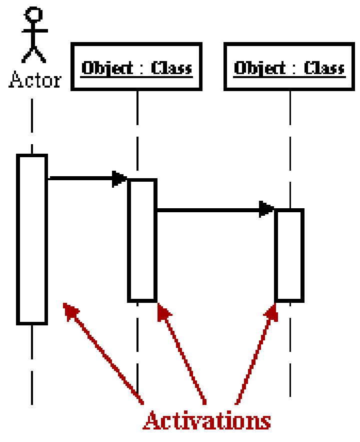
1.1.3 UML Sequence Diagrams

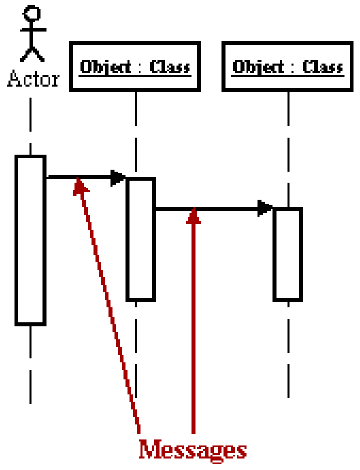
Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.



Class roles
Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.

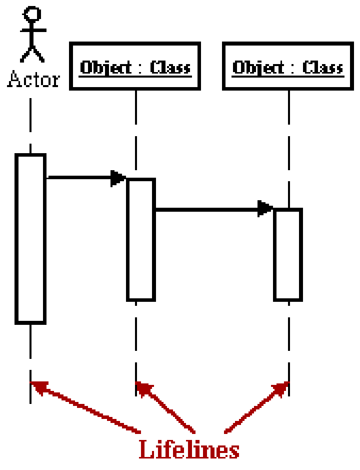
Activation
Activation boxes represent the time an object needs to complete a task.





Messages
Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

Lifelines
Lifelines are vertical dashed lines that indicate the object's presence over time.



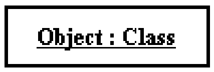
Destroying Objects
Objects can be terminated early using an arrow labeled "< < destroy > >" that points to an X.

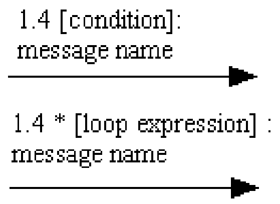
Loops
A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].

1.1.4 UML COLLABORATION DIAGRAMS

A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.

<ins>*Class roles*</ins>
Class roles describe how objects behave. Use the UML object symbol to illustrate class roles, but don't list object attributes.





<ins>*Messages*</ins>
Unlike sequence diagrams, collaboration diagrams do not have an explicit way to denote time and instead number messages in order of execution. Sequence numbering can become nested using the Dewey decimal system. For example, nested messages under the first message are labeled 1.1, 1.2, 1.3, and so on. The a condition for a message is usually placed in square brackets immediately following the sequence number. Use a * after the sequence number to indicate a loop.

1.2 XML and XML Schema

1.2.1 eXtensible Markup Language

XML stands for eXtensible Markup Language. XML is a markup language much like HTML. XML was designed to describe data. XML tags are not predefined. You must define your own tags. XML uses a Document Type Definition (DTD) or an XML Schema to describe the data. XML with a DTD or XML Schema is designed to be self-descriptive.

XML was designed to carry data. XML is not a replacement for HTML. XML and HTML were designed with different goals:

- XML was designed to describe data and to focus on what data is.
- HTML was designed to display data and to focus on how data looks.
- HTML is about displaying information, while XML is about describing information.

XML was not designed to DO anything. Maybe it is a little hard to understand, but XML does not DO anything. XML is created to structure, store and to send information. E.g. the following example is a note to Tove from Jani, stored as XML:

```
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The note has a header and a message body. It also has sender and receiver information. But still, this XML document does not DO anything. It is just pure information wrapped in XML tags. Someone must write a piece of software to send, receive or display it.

1.2.2 XML SCHEMA

XML Schema is an XML based schema description language. An XML schema describes the structure, contents and semantics of an XML document. It is an W3C recommendation. It can be referenced through any standard XML parser.

The XML Schema language is also referred to as XML Schema Definition (XSD). The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

An XML Schema defines elements and attributes that can appear in a document, which elements are child elements, their order and number, whether an element is empty or can include text, data types for elements and attributes, and default and fixed values for elements and attributes

Very soon XML Schemas will be used in most Web applications as a replacement for DTDs because, XML Schemas are extensible to future additions, richer and more useful than DTDs, written in XML, support data types, entities, constraints, namespaces, and OOP features like inheritance, encapsulation etc

XML Schema was originally proposed by Microsoft, but became an official W3C recommendation in May 2001. The specification is now stable and has been reviewed by the W3C Membership.

One of the greatest strength of XML Schemas is the support for data types. With the support for data types it is easier to describe permissible document content, validate the correctness of data, work with data from a database, define data facets (restrictions on data), define data patterns (data formats), and convert data between different data types

Another great strength about XML Schemas is that they are written in XML. Since XML Schemas are written in XML, you don't have to learn another language, use your XML editor to edit your Schema files, use your XML parser to parse your Schema files, manipulate your Schema with the XML DOM, and transform your Schema with XSLT

When data is sent from a sender to a receiver it is essential that both parts have the same "expectations" about the content. With XML Schemas, the sender can describe the data in a way that the receiver will understand. E.g. a date like 1999-03-11 might (in some countries) be interpreted as 3. November or (in some other countries) as 11. March, but an XML element with a data type like this:

```
<date type="date">1999-03-11</date>
```

ensures a mutual understanding of the content because the XML data type date requires the format CCYY-MM-DD.

XML Schemas are extensible, just like XML, because they are written in XML. With an extensible Schema definition you can reuse your Schema in other Schemas, create your own data types derived from standard types, and reference multiple schemas from the same document

A well-formed XML document is a document that conforms to the XML syntax rules. It must begin with the XML declaration, must have one unique root element, all start tags must match end-tags with case sensitive XML tags, all elements must be closed and properly nested, the attribute values must be quoted, and XML entities must be used for special characters. Even if documents are Well-Formed they can still contain errors, and those errors can have serious consequences. Think of this situation: you order 5 gross of laser printers, instead of 5 laser printers. With XML Schemas, your validating software can catch most of these errors.

1.3 Need for Mapping

As the application of the computer-based solutions has increased, these solutions started to grow more and more complex in nature. This prompted the increase in size of the development staff as well as the development site. The development sites tend to be scattered across geographical locations across the globe. Thus the development data was also required to be moved across these locations. Since the parties involved in the development process share large volumes of data, mostly the models of the system being developed, so it was required that this data be transferred across the sites over the networks. As UML is a standard design modeling language and XML is widely being accepted as information representation, sharing, and exchange language over Internet, we need to map the UML diagrams to XML documents.

Also, the development process nowadays has become more and more model-driven. Developers all around the world are striving hard to cut the design costs, prompting them to share design models for similar problems. So they are also pressing hard for such efforts which help them share these models over Internet.

Thus, we find that there has been enough encouragement to work for such an mapping scheme, which will be helping us convert the UML diagrams to XML documents.

1.4 Previous Work

Since both the technologies UML and XML are just beginning to grow, not much work has been done in the field of mapping between these two. There have been efforts from David Carlson, Rational Software Corporation, I-Chen Wu and Shang-Hsien Hsieh in this area. All of them have followed the approach of mapping between XML Schema Documents and UML.

Before working on our mapping scheme we went through the approaches used by David Carlson, Grady Booch et al at Rational, Wu and Hsieh to map class diagrams to XML documents using XML Schema. All of them have converted the UML class diagram to XML Schema or vice versa.

Booch et al worked to map the XSD to UML class diagrams, in order to model the XML Schemas using UML. Their intention was not to map the UML diagrams to XML. David Carlson also worked to map the XML vocabularies with UML with slightly different approach by using stereotypes in UML, unlike Booch et al, who approached by trying to use more conventional features of UML. Both these works helped a lot in the mapping scheme, which we developed.

Wu and Hsieh worked on the same lines as ours i.e. mapping UML diagrams to XML. They created XSD from the class diagrams, just opposite to what Booch and Carlson did. But it is a bit complex approach to generate the XSD for each class diagram. Again this way we can't map the sequence and collaboration diagrams.

1.4.1 Wu and Hsieh Approach

In this approach, XML Schema is constructed from UML model through the concept of the XML Metadata Interchange specification (XMI), which defines a rigorous approach for generating an XML DTD from a metamodel definition, and slightly extends the approach of XMI for mapping object-oriented data model expressed by UML to XML Schema. The transformation rules employed in the mapping process are discussed as follows:

Mapping UML Classes to XML Elements

The UML Classes show the structural and behavioral features in the object-oriented Model. These features include attributes, association, aggregation, and composition. On the other hand, XML elements serve as a container for attribute and child elements. Thus, mapping UML classes to XML elements are quite straightforward.

Mapping UML Attributes to XML Attributes or Elements

Basically, either a primitive data type or an enumeration of UML attributes may be represented as an XML attribute. However, XML parser removes all extra whitespace characters, such as tabs, linefeeds, etc. That makes XML attributes mainly appropriate for simple datatypes of short string values. On the other hand, one can map attributes of an UML class to separate child elements of the corresponding XML element of the class.

Flagging UML Object Relationships by an XML Attribute

The current version (Version 1.0) of XML Schema does not yet have direct and full supports for expressing a complete object-oriented model, especially the distinction between the delegation and aggregation relationships that commonly exist in the model. Therefore, this work employs a special attribute named “relation” with a value of either “delegation” or “aggregation” to flag the relationship between the owning XML element and its child elements.

Constraints on Naming XML Elements

In general, the UML class name is directly used as the XML tag name in the mapping process. However, there are certain constraints we must comply when naming the XML elements:

1. The tag name cannot have spaces in it, but symbols like “.”, “-”, and “_” are allowed.
2. The tag name should not start with the string “XML”.

1.5 OUR APPROACH

In this work we have proposed an approach to map the most commonly used UML diagrams to XML documents. These include class diagrams, sequence diagrams, and collaboration diagrams. The rest of the models can be mapped with the same approach.

This approach is slightly different than the approach proposed by Wu and Hsieh. We have defined an XML Schema Definition for the class diagram, the sequence diagram, and the collaboration diagram, which will be defining the structure of the documents representing the class, sequence and collaboration diagrams. The XML document mapped from the UML diagrams will reference these XSDs to get the document structure. These XSDs will be discussed in the Chapter 2. Thus a single schema definition for class diagrams will define the structure for all the class diagrams. Similarly a single schema definition for sequence diagrams will be enough for all the sequence diagrams, and there will be a single schema definition for all the collaboration diagrams. This is done because basic elements of a class diagram are similar. It is true for the sequence diagram and the collaboration diagrams as well.

We have also developed a mapping scheme for mapping the UML diagrams to XML documents with these structures. This scheme will also be discussed in the Chapter 2. Since the structure of all the class diagrams are similar so with this slight variation in approach of Wu and Hseih, we can easily perform the mappings in a more general and efficient way.

We have also discussed the validations required for these mapping schemes. These will validate the generated class diagrams, sequence diagrams, and collaboration diagrams. Since, this restrictions can't be put later on in the document, we will force these prior to the mapping on the structures to be mapped to XML. The validations for the UML diagrams are discussed in Chapter 2.

The Chapter 3 discusses the design of the system worked out to automate this mapping scheme. This will present a Use case, and class diagram model of the system apart from providing the system architecture, input design and output design.

The Chapter 4 discusses the implementation of the design discussed in Chapter 3, thus realization of the system design. It discusses the methodology, user interface, and the outputs produced by the system.

The Chapter 5 demonstrates the automated system with an example taken from the UML text.

CHAPTER 2

THE MAPPING

The work of mapping the UML diagrams, i.e. class diagram, sequence diagram and collaboration diagram, is divided in several sub-tasks in our approach. This has been one of the factors in favor of the adoption of this approach apart from being logical, efficient, and general approach.

First, we have defined the XML Schema Definition (XSD) for class diagram. A similar XSD for sequence diagram, and collaboration diagram is also defined. These XSDs are stored in the folder where we have to put the XML document of the diagram in order to get a reference to these schemas. First section of this chapter explains these XSDs.

After that we have specified a mapping scheme that will convert a class diagram into an XML document referencing the above mentioned XSD for class diagrams. Similarly, we have specified the mapping schemes for sequence diagrams and collaboration diagrams as well, which will be referencing their respective XSDs. The second section of this chapter explains the mapping scheme.

Finally, we have specified the ways of validating the class diagrams, sequence diagrams and the collaboration diagrams. The class diagram can be validated on its own, but the validation of the sequence and the collaboration diagrams involve the class diagrams they are using. So, we must specify the class diagrams against which they can be validated. The validation rules of these diagrams are listed in the third section of the chapter.

This chapter deals with the capture of the above specifications and ways of integrating them.

2.1 XML Schema Definitions

The first part of the research involved the designing of XML Schema Definition for the class diagram, sequence diagram, and the collaboration diagram. In this section we have presented the XML Schema Definition for these diagram. In order to make these diagrams more understandable we have also written the corresponding definitions in Bachaus-Naur Form.

2.1.1 XML Schema Definition for Class Diagram

The XML Schema Definition used for a class diagram is given below:

```
<?xml version="1.0" ?>
<Schema              name="classDiagram.xsd"              xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
<!-- 'diagram' is the root node, including 'class' elements -->
        <ElementType name="diagram">
                <group minOccurs="1" maxOccurs="*">
                        <element type="class" />
                </group>
        </ElementType>
        <!-- 'class' schema definition -->
        <ElementType name="class" content="eltOnly">
                <attribute type="name" />
                <element type="instance-variable" />
                <element type="operator" />
                <element type="constraint" />
                <element type="association" />
                <element type="composition" />
                <element type="aggregation" />
                <element type="generalization" />
        </ElementType>
        <!-- 'name'attribute definition -->
        <AttributeType name="name" dt:type="string" required="yes" />
        <!-- 'instance-variable' schema definition -->
        <ElementType name="instance-variable" content="textOnly" dt:type="id" />
        <!-- 'operator' schema definition -->
        <ElementType name="operator" content="eltOnly">
                <attribute type="name" />
                <element type="args-type" />
                <element type="ret-type" />
        </ElementType>
        <!-- 'constraint' schema definition -->
        <ElementType name="constraint" content="textOnly" />
        <!-- 'association' schema definition -->
        <ElementType name="association" content="eltOnly">
                <attribute type="role-name" />
                <attribute type="multiplicity" />
                <element type="class-name" />
        </ElementType>
        <!-- 'composition' schema definition -->
        <ElementType name="composition" content="eltOnly">
                <attribute type="multiplicity" />
                <element type="class-name" />
        </ElementType>
        <!-- 'aggregation' schema definition -->
        <ElementType name="aggregation" content="eltOnly">
                <attribute type="multiplicity" />
                <element type="class-name" />
        </ElementType>
        <!-- 'generalization' schema definition -->
        <ElementType name="generalization" content="eltOnly">
                <element type="class-name" />
        </ElementType>
        <!-- 'args-type' schema definition -->
        <ElementType name="args-type" content="textOnly" />
        <!-- 'ret-type' schema definition -->
        <ElementType name="ret-type" content="textOnly" />
        <!-- 'multiplicity' attribute definition -->
        <AttributeType  name="multiplicity"  dt:type="enumeration"  dt:values="optional  one  many"
required="no" default="optional" />
        <!-- 'role-name' attribute definition -->
        <AttributeType name="role-name" dt:type="string" required="no" />
        <!-- 'class-name' Schema Definition -->
        <ElementType name="class-name" content="textOnly" />
</Schema>
```

The equivalent BNF for the class diagram schema is:

1. <role-name> -> string
2. <name> -> string
3. <multiplicity> -> optional | one | many
4. <class-name> -> text
5. <ret-type> -> text
6. <args-type> -> text
7. <constraint> -> text
8. <instance-variable> -> text
9. <generalization> -> <class-name>
10. <aggregation> -> <multiplicity> <class-name>
11. <composition> -> <multiplicity> <class-name>
12. <association> -> <role-name> <multiplicity> <class-name>
13. <operator> -> <name> <args-type> <ret-type>
14. <class> -> <instance-variable>* <operator>* <constraint>* <association>* <composition>* <aggregation>* <generalization>*
15. <diagram> -> <class>$^+$

2.1.2 XML Schema Definition for Sequence Diagram

The XML Schema Definition used for Sequence Diagram is given below:

```
<?xml version="1.0" ?>
<Schema name="sequenceDiagram.xsd" xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
        <!-- 'diagram' is the root node, including 'class-operator' elements -->
        <ElementType name="diagram">
                <group minOccurs="1" maxOccurs="*">
                        <element type="class-name" /></group>
        </ElementType>
        <!-- 'class-name' schema definition -->
        <ElementType name="class-name">
                <attribute type="name" />
                <group minOccurs="0" maxOccurs="*">
                        <element type="class-operator" /></group>
        </ElementType>
        <!-- 'class-operator' schema definition -->
        <ElementType name="class-operator" content="eltOnly">
                <attribute type="multiplicity" />
                <attribute type="condition" />
                <element type="method-name" />
                <element type="args-type" />
                <element type="class-name" />
        </ElementType>
        <!-- 'method-name' schema definition -->
        <ElementType name="method-name" content="textOnly" dt:type="id" />
        <!-- 'args-type' schema definition -->
        <ElementType name="args-type" content="textOnly" />
        <!-- 'multiplicity' attribute definition -->
        <AttributeType name="multiplicity" dt:type="enumeration" dt:values="optional one many"
required="no" default="one" />
        <!-- 'name' attribute definition -->
        <AttributeType name="name" dt:type="String" required="yes" />
        <!-- 'condition' attribute definition -->
        <AttributeType name="condition" dt:type="string" required="no" default="true" />
</Schema>
```

The equivalent BNF for the sequence diagram schema is:

```
1.  <condition> -> string
2.  <name> -> string
3.  <multiplicity> -> optional | one | many
4.  <args-type> -> text
5.  <method-name> -> text
6.  <class-operator> -> <condition> <multiplicity> <method-name> <args-type> <class-name>
7.  <class-name> -> <name> <class-operator>*
8.  <diagram> -> -> <class-name>*
```

2.1.3 XML Schema Definition for Collaboration Diagram

The XML Schema Definition used for Collaboration Diagram is given below:

```
<?xml version="1.0" ?>
<Schema name="collaborationDiagram.xsd" xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
        <!-- 'diagram' is the root node, including 'class-operator' elements -->
        <ElementType name="diagram">
                <group minOccurs="1" maxOccurs="1">
                        <element type="class" />
                </group>
        </ElementType>
        <!-- 'class-operator' schema definition -->
        <ElementType name="class">
                <attribute type="name" />
                <element type="instance-name" />
                <element type="class-operator" />
        </ElementType>
        <!-- 'class-operator' schema definition -->
        <ElementType name="class-operator" content="eltOnly">
                <attribute type="name" />
                <attribute type="sequence-number" />
                <attribute type="multiplicity" />
                <attribute type="condition" />
                <element type="args-type" />
                <element type="class" />
        </ElementType>
        <!-- 'name'attribute definition -->
        <AttributeType name="name" dt:type="string" required="yes" />
        <!-- 'args-type' schema definition -->
        <ElementType name="args-type" content="textOnly" />
        <!-- 'instance-name' schema definition -->
        <ElementType name="instance-name" content="textOnly" />
        <!-- 'multiplicity' attribute definition -->
        <AttributeType name="multiplicity" dt:type="enumeration" dt:values="none one many"
required="no" default="one" />
        <!-- 'condition' attribute definition -->
        <AttributeType name="condition" dt:type="string" required="no" default="true" />
        <!-- 'sequence-number' attribute definition -->
        <AttributeType name="sequence-number" dt:type="string" required="yes" />
</Schema>
```

The equivalent BNF for the collaboration diagram schema is:

1. <sequence-number> -> string
2. <condition> -> string
3. <name> -> string
4. <multiplicity> -> none | one | many
5. <instance-name> -> text
6. <args-type> -> text
7. <class-operator> -> <name> <sequence-number> <condition> <multiplicity> <args-type> <class>
8. <class> -> <name> <instance-name> <class-operator>*
9. <diagram> -> <class>

The above mentioned XML Schema Definitions have been used in the mapping schemes for class diagrams, sequence diagrams and collaboration diagrams respectively. These will act as the structure definitions for the XML documents produced through the mapping schemes for the UML class, sequence, and collaboration diagrams respectively. The XML document written on the basis of these structures will have to reference these definitions to get their structures.

2.2 The Scheme of Mapping

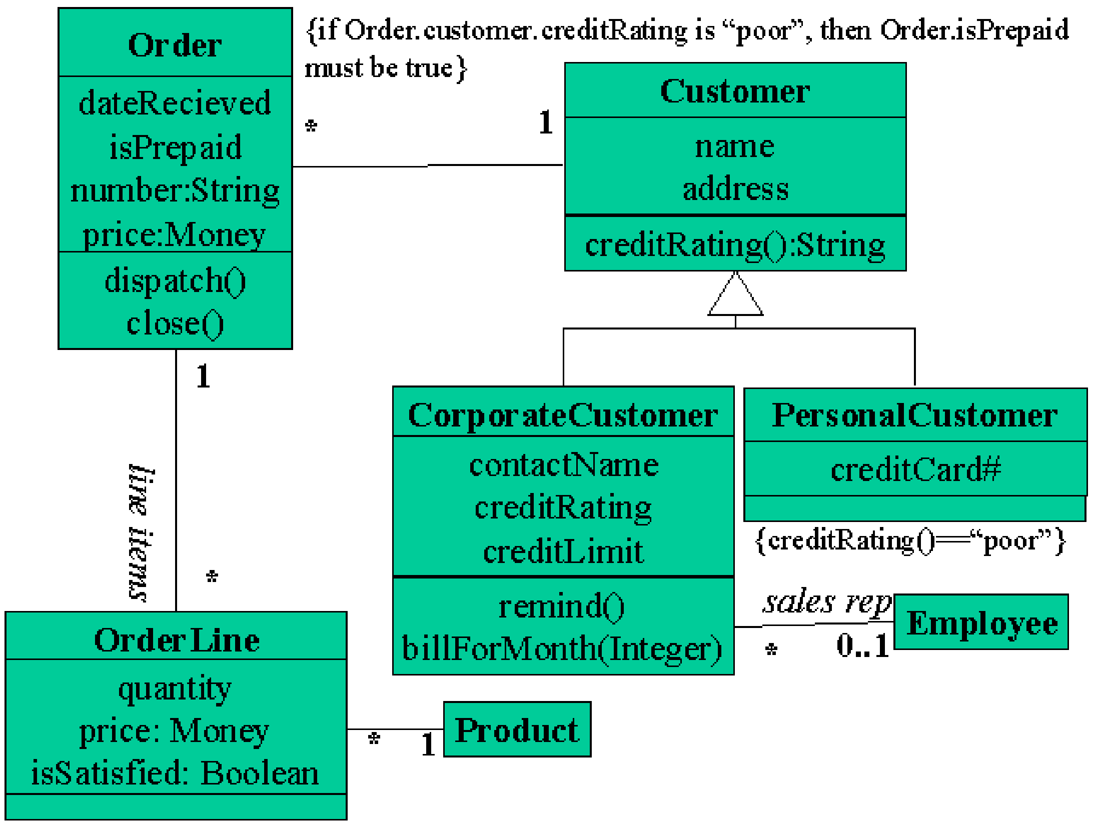
The scheme of mapping is the core of the thesis work. It explains the way the actual mapping from UML diagrams to XML will be taking place. We will be discussing the mapping schemes for class diagrams, sequence diagrams, and collaboration diagrams one by one in the next three sub-sections.

2.2.1 Class Diagram Mapping

The mapping of UML class diagram to XML is performed in the following manner:

1. First of all, the root node <diagram> with property value '*xmlns*' set to *class diagram XML Schema Definition* is generated.
2. Now, all the classes in the class diagram are taken one by one and mapped inside the <diagram> node as <class> nodes with property '*name*' set to *class name*.
3. Inside node <class>, first of all *instance variable*s are mapped as <instance-variable> nodes.
4. Next, the *operator*s are mapped as <operator> nodes with property '*name*' set to *operator name*, and child nodes <args-type> and <ret-type> containing the *argument list* and *return type* of the operator respectively.
5. The *constraint*s, if any, of the class are enlisted next in the <constraint> node.
6. Following constraints will be the association contribution of the class in an *association* of the class diagram. This will be enlisted in the <association> node. This node will have two properties, namely '*role-name*' set to *role name* of the class in the association, and '*multiplicity*'. It will also have a child node <class-name> containing the name of the *class* with which the association is made.
7. Next are the *composition* and *aggregation* contribution of the class, enlisted in the nodes <composition> and <aggregation>. These will have an attribute '*multiplicity*', and a child node <class-name> having the name of the *class* with which the aggregation/composition relationship is there.
8. Finally, there is the <generalization> node is there, which will have the name of the *class* in a child node <class-name> that has been inherited by the class.

Using above mapping scheme we can map the class diagram on the next page to the XML document followed by it.



```
<diagram xmlns="x-schema:classDiagram.xsd">
 <class name="Order">
  <instance-variable>dateRecieved</instance-variable>
  <instance-variable>isPrepaid</instance-variable>
  <instance-variable>number:String</instance-variable>
  <instance-variable>price:Money</instance-variable>
  <operator name="dispatch" />
  <operator name="close" />
  <constraint>{if Order.customer.creditRating is "poor" then Order.isPrepaid must be
"true"}</constraint>
  <association multiplicity="many">
   <class-name>Customer</class-name>
  </association>
  <association multiplicity="one">
   <class-name>OrderLine</class-name>
  </association>
 </class>

 <class name="Customer">
  <instance-variable>name</instance-variable>
  <instance-variable>address</instance-variable>
  <operator name="creditRating">
   <ret-type>String</ret-type>
  </operator>
  <association multiplicity="one">
   <class-name>Order</class-name>
  </association>
 </class>

 <class name="OrderLine">
```

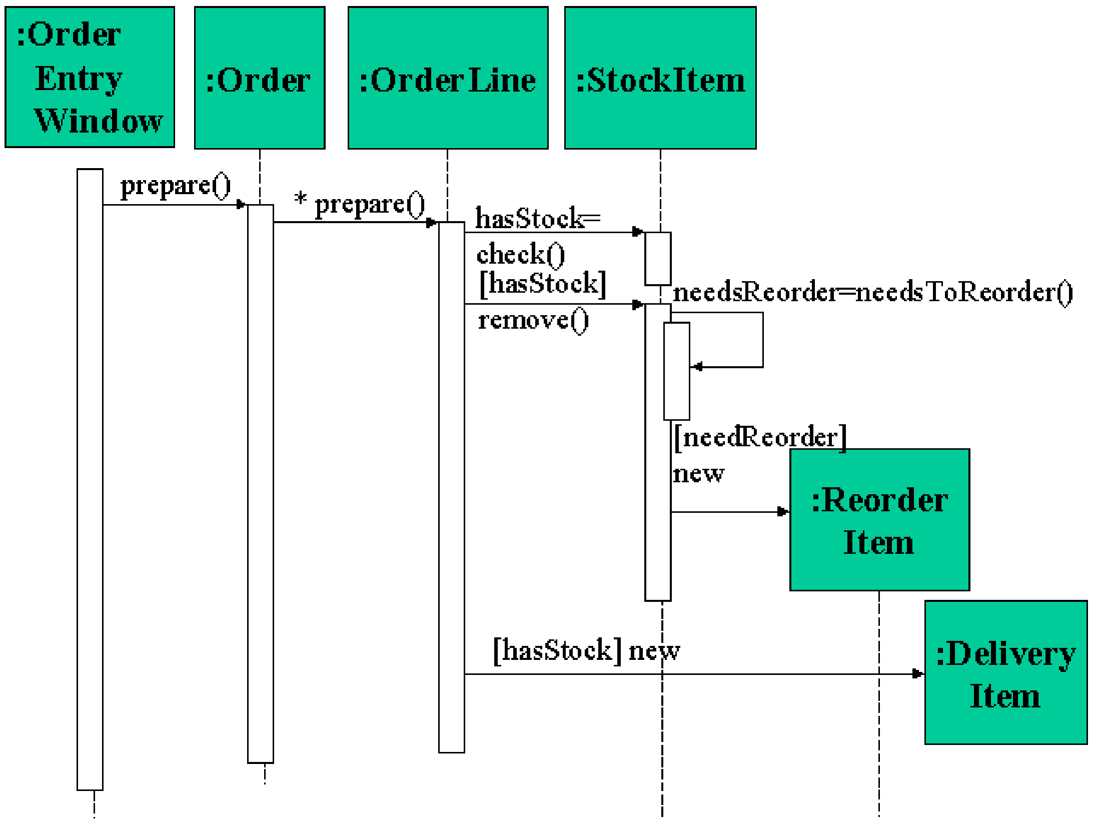
```
  <instance-variable>quantity:Integer</instance-variable>
  <instance-variable>price:Money</instance-variable>
  <instance-variable>isSatisfied:Boolean</instance-variable>
  <association multiplicity="many" role-name="line items">
   <class-name>Order</class-name>
  </association>
  <association multiplicity="many">
   <class-name>Product</class-name>
  </association>
 </class>

 <class name="CorporateCustomer">
  <instance-variable>contactName</instance-variable>
  <instance-variable>creditRating</instance-variable>
  <instance-variable>creditLimit</instance-variable>
  <operator name="remind" />
  <operator name="billForMonth">
   <args-type>Integer</args-type>
  </operator>
  <association multiplicity="many">
   <class-name>Employee</class-name>
  </association>
  <generalization>
   <class-name>Customer</class-name>
  </generalization>
 </class>

 <class name="PersonalCustomer">
  <instance-variable>creditCard#</instance-variable>
  <constraint>{creditRating() == "poor"}</constraint>
  <generalization>
   <class-name>Customer</class-name>
  </generalization>
 </class>

 <class name="Employee">
  <association multiplicity="optional" role-name="sales rep">
   <class-name>CorporateCustomer</class-name>
  </association>
 </class>

 <class name="Product">
  <association multiplicity="one">
   <class-name>OrderLine</class-name>
  </association>
 </class>
</diagram>
```

2.2.2 Sequence Diagram Mapping

The mapping scheme for the UML sequence diagrams is as follows:

1. The root node <diagram> is set with the property '*xmlns*' as the name of the *XML Schema Definition for the UML sequence diagram*.
2. The first instance object is taken as the property '*name*' of the <class-name> node. The <class-name> node will act as the parent node to all the messages represented by the <class-operator> nodes.
3. The <class-operator> node will have two properties '*multiplicity*' and '*condition*', and three child nodes, namely <method-name>, <args-type>, and <class-name>.
4. The node <method-name> will mark up the *name of the message*, and the node <args-type> will mark up the *return-type and arguments* of the message. The third node <class-name> will enclose the *class instance* to which the message goes.

For demonstration we can map the following sequence diagram to the following XML document.



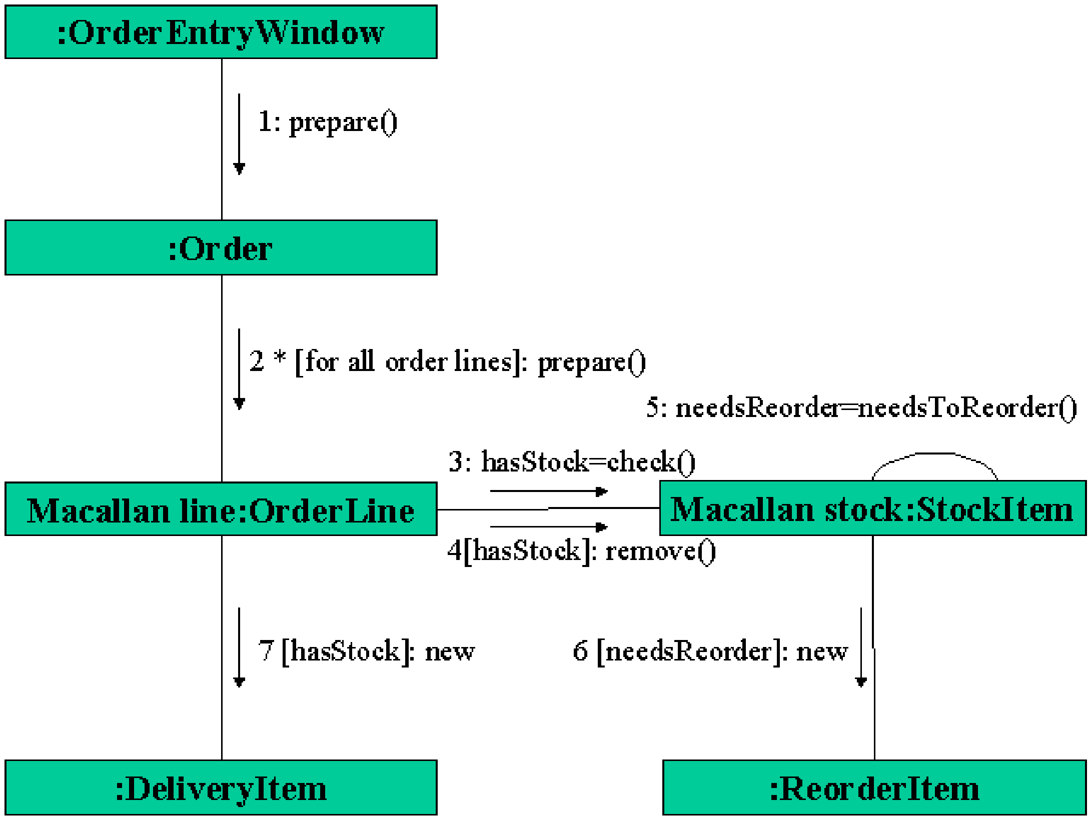
```xml
<diagram xmlns="x-schema:sequenceDiagram.xsd">
 <class-name name="OrderEntryWindow">
  <class-operator condition="true" multiplicity="one">
   <method-name>prepare</method-name>
   <class-name name="Order">
    <class-operator multiplicity="many" condition="true">
     <method-name>prepare</method-name>
     <class-name name="OrderLine">
      <class-operator condition="true" multiplicity="one">
       <method-name>checkStock</method-name>
       <args-type>retval=hasStock</args-type>
       <class-name name="StockItem" />
      </class-operator>
      <class-operator condition="hasStock" multiplicity="one">
       <method-name>remove</method-name>
       <class-name name="StockItem">
        <class-operator condition="true" multiplicity="one">
         <method-name>needToReorder</method-name>
         <args-type>retval=needsReorder</args-type>
         <class-name name="StockItem" />
        </class-operator>
        <class-operator condition="needsReorder" multiplicity="one">
         <method-name>new</method-name>
         <class-name name="ReorderItem" />
        </class-operator>
       </class-name>
      </class-operator>
      <class-operator condition="hasStock" multiplicity="one">
       <method-name>new</method-name>
       <class-name name="DeliveryItem" />
      </class-operator>
     </class-name>
    </class-operator>
   </class-name>
  </class-operator>
 </class-name>
</diagram>
```

2.2.3 Collaboration Diagram Mapping

The following mapping scheme will be used for mapping the UML collaboration diagram:

1. The root node <diagram> will contain all the diagram structure. The '*xmlns*' property of the <diagram> node is set as the *XML Schema Definition corresponding to collaboration diagram*.
2. The <diagram> node will have a child node <class>, which will be referring the *instance of the first class* in the collaboration diagram. The node <class> will have an attribute name, and two child nodes <instance-name> and <class-operator>.
3. The node <instance-name> will mark up the name of the instance, if any. The node <class-operator> will have four attributes, '*name*', '*sequence-number*', '*multiplicity*', and '*condition*' of the message; and two child nodes <args-type> and <class>.
4. The node <args-type> will represent the *return type and arguments* of the message. The node <class> will mark up the *class* to which the message goes.

The above scheme will map the following collaboration diagram as under:



```
<diagram xmlns="x-schema:collaborationDiagram.xsd">
 <class name="OrderEntryWindow">
  <class-operator name="prepare" sequence-number="1" condition="true" multiplicity="one">
   <class name="Order">
    <class-operator name="prepare" sequence-number="2" multiplicity="many" condition="for all
order lines">
     <class name="OrderLine">
      <instance-name>Macallan Line</instance-name>
      <class-operator name="checkStock" sequence-number="3" condition="true"
multiplicity="one">
       <args-type>retval=hasStock</args-type>
       <class name="StockItem">
        <instance-name>Macallan Stock</instance-name>
       </class>
      </class-operator>
      <class-operator name="remove" sequence-number="4" condition="hasStock"
multiplicity="one">
       <class name="StockItem">
        <instance-name>Macallan Stock</instance-name>
        <class-operator name="needToReorder" sequence-number="5" condition="true"
multiplicity="one">
         <args-type>retval=needsReorder</args-type>
         <class name="StockItem">
          <instance-name>Macallan Stock</instance-name>
         </class>
        </class-operator>
        <class-operator name="new" sequence-number="6" condition="needsReorder"
multiplicity="one">
         <class name="ReorderItem" />
        </class-operator>
       </class>
      </class-operator>
      <class-operator name="new" sequence-number="7" condition="hasStock"
multiplicity="one">
       <class name="DeliveryItem" />
      </class-operator>
     </class>
    </class-operator>
   </class>
  </class-operator>
 </class>
</diagram>
```

2.3 VALIDATIONS

Generally speaking, software engineering researchers seek better ways to develop and evaluate software. They are motivated by practical problems, and key objectives of the research are often quality, cost, and timeliness of software products. This section presents a character of the validation we will provide. This has been derived from the text example and demonstration problem we have studied. More rigorous validations can be forced, but we have kept it general as we are not referring to a specific domain of software solution but the process of the development of the process by using the methods of UML.

Good research requires not only a result, but also clear and convincing evidence that the result is sound. This evidence should be based on experience or systematic analysis, not simply persuasive argument or textbook examples. So while dealing with the domains of applications we should apply the validation more strictly than the way we have followed. Our approach of not going stricter is an indication that validation in practice is not always clear and convincing.

The system validation refers to the development of the right solution for a given problem. So, in order to ensure the right way of mapping we must be aware of the ways to validate the class diagrams, sequence diagrams, and collaboration diagrams with respect to the class diagram(s). This will ensure that we are doing the right sort of mapping once we have a correct class, sequence, or collaboration diagram. If this check is not performed we may end up doing a mapping for virtually no use. Hence, the process of validation of the diagram is one of the most important aspect in our mapping scheme.

Keeping the above philosophy we will be proposing the validation approach for us as follows.

2.3.1 CLASS DIAGRAMS

Validating of the class diagrams is most important for getting an error free product. The most important aspects are:

1. The class names in a namespace must be unique. These are case sensitive.
2. The association should be contributed by both the classes. It means that if class A shows an association with class B, then class B must also have an association to class A. Both the classes A and B must be there in the same class diagram.
3. Similarly, the composition and the aggregation will also be contributed by both classes, which must be in the same class diagram. Although it does not seem to be following the actual UML definitions of these relationships, but have done so in order to keep the contribution of each class in the relationships with their definition.
4. Generalization should be inherited from a class in the same class diagram.
5. Additionally we can also add the UML naming conventions as the validation rules, but it is better that these things be put on only if we are following the stricter rules, in real time development environment as well.

2.3.2 Sequence Diagrams

Validating sequence diagrams is a bit more complex than the class diagrams, as it involves the class diagrams, which we are referring to in the sequence diagrams as well. So we can follow the following restrictions:

1. The class diagram(s) we are referring to in the sequence diagram must be valid.
2. The classes in the sequence diagram must be in the class diagram(s).
3. The message names must be either specified as a general operator, like '*new*', or in the corresponding class as operator, or the class inherited by this class, of the class diagram.
4. Stricter rules may involve the prototype of the messages, although at every place it is not recommended, since the purpose of sequence diagram is not to enforce the structure.

2.3.3 Collaboration Diagrams

Like sequence diagram the collaboration diagrams are also validating against the classes. The above rules may be reproduced for the collaboration diagram:

1. The class diagrams we are referring to in the collaboration diagram must be valid.
2. The classes in the collaboration diagram must be in the class diagram(s).
3. The message names must be either specified as a general operator, like '*new*', or in the corresponding class as operator, or the class inherited by this class, of the class diagram.
4. Stricter rules may involve the prototype of the messages, although at every place it is not recommended, since the purpose of collaboration diagram is not to enforce the structure.

CHAPTER 3

DESIGN

In this chapter we have presented the logical design of the system we have developed to automate the process of mapping UML diagrams to XML documents using the mapping scheme presented in the previous chapter. Thus, this system is as per the specifications provided in chapter 2. The major activities involved in the development of this design were reviewing the previous specifications, designing the system architecture and components, and presenting the complete design. During this process, we have sticked to the object-oriented software engineering design guidelines that helped us achieve an efficient, reliable and maintainable system design.

The design of the system developed by us has considered the requirements, data and processing characteristics as specified previously to specify its architecture and the input and output design. We have developed a modular and object-oriented system design as presented later in this chapter with the help of use case and class diagram models. Since the output of the system must be according to the requirements, we have to keep on reviewing these continuously during the design process as and when required.

The chapter includes four sections: Architecture, Input Design, Output Design, and System Design.

The first section discusses the architecture of the solution we have developed. It will explain various layers and components of the system, called "**UML to XML converter**", along with the system flow.

The second section discusses the Input Design of the system. It explains the way inputs will be provided to the system. We have also discussed the format of the input and the input media.

The third section is on Output Design. It discusses the way the system will produce the outputs. It will also explain the format of the output and output media.

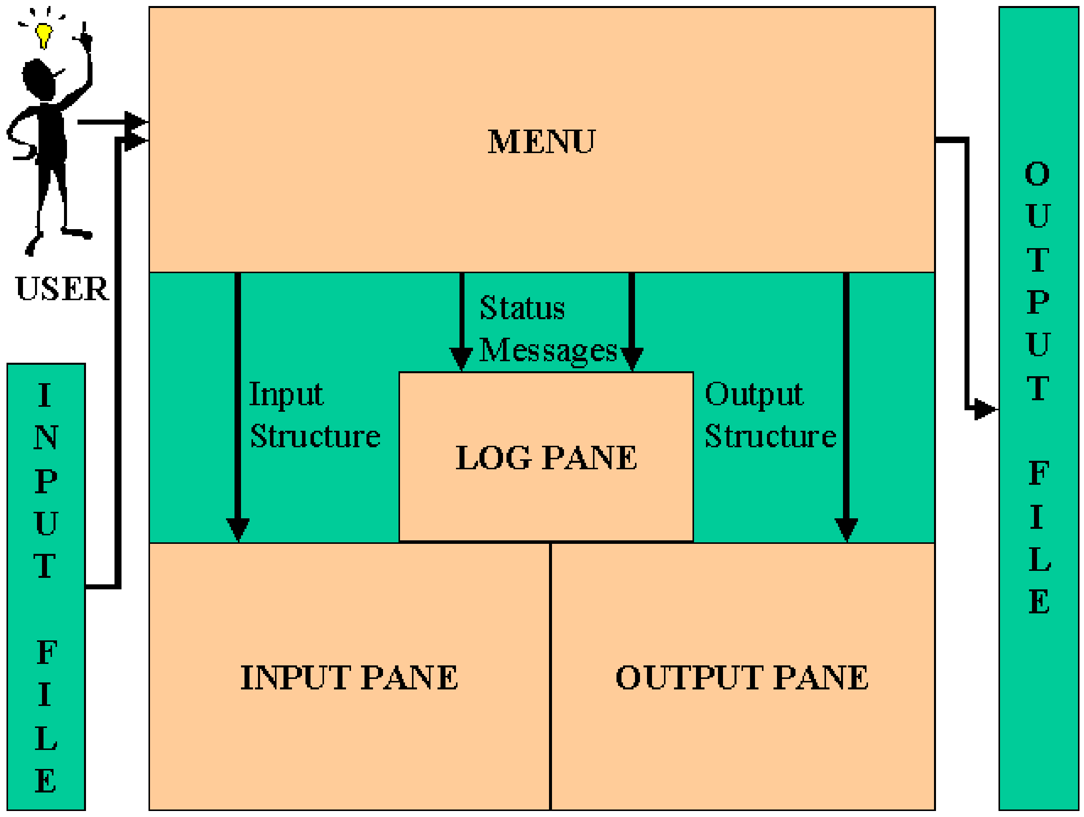
The last section explains the System Design. It expresses the complete system design of our system. It presents the use case analysis and class diagrams of our system. It expresses the integration of the components of our system in one design.

3.1 Architecture

Our system will be having the abstract machine model of the software architecture. The system will be organized into a series of layers of sub-systems, each of which will be providing a set of services to our systems. Each of these layers may again be termed as abstract machine performing the specific task.

The system will be having a menu based GUI interface with file based input and the output will directed to the files as well, however the input will be read and displayed in the input pane of the system and the output will also be displayed in the output pane of the GUI. A system log will also be maintained for the current job.

The system architecture of our system can be demonstrated through the following figure.



The system will be functioning as per the above figure. User will interact with the system through the menu, i.e. the menu will act as the interface for the user. The input structure will be specified in the input file. The input structure will be displayed in the Input Pane. After generating the output in XML format it will be displayed in the Output Pane. The tasks performed and the status of the tasks will be displayed in the Log Pane. Finally the generated output is stored in the Output File which will be an .XML file.

3.2 Input Design

Input quality and accuracy are essential for every successful information system. With today’s technology, a wide variety of input media is available, including optical, voice and magnetic recognition devices: special-purpose terminals; and graphical input devices. Input design includes selecting appropriate input media and methods, developing efficient input procedures, reducing input volume and avoiding input errors. In carrying out these tasks, the systems analyst must consider three key procedures: data capture, data entry and data input. Data capture involves identifying and recording source data. Data entry involves converting source data into a computer-readable form. Data input involves the actual introduction of data into the information system.

To help reduce input errors, data is validated by one or more checks of sequence, existence, range and limit, reasonableness, validity, combination and batch control. Source document design involves forms used to request and collect input data, to trigger or authorize and input action and to provide a record of the original transaction. Input record design for batch input data involves placing data in a temporary file that becomes the input file for data entry. We have followed the same method for our input design. We will be providing the input as a diagram file saved in the text format. The format of our diagrams can be specified in Backaus-Naur Form as given in next three sub-sections.

3.2.1 Class Diagram Input File

1. *Class-name* => text
2. *Instance-number* => number
3. *Operator-number* => number
4. *Association-number* => number
5. *Composition-number* => number
6. *Aggregation-number* => number
7. *Generalization-number* => number
8. *Operator-name* => text
9. *Multiplicity* => one | many | optional
10. *Role-name* => text
11. *Ret-type* => text
12. *Args-type* => text
13. *Operator* => *Operator-name*; *Args-type*; *Ret-type*
14. *Constraint* => {text}
15. *Association* => *Multiplicity*; *Role-name*; *Class-name*
16. *Composition* => *Multiplicity*; *Class-name*
17. *Aggregation* => *Multiplicity*; *Class-name*
18. *Generalization* => *Class-name*
19. *Instance-variable* => text
20. *Class-record* => *Class-name*; *Instance-number*; {*Instance-variable*;}$^{\textit{Instance-number}}$ *Operator-number*; {*Operator*;}$^{\textit{Operator-number}}$ *Constraint*; *Association-number*; {*Association*;}$^{\textit{Association-number}}$ *Composition-number*; {*Composition*;}$^{\textit{Composition-number}}$ *Aggregation-number*; {*Aggregation*;}$^{\textit{Aggregation-number}}$ *Generalization-number*; {*Generalization*;}$^{\textit{Generalization-number}}$
21. *Class-diagram* => {*Class-record <return>*}*

Hence the sample input file for the class diagram in the previous chapter is as given on the next page.

Order; 4; dateRecieved; isPrepaid; number:String; price:Money; 2; dispatch;;; close;;; {if Order.customer.creditRating is "poor", then Order.isPrepaid must be true}; 2; many;;Customer; one;;OrderLine; 0; 0; 0;
Customer; 2; name; address; 1; creditRating;;String; ; 1; one;;Order; 0; 0; 0;
OrderLine; 3; quantity:Integer; price:Money; isSatisfied:Boolean; 0; ; 2; many;line items;Order; many;;Product; 0; 0; 0;
Product; 0; 0; ; 1; one;;OrderLine; 0; 0; 0;
CorporateCustomer; 3; contactName; creditRating; creditLimit; 2; remind;;; billForMonth;Integer;; ; 1; many;;Employee; 0; 0; 1; Customer;
PersonalCustomer; 1; creditCard#; 0; {creditRating()=="poor"}; 0; 0; 0; 1; Customer;
Employee; 0; 0; ; 1; Optional; sales rep;CorporateCustomer; 0; 0; 0;

3.2.2 SEQUENCE DIAGRAM INPUT FILE

1. *Message-name* => text
2. *Message-number* => number
3. *Class-name* => text
4. *Condition* => text
5. *Args-type* => text
6. *Multiplicity* => one | many | optional
7. *Message* => *Message-name*; *Condition*; *Args-type*; *Multiplicity <return> Class*
8. *Class* => *Class-name*; *Message-number*; {*Message*;}$^{Message\text{-}number}$
9. *Sequence-diagram* => *Class*

Thus we can write the input file for the sequence diagram in the previous chapter as given below:

```
OrderEntryWindow;1;
 Prepare;;;;
 Order;1;
  Prepare;;;many;
  OrderLine;3;
   CheckStock;;retval=hasStock;;
   StockItem;0;
   Remove;hasStock;;;
   StockItem;2;
    NeedToReorder;;retval=needsReorder;;
    StockItem;0;
    New;needsReorder;;;
    ReorderItem;0;
   New;hasStock;;;
   DeliveryItem;0;
```

3.2.3 COLLABORATION DIAGRAM INPUT FILE

1. *Message-name* => text
2. *Message-number* => number
3. *Sequence-number* => number
4. *Class-name* => text
5. *Instance-name* => text
6. *Class-Instance* => *Class-name* | *Instance-name :Class-name*
7. *Condition* => text
8. *Args-type* => text
9. *Multiplicity* => one | many | optional
10. *Message* => *Message-name*; *Sequence-number*; *Condition*; *Args-type*; *Multiplicity <return> Class*
11. *Class* => *Instance-name*; *Message-number*; {*Message*;}$^{Message\text{-}number}$

12. *Collaboration-diagram* => *Class*

The input file for the collaboration diagram in the previous chapter will look like given below:

```
OrderEntryWindow;1;
 prepare;1;;;;
 Order;1;
  prepare;2;for all order lines;;many;
  Macallan Line:OrderLine;3;
   checkStock;3;;retval=hasStock;;
   Macallan Stock:StockItem;0;
   remove;4;hasStock;;;
   Macallan Stock:StockItem;2;
    needToReorder;5;;retval=needsReorder;;
    Macallan Stock:StockItem;0;
    new;6;needsReorder;;;
    ReorderItem;0;
   new;7;hasStock;;;
   DeliveryItem;0;
```

3.3 OUTPUT DESIGN

This section explains the output design of our system. Since a system can produce various types of outputs, we must specify the output design of our system as well. In addition to printed output and screen output, which are the most common form, other examples include audio output, automated facsimile and faxback, e-mail and links to Web pages. Retail point-of-sale terminals and ATM's produce other specialized forms of output, for example. There are several types of output reports and are classified as detail, exception or summary reports and as internal or external reports. There are printed report designs for both stock paper and specialty forms. Screen output can be both character and graphical. The objective of output control is to ensure that information is correct, complete and secure. Data integrity must be maintained during the output process, reports should have appropriate titles, they should be dated and pages should be numbered and the record counts should be reconciled against the input totals. Output security should protect the privacy and rights of individuals and organizations and protect the data from theft or unauthorized access. Reports should be clearly marked as confidential and should be distributed only to authorized personnel. Sensitive reports should be stored in secure areas. There are many products available that can provide online report and screen generators that help design reports and screen displays.

The output design for our proposed system will be in the following way:

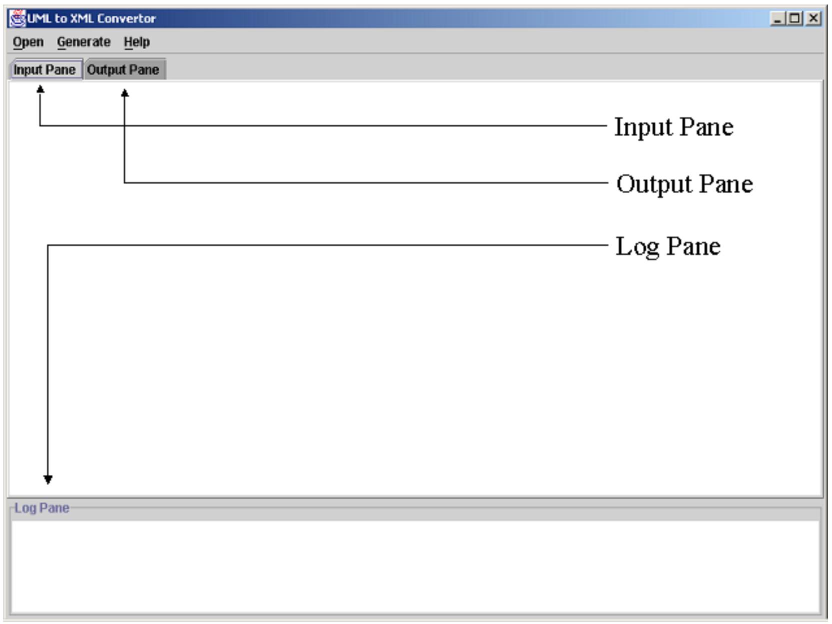
1. The system shall be displaying the input structure (i.e. Class, Sequence, or Collaboration Diagram) in a formatted text format so that the input structure is easier to understand.
2. The output will be in the XML format so that it can be displayed in the XML tagged format in the output pane.
3. The status of the system should be displayed in the log pane in an informative way.

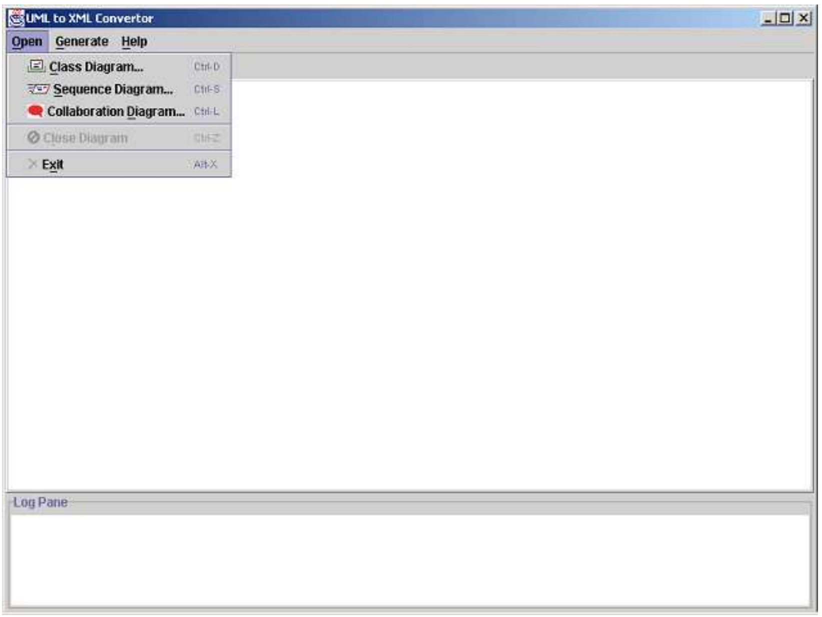
Although we could have easily improved the format of the input structures by displaying the diagrams properly, but since it is beyond the scope of the work, so it was not concentrated upon. Furthermore, since we need the output in XML format, so we have displayed it as it is generated.

The log pane is very important as it shows us the status of the work going on. We can produce the warning messages as well, but it is better to have a log pane as it will make user understand what is going on in the process.

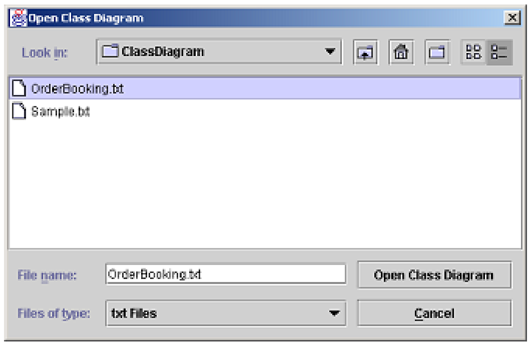
3.3.1 User Interface

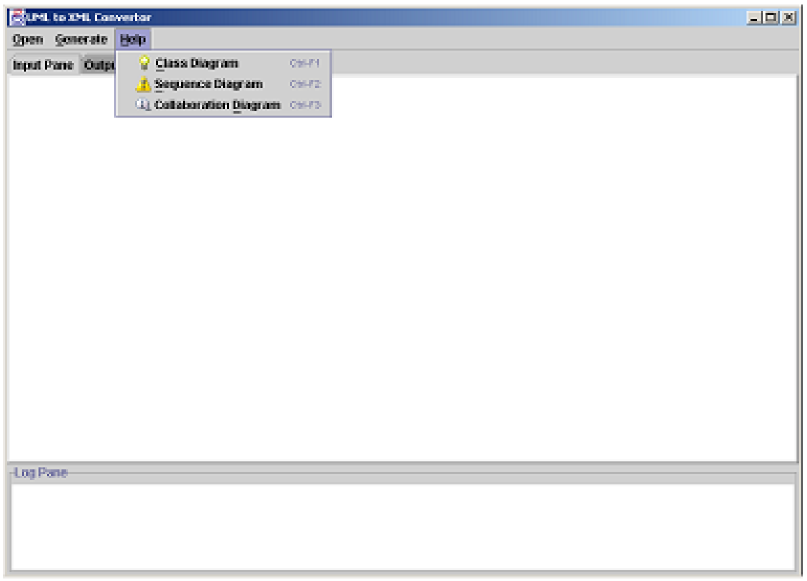
The UTXGUI user interface is a menu-based desktop GUI. It will be having a menubar for accessing various functions and three panes to display the results: Input Pane, Output Pane, and Log Pane. The Input pane will display the structure of the input. Output Pan will display the generated output, while the Log Pane will display the status of the current process.





The 'Open' menu will have the options of opening the class diagram, sequence diagram, and collaboration diagram. The opening of these diagram will bring the file chooser dialog from where we can choose the input file. This dialog is shown in the adjacent figure. We can close a diagram or we can generate the corresponding XML document, if it is a valid structure.

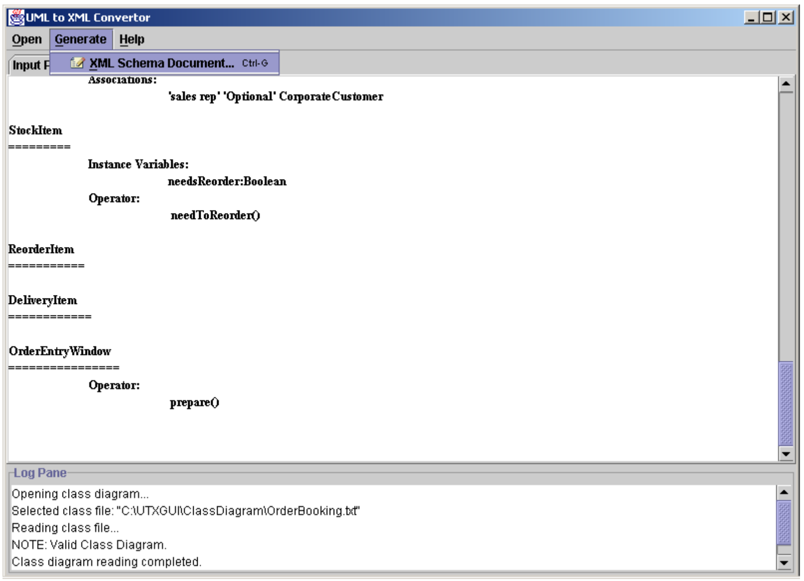




We can also take help on the class diagram, sequence diagram or the collaboration diagram. These options will provide the online help on these documents.

When we open a diagram, we will get its structure in the input pane. The log of each and every step will be displayed in the log pane. After opening the diagram the option of generating the XML document will also get enabled. Now, we can do the mapping and generate the corresponding XML document. If the document is invalid it will not get enabled.

This is the entire functioning of the UTXGUI tool for the converting of UML diagram to XML document. The same procedure applies for sequence diagram and the collaboration diagram. The only difference is that here it will ask the input structure for the class diagram that are to be used to validate the sequence and the collaboration diagrams.

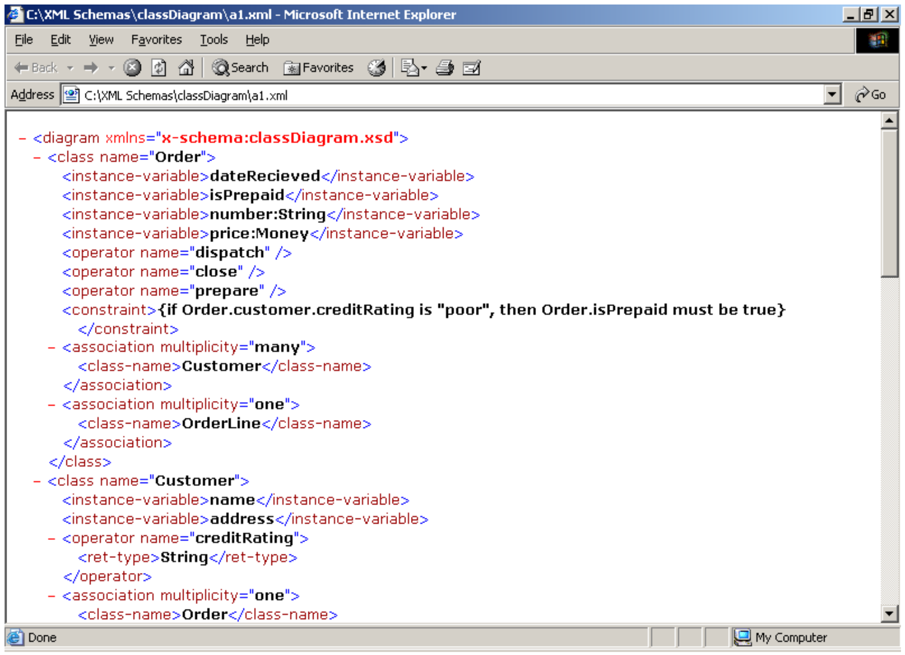


```
          Associations:
                    'sales rep' 'Optional' CorporateCustomer

StockItem
=========
          Instance Variables:
                    needsReorder:Boolean
          Operator:
                    needToReorder()

ReorderItem
===========

DeliveryItem
============

OrderEntryWindow
================
          Operator:
                    prepare()
```

```
Opening class diagram...
Selected class file: "C:\UTXGUI\ClassDiagram\OrderBooking.txt"
Reading class file...
NOTE: Valid Class Diagram.
Class diagram reading completed.
```

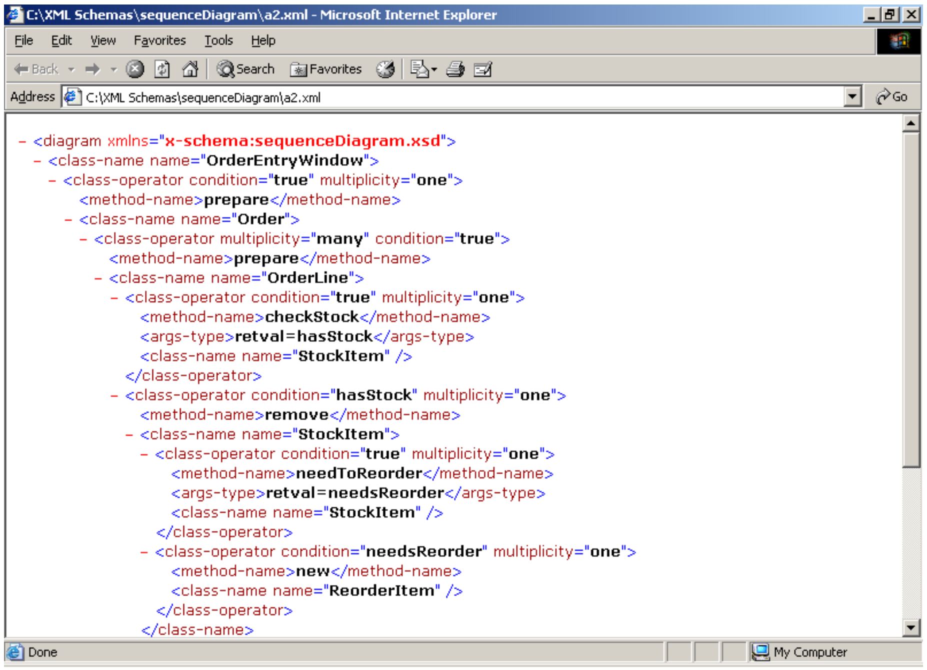
3.3.2 Outputs

The UTXGUI system output include the class diagram XML document generated by our system. The following is the screenshot of the explorer window when we opened the class diagram document generated by our system. This figure corresponds to the example class diagram used in the chapter 1 of the dissertation.


C:\XML Schemas\classDiagram\a1.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XML Schemas\classDiagram\a1.xml
Go
Done
My Computer

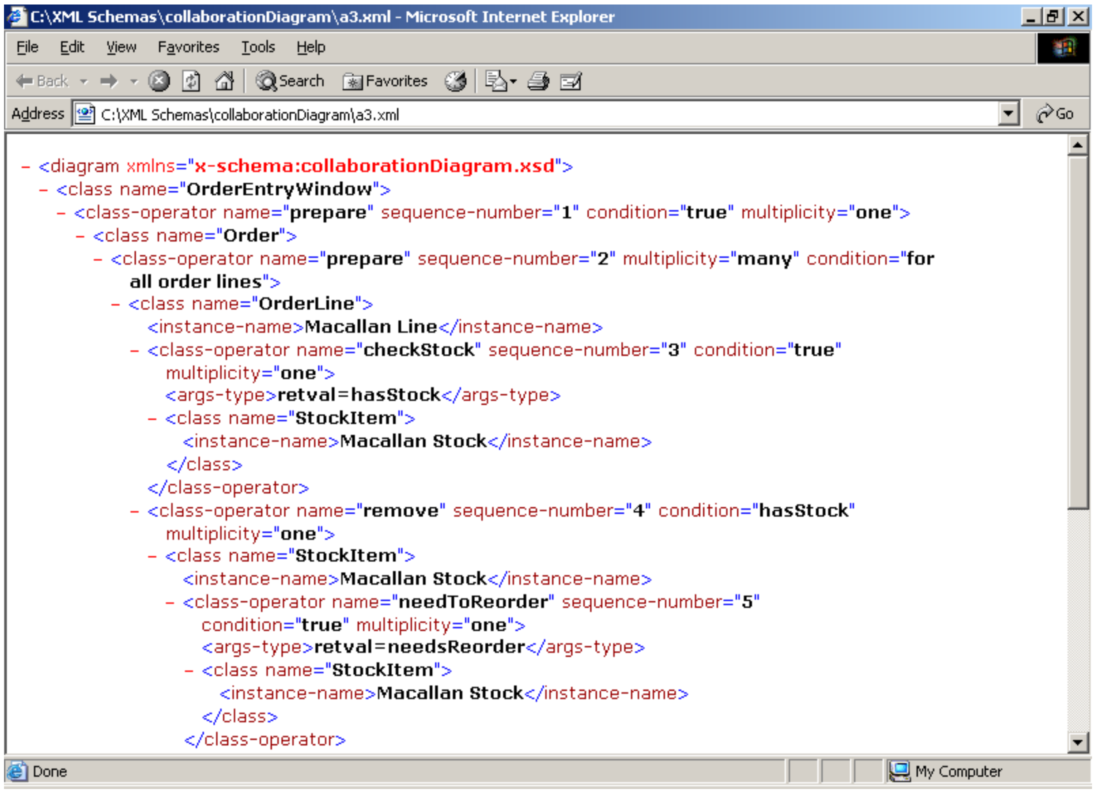
```
- <diagram xmlns="x-schema:classDiagram.xsd">
  - <class name="Order">
      <instance-variable>dateRecieved</instance-variable>
      <instance-variable>isPrepaid</instance-variable>
      <instance-variable>number:String</instance-variable>
      <instance-variable>price:Money</instance-variable>
      <operator name="dispatch" />
      <operator name="close" />
      <operator name="prepare" />
      <constraint>{if Order.customer.creditRating is "poor", then Order.isPrepaid must be true}
        </constraint>
    - <association multiplicity="many">
        <class-name>Customer</class-name>
      </association>
    - <association multiplicity="one">
        <class-name>OrderLine</class-name>
      </association>
    </class>
  - <class name="Customer">
      <instance-variable>name</instance-variable>
      <instance-variable>address</instance-variable>
    - <operator name="creditRating">
        <ret-type>String</ret-type>
      </operator>
    - <association multiplicity="one">
        <class-name>Order</class-name>
```

The output corresponding to the example sequence diagram of chapter 1 produced by the system, when viewed in the explorer is as under.


C:\XML Schemas\sequenceDiagram\a2.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XML Schemas\sequenceDiagram\a2.xml
Go
Done
My Computer

```
- <diagram xmlns="x-schema:sequenceDiagram.xsd">
  - <class-name name="OrderEntryWindow">
    - <class-operator condition="true" multiplicity="one">
        <method-name>prepare</method-name>
      - <class-name name="Order">
        - <class-operator multiplicity="many" condition="true">
            <method-name>prepare</method-name>
          - <class-name name="OrderLine">
            - <class-operator condition="true" multiplicity="one">
                <method-name>checkStock</method-name>
                <args-type>retval=hasStock</args-type>
                <class-name name="StockItem" />
              </class-operator>
            - <class-operator condition="hasStock" multiplicity="one">
                <method-name>remove</method-name>
              - <class-name name="StockItem">
                - <class-operator condition="true" multiplicity="one">
                    <method-name>needToReorder</method-name>
                    <args-type>retval=needsReorder</args-type>
                    <class-name name="StockItem" />
                  </class-operator>
                - <class-operator condition="needsReorder" multiplicity="one">
                    <method-name>new</method-name>
                    <class-name name="ReorderItem" />
                  </class-operator>
                </class-name>
```

Similarly, the view of the output of the collaboration diagram is in the following diagram.


C:\XML Schemas\collaborationDiagram\a3.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XML Schemas\collaborationDiagram\a3.xml
Go
Done
My Computer

```
- <diagram xmlns="x-schema:collaborationDiagram.xsd">
  - <class name="OrderEntryWindow">
    - <class-operator name="prepare" sequence-number="1" condition="true" multiplicity="one">
      - <class name="Order">
        - <class-operator name="prepare" sequence-number="2" multiplicity="many" condition="for
            all order lines">
          - <class name="OrderLine">
              <instance-name>Macallan Line</instance-name>
            - <class-operator name="checkStock" sequence-number="3" condition="true"
                multiplicity="one">
                <args-type>retval=hasStock</args-type>
              - <class name="StockItem">
                  <instance-name>Macallan Stock</instance-name>
                </class>
              </class-operator>
            - <class-operator name="remove" sequence-number="4" condition="hasStock"
                multiplicity="one">
              - <class name="StockItem">
                  <instance-name>Macallan Stock</instance-name>
                - <class-operator name="needToReorder" sequence-number="5"
                    condition="true" multiplicity="one">
                    <args-type>retval=needsReorder</args-type>
                  - <class name="StockItem">
                      <instance-name>Macallan Stock</instance-name>
                    </class>
                  </class-operator>
```

3.4 System Design

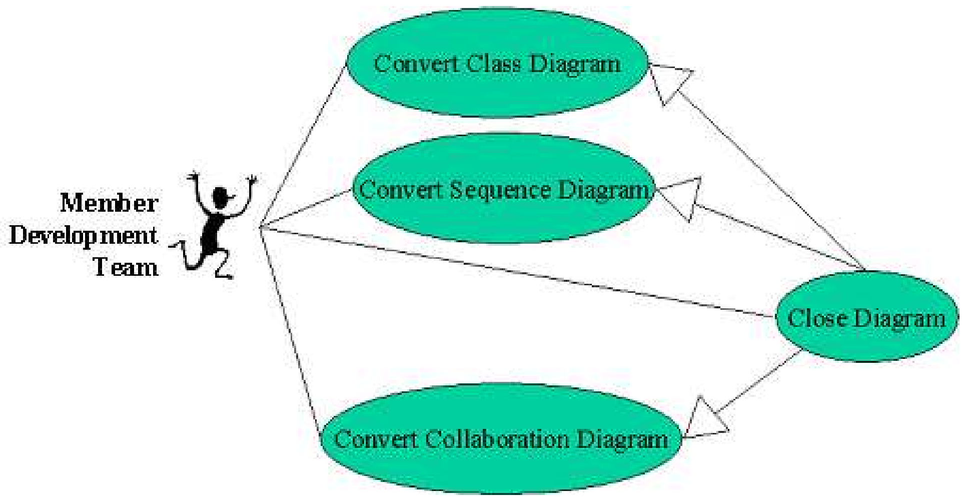
The system developed has been according to the object-oriented design of software development. The following strategy has been followed through this development process:

- Object-oriented Analysis. It is concerned with developing an object-oriented model of application domain. The identified objects reflect entities and operations that are associated with the problem that is solved.
- Object-oriented Design. It is concerned with the developing an object oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution to the problem that is being solved.
- Object-oriented Programming. It is concerned with the realizing a software design using an object-oriented programming language. We have used Java as the programming language.

We have realized the first two steps till now using the use case model and class diagram models and now will represent these two using UML. The third step will be discussed in the next chapter. Although we should have presented these diagrams earlier as per the standard process, but these have been avoided in order to first make the things clearer with the vision followed for the mappings, as that is the prime objective of the dissertation.

3.4.1 USE CASE MODEL

The following is the use case diagram for the system:



There are four use cases in the system. These are Convert Class Diagram, Convert Sequence Diagram, Convert Collaboration Diagram, and Close Diagram. The use case description of these use cases is given ahead.

Use Case I: Convert Class Diagram

Precondition	We have a class diagram structure in input file, along with the specified XML Schema Definition for Class Diagram
Start State	User selects the class diagram from the menu.
Initiator	Member Development Team
Order of Actions	1. Choose the class diagram to be converted. 2. Validate the class diagram. 3. Convert the class diagram to XML document.
Possible End State	We have the XML document corresponding to the class diagram structure.
Paths of Execution not Allowed	If class diagram is invalid, we can’t generate XML document.
Alternate Paths that are Inlined or Extracted from Basic Path	After the conversion we can close the class diagram.
Interactions between Actors and the System	1. User will choose class diagram from the menu. 2. He will then specify the input file where class diagram structure is stored. 3. He will then choose the generate option from the menu, if the class diagram structure is validated. 4. After generating the XML class diagram document, he can opt for closure of the class diagram.
Usage of Resources	1. Storage space for Input File, and Output File.

Use Case II: Convert Sequence Diagram

Precondition	We have a sequence diagram structure in input file, along with the specified XML Schema Definition for Sequence Diagram
Start State	User selects the sequence diagram from the menu.
Initiator	Member Development Team
Order of Actions	1. Choose the Sequence diagram, and the class diagram(s) used by this sequence diagram, to be converted. 2. Validate the sequence diagram against the specified class diagram. 3. Convert the sequence diagram to XML document.
Possible End State	We have the XML document corresponding to the sequence diagram structure.
Paths of Execution not Allowed	If sequence diagram is invalid, we can't generate XML document.
Alternate Paths that are Inlined or Extracted from Basic Path	After the conversion we can close the sequence diagram.
Interactions between Actors and the System	1. User will choose sequence diagram from the menu. 2. User will be specifying the class diagrams that the sequence diagram has used. 3. He will then specify the input file where sequence diagram structure is stored. 4. He will then choose the generate option from the menu, if the sequence diagram structure is validated. 5. After generating the XML sequence diagram document, he can opt for closure of the sequence diagram.
Usage of Resources	1. Storage space for Input File, and Output File.

Use Case III: Convert Collaboration Diagram

Precondition	We have a collaboration diagram structure in input file, along with the specified XML Schema Definition for Collaboration Diagram
Start State	User selects the collaboration diagram from the menu.
Initiator	Member Development Team
Order of Actions	1. Choose the Collaboration diagram, and the class diagram(s) used by this collaboration diagram, to be converted. 2. Validate the Collaboration diagram against the specified class diagram. 3. Convert the collaboration diagram to XML document.
Possible End State	We have the XML document corresponding to the collaboration diagram structure.
Paths of Execution	If collaboration diagram is invalid, we can't generate XML

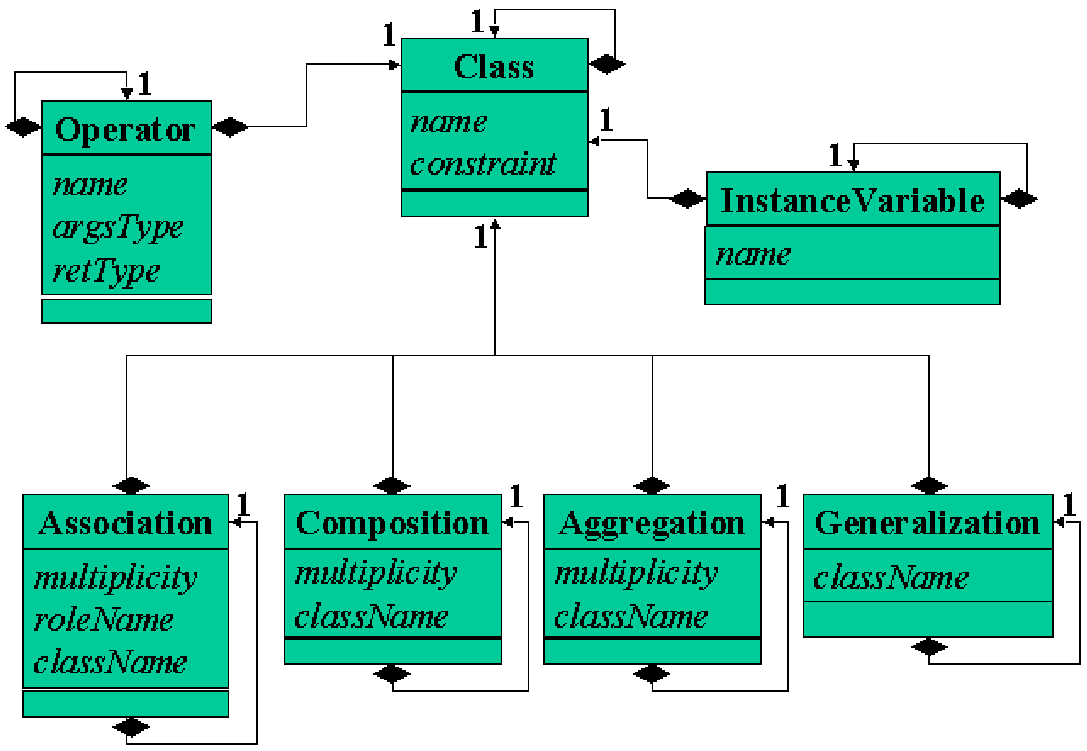
not Allowed	document.
Alternate Paths that are Inlined or Extracted from Basic Path	After the conversion we can close the collaboration diagram.
Interactions between Actors and the System	1. User will choose collaboration diagram from the menu. 2. User will be specifying the class diagrams that the collaboration diagram has used. 3. He will then specify the input file where collaboration diagram structure is stored. 4. He will then choose the generate option from the menu, if the collaboration diagram structure is validated. 5. After generating the XML collaboration diagram document, he can opt for closure of the collaboration diagram.
Usage of Resources	1. Storage space for Input File, and Output File.

Use Case IV: Close Diagram

Precondition	We have a class, sequence or collaboration diagram opened with our system.
Start State	User selects to close the diagram from the menu.
Initiator	Member Development Team
Order of Actions	1. Choose the close diagram option from the menu. 2. System will check which diagram is opened with the system. 3. It will close the diagram and all resources are freed.
Possible End State	The open diagram is closed.
Paths of Execution not Allowed	Nil.
Alternate Paths that are Inlined or Extracted from Basic Path	Nil.
Interactions between Actors and the System	1. User will choose close diagram from the menu.
Usage of Resources	Nil.

3.4.2 CLASS DIAGRAM MODEL

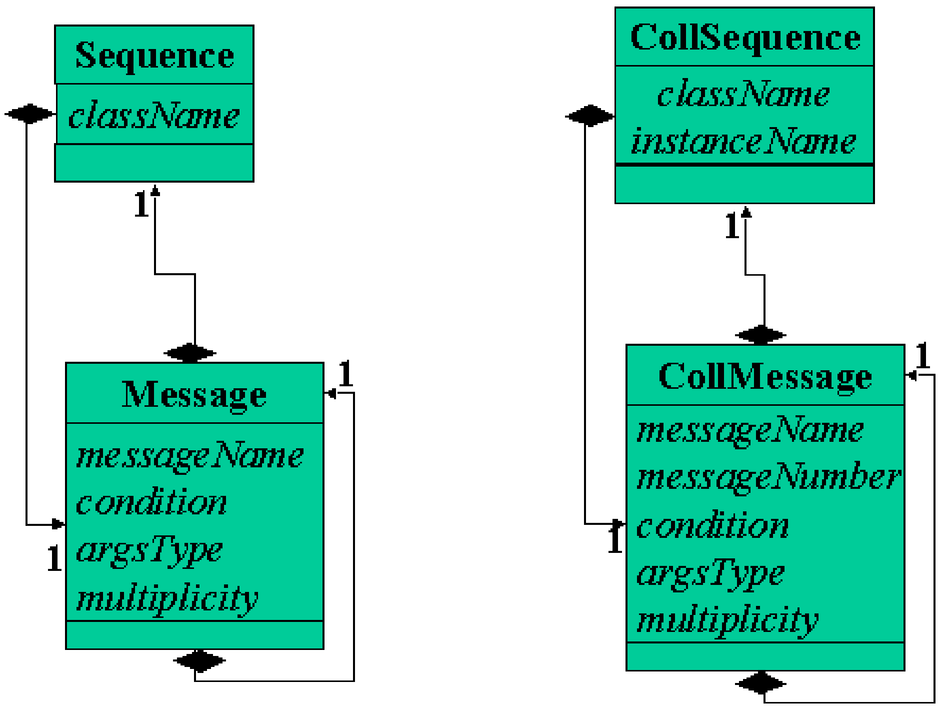
The class diagram of the system represents the static model of the system. It is very important for the forward as well as backward engineering of the product. We will be presenting the class diagram of our system now. These figures will present the static view of our system. We can then concentrate on the dynamic aspects of our system.



The previous diagram represents the class diagram of the structure representing the class diagram. It is an entity class. The class 'Class' will be representing the class diagram. It will be having two attributes: name, and constraint. It will be composing of the InstanceVariable, Operator, Association, Composition, Aggregation, and Generalization classes as well. We have used the linked structure of all these classes. So we have a 'composition' relationship amongst these classes. Also all these classes have the same relationship with themselves. Thus we will have a multilinked structure of these classes representing the class diagram.

The sequence diagram can be represented by the structure in the following figure. Here we will have a 'Sequence' class, which will be representing the sequence diagram. The sequence will be an entity class, and it will have a className and a message. The message will have the attributes for a message and will be composing of a message and a sequence. This will be enabling us to represent us a sequence diagram.

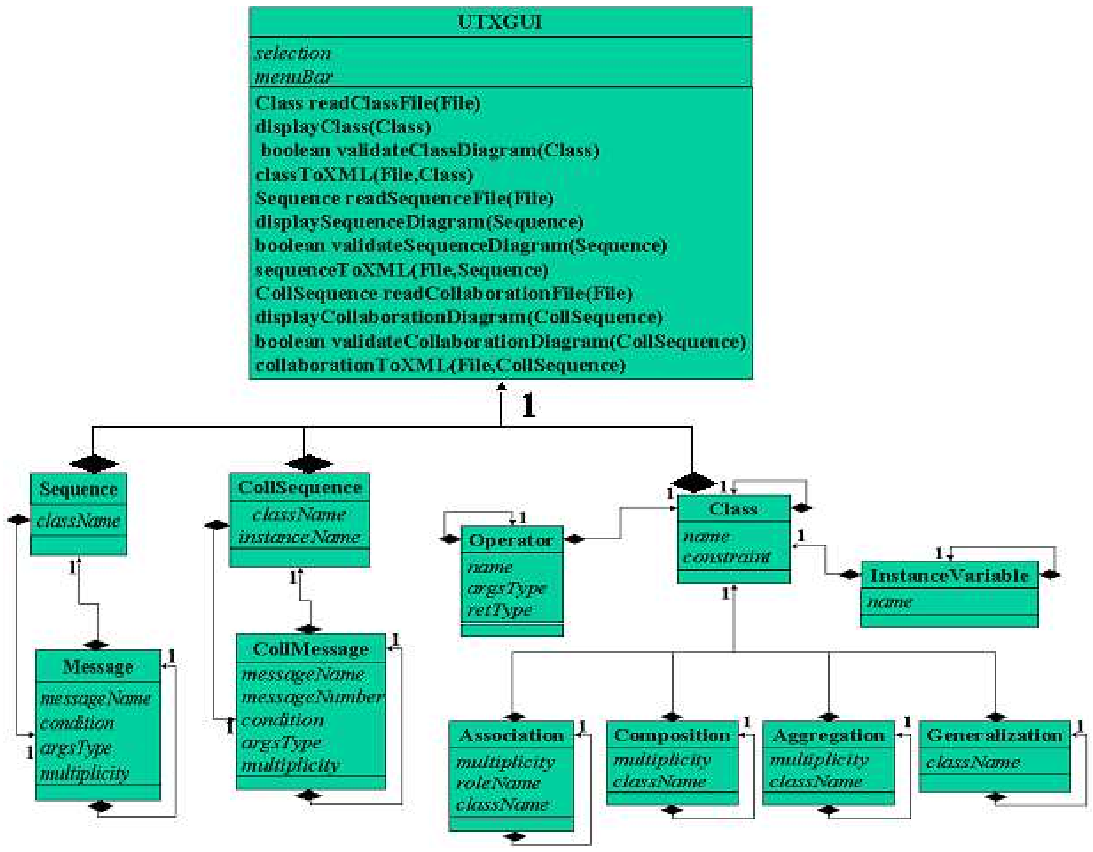
A collaboration diagram is much of the same, except that it may have the instance names, and a sequence number for the messages. The figure next to the sequence diagram is having the class structure for the collaboration diagram. The class 'CollSequence' class, which is an entity class, will be representing the collaboration diagram. It will also be having a multilinked structure.



Finally, we will have to have a boundary class as well. A boundary class models the interactions between the user and the system. It is typically an user interface class. Our boundary class is UTXGUI. It will be having the menuBar, and the selected object (class, sequence, and collaboration diagram). It will also be taking care of all the responsibilities of doing the tasks like reading, validating, and converting the diagrams (class, sequence, and collaboration). This class will be providing access to each and every function that is to be performed by the system. The menuBar object of MenuBar will be implementing the interface menu for providing access to these operators. So, all these operators need to have private access specifier.

The UTXGUI class will be having objects of Class, Sequence, and CollSequence classes to store the class diagram, sequence diagram, and collaboration diagram. The will be processed as and when required by the corresponding user action. Hence, now on we can finally present the final and complete class diagram structure. The figure on the next page specifies the whole static structure of our system.

We have no control class, but the boundary class UTXGUI itself takes care of the some of the controls like menu management, event handling etc.


UTXGUI
selection
menuBar
Class readClassFile(File)
displayClass(Class)
boolean validateClassDiagram(Class)
classToXML(File,Class)
Sequence readSequenceFile(File)
displaySequenceDiagram(Sequence)
boolean validateSequenceDiagram(Sequence)
sequenceToXML(File,Sequence)
CollSequence readCollaborationFile(File)
displayCollaborationDiagram(CollSequence)
boolean validateCollaborationDiagram(CollSequence)
collaborationToXML(File,CollSequence)
1
Sequence
className
Message
messageName
condition
argsType
multiplicity
CollSequence
className
instanceName
CollMessage
messageName
messageNumber
condition
argsType
multiplicity
Operator
name
argsType
retType
Class
name
constraint
InstanceVariable
name
Association
multiplicity
roleName
className
Composition
multiplicity
className
Aggregation
multiplicity
className
Generalization
className
1

CHAPTER 4

IMPLEMENTATION

This chapter discusses the implementation of our automated system designed in the previous chapter. After the designing the system, we must implement it to realize it into a software product. This will ensure that the mapping scheme we have proposed and developed is working in the practical application domains. Thus the implementation of the design presented in previous chapter realizes a system which automates the mappings from UML diagrams to XML documents as per our mapping scheme.

The mapping scheme followed has necessarily been as presented in the chapter 2. It uses the XML Schema Definition document presented in the same chapter. The system will be taking in the input a file having the structure of the input structure, and will produce in output an XML Schema document referencing the prescribed schema definitions. The automation ensures that we can automate the whole process, thus avoiding transferring of large data in the form of UML diagrams over the networks.

We have developed the system using Java, Java Foundation Classes (The Java “Swing” Technology. We have also used the object oriented system analysis and design. The first section of this chapter concentrates on an introduction to these technologies and tools.

The second section will explain the program logic and the method of the mappings from the input to the output.

The third section explains the user interface and its functions.

The fourth sections emphasizes on the output screens of the developed system. It will explain the outputs produced by the system.

4.1 Tools and Technologies Used

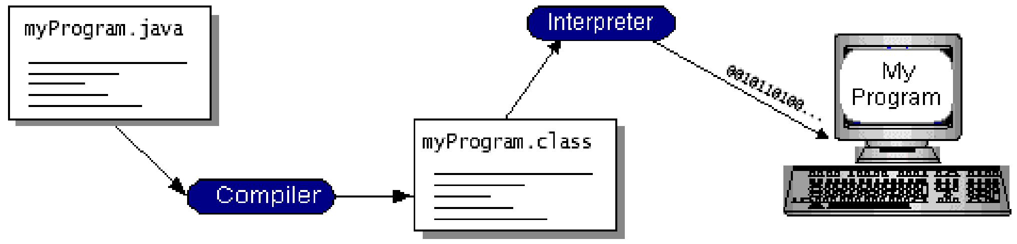
This section explains the technologies used in the implementation of the system developed for the automation of the mapping from UML Diagrams to XML documents. Since UML and XML are already explained elsewhere, we have avoided explaining these right here. The next sections explain Java, Java Swing, and OOA&D.

4.1.1 Java

The Java programming language is a high-level language that has following characteristics:

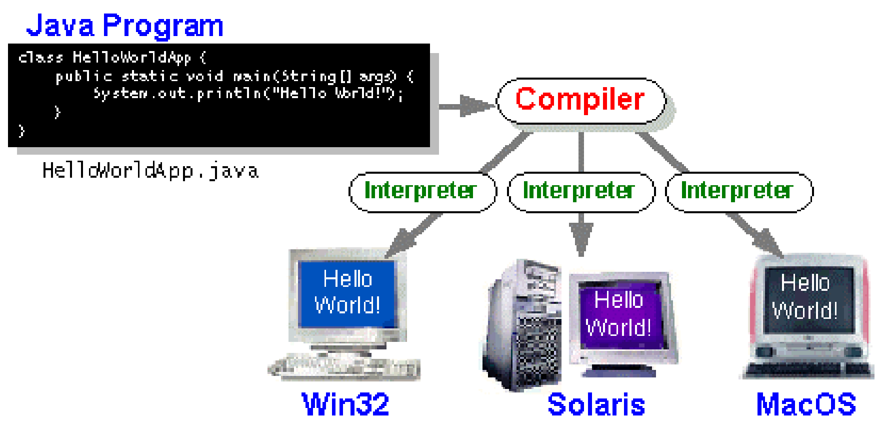
- Simple
- Object oriented
- Distributed
- Interpreted
- Robust
- Secure
- Architecture neutral
- Portable
- High performance
- Multithreaded
- Dynamic

With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. With the compiler, first you translate a program into an intermediate language called *Java bytecodes* —the platform-independent codes interpreted by the interpreter on the Java platform. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed. The following figure illustrates how this works.



You can think of Java bytecodes as the machine code instructions for the *Java Virtual Machine* (Java VM). Every Java interpreter, whether it's a development tool or a Web browser that can run applets, is an implementation of the Java VM.

Java bytecodes help make "write once, run anywhere" possible. You can compile your program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java VM. That means that as long as a computer has a Java VM, the same program written in the Java programming language can run on Windows 2000, a Solaris workstation, or on an iMac.



```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

The Java Platform

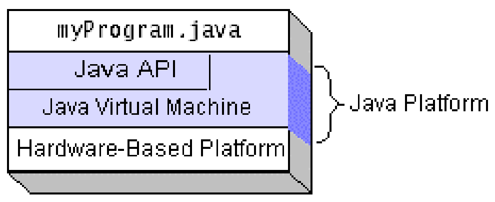
A *platform* is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Windows 2000, Linux, Solaris, and MacOS. Most platforms can be described as a combination of the operating system and hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

- The *Java Virtual Machine* (Java VM)
- The *Java Application Programming Interface* (Java API)

You've already been introduced to the Java VM. It's the base for the Java platform and is ported onto various hardware-based platforms.

The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.



The figure depicts a program that's running on the Java platform. As the figure shows, the Java API and the virtual machine insulate the program from the hardware.

Native code is code that after you compile it, the compiled code runs on a specific hardware platform. As a platform-independent environment, the Java platform can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time bytecode compilers can bring performance close to that of native code without threatening portability.

4.1.2 JAVA SWING AND JFC

JFC is short for Java^{TM} Foundation Classes, which encompass a group of features to help people build graphical user interfaces (GUIs). The JFC was first announced at the 1997 JavaOne developer conference and is defined as containing the following features:

The Swing Components

Include everything from buttons to split panes to tables.

Pluggable Look and Feel Support

Gives any program that uses Swing components a choice of looks and feels. For example, the same program can use either the Java^{TM} look and feel or the Windows look and feel. We expect many more look-and-feel packages -- including some that use sound instead of a visual "look" -- to become available from various sources.

Accessibility API

Enables assistive technologies such as screen readers and Braille displays to get information from the user interface.

Java 2D^{TM} API (Java 2 Platform only)

Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and in applets.

Drag and Drop Support (Java 2 Platform only)

Provides the ability to drag and drop between a Java application and a native application.

The first three JFC features were implemented without any native code, relying only on the API defined in JDK 1.1. As a result, they could and did become available as an extension to JDK 1.1. This extension was released as JFC 1.1, which is sometimes called "the Swing release." The API in JFC 1.1 is often called "the Swing API."

Note: "Swing" was the codename of the project that developed the new components. Although it's an unofficial name, it's frequently used to refer to the new components and related API. It's immortalized in the package names for the Swing API, which begin with javax.swing.

4.2 Program Structure

This section explains the logic of the system implemented. Apart from the structure of all the classes used we will be presenting the way we have implemented the various mechanics of our system. Although the core logic remains as discussed in the previous chapters, we have altered only the technical aspects as and when required. Since we can easily convert the UML diagrams into a feasible system in Java. So, most of it was done pretty easily.

The following pseudo codes present the structure of the classes of the class diagram in the way they were implemented. The code of the methods are avoided in order to save the space. Also since all the classes except for the UTXGUI class are entity classes, so everything is self explanatory for these class definitions.

The class ‘Class’ is used to represent the class diagram using the classes InstanceVariable, Operator, Association, Composition, Aggregation, and Generalization.

Class
```
public class Class
{
 String name;
 InstanceVariable instanceVariable;
 Operator operator;
 String constraint;
 Association association;
 Composition composition;
 Aggregation aggregation;
 Generalization generalization;
 Class nextClass;

 public Class()
 {
 // constructor
 }
}
```

InstanceVariable
```
public class InstanceVariable
{
 String name;
 InstanceVariable nextInstanceVariable;

 public InstanceVariable()
 {
  // constructor
 }
}
```

Operator
```
public class Operator
{
 String name;
 String argsType;
 String retType;
 Operator nextOperator;

 public Operator()
 {
  // constructor
 }
}
```

Association

```
public class Association
{
 String multiplicity;
 String roleName;
 String className;
 Association nextAssociation;

 public Association()
 {
  // constructor
 }
}
```

Composition

```
public class Composition
{
 String multiplicity;
 String className;
 Composition nextComposition;

 public Composition()
 {
  // constructor
 }
}
```

Aggregation

```
public class Aggregation
{
 String multiplicity;
 String className;
 Aggregation nextAggregation;

 public Aggregation()
 {
  // constructor
 }
}
```

Generalization

```
public class Generalization
{
 String className;
 Generalization nextGeneralization;
 public Generalization()
 {
  // constructor
 }
}
```

The class ‘Sequence’ will represent the sequence diagram, using the ‘Message’ class as well.

Sequence

```
Public class Sequence
{
 String className;
 Message message;

 public Sequence()
 {
  // constructor
 }
}
```

Message

```
Public class Message
{
 String messageName;
 String condition;
 String argsType;
 String multiplicity;
 Sequence sequence;
 Message nextMessage;


 public Message()
 {
  // constructor
 }
}
```

The class ‘CollSequence’ will represent a collaboration diagram, using the class ‘CollMessage’.

CollSequence

```
Public class CollSequence
{
 String className;
 String instanceName;
 CollMessage collMessage;

 public CollSequence()
 {
  // constructor
 }
}
```

CollMessage

```
Public class CollMessage
{
 String messageName;
 String messageNumber;
 String condition;
 String argsType;
 String multiplicity;
 CollSequence collSequence;
 CollMessage nextMessage;


 public CollMessage()
 {
  // constructor
 }
}
```

The class 'UTXGUI' will be presenting user interface elements and their functionality. It inherits the class JFrame of the JFC, which helps in managing the desktop windows more conveniently. Various menu items are used to manage the user input to the system. We have only presented the methods, which the user interface accesses directly. The program logic involves more functions. All of the functions are declared private as these are to be accessed by the function managing the menubar event processing, *actionPerformed*.

UTXGUI

```
public class UTXGUI extends JFrame
{
 JTextArea log, input, output;
 JScrollPane logPane, inputPane, outputPane;
 JTabbedPane tabbedPane;
 String newline = "\n", selection="";
 Class classDiagram;
 CollSequence collaborationDiagram;
 Sequence sequenceDiagram;
 JMenuBar menuBar;
 JMenu menu1, menu2, menu3;
 JMenuItem menuItem11, menuItem12, menuItem13, menuItem14, menuItem15, menuItem21,
menuItem31, menuItem32, menuItem33;

// method to handle the menu events.
 public void actionPerformed(ActionEvent e)

// method to convert a class diagram to XML document.
 private void classToXML(String fileName, Class c)

// method to convert a sequence diagram to XML document.
 private void sequenceToXML(Sequence ss, String fileName)

// method to convert a collaboration diagram to XML document
 private void collaborationToXML(CollSequence ss, String fileName)

// method to read a class file
 private Class readClassFile(String fileName)
// method to display a class diagram in the input pane
 private void displayClass(Class c)

// method to validate a class diagram: valid=true
 private boolean validateClassDiagram(Class c)

// method to read a collaboration diagram input file
 private CollSequence readCollaborationFile(String fileName)

// method to display a collaboration diagram in the input pane
 private void displayCollaborationDiagram(CollSequence ss)

// method to validate a collaboration diagram: valid=true
 private boolean validateCollaborationDiagram(CollSequence ss, Class c)

// method to read a sequence diagram input file
 private Sequence readSequenceFile(String fileName)

// method to display a sequence diagram in the input pane
 private void displaySequenceDiagram(Sequence ss)

// method to validate a sequence diagram: valid=true
 private boolean validateSequenceDiagram(Sequence ss, Class c)
}
```

CHAPTER 5

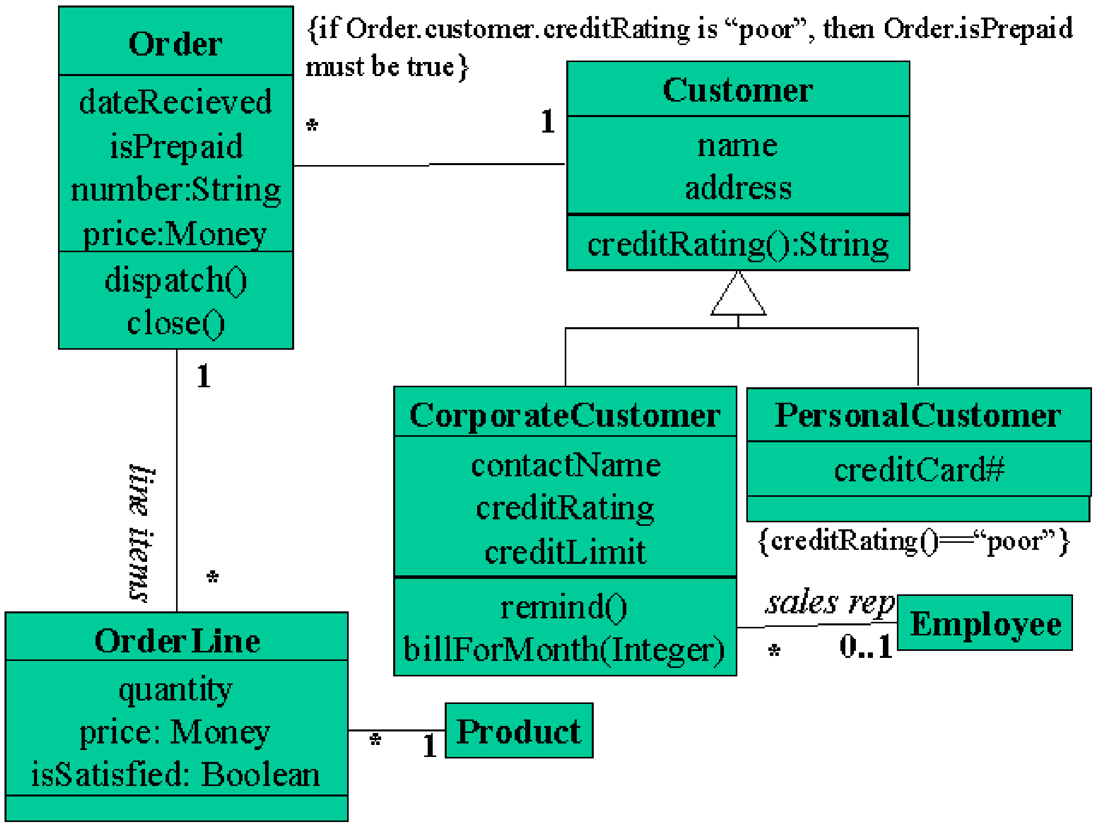
DEMONSTRATION

This chapter is dedicated to the demonstration of the developed system. In order to explain the use of system we must explain each and every step involved in the demonstration. This also ensures that the system gives out proper output, provided the input design is followed.

5.1 CLASS DIAGRAM

Step I:

We must have a class diagram to be converted to XML document. This class diagram must be in the format as per the specification of the Input Design of Chapter 3. In the example run we have taken the following class diagram.



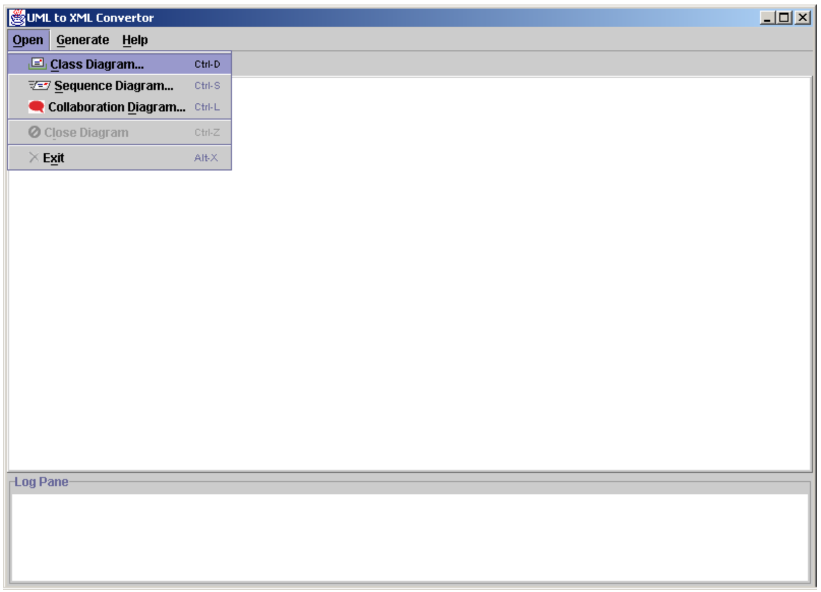
The equivalent input file for this class diagram is as given below:

Order; 4; dateRecieved; isPrepaid; number:String; price:Money; 2; dispatch;;; close;;; {if Order.customer.creditRating is "poor", then Order.isPrepaid must be true}; 2; many;;Customer; one;;OrderLine; 0; 0; 0;
Customer; 2; name; address; 1; creditRating;;String; ; 1; one;;Order; 0; 0; 0;

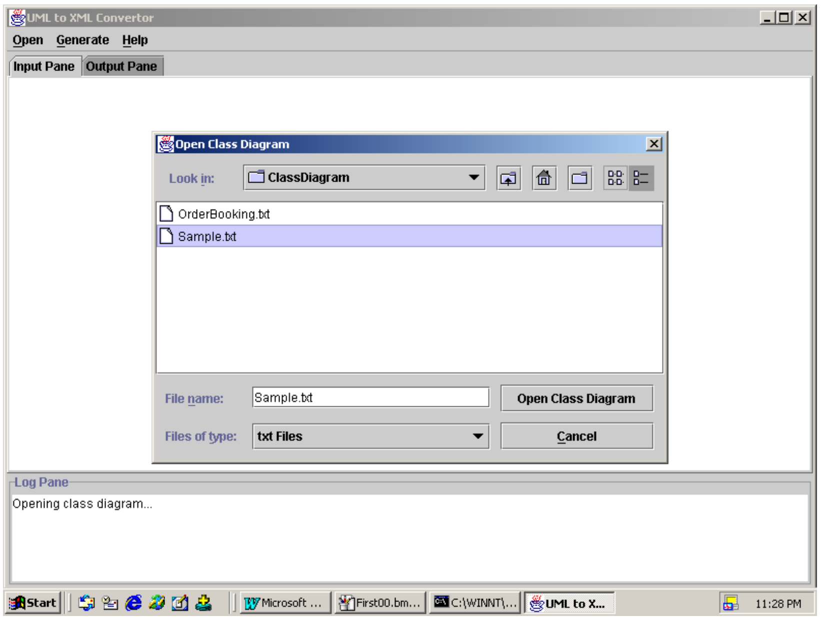
OrderLine; 3; quantity:Integer; price:Money; isSatisfied:Boolean; 0; ; 2; many;line items;Order; many;;Product; 0; 0; 0;
Product; 0; 0; ; 1; one;;OrderLine; 0; 0; 0;
CorporateCustomer; 3; contactName; creditRating; creditLimit; 2; remind;;; billForMonth;Integer;; ; 1; many;;Employee; 0; 0; 1; Customer;
PersonalCustomer; 1; creditCard#; 0; {creditRating()=="poor"}; 0; 0; 0; 1; Customer;
Employee; 0; 0; ; 1; Optional; sales rep;CorporateCustomer; 0; 0; 0;

Step II: (Open -> Class Diagram…)

After having the above class diagram input structure we can run our system and choose the *Class Diagram* option from the *Open* menu. The system will now ask for the location of the input file structure using a file chooser dialog.

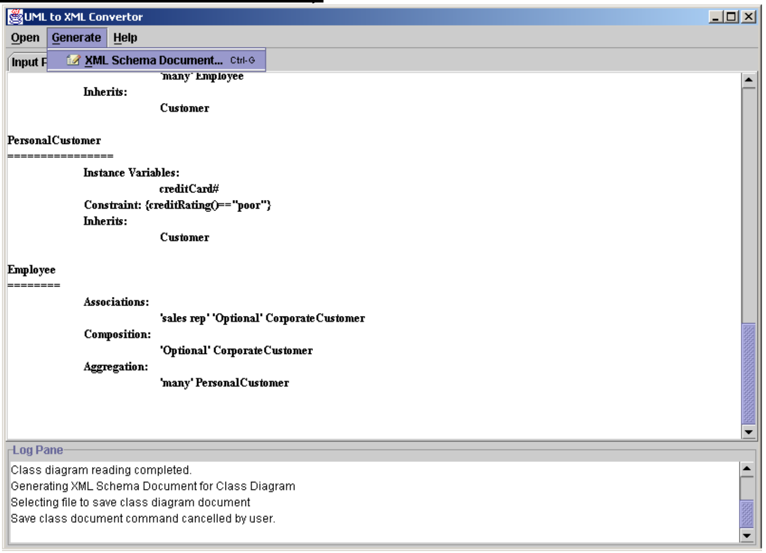


This will result in the opening of the class diagram and the creation of the class diagram structure, which will be displayed in the Input Pane. The system will check the input class file for validity and display whether it is valid or not in the Log Pane.



Opening class diagram...

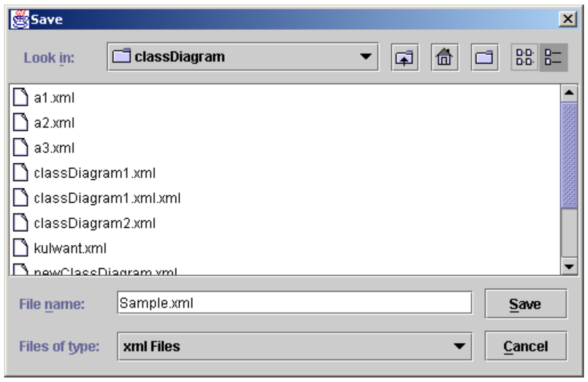
<u>Step III: (Generate -> XML Schema Document…)</u>

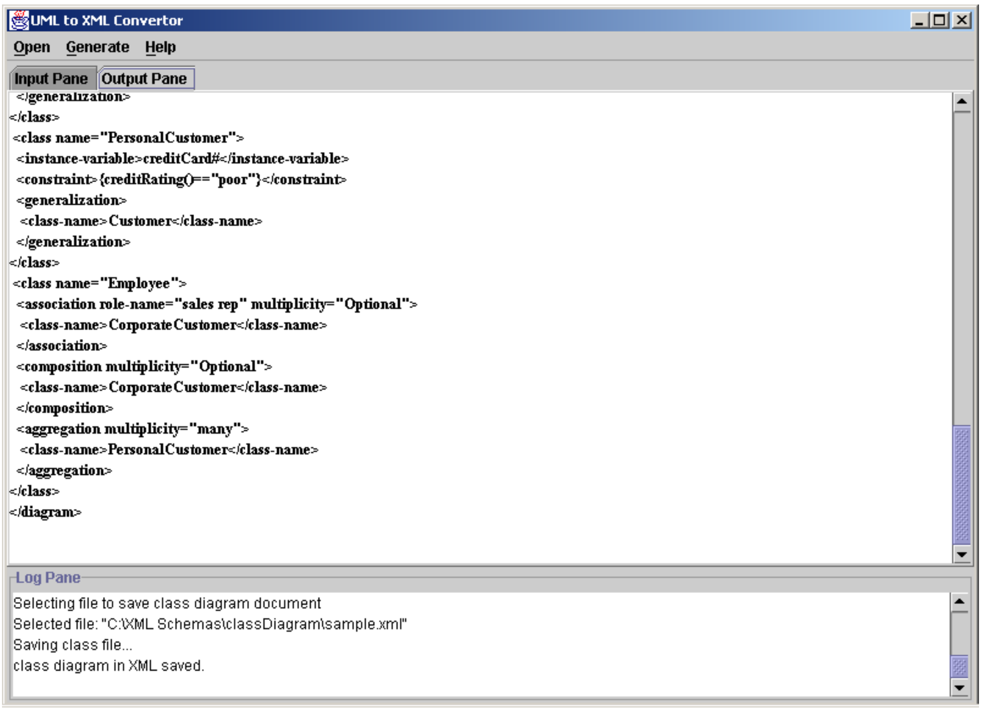


```
                    'many' Employee
        Inherits:
                    Customer

PersonalCustomer
================
        Instance Variables:
                    creditCard#
        Constraint: {creditRating()=="poor"}
        Inherits:
                    Customer

Employee
========
        Associations:
                    'sales rep' 'Optional' CorporateCustomer
        Composition:
                    'Optional' CorporateCustomer
        Aggregation:
                    'many' PersonalCustomer
```

```
Class diagram reading completed.
Generating XML Schema Document for Class Diagram
Selecting file to save class diagram document
Save class document command cancelled by user.
```

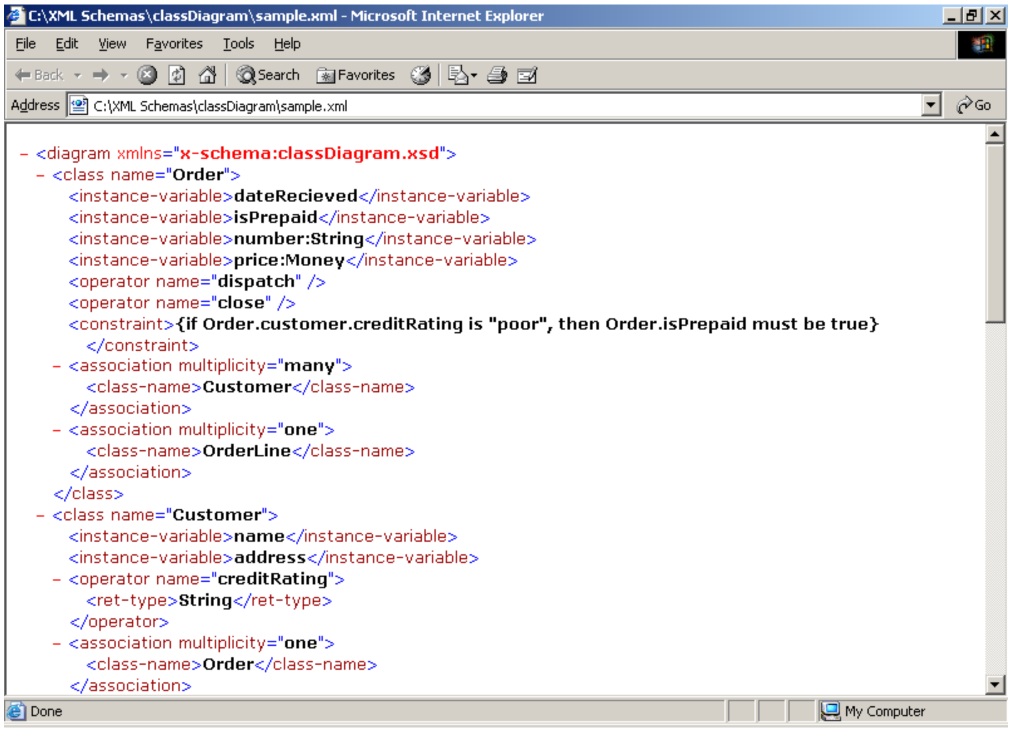
Now we can generate the class diagram XML document by choosing *XML Schema Document* from the *Generate* menu. This will ask for the location for the saving of the generated document. This location must be having the XML Schema Definition document, so that the document we generated can reference it for its structure. This will result in the generation of the XML Schema Document for the class diagram input. This will be displayed in the Output Pane as well. Also, we can view it in any Internet browser supporting XML Schemas.





```
  </generalization>
</class>
 <class name="PersonalCustomer">
  <instance-variable>creditCard#</instance-variable>
  <constraint>{creditRating()=="poor"}</constraint>
  <generalization>
   <class-name>Customer</class-name>
  </generalization>
</class>
 <class name="Employee">
  <association role-name="sales rep" multiplicity="Optional">
   <class-name>CorporateCustomer</class-name>
  </association>
  <composition multiplicity="Optional">
   <class-name>CorporateCustomer</class-name>
  </composition>
  <aggregation multiplicity="many">
   <class-name>PersonalCustomer</class-name>
  </aggregation>
</class>
</diagram>
```

```
Selecting file to save class diagram document
Selected file: "C:\XML Schemas\classDiagram\sample.xml"
Saving class file...
class diagram in XML saved.
```


C:\XML Schemas\classDiagram\sample.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XML Schemas\classDiagram\sample.xml
Go
Done
My Computer

```
- <diagram xmlns="x-schema:classDiagram.xsd">
 - <class name="Order">
     <instance-variable>dateRecieved</instance-variable>
     <instance-variable>isPrepaid</instance-variable>
     <instance-variable>number:String</instance-variable>
     <instance-variable>price:Money</instance-variable>
     <operator name="dispatch" />
     <operator name="close" />
     <constraint>{if Order.customer.creditRating is "poor", then Order.isPrepaid must be true}
       </constraint>
   - <association multiplicity="many">
       <class-name>Customer</class-name>
     </association>
   - <association multiplicity="one">
       <class-name>OrderLine</class-name>
     </association>
   </class>
 - <class name="Customer">
     <instance-variable>name</instance-variable>
     <instance-variable>address</instance-variable>
   - <operator name="creditRating">
       <ret-type>String</ret-type>
     </operator>
   - <association multiplicity="one">
       <class-name>Order</class-name>
     </association>
```

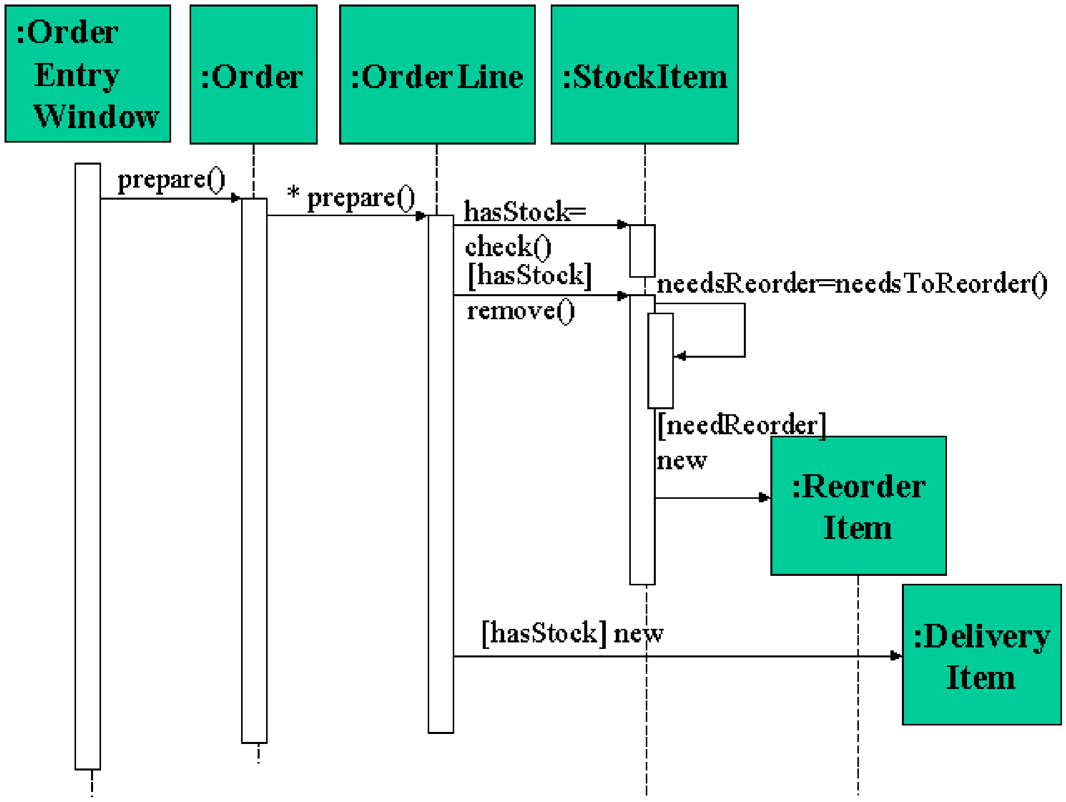
The output file generated is given as under:

```
<diagram xmlns="x-schema:classDiagram.xsd">
 <class name="Order">
  <instance-variable>dateRecieved</instance-variable>
  <instance-variable>isPrepaid</instance-variable>
  <instance-variable>number:String</instance-variable>
  <instance-variable>price:Money</instance-variable>
  <operator name="dispatch" />
  <operator name="close" />
  <constraint>{if Order.customer.creditRating is "poor" then Order.isPrepaid must be
"true"}</constraint>
  <association multiplicity="many">
   <class-name>Customer</class-name>
  </association>
  <association multiplicity="one">
   <class-name>OrderLine</class-name>
  </association>
 </class>

 <class name="Customer">
  <instance-variable>name</instance-variable>
  <instance-variable>address</instance-variable>
  <operator name="creditRating">
   <ret-type>String</ret-type>
  </operator>
  <association multiplicity="one">
   <class-name>Order</class-name>
  </association>
 </class>

 <class name="OrderLine">
  <instance-variable>quantity:Integer</instance-variable>
  <instance-variable>price:Money</instance-variable>
  <instance-variable>isSatisfied:Boolean</instance-variable>
  <association multiplicity="many" role-name="line items">
   <class-name>Order</class-name>
  </association>
  <association multiplicity="many">
   <class-name>Product</class-name>
  </association>
 </class>

 <class name="CorporateCustomer">
  <instance-variable>contactName</instance-variable>
  <instance-variable>creditRating</instance-variable>
  <instance-variable>creditLimit</instance-variable>
  <operator name="remind" />
  <operator name="billForMonth">
   <args-type>Integer</args-type>
  </operator>
  <association multiplicity="many">
   <class-name>Employee</class-name>
  </association>
  <generalization>
   <class-name>Customer</class-name>
  </generalization>
 </class>

 <class name="PersonalCustomer">
  <instance-variable>creditCard#</instance-variable>
  <constraint>{creditRating() == "poor"}</constraint>
  <generalization>
   <class-name>Customer</class-name>
  </generalization>
 </class>

 <class name="Employee">
  <association multiplicity="optional" role-name="sales rep">
   <class-name>CorporateCustomer</class-name>
  </association>
 </class>

 <class name="Product">
  <association multiplicity="one">
   <class-name>OrderLine</class-name>
  </association>
 </class>
</diagram>
```

5.2 Sequence Diagram

Step I:

We must have a sequence diagram to be converted to XML document. This sequence diagram must be in the format as per the specification of the Input Design of Chapter 3. In the example run we have taken the following sequence diagram.



The equivalent input file for this sequence diagram is given below.

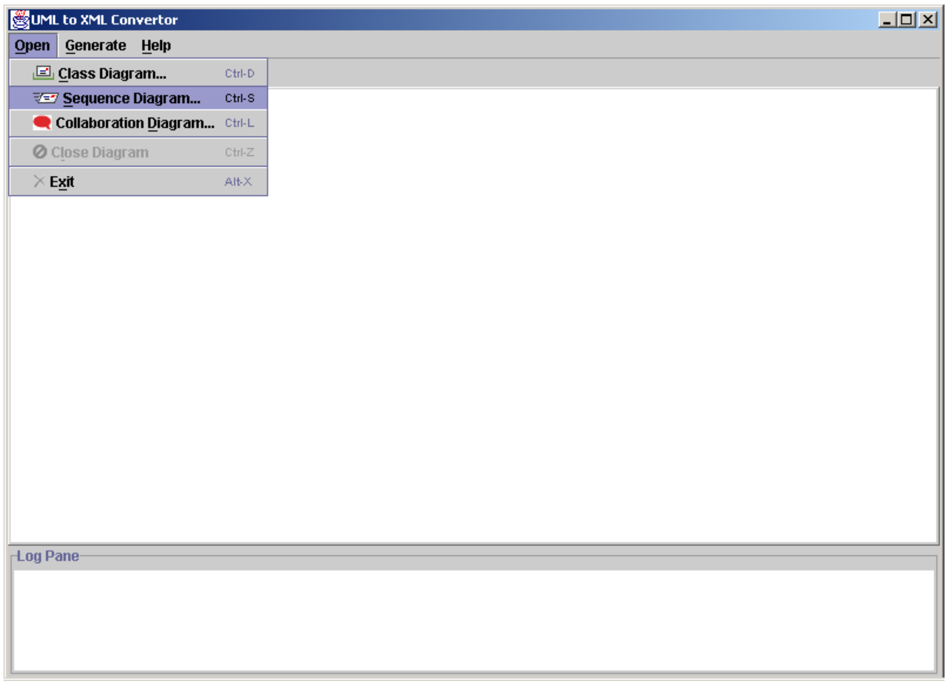
```
OrderEntryWindow;1;
 Prepare;;;;
 Order;1;
  Prepare;;;many;
  OrderLine;3;
   CheckStock;;retval=hasStock;;
   StockItem;0;
   Remove;hasStock;;;
   StockItem;2;
    NeedToReorder;;retval=needsReorder;;
    StockItem;0;
    New;needsReorder;;;
    ReorderItem;0;
   New;hasStock;;;
   DeliveryItem;0;
```

The class diagram structure used to validate this sequence diagram is as given below:

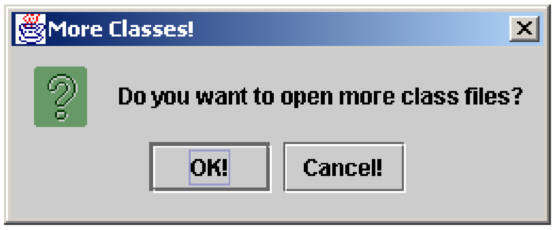
Order;4;dateRecieved;isPrepaid;number:String;price:Money;3;dispatch;;;close;;;prepare;;;{if Order.customer.creditRating is "poor", then Order.isPrepaid must be true};2;many;;Customer;one;;OrderLine;0;0;0;
Customer;2;name;address;1;creditRating;;String;;1;one;;Order;0;0;0;
OrderLine;4;quantity:Integer;price:Money;isSatisfied:Boolean;hasStock;2;checkStock;;;remove;;;;2;many;line items;Order;many;;Product;0;0;0;
Product;0;0;;1;one;;OrderLine;0;0;0;
CorporateCustomer;3;contactName;creditRating;creditLimit;2;remind;;;billForMonth;Integer;;;1;many;;Employee;0;0;1;Customer;
PersonalCustomer;1;creditCard#;0;{creditRating()=="poor"};0;0;0;1;Customer;
Employee;0;0;;1;Optional;sales rep;CorporateCustomer;0;0;0;
StockItem;1;needsReorder:Boolean;1;needToReorder;;;;0;0;0;0;
ReorderItem;0;0;;0;0;0;0;
DeliveryItem;0;0;;0;0;0;0;
OrderEntryWindow;0;1;prepare;;;;0;0;0;0;

Step II: (Open -> Sequence Diagram…)

Now we can run our system to convert the above specified sequence diagram to XML document by choosing *Sequence Diagram* from the *Open* menu.

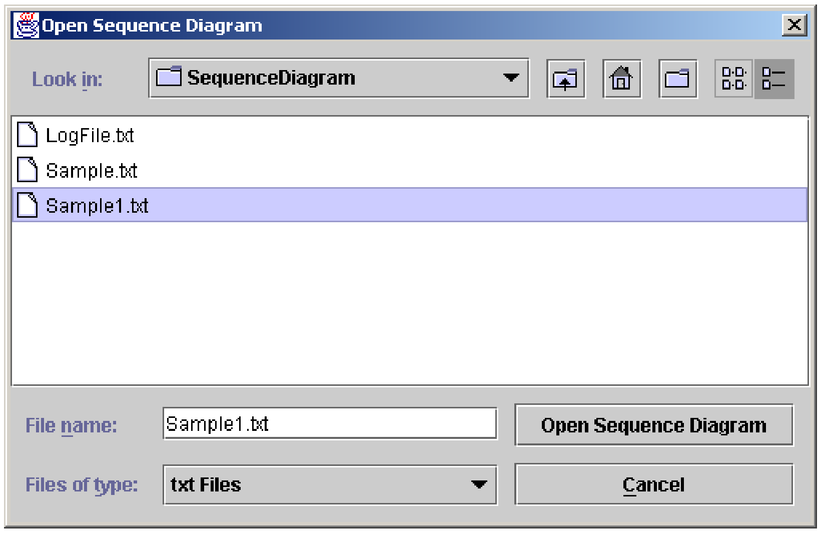


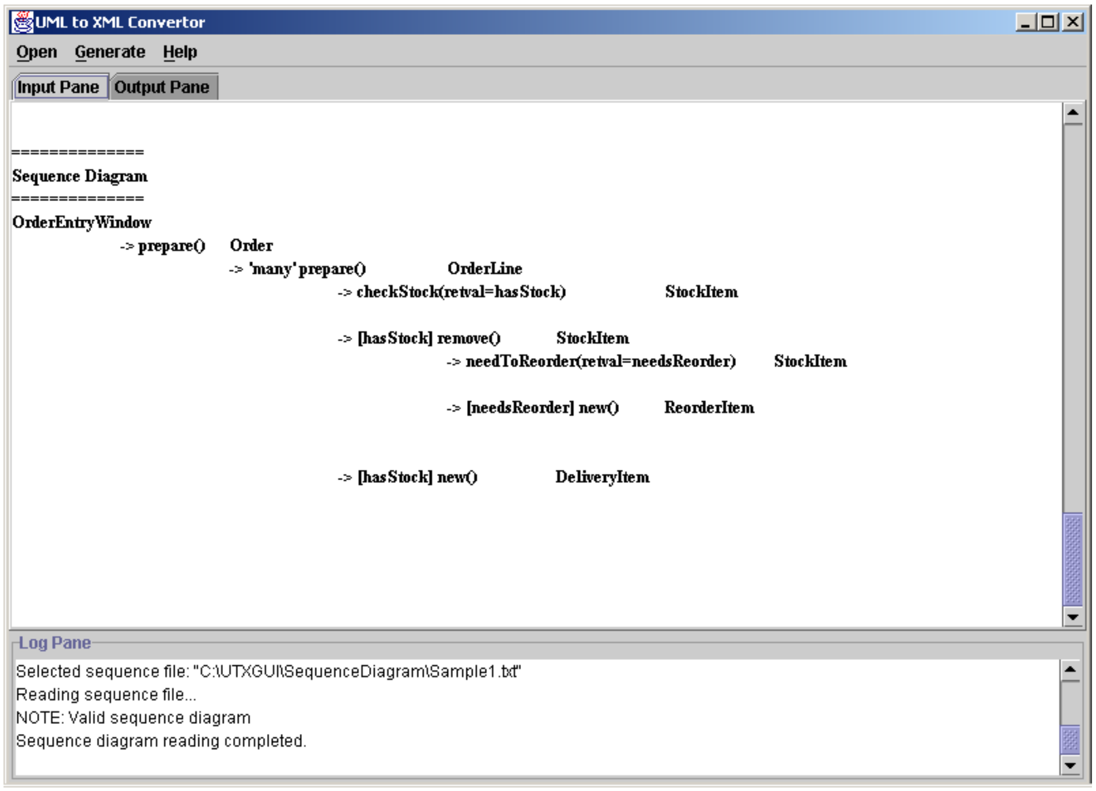
The system will now ask for the class diagram that is to be used to validate the sequence diagram. We can choose that using the file chooser dialog box. The system will now ask whether we want to use another class diagram to validate the class diagram,



we can do so if we have to use more than one class diagrams to validate a single sequence diagram.

Now the system will ask for the input file for the sequence diagram structure. We can choose it using the file chooser dialog. The system will validate the input sequence diagram and display the structure of the class diagram(s) and sequence diagram in the input pane. It will display the status of all the steps including validity in the log pane of the system.



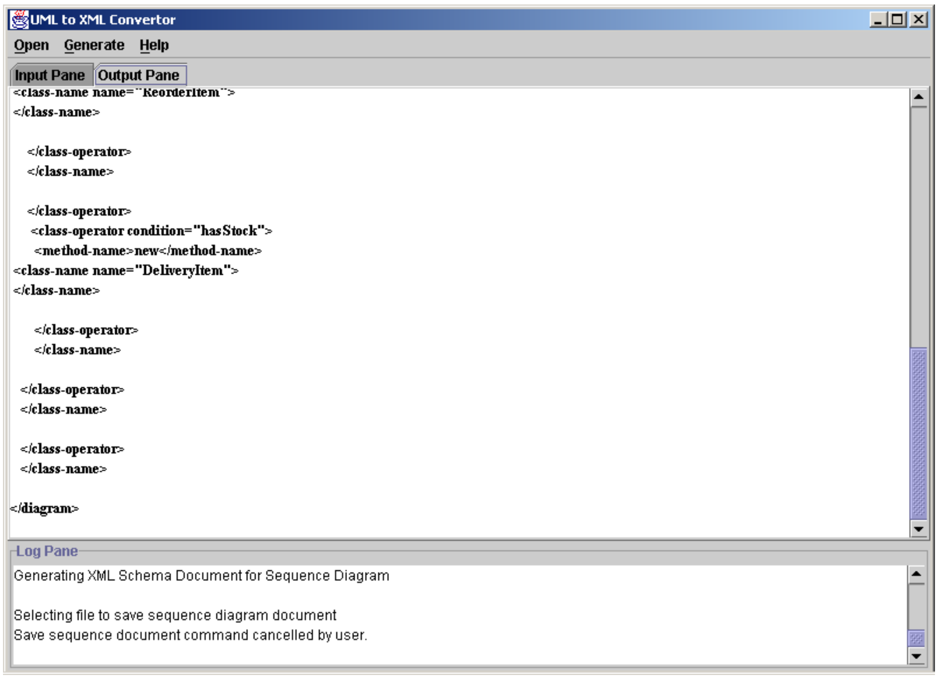


```
==============
Sequence Diagram
==============
OrderEntryWindow
              -> prepare()    Order
                              -> 'many'prepare()            OrderLine
                                                            -> checkStock(retval=hasStock)            StockItem

                                                            -> [hasStock] remove()         StockItem
                                                                                           -> needToReorder(retval=needsReorder)      StockItem

                                                                                           -> [needsReorder] new()        ReorderItem


                                                            -> [hasStock] new()            DeliveryItem
```

```
Selected sequence file: "C:\UTXGUI\SequenceDiagram\Sample1.txt"
Reading sequence file...
NOTE: Valid sequence diagram
Sequence diagram reading completed.
```

Step III: (Generate -> XML Schema Document…)

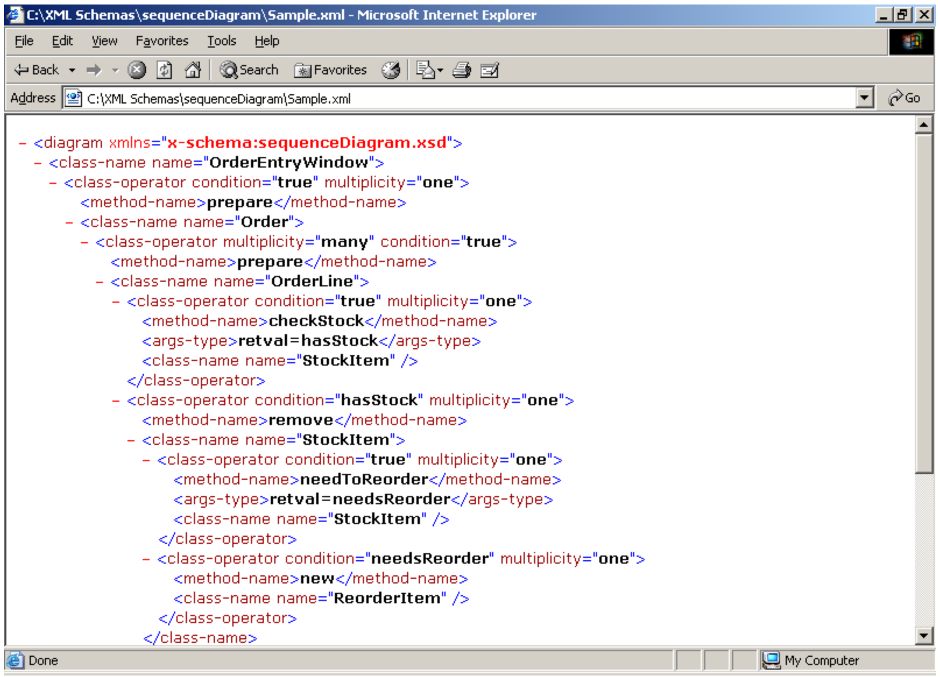
Now we can generate the XML document for the given sequence diagram. For this we will choose *XML Schema Document* option from the *Generate* menu. It will ask for the location of the output XML document, which we can specify using the file chooser dialog box. It should be stored at the same location where we have saved the XML Schema Definition for sequence diagram, so that it can reference the XML Schema to get its document structure. This output will be displayed in the output pane of the system. The output file can be viewed in any Internet browser supporting XML Schemas.



```
<class-name name="ReorderItem">
</class-name>

   </class-operator>
   </class-name>

   </class-operator>
    <class-operator condition="hasStock">
     <method-name>new</method-name>
<class-name name="DeliveryItem">
</class-name>

     </class-operator>
     </class-name>

  </class-operator>
  </class-name>

  </class-operator>
  </class-name>

</diagram>
```

Log Pane

```
Generating XML Schema Document for Sequence Diagram

Selecting file to save sequence diagram document
Save sequence document command cancelled by user.
```


C:\XML Schemas\sequenceDiagram\Sample.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XML Schemas\sequenceDiagram\Sample.xml
Go
Done
My Computer

```
- <diagram xmlns="x-schema:sequenceDiagram.xsd">
  - <class-name name="OrderEntryWindow">
    - <class-operator condition="true" multiplicity="one">
        <method-name>prepare</method-name>
      - <class-name name="Order">
        - <class-operator multiplicity="many" condition="true">
            <method-name>prepare</method-name>
          - <class-name name="OrderLine">
            - <class-operator condition="true" multiplicity="one">
                <method-name>checkStock</method-name>
                <args-type>retval=hasStock</args-type>
                <class-name name="StockItem" />
              </class-operator>
            - <class-operator condition="hasStock" multiplicity="one">
                <method-name>remove</method-name>
              - <class-name name="StockItem">
                - <class-operator condition="true" multiplicity="one">
                    <method-name>needToReorder</method-name>
                    <args-type>retval=needsReorder</args-type>
                    <class-name name="StockItem" />
                  </class-operator>
                - <class-operator condition="needsReorder" multiplicity="one">
                    <method-name>new</method-name>
                    <class-name name="ReorderItem" />
                  </class-operator>
                </class-name>
```

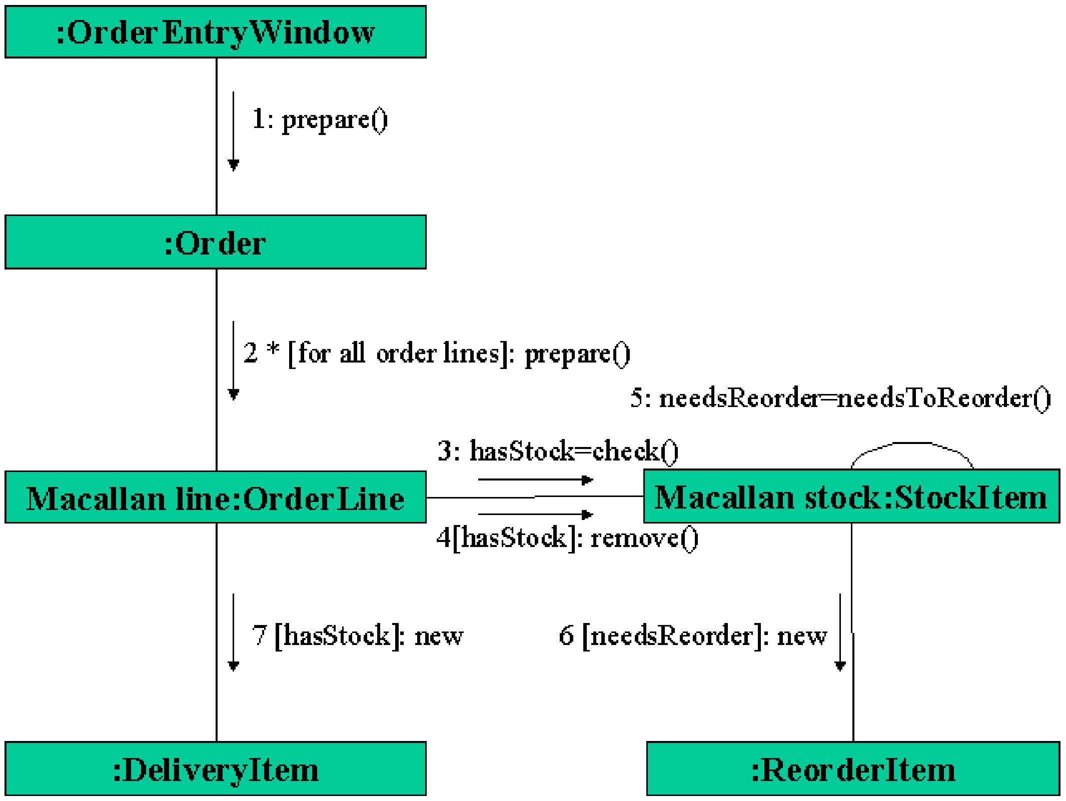
The generated output file is as given under.

```
<diagram xmlns="x-schema:sequenceDiagram.xsd">
 <class-name name="OrderEntryWindow">
  <class-operator condition="true" multiplicity="one">
   <method-name>prepare</method-name>
   <class-name name="Order">
    <class-operator multiplicity="many" condition="true">
     <method-name>prepare</method-name>
     <class-name name="OrderLine">
      <class-operator condition="true" multiplicity="one">
       <method-name>checkStock</method-name>
       <args-type>retval=hasStock</args-type>
       <class-name name="StockItem" />
      </class-operator>
      <class-operator condition="hasStock" multiplicity="one">
       <method-name>remove</method-name>
       <class-name name="StockItem">
        <class-operator condition="true" multiplicity="one">
         <method-name>needToReorder</method-name>
         <args-type>retval=needsReorder</args-type>
         <class-name name="StockItem" />
        </class-operator>
        <class-operator condition="needsReorder" multiplicity="one">
         <method-name>new</method-name>
         <class-name name="ReorderItem" />
        </class-operator>
       </class-name>
      </class-operator>
      <class-operator condition="hasStock" multiplicity="one">
       <method-name>new</method-name>
       <class-name name="DeliveryItem" />
      </class-operator>
     </class-name>
    </class-operator>
   </class-name>
  </class-operator>
 </class-name>
</diagram>
```

5.2 Collaboration Diagram

Step I:

We must have a collaboration diagram to be converted to XML document. This collaboration diagram must be in the format as per the specification of the Input Design of Chapter 3. In the example run we have taken the adjacent collaboration diagram.



The equivalent input file for this collaboration diagram is given below.

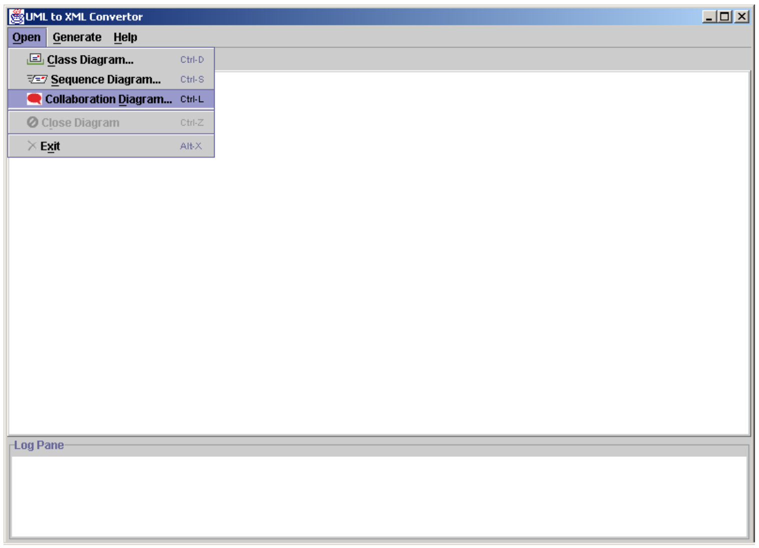
```
OrderEntryWindow;1;
 prepare;1;;;;
 Order;1;
  prepare;2;for all order lines;;many;
  Macallan Line:OrderLine;3;
   checkStock;3;;retval=hasStock;;
   Macallan Stock:StockItem;0;
   remove;4;hasStock;;;
   Macallan Stock:StockItem;2;
    needToReorder;5;;retval=needsReorder;;
    Macallan Stock:StockItem;0;
    new;6;needsReorder;;;
    ReorderItem;0;
   new;7;hasStock;;;
   DeliveryItem;0;
```

The class diagram structure used to validate this collaboration diagram is as given below:

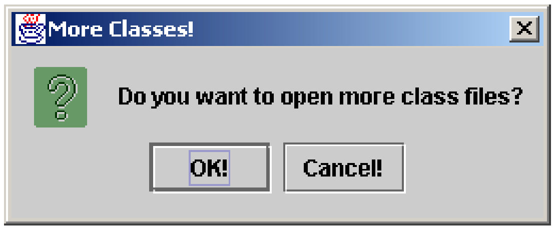
Order;4;dateRecieved;isPrepaid;number:String;price:Money;3;dispatch;;;close;;;prepare;;;{if Order.customer.creditRating is "poor", then Order.isPrepaid must be true};2;many;;Customer;one;;OrderLine;0;0;0;
Customer;2;name;address;1;creditRating;;String;;1;one;;Order;0;0;0;
OrderLine;4;quantity:Integer;price:Money;isSatisfied:Boolean;hasStock;2;checkStock;;;remove;;;;
2;many;line items;Order;many;;Product;0;0;0;
Product;0;0;;1;one;;OrderLine;0;0;0;
CorporateCustomer;3;contactName;creditRating;creditLimit;2;remind;;;billForMonth;Integer;;;1;many;;Employee;0;0;1;Customer;
PersonalCustomer;1;creditCard#;0;{creditRating()=="poor"};0;0;0;1;Customer;
Employee;0;0;;1;Optional;sales rep;CorporateCustomer;0;0;0;
StockItem;1;needsReorder:Boolean;1;needToReorder;;;;0;0;0;0;
ReorderItem;0;0;;0;0;0;0;
DeliveryItem;0;0;;0;0;0;0;
OrderEntryWindow;0;1;prepare;;;;0;0;0;0;

Step II: (Open -> Collaboration Diagram…)

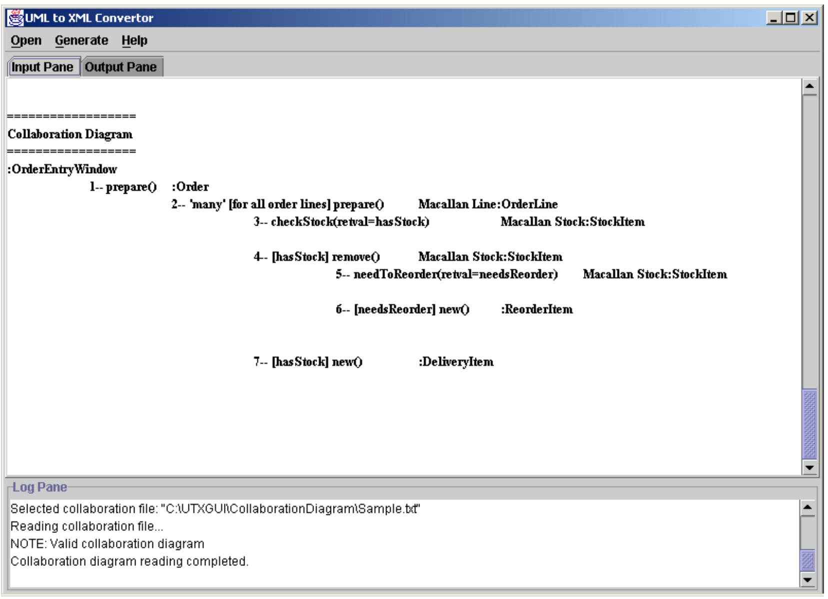
Now we can run our system to convert the above specified collaboration diagram to XML document by choosing Collaboration Diagram from the Open menu.



The system will now ask for the class diagram that is to be used to validate the collaboration diagram. We can choose that using the file chooser dialog box. The system will now ask whether we want to use another class diagram to validate the collaboration diagram, we can do so if we have to use more than one class diagrams to validate a single collaboration diagram.



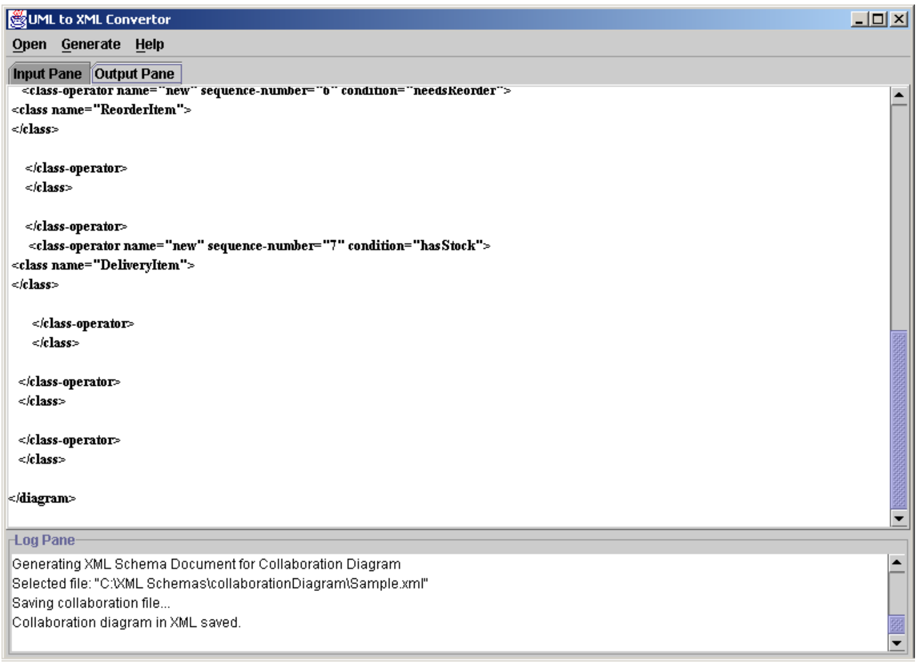
Now the system will ask for the input file for the collaboration diagram structure. We can choose it using the file chooser dialog. The system will validate the input collaboration diagram and display the structure of the class diagram(s) and collaboration diagram in the input pane. It will display the status of all the steps including validity in the log pane of the system.



```
==================
Collaboration Diagram
==================
:OrderEntryWindow
            1-- prepare()   :Order
                        2-- 'many' [for all order lines] prepare()       Macallan Line:OrderLine
                                    3-- checkStock(retval=hasStock)             Macallan Stock:StockItem

                                    4-- [hasStock] remove()        Macallan Stock:StockItem
                                                5-- needToReorder(retval=needsReorder)     Macallan Stock:StockItem

                                                6-- [needsReorder] new()       :ReorderItem


                                    7-- [hasStock] new()           :DeliveryItem
```

```
Selected collaboration file: "C:\UTXGUI\CollaborationDiagram\Sample.txt"
Reading collaboration file...
NOTE: Valid collaboration diagram
Collaboration diagram reading completed.
```

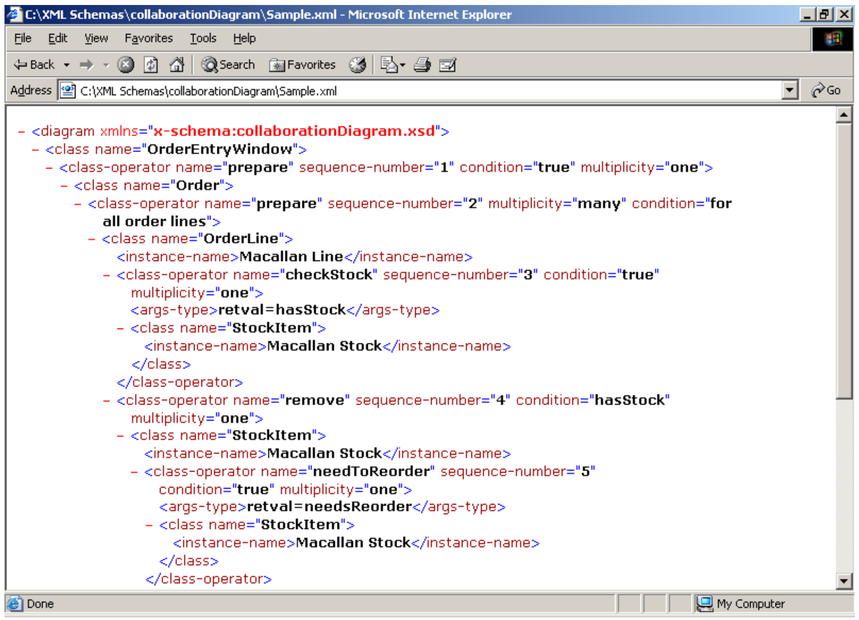
__Step III: (Generate -> XML Schema Document…)__



```
  <class-operator name="new" sequence-number="6" condition="needsReorder">
<class name="ReorderItem">
</class>

   </class-operator>
   </class>

   </class-operator>
    <class-operator name="new" sequence-number="7" condition="hasStock">
<class name="DeliveryItem">
</class>

     </class-operator>
     </class>

  </class-operator>
  </class>

  </class-operator>
  </class>

</diagram>
```

```
Generating XML Schema Document for Collaboration Diagram
Selected file: "C:\XML Schemas\collaborationDiagram\Sample.xml"
Saving collaboration file...
Collaboration diagram in XML saved.
```

Now we can generate the XML document for the given collaboration diagram. For this we will choose XML Schema Document option from the Generate menu. It will ask for the location of the output XML document, which we can specify using the file chooser dialog box. It should be stored at the same location where we have saved the XML Schema Definition for collaboration diagram, so that it can


C:\XML Schemas\collaborationDiagram\Sample.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\XML Schemas\collaborationDiagram\Sample.xml
Go
Done
My Computer

```
- <diagram xmlns="x-schema:collaborationDiagram.xsd">
 - <class name="OrderEntryWindow">
  - <class-operator name="prepare" sequence-number="1" condition="true" multiplicity="one">
   - <class name="Order">
    - <class-operator name="prepare" sequence-number="2" multiplicity="many" condition="for
        all order lines">
     - <class name="OrderLine">
         <instance-name>Macallan Line</instance-name>
       - <class-operator name="checkStock" sequence-number="3" condition="true"
           multiplicity="one">
           <args-type>retval=hasStock</args-type>
         - <class name="StockItem">
             <instance-name>Macallan Stock</instance-name>
           </class>
         </class-operator>
       - <class-operator name="remove" sequence-number="4" condition="hasStock"
           multiplicity="one">
         - <class name="StockItem">
             <instance-name>Macallan Stock</instance-name>
           - <class-operator name="needToReorder" sequence-number="5"
               condition="true" multiplicity="one">
               <args-type>retval=needsReorder</args-type>
             - <class name="StockItem">
                 <instance-name>Macallan Stock</instance-name>
               </class>
             </class-operator>
```

reference the XML Schema to get its document structure. The output file can be viewed in any Internet browser supporting XML Schemas.

The generated output file is as given under.

```
<diagram xmlns="x-schema:collaborationDiagram.xsd">
  <class name="OrderEntryWindow">
   <class-operator name="prepare" sequence-number="1" condition="true" multiplicity="one">
    <class name="Order">
     <class-operator name="prepare" sequence-number="2" multiplicity="many" condition="for all
order lines">
      <class name="OrderLine">
       <instance-name>Macallan Line</instance-name>
       <class-operator   name="checkStock"   sequence-number="3"   condition="true"
multiplicity="one">
        <args-type>retval=hasStock</args-type>
        <class name="StockItem">
         <instance-name>Macallan Stock</instance-name>
        </class>
       </class-operator>
       <class-operator   name="remove"   sequence-number="4"   condition="hasStock"
multiplicity="one">
        <class name="StockItem">
         <instance-name>Macallan Stock</instance-name>
         <class-operator   name="needToReorder"   sequence-number="5"   condition="true"
multiplicity="one">
          <args-type>retval=needsReorder</args-type>
          <class name="StockItem">
           <instance-name>Macallan Stock</instance-name>
          </class>
         </class-operator>
         <class-operator   name="new"   sequence-number="6"   condition="needsReorder"
multiplicity="one">
          <class name="ReorderItem" />
         </class-operator>
        </class>
       </class-operator>
       <class-operator          name="new"          sequence-number="7"          condition="hasStock"
multiplicity="one">
        <class name="DeliveryItem" />
       </class-operator>
      </class>
     </class-operator>
    </class>
   </class-operator>
  </class>
</diagram>
```

CHAPTER 6

CONCLUSION

While dealing with the large volumes of data to be shared by the development team during the software solution development life cycles, it is very important that each and every member gets the data required by him as soon as possible. Although, UML has become the widely accepted standard for modeling the physical as well as software systems, the UML diagrams are not very convenient travelers over the Internet due to their large size. Additionally, with the growth of software industry the application domains have increased rapidly. This has resulted in the increase in size of the UML diagrams for a software system. Hence, the sharing of these diagrams over the networks by large project teams has become more and more difficult. XML, being a standard information representation and sharing language across Internet, can be a better solution for the purpose of sharing it over network. Still UML is better alternative for modeling. So, in order to make the things convenient for such parties who want to share these diagrams over the networks, a better strategy can be to model the systems using UML and then in order to share it over network we can convert the UML diagrams to XML documents.

In this thesis, we have discussed a mapping scheme, which will enable us to map the UML diagrams, like class diagrams, sequence diagrams and collaboration diagrams, to XML documents using XML Schema. For this we have defined an XML Schema Definition document for each of these diagrams. The mapping scheme takes a UML diagram and convert it into an XML document, whose structure can be received by referencing the appropriate XSD. This ensures that we don’t have to generate schema for each and every UML diagrams, which is a more complex task. We have also designed and implemented a software, which will automate the process of mapping UML diagrams to XML documents. The mapping scheme provides the XML Schema definition for three most widely used UML diagrams, namely class diagram, sequence diagram, and collaboration diagram, and presents a mapping methodology to convert these diagrams into XML documents.

6.1 FUTURE WORK

We have worked on the mapping scheme, which is sufficient enough to map the three types of UML diagrams, class diagrams, sequence diagrams, and collaboration diagrams to XML documents. Further, we can extend this work by mapping some other important models of UML like use case model, state chart model, and activity model to XML. Also since XML documents are not very easy to understand, we can also work to model the XML documents representing UML diagrams back to UML.

Finally, after all this work is done, we will be able to share the UML models over the networks in a very convenient way amongst large development parties, thus reducing the resource usage in development process.

BIBLIOGRAPHY

1. CT Arrington; "Enterprise Java with UML"; OMG Press, 2001.
2. David Carlson; "Modeling XML Vocabularies with UML: Part – I"; http://www.xml.com /pub/a/2001/08/22/uml.html
3. David Carlson; "Modeling XML Vocabularies with UML: Part – II"; http://www.xml.com /pub/a/2001/09/19/uml.html
4. David Carlson; "Modeling XML Vocabularies with UML: Part – III"; http://www.xml.com/pub/a/2001/10/10/uml.html.
5. Grady Booch, Magnus Christerson, Mathew Fuchs, Jari Koistinen; "UML for XML Schema Mapping Specification"; http://www.rational.com/media/uml/resources/media /uml_xmlschema33.pdf?SMSESSION=NO
6. I-Chen Wu, Shang-Hsien Hsieh; "An UML-XML-RDB Model Mapping Solution for Facilitating Information Standardization and Sharing in Construction Industry"; http://fire.nist.gov/bfrlpubs/build02/PDF/b02158.pdf.
7. James Rumbaugh, Ivar Jacobson, Grady Booch; "The Unified Modeling Language Reference Manual"; Addison-Wesley.
8. James Rumbaugh, Ivar Jacobson, Grady Booch; "The Unified Modeling Language User Guide"; Addison-Wesley.
9. Kurt Cagle; "XML Developer's Handbook"; BPB Publications, 2000.
10. Martin Fowler, Kendall Scott; "UML Distilled"; Pearson Education, 2001 (2/e).
11. P. Naughton, H. Schildt; "Java2: The Complete Reference"; Tata McGraw Hill.
12. Sandra E. Eddy, BK DeLong; "XML Programming Reference"; IDG Books, 2001(2/e).
13. W3C Recommendation; "XML Schema Part 0: Primer"; W3C, http://www.w3.org /TR/xmlschema-0/.
14. W3C Recommendation; "XML Schema Part 1: Structures"; W3C, http://www.w3.org /TR/xmlschema-1/.
15. W3C Recommendation; "XML Schema Part 2: Datatypes"; W3C, http://www.w3.org /TR/xmlschema-2/.
16. W3C Recommendation; "XML Schema Requirements"; W3C, http://www.w3.org/TR/NOTE-xml-schema-req.
17. W3C; "XML Schema"; http://www.w3c.com.
18. The JavaTM Tutorial; "Trail: Creating a GUI with JFC/Swing (The Swing Tutorial)"; Sun Microsystems, Inc.; http://java.sun.com/docs/books/tutorial/uiswing/.
19. SmartDraw UML Center; "How to draw UML Diagrams"; SmartDraw.com; http://www.smartdraw.com/resources/centers/uml/uml.htm.
20. W3Schools; "XML Schema Tutorials"; W3Schools.com; http://www.w3schools.com/ schema/default.asp.
21. W3Schools; "XML Tutorial"; W3Schools.com; http://www.w3schools.com/xml/ default.asp.
22. John E. Hopcroft and Jeffrey D. Ullman; "Introduction to Automata Theory, Languages, and Computation"; Narosa Publishing House, 1999.