Virtual design office: A collaborative unified modeling language tool

Hara Totapally

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Computer Engineering Commons

Recommended Citation
https://scholarworks.lib.csusb.edu/etd-project/1994
VIRTUAL DESIGN OFFICE, A COLLABORATIVE UNIFIED MODELING LANGUAGE TOOL

A Project
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Hara Totapally
December 2001
VIRTUAL DESIGN OFFICE, A COLLABORATIVE UNIFIED MODELING LANGUAGE TOOL

A Project
Presented to the
Faculty of
California State University,
San Bernardino

by
Hara Totapally
December 2001

Approved by:
Arturo Concepcion, Chair, Computer Science
George Georgiou
Key Zemoudah
ABSTRACT

Real-time conferencing and collaborative computing is a great way to make developers more effective, increase productivity and teamwork, improve decision making, ideal way to break down geographic barriers, enabling technical and creative professionals to collaborate. The benefits of collaborative computing emphasize the need for a collaborative modeling tool.

In this project a collaborative framework was developed comprising of configurable client and server components. The framework was extended to create an interactive and collaborative modeling tool, VIDEO (Virtual Design Office) supporting Unified Modeling Language limited to class diagram, in a collaborative environment. VIDEO client was designed as an applet and its graphical user interface resembled Rational Rose, a commercial modeling tool.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td><strong>CHAPTER ONE: SOFTWARE REQUIREMENT SPECIFICATIONS</strong></td>
<td></td>
</tr>
<tr>
<td>Collaborative Computing</td>
<td>1</td>
</tr>
<tr>
<td>Need for a Collaborative Modeling Tool</td>
<td>2</td>
</tr>
<tr>
<td>Methodology</td>
<td>3</td>
</tr>
<tr>
<td>Central Server</td>
<td>4</td>
</tr>
<tr>
<td>Distributed Client</td>
<td>7</td>
</tr>
<tr>
<td>VIDEO Scope</td>
<td>7</td>
</tr>
<tr>
<td>Definitions, Acronyms, and Abbreviations</td>
<td>11</td>
</tr>
<tr>
<td>Summary</td>
<td>15</td>
</tr>
<tr>
<td><strong>CHAPTER TWO: ARCHITECTURE</strong></td>
<td></td>
</tr>
<tr>
<td>Overview</td>
<td>16</td>
</tr>
<tr>
<td>Architecture</td>
<td>16</td>
</tr>
<tr>
<td>VIDEO Packages</td>
<td>19</td>
</tr>
<tr>
<td>Classes Common to Client and Server</td>
<td>24</td>
</tr>
<tr>
<td>Registry</td>
<td>24</td>
</tr>
<tr>
<td>LockRegistry</td>
<td>25</td>
</tr>
<tr>
<td>Attribute</td>
<td>26</td>
</tr>
<tr>
<td>Operation</td>
<td>28</td>
</tr>
<tr>
<td>VideoClass</td>
<td>29</td>
</tr>
<tr>
<td>Classdiagram</td>
<td>32</td>
</tr>
</tbody>
</table>
CHAPTER FIVE: CONCLUSIONS AND FUTURE DIRECTIONS

Overview ................................................. 125

Future Directions ......................................... 126

Authentication and Authorization ............ 126

Load Balancing and Session Management ...... 126

Support for Multiple Channels ............... 127

Support for Non-Web Applications ............ 127

Support for Different Protocols ............... 127

Design as Thin Client ....................... 127

Including Activation Framework ............... 127
LIST OF FIGURES

Figure 1.1  A Collaborative System ........................................ 4
Figure 1.2  Use Cases Supported by Client ................................ 9
Figure 1.3  Use Cases Supported by Server ............................... 10
Figure 2.1  VIDEO State Diagram .......................................... 17
Figure 2.2  VIDEO Architecture ........................................... 18
Figure 2.3  VIDEO Packages ................................................ 19
Figure 2.4  Package video Class Diagram ................................ 20
Figure 2.5  Package server Class Diagram ............................... 21
Figure 2.6  Package client Class Diagram ............................... 22
Figure 2.7  Registry .......................................................... 25
Figure 2.8  LockRegistry ..................................................... 26
Figure 2.9  Attribute ......................................................... 27
Figure 2.10 Operation .......................................................... 29
Figure 2.11 VideoClass ....................................................... 31
Figure 2.12 Classdiagram ..................................................... 33
Figure 2.13 Interface IRequest .............................................. 34
Figure 2.14 Request Facade .................................................. 35
Figure 2.15 AttributeRequest ............................................... 36
Figure 2.16 OperationRequest ............................................... 38
Figure 2.17 ClassRequest ..................................................... 39
Figure 2.18 ClassdiagramRequest .......................................... 41
Figure 2.19 LockRequest ...................................................... 42
Figure 3.6  Add New Attribute                      104
Figure 3.7  Adding a New Operation                105
Figure 3.8  Renaming an Attribute                 107
Figure 3.9  Deleting an Attribute                 108
Figure 3.10 Renaming an Operation                109
Figure 3.11 Delete an Operation                  110
Figure 3.12 Creating a New Class Diagram         112
Figure 3.13 Deleting a Class Diagram             113
Figure 3.14 Renaming a Class Diagram             114
Figure 3.15 Checking-Out a Class Diagram          115
Figure 3.16 Check-In a Class Diagram              116
Figure 3.17 Add Class Tool                       117
Figure 3.18 Class Diagram Manager                118
CHAPTER ONE
SOFTWARE REQUIREMENT SPECIFICATIONS

Collaborative Computing

Real-time conferencing and collaborative computing is a great way to make employees more effective, increase productivity and teamwork, improve decision making, ideal way to break down geographic barriers, enabling technical and creative professionals to collaborate. Several studies were done and several educational and commercial software packages were developed targeting collaborative computing. Following are some of them:

- "Java Shared Data Toolkit" [9] provides a set of application programming interfaces for collaborative works. Advantages of collaborative computing were explained in detail.

- "A Collaborative Software Engineering System" by Gary Brewer [10] developed a collaborative system. The shortcomings of groupware were discussed.

• SunForum 3.2 Data Collaboration software supports audio- and video-conferencing and whiteboard sessions, VoIP telephony communication, share applications, and transfer files and multimedia information.

• Habanero [12] is a software framework of intercomputer object transportation capabilities, which facilitates the construction of software for synchronous communication over the Internet.

Need for a Collaborative Modeling Tool

Software development requires the use of modeling tools throughout its life cycle. There are several modeling tools available in the market. Some of them are: Rational Rose, System Architect, Structure Builder, Visio etc. Most of these products operate in a single user, standalone mode.

Software development is a complex team effort and requires the services of several team members who could be developers, domain experts, project managers, business system analysts, graphic designers, and many others, possibly geographically spread over. The diversity of software development team members coupled with the benefits
of collaborative computing justifies the development of a collaborative modeling tool.

Accordingly the present project, Virtual Design Office (VIDEO) was aimed at developing, a flexible collaborative framework, and extending the framework to create a collaborative modeling tool supporting Unified Modeling Language (UML) specifications.

Methodology

A distributed collaborative framework was developed using design patterns. Making use of web technology the collaborative framework was developed as a web based application. In view of cross platform requirements the framework was developed using Java. The framework was modeled as a client (distributed clients)/server (central server) technology using the Publisher/Subscriber pattern. The distributed clients would send requests to the central server and the central server would process the requests and publish the response. All connected clients, as response subscribers would receive the response from the central server and synchronize the application’s state. The central server and distributed clients are shown in Figure 1.1. The framework was then extended to develop a
collaborative modeling tool supporting the desired functionality of drawing UML diagrams.

Central Server

Central server is the central point of request processing and response publishing. While designing the central server the following approaches were evaluated:
- Remote Method Invocation: A typical Remote Method Invocation (RMI) method would be a request-response scenario. For the server to broadcast the response to all connected clients there are two alternatives: a) either all the connected clients should define their own Remote Servers or b) the clients have to poll the server at a predefined time. The first alternative requires writing code overcoming the browsers' security restrictions. The second alternative increases network traffic, network overhead as the number of requests to the server. This restricts the scalability of the server.

- TCP/IP Sockets: In this approach for every connected client a new thread is spawned on the server. The overhead of thread creation and thread per client and the associated resource requirements again restrict the scalability the application.

- Alternative Approaches: The limitations of RMI and TCP/IP socket approaches demand a better alternative. The alternative system should be
capable of broadcasting the response to the connected clients without the overhead of multiple threads and writing code, which is within browsers' security restrictions and avoiding polling the server. A combination of RMI and Sockets with an RMI server for queuing the incoming requests and a ServerSocket maintaining a list of connected clients to broadcast the response is very promising. Java servlet technology replacing the RMI server above can also be used for dispatching a request from the client. Again the servlet technology requires a servlet engine to process the servlets and inter process communications on the server. In view of its simplicity the RMI-Socket approach is used in the present study.
Distributed Client

To support user interaction the client was designed and developed as an applet. The Graphical User Interface was developed using Swing components. The client could be run in any web browser supporting Swing components or using java appletviewer.

VIDEO Scope

Unified Modeling Language (UML) specification is big and as a result, the scope of the project was limited to the following UML functionality:

- Class: The following operations related to a class are supported
  - Create a new class
  - Checkout a class
  - Check in a class
  - Rename a class
  - Delete a class
  - Add an attribute to a class
  - Add an operation to a class
• Attribute: The following operations related to an attribute are supported
  o Rename an attribute
  o Delete an attribute from a class

• Operation: The following operations related to an operation are supported
  o Rename an operation
  o Delete an operation from a class

• Classdiagram: The following operations related to a class diagram are supported
  • Create a new class diagram
  • Delete a class diagram
  • Rename a class diagram
  • Check out a class diagram
  • Check in a class diagram
  • Add a class to a class diagram
  • Delete a class from a class diagram

Supporting the above functionality, the use cases on the client side development are shown in Figure 1.2 and the use cases on the server side are shown in Figure 1.3.
Figure 1.2 Use Cases Supported by Client
Figure 1.3 Use Cases Supported by Server
Definitions, Acronyms, and Abbreviations

The definitions, acronyms, and abbreviations used in the document are described in this section.

**Applet.** An applet is small program that is intended not to be run on its own, but rather to be embedded inside another application.

**Abstract Factory.** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Builder.** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Chain of Responsibility.** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**Command.** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**Composite.** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients
treat individual objects and compositions of objects uniformly.

**Decorator.** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

**Facade.** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

**Factory Method.** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Flyweight.** Use sharing to support large numbers of fine-grained objects efficiently.

**Iterator.** Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Mediator.** Defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.
Naming. The naming class provides methods for storing and obtaining references to remote objects in the remote object registry.

Observer. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Singleton. Ensure a class only has one instance and provide global point of access to it.

Persistency. Persistence indicates that a collection of data remains intact even if its source is no longer attached to the network. The objects in JavaSpaces for example, remain available to other users even if the source has temporarily disconnected from the network. This feature is critical to keep distributed systems highly available.

Java. Java is a cross-platform programming language from Sun Microsystems.

RMIRegistry. A remote object registry is a bootstrap naming service that is used by RMI servers on the same host to bind remote objects to names. Clients on local and remote hosts can look up remote objects and make remote method invocations.
Serialization. Serialization is the process of writing/reading the persistent data to/from the storage media.

Server Socket. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

Socket. A socket is an endpoint for communication between two machines.

Swing. Swing is the part of the Java Foundation Classes (JFC) that implements a new set of GUI components with a pluggable look and feel.

TCP/IP. Transmission Control Protocol on top of the Internet Protocol provides a reliable, point-to-point communication channel that client-server applications on the internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

UML. The Unified Modeling Language is the industry-standard language for specifying, visualizing,
constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction.

Web Server. A web server is an application, which would serve contents to the World Wide Web.

XML. XML is an abbreviation for extensible markup language.

Summary

Describing the advantages of collaborative computing the necessity of developing a collaborative modeling tool was explained. Discussing various options a multi-client, server architecture was proposed for implementation. The details of client server architecture were described. The scope of the project was discussed. Various technical terms were explained. In the next chapter VIDEO architecture is explained in detail and the implementation details of various components are discussed.
CHAPTER TWO
ARCHITECTURE

Overview

In this chapter VIDEO architecture, video package structure, video components, and use case realization with the help of sequence diagrams are explained.

Architecture

VIDEO was designed as a multi-client multi-threaded server using RMIServer for receiving incoming requests and TCP/IP sockets for broadcasting the response to the connected clients. Using client's GUI a request could be created and dispatched to the server. At a high level the server logs the request, queues the request to a request queue, sorts the request queue, dequeues the request queue, instantiates a request processor, processes the request, logs the response, and broadcasts the response. A client processes the response synchronizing its object states with the objects on the server. The state diagram explaining the complete cycle of request processing and response publishing is shown in Figure 2.1. VIDEO architecture is shown in Figure 2.2.
Figure 2.1 VIDEO State Diagram
Figure 2.2 VIDEO Architecture
VIDEO Packages

The package structure of the application is shown in Figure 2.3. Various classes used in the project are packaged as follows:

- **video**: Contains classes used by client and server and is shown in Figure 2.4
- **server**: Contains classes used by server and is shown in Figure 2.5
- **client**: Contains classes used by client and is shown in Figure 2.6
- **event**: Contains events used in the application
- **utilities**: Contains utility classes
- **drawingtools**: Contains general purpose drawing tools

![Diagram showing the VIDEO Packages]
Figure 2.4 Package video Class Diagram
Figure 2.5 Package server Class Diagram
Figure 2.6  Package client Class Diagram
Figure 2.6 Package client Class Diagram (Contd.)
Classes Common to Client and Server

There are several classes used by classes in client and server packages. All these classes were placed in the package video. In this section important classes in video package, and used in client and server packages are described with class diagrams.

Registry

Registry is the object repository and maintains the state of objects used in the application. While the central server maintains master copy of the application, every client maintains its own registry and client’s registry is always synchronized with the central server’s registry. To avoid creation of multiple instances, Registry is designed as a Singleton. The Registry has a reference to a hashmap. Objects are stored in the hashmap as key value pairs in the Registry with object IDs as keys. Hashmap could be replaced with a database for database services. In addition to being a repository, Registry also implements several listener interfaces, which are explained in the subsequent sections. Registry class is shown in Figure 2.7.
Similar to Registry LockRegistry is designed as a Singleton and is shown in Figure 2.8. LockRegistry maintains the list of objects locked.
Attribute

Attribute is a model of UML Attribute. Attribute has a reference to an Observable class so that observers could be registered to receive state change notification. A VideoClass described in the subsequent section, which is a parent class of the Attribute would be registered as an Observer of the Attribute so that the Attribute can notify its state change to its parent, VideoClass and in turn the
parent could update its state and notify its state change to its registered observers. Attribute class is shown in Figure 2.9.

```
Serializable

t=VideoAttribute

+int ATTRIBUTE_PUBLIC=0
+int ATTRIBUTE_PROTECTED=1
+int ATTRIBUTE_PRIVATE=2
+int ATTRIBUTE_IMPLEMENTATION=3

-int lAttributeModifier
-String sAttributeName
-String sStereoType
-String sClass
-String sType
-long oid=-1
-VideoClass videoClass=null
-long videoClassID=-1
-ObservableClass observableClass=null

Attribute()
Attribute(int lAttributeModifier)
Attribute(String sAttributeName)
Attribute(int lAttributeModifier, String sAttributeName)
+long getOID()
+void setOID(long oid)
+String getAttributeName()
+void setAttributeName(String sAttributeName)
+int getAttributeModifier()
+void setAttributeModifier(int iAttributeModifier)
+String getStereoType()
+String toString()
+void setVideoClass(VideoClass videoClass)
+VideoClass getVideoClass()
+void setVideoClassID(long videoClassID)
+long getVideoClassID()
+void addObserver(Observer observer)
-void update()
```

Figure 2.9 Attribute
Operation is a model representation of UML Operation. Similar to Attribute, an Operation has a reference to an Observable class, observers could be registered and a state change notification can be sent to its registered observers. Again the parent, VideoClass would be registered as an Observer of the Operation. Operation is shown in Figure 2.10.
VideoClass

VideoClass is a model representation of UML Class.

VideoClass provides several accessor and mutator methods
for getting and setting various members. VideoClass also has a reference to an Observable class so that observers could be registered and a state change notification could be sent. Classdiagram discussed in a subsequent section would be registered as an Observer of VideoClass. VideoClass implements Observer interface so that it can be registered as an observer of its members namely Attributes, and Operations. When a member's state change notification is received, VideoClass would update its state and in turn would notify of its state change to its registered observers. VideoClass is shown in Figure 2.11.
## VideoClass

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoClass()</td>
<td>Constructor for VideoClass</td>
</tr>
<tr>
<td>VideoClass(long oid)</td>
<td>Constructor that takes a long oid as an argument</td>
</tr>
<tr>
<td>VideoClass(String className)</td>
<td>Constructor that takes a string className as an argument</td>
</tr>
<tr>
<td>VideoClass(String className, long oid)</td>
<td>Constructor that takes a string className and a long oid</td>
</tr>
<tr>
<td>+long getOID()</td>
<td>Method to get the OID of the VideoClass</td>
</tr>
<tr>
<td>+void setClassName(String sName)</td>
<td>Method to set the className of the VideoClass</td>
</tr>
<tr>
<td>+String getClassName()</td>
<td>Method to get the className of the VideoClass</td>
</tr>
<tr>
<td>+void addOperation(Operation operation)</td>
<td>Method to add an operation to the VideoClass</td>
</tr>
<tr>
<td>+void deleteOperation(Operation operation)</td>
<td>Method to delete an operation from the VideoClass</td>
</tr>
<tr>
<td>+Iterator getOperations()</td>
<td>Method to get an iterator of all operations in the VideoClass</td>
</tr>
<tr>
<td>+void addAttribute(Attribute attribute)</td>
<td>Method to add an attribute to the VideoClass</td>
</tr>
<tr>
<td>+void deleteAttribute(Attribute attribute)</td>
<td>Method to delete an attribute from the VideoClass</td>
</tr>
<tr>
<td>+Iterator getAttributes()</td>
<td>Method to get an iterator of all attributes in the VideoClass</td>
</tr>
<tr>
<td>+void removeCurrentNode()</td>
<td>Method to remove the current node from the VideoClass</td>
</tr>
<tr>
<td>+String toString()</td>
<td>Method to get a string representation of the VideoClass</td>
</tr>
<tr>
<td>+void attachToTree(JTree tree, DefaultMutableTreeNode parent, boolean shouldBeVisible)</td>
<td>Method to attach the VideoClass to a tree node</td>
</tr>
<tr>
<td>+String getName()</td>
<td>Method to get the name of the VideoClass</td>
</tr>
<tr>
<td>#Object clone()</td>
<td>Method to clone the VideoClass</td>
</tr>
<tr>
<td>+VideoClass getClone()</td>
<td>Method to get a clone of the VideoClass</td>
</tr>
<tr>
<td>+String getStringRepresentation()</td>
<td>Method to get a string representation of the VideoClass</td>
</tr>
<tr>
<td>+void addObserver(Observer observer)</td>
<td>Method to add an observer to the VideoClass</td>
</tr>
<tr>
<td>+void update()</td>
<td>Method to update the VideoClass</td>
</tr>
<tr>
<td>+void update(Observable observable, Object object)</td>
<td>Method to update the VideoClass with an observer and object</td>
</tr>
</tbody>
</table>

**Figure 2.11 VideoClass**
**ClassDiagram**

ClassDiagram is a model representation of UML class diagram. All classes participating in a ClassDiagram are stored in a Hashtable with the VideoClass as the key and VideoClass's location as the value. ClassDiagram implements the Observer interface so that it can be registered as an Observer of an Observable class. A ClassDiagram would be registered as an Observer of all VideoClasses which participate in the class diagram. A ClassDiagram would update its state upon state change notification from any of its participating classes. The ClassDiagram is shown in Figure 2.12.
### ClassDiagram

<table>
<thead>
<tr>
<th>Method/Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassDiagram()</td>
<td>Constructor.</td>
</tr>
<tr>
<td>ClassDiagram(long oid)</td>
<td>Constructor.</td>
</tr>
<tr>
<td>ClassDiagram(String classDiagramName)</td>
<td>Constructor.</td>
</tr>
<tr>
<td>ClassDiagram(String classDiagramName, long oid)</td>
<td>Constructor.</td>
</tr>
<tr>
<td>+String toString()</td>
<td>toString.</td>
</tr>
<tr>
<td>+long getOID()</td>
<td>getOID.</td>
</tr>
<tr>
<td>+void setClassDiagramName(String classDiagramName)</td>
<td>setClassDiagramName.</td>
</tr>
<tr>
<td>+String getClassDiagramName()</td>
<td>getClassDiagramName.</td>
</tr>
<tr>
<td>+void addClass(VideoClass videoClass)</td>
<td>addClass.</td>
</tr>
<tr>
<td>+void removeClass(VideoClass videoClass)</td>
<td>removeClass.</td>
</tr>
<tr>
<td>+void deleteSelected(long[] ids)</td>
<td>deleteSelected.</td>
</tr>
<tr>
<td>+void setLocation(VideoClass videoClass, Point point)</td>
<td>setLocation.</td>
</tr>
<tr>
<td>+void setLocation(long[] ids, Point[] point)</td>
<td>setLocation.</td>
</tr>
<tr>
<td>+Point getLocation(VideoClass videoClass)</td>
<td>getLocation.</td>
</tr>
<tr>
<td>+Enumeration getClasses()</td>
<td>getClasses.</td>
</tr>
<tr>
<td>+Enumeration getLocations()</td>
<td>getLocations.</td>
</tr>
<tr>
<td>+Hashtable getElements()</td>
<td>getElements.</td>
</tr>
<tr>
<td>+String getName()</td>
<td>getName.</td>
</tr>
<tr>
<td>+void processClassResponseEvent(ClassResponse classResponse)</td>
<td>processClassResponseEvent.</td>
</tr>
<tr>
<td>+void processResponse(IResponse response)</td>
<td>processResponse.</td>
</tr>
<tr>
<td>+long getClassOID(Point point)</td>
<td>getClassOID.</td>
</tr>
<tr>
<td>+void addRelation(IUserInterface userInterface)</td>
<td>addRelation.</td>
</tr>
<tr>
<td>+void removeRelation(IUserInterface userInterface)</td>
<td>removeRelation.</td>
</tr>
<tr>
<td>+VideoClass getVideoClass(long oid)</td>
<td>getVideoClass.</td>
</tr>
<tr>
<td>+boolean containsClass(long oid)</td>
<td>containsClass.</td>
</tr>
<tr>
<td>-void update()</td>
<td>update.</td>
</tr>
<tr>
<td>+void addObserver(Observer observer)</td>
<td>addObserver.</td>
</tr>
<tr>
<td>+void update(Observable observable, Object object)</td>
<td>update.</td>
</tr>
</tbody>
</table>

**Figure 2.12 ClassDiagram**
Request Facade

The design pattern "Facade" is used for sending requests from client to the server. The interface IRequest is the base interface of the request facade. The interface IRequest extends the "Comparator" interface. The comparator interface is used while sorting the request queue and the initial timestamp is used for comparison. The method "getInitTime()" returns the time at which the request was initiated and the method "setInitTime()" sets the time at which the request was initiated. The method "getSource()" returns the InetAddress of the source and "setSource()" sets the InetAddress of the request origin. The interface IRequest is shown in Figure 2.13 and the request facade is shown in Figure 2.14.

```
Comparator

interface IRequest

video.IRequest

+String getComponentName()
+long getInitTime()
+void setInitTime(long initTime)
+InetAddress getSource()
+void setSource(InetAddress inetAddress)
```

Figure 2.13 Interface IRequest
AttributeRequest

AttributeRequest details are shown in Figure 2.15. The constants CREATE_NEW_ATTRIBUTE, ATTRIBUTE_RENAME, and DELETE_ATTRIBUTE define the supported functionality and are used by AttributeRequestProcessor while processing AttributeRequest. The attribute request contains a
reference to its parent class. The parent class is used
during request processing and response processing.

Figure 2.15 AttributeRequest
OperationRequest

The OperationRequest is shown in Figure 2.16. The constants CREATE_NEW_OPERATION, OPERATION_RENAME, and DELETE_OPERATION define the functionality supported and are used by OperationRequestProcessor, while processing OperationRequest. Similar to attribute request operation request has a reference to its parent class.
Figure 2.16 OperationRequest

The constants CREATE_NEW_CLASS, CLASS_RENAME, ADD_NEW_ATTRIBUTE, ADD_NEW_OPERATION, MODIFY_CLASS, DELETE_CLASS define the supported functionality and are
used by ClassRequestProcessor while processing ClassRequest. ClassRequest is shown in Figure 2.17.

| video.ClassRequest | Serializable
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+int CREATE_NEW_CLASS=0</td>
<td></td>
</tr>
<tr>
<td>+int CLASS_RENAME=1</td>
<td></td>
</tr>
<tr>
<td>+int ADD_NEW_ATTRIBUTE=2</td>
<td></td>
</tr>
<tr>
<td>+int ADD_NEW_OPERATION=7</td>
<td></td>
</tr>
<tr>
<td>+int MODIFY_CLASS=13</td>
<td></td>
</tr>
<tr>
<td>+int DELETE_CLASS=16</td>
<td></td>
</tr>
<tr>
<td>-long classid=-1</td>
<td></td>
</tr>
<tr>
<td>-String className=null</td>
<td></td>
</tr>
<tr>
<td>-InetAddress inetAddress=null</td>
<td></td>
</tr>
<tr>
<td>-VideoClass videoClass=null</td>
<td></td>
</tr>
<tr>
<td>-Attribute attribute=null</td>
<td></td>
</tr>
<tr>
<td>-Operation operation=null</td>
<td></td>
</tr>
<tr>
<td>-int action=-1</td>
<td></td>
</tr>
<tr>
<td>-long initTime=0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.17 ClassRequest
ClassdiagramRequest

ClassdiagramRequest is shown in Figure 2.18. The constants CREATE_NEW_CLASSDIAGRAM, CLASSDIAGRAM_RENAME, ADD_CLASS, REMOVE_CLASS, DELETE_CLASSDIAGRAM define the supported functionality and are used by ClassdiagramRequestProcessor.
<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE NEW CLASSDIAGRAM</td>
<td>int newLocations</td>
</tr>
<tr>
<td>CLASSDIAGRAM RENAME</td>
<td>long ids</td>
</tr>
<tr>
<td>REMOVE CLASS</td>
<td>long objectsTobeDeleted</td>
</tr>
<tr>
<td>MODIFY CLASS LOCATION</td>
<td>long toOID</td>
</tr>
<tr>
<td>CREATE NEW RELATION</td>
<td>long fromOID</td>
</tr>
<tr>
<td>MODIFY START CLASS</td>
<td>Point point</td>
</tr>
<tr>
<td>MODIFY END CLASS</td>
<td>String classDiagramName</td>
</tr>
<tr>
<td>LOCK REQUEST REJECTED</td>
<td>InetAddress inetAddress</td>
</tr>
<tr>
<td>LOCK REQUEST REJECTED</td>
<td>long initTime</td>
</tr>
<tr>
<td>DELETE SELECTED</td>
<td>int action</td>
</tr>
<tr>
<td>DELETE CLASSDIAGRAM</td>
<td>long classID</td>
</tr>
</tbody>
</table>

```java
public class ClassDiagramRequest {
    private int action;
    private long initTime;
    private long classID;
    private long classDiagramID;
    private InetAddress source;
    private String classDiagramName;
    private Point point;
    private long toOID;
    private long fromOID;
    private long[] newLocations;
    private long[] ids;
    private long[] objectsTobeDeleted;
    private long[] toOIDs;
    private long[] fromOIDs;
    private boolean[] setids;
    private void setComponentName(String componentName) {
    private void setAction(int action) {
    private void setInitTime(long initTime) {
    private void setClassID(long classID) {
    private void setClassDiagramID(long classDiagramID) {
    private void setSource(InetAddress source) {
    private void setClassDiagramName(String classDiagramName) {
    private void setPoint(Point point) {
    private void setFromOID(long fromOID) {
    private void setToOID(long toOID) {
    private void setObjectsTobeDeleted(long[] objectsTobeDeleted) {
    private void setNewLocations(Point[] newLocations) {

Figure 2.18 ClassDiagramRequest
LockRequest

LockRequest is used for either requesting a lock or releasing the lock. Two constants REQUEST, and RELEASE define the two lock actions supported by the LockRequest class. The LockRequest's constructor requires two arguments: 1. an Object ID, the object's oID on which a lock action is required and 2. the requested action. REQUEST and RELEASE actions are implemented for a Class and a Classdiagram.

Figure 2.19 LockRequest
Response Facade

For every request in the request facade there is a corresponding response class in the response facade. Similar to request facade a response facade is used. The base interface for the response facade is IResponse and is shown in Figure 2.20 and the response facade is shown in Figure 2.21

```
interface IResponse

+String getComponentName()

Figure 2.20 Interface IResponse
```
ApplicationResponse

ApplicationResponse is shown in Figure 2.22. Initially when a client connects to the server the client served with ApplicationResponse, which has the Application object being served by the server. From the accessor method "getApplication()" the application being served by the server could be obtained.
AttributeResponse

AttributeResponse is the response to an AttributeRequest. The constants CREATE_NEW_ATTRIBUTE, ATTRIBUTE_RENAME, DELETE_ATTRIBUTE define the supported functionality and are used by AttributeResponseProcessor.
**Serializable Response**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_NEW_ATTRIBUTE</td>
<td>+int CREATE_NEW_ATTRIBUTE=0</td>
</tr>
<tr>
<td>ATTRIBUTE_RENAME</td>
<td>+int ATTRIBUTE_RENAME=1</td>
</tr>
<tr>
<td>DELETE_ATTRIBUTE</td>
<td>+int DELETE_ATTRIBUTE=2</td>
</tr>
<tr>
<td>MODIFY_ATTRIBUTE_MODIFIER</td>
<td>+int MODIFY_ATTRIBUTE_MODIFIER=4</td>
</tr>
<tr>
<td>MODIFY_ATTRIBUTE_TYPE</td>
<td>+int MODIFY_ATTRIBUTE_TYPE=5</td>
</tr>
<tr>
<td>MODIFY_ATTRIBUTE</td>
<td>+int MODIFY_ATTRIBUTE=7</td>
</tr>
<tr>
<td>attributeName</td>
<td>String attributeName=null</td>
</tr>
<tr>
<td>videoClassId</td>
<td>long videoClassId=-1L</td>
</tr>
<tr>
<td>attributeId</td>
<td>long attributeId=-1L</td>
</tr>
<tr>
<td>videoClass</td>
<td>VideoClass videoClass=null</td>
</tr>
<tr>
<td>attribute</td>
<td>Attribute attribute=null</td>
</tr>
<tr>
<td>action</td>
<td>int action=-1</td>
</tr>
</tbody>
</table>

```java
+ AttributeResponse()
+ String getComponentName()
+ VideoClass getVideoClass()
+ void setVideoClass(VideoClass videoClass)
+ Attribute getAttribute()
+ void setAttribute(Attribute attribute)
+ int getAction()
+ void setAction(final int action)
+ void setAttributeId(long attributeId)
+ long getAttributeId()
+ void setVideoClassId(long videoClassId)
+ long getVideoClassId()
+ String getAttributeName()
+ void setAttributeName(String attributeName)
```

**Figure 2.23 AttributeResponse**

**OperationResponse**

OperationResponse is shown in Figure 2.24. An OperationResponse is a response to an OperationRequest. The constants CREATE_NEW_OPERATION, OPERATION_RENAME, DELETE_OPERATION define the supported functionality and are
used by OperationResponseProcessor for processing on the client side.

```java
import java.io.Serializable;

public class OperationResponse {
    private int createNewOperation = 0;
    private int operationRename = 1;
    private int deleteOperation = 2;
    private int modifyOperationModifier = 4;
    private int modifyOperationType = 5;
    private int modifyOperation = 7;
    private String operationName = null;
    private long videoClassId = -1L;
    private long operationId = -1L;
    private VideoClass videoClass = null;
    private Operation operation = null;
    private int action = -1;

    public OperationResponse() {
    }

    public String getComponentName() {
    }

    public VideoClass getVideoClass() {
    }

    public void setVideoClass(VideoClass videoClass) {
    }

    public Operation getOperation() {
    }

    public void setOperation(Operation operation) {
    }

    public int getAction() {
    }

    public void setAction(int action) {
    }

    public long getOperationId() {
    }

    public void setOperationId(long operationId) {
    }

    public long getVideoClassId() {
    }

    public void setVideoClassId(long videoClassId) {
    }

    public String getOperationName() {
    }

    public void setOperationName(String operationName) {
    }
}
```

Figure 2.24 OperationResponse
ClassdiagramResponse

ClassdiagramResponse is shown in Figure 2.25. ClassdiagramResponse is the response to a ClassdiagramRequest. The constants CREATE_NEW_OPERATION, OPERATION_RENAME, DELETE_OPERATION define the supported functionality and are used by ClassdiagramResponseProcessor for processing on the client side.
Figure 2.25 ClassdiagramResponse
LockResponse

LockResponse is shown in Figure 2.26. LockResponse is the response for a LockRequest. The constants LOCK_ISSUED, LOCK_REJECTED, LOCK_RELEASED define the supported functionality and are used on the client.

<table>
<thead>
<tr>
<th>Serializable IResponse</th>
</tr>
</thead>
<tbody>
<tr>
<td>video.LockResponse</td>
</tr>
<tr>
<td>+int LOCK_ISSUED=0</td>
</tr>
<tr>
<td>+int LOCK_REJECTED=1</td>
</tr>
<tr>
<td>+int LOCK_RELEASED=2</td>
</tr>
<tr>
<td>-inetAddress destination=null</td>
</tr>
<tr>
<td>-int action=LOCK_REJECTED</td>
</tr>
<tr>
<td>-long oid=-1</td>
</tr>
<tr>
<td>+ LockResponse(long oid)</td>
</tr>
<tr>
<td>+String getComponentName()</td>
</tr>
<tr>
<td>+int getAction()</td>
</tr>
<tr>
<td>+void setAction(int action)</td>
</tr>
<tr>
<td>+inetAddress getDestination()</td>
</tr>
<tr>
<td>+void setDestination/inetAddress destination)</td>
</tr>
<tr>
<td>+long getOID()</td>
</tr>
</tbody>
</table>

Figure 2.26 LockResponse
Server

The class VideoServer is the central server of the server component and is responsible for:

- Binding the VideoRMIServerImpl to the RMIRegistry
- Opening the server socket and managing the connected clients
- Creating listeners and registering the listeners correctly

VideoServer

The class VideoServer is shown in Figure 2.27. The static "main" method is used for starting the VideoServer.

```java
server.VideoServer

Runnable

+ Application application=null
- Socket socket=null
- ApplicationResponse applicationResponse=null
- IRequestPublisher requestPublisher=null
- ListObserver listClients=null
- ServerSocket server=null
- Properties properties=null
- Thread thread=null

+ VideoServer(String[] args)
+ void run()
- void init()
+ void main(String[] args)

Figure 2.27 VideoServer
```
VideoRMIServer

VideoRMIServer and is shown in Figure 2.28. VideoRMIServer defines the remote method "dispatchRequest" used by:

- Client to dispatch a request
- Server to receive the request from a client

![Figure 2.28 VideoRMIServer Remote Interface](image)

RequestPublisher Interface

The interface IRequestPublisher defines the interface for publishing the incoming requests. While the method "publishRequest()" defines the method for publishing requests, the method "addRequestSubscriber()" defines the method for adding a subscriber, and the method "removeSubscriber()" defines the method for removing a
subscriber. The interface IRequestPublisher is shown in Figure 2.29

<table>
<thead>
<tr>
<th>Interface</th>
<th>server.IRequestPublisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>+void publishRequest(final IRequestPublisher requestPublisher, final IRequest request)</td>
<td></td>
</tr>
<tr>
<td>+void publishRequest(final IRequestPublisher requestPublisher, final IRequest request, final InetAddress inetAddress)</td>
<td></td>
</tr>
<tr>
<td>+void addRequestSubscriber(IRequestSubscriber requestSubscriber)</td>
<td></td>
</tr>
<tr>
<td>+void removeRequestSubscriber(IRequestSubscriber requestSubscriber)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.29 IRequestPublisher Interface.

**VideoRMIServerImpl**

VideoRMIServer implementation class is VideoServerImpl. VideoRMIServerImpl receives the requests from clients. VideoRMIServerImpl also implements the interface IRequestPublisher so that every request received by VideoRMIServerImpl class can be published and the registered subscribers can process the request. Accordingly for logging the requests RequestLogger is registered as a subscriber and for processing the requests DefaultRequestProcessor is registered as a subscriber. The class diagram of VideoRMIServerImpl class is shown in Figure 2.30.
RequestProcessorFacade

RequestProcessorFacade is used for request processing. The interface IRequestProcessor defines the single method "processRequest()", the base interface of the RequestProcessor facade. IRequestProcessorInterface is shown in Figure 2.31 and the request processor facade is shown in Figure 2.32.
Figure 2.31 IRequestProcessorInterface

Figure 2.32 RequestProcessorFacade
**RequestProcessorFactory**

The interface for creating a request processor is defined by the factory method `IRequestProcessorFactoryMethod` and `RequestProcessorFactory` class implements the `IRequestProcessorFactoryMethod`. The `RequestProcessorFactory` class diagram is shown in Figure 2.33.

![RequestProcessorFactory Diagram](image)

**IResponsePublisher**

The interface `IResponsePublisher` defines the interface for publishing the response. The interface defines the
methods for adding/removing ResponseSubscribers, and the method "publishResponse()" for publishing the response. The IResponsePublisher interface is shown in Figure 2.34.

<table>
<thead>
<tr>
<th>Interface</th>
<th>IResponsePublisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>+void publishResponse(IResponse response)</td>
<td></td>
</tr>
<tr>
<td>+void addResponseSubscriber(IResponseSubscriber responseSubscriber)</td>
<td></td>
</tr>
<tr>
<td>+void removeResponseSubscriber(IResponseSubscriber responseSubscriber)</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 2.34 IResponsePublisher](image)

**DefaultRequestProcessor**

DefaultRequestProcessor is designed as a multi-threaded component and is shown in Figure 2.35. As a request subscriber DefaultRequestProcessor enqueues the requests in one thread and dequeues the request queue and delegates the request processing to a dynamic request processor in another thread. Dynamic loading of the request processors and the request processing is shown in Figure 2.36. The file "server/config/RequestProcessors.prop" defines the mapping
between the request class name and request processor name. The actual class name of the request object is obtained using request.getClass().getName(). By looking up the request processor name from the property file, the request processor is dynamically loaded using Class.forName(request processor name).newInstance() and casting it to the interface IRequestProcessor. DefaultRequestProcessor also implements the interface IResponsePublisher. As an IResponsePublisher DefaultRequestProcessor publishes the response returned by the RequestProcessor.

![Diagram of DefaultRequestProcessor](image)

**Figure 2.35 DefaultRequestProcessor**
Figure 2.36 Dynamic Loading of Request Processors
RequestValidator

Every request is subject to request validation by RequestValidator. The granularity of validation is maintained at the request level. Request validation details are discussed below:

- LockRequest: A LockRegistry maintains the list of the locks on different objects. An Integer wrapping the object id as key is mapped to the InetAddress of the client holding the lock as value. LockRequest provides accessor methods to obtain the object id on which the lock is requested, lock action, and the InetAddress of the client which sent the request. An Integer wrapping the object id, as a key can be used to check if a lock was issued on the object. A lock is issued if no lock was already issued on the object. A LockRequest with a RELEASE action is subject to the equality test that the InetAddress of the requesting client and the InetAddress in the LockRegistry are the same.

- AttributeRequest: Every AttributeRequest is subjected to the test that the parent class was
locked and the request was from the client which has the lock.

- **OperationRequest**: Every OperationRequest is subjected to the test that the parent class was locked and the request was from the client which has the lock.

- **ClassRequest**: Creating a new class does not require a validation. Any other related request is validated for the class being locked and the request is from the same client, which possesses the lock.

- **ClassdiagramRequest**: Creation of a new Classdiagram is not validated. Any other related request is validated for a lock and the requesting client’s identity.

**Managing Disconnected Clients**

When a client sends a request to close the connection, the socket is removed from the list of the connected clients. All locks with the client’s InetAddress are cleared from the LockRegistry.
Client

VIDEO Client was implemented as an Applet. The client is responsible for:

- Establishing connection with server
- Sending requests to the server
- Listening for responses from the server
- Processing responses from the server

Accordingly the client is designed and implemented with the following main components:

- VideoClient: Graphical User Interface of the client is represented by VideoClient.
- RequestDispatcher: RequestDispatcher is responsible for dispatching the requests to the server.
- ClientListener: ClientListener is responsible for listening for the incoming responses from the server.
- Registry: Registry is the repository for the application. The Registry would be a mirror image of the registry on the server.
- LockRegistry: Maintains a list of locks and the associated objects.
- **WindowManager**: WindowManager is responsible for managing various windows of the application.
- **VideoTOC**: VideoTOC is a hierarchical display of various components of the application.
- **StatusBar**: StatusBar displays the status of the client.

Encompassing the above components, the client was designed and the client's composition is shown in Figure 2.37.
Several components are constructed during client's initialization. The VideoClient construction is shown in Figure 2.38.
Figure 2.38 VIDEO Client Construction
RequestDispatcher

Every request by the client is dispatched to the server by through a single instance of a request dispatcher. The requirements of a request dispatcher would be:

- Should implement the interface IRequestDispatcher for dispatching the requests
- Should extend Observable so that its state change can be notified to its registered observers
- Should implement the interface IRequestEventListener so that it could be registered as a RequestEventListener of VideoClient

AbstractRequestDispatcher is an implementation of the above bulleted requirements. IRequestDispatcherFactory method defines the interface for creating a request dispatcher and RequestDispatcherFactory implements the interface IRequestDispatcherFactory. RequestDispatcherFactory class diagram is shown in Figure 2.39
The RequestDispatcher implemented in the project uses the Remote Method Invocation. Accordingly, RequestDispatcher has reference to the VideoRMIServer. RequestDispatcher's Class Diagram is shown in Figure 2.40.
Figure 2.40 RequestDispatcher Composition

Connecting to Server. The client connects to the server by finding the VideoRMIServer on the network. The hostname is determined from the applet's
getCodebase().getHost() method. VideoRMIServer's configuration parameters are bindName and bindPort are read from "client/clientinit.prop" file on the server. The process of establishing the connection with the server is shown in Figure 2.41

**Figure 2.41 Connecting to Server**

**Dispatching Request.** RequestDispatcher dispatches the request to the server using VideoRMIServer. Sequence diagram for dispatching a request is shown in Figure 2.42.
ClientListener

ClientListener is implemented as an Observable multithreaded object. StatusBar is registered as an Observer of ClientListener so that ClientListener's state change can be displayed on the status bar. ClientListener creates a socket and connects to the server. ClientListener listens for the incoming responses from the server. ClientListener is shown in Figure 2.43.
<table>
<thead>
<tr>
<th>Class: ClientListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Thread listener=null</td>
</tr>
<tr>
<td>-List responseProcessors=null</td>
</tr>
<tr>
<td>-Socket socket=null</td>
</tr>
<tr>
<td>+ClientListener()</td>
</tr>
<tr>
<td>+void connect(String host, String port)</td>
</tr>
<tr>
<td>+void disconnect()</td>
</tr>
<tr>
<td>+void run()</td>
</tr>
<tr>
<td>+void addResponseProcessor(IResponseProcessor responseProcessor)</td>
</tr>
<tr>
<td>+void removeResponseProcessor(IResponseProcessor responseProcessor)</td>
</tr>
<tr>
<td>-void processResponse(IResponse videoResponse)</td>
</tr>
</tbody>
</table>

**Figure 2.43 ClientListener**

Sequence diagram showing ClientListener construction is shown in Figure 2.44.
Figure 2.44 ClientListener Construction
Establishing Connection with the Server. A connection to the server is established by ClientListener, which opens a TCP/IP socket and connects to the server. The sequence diagram for establishing connection to the server is shown in Figure 2.45.
Receiving Response from Server. ClientListener constantly listens for server's broadcast in its runnable
implementation. The sequence diagram showing the run method of the ClientListener is shown in Figure 2.46.
WindowManager

As a multiple document interface displaying the active diagram is delegated to WindowManager. WindowManager is shown in Figure 2.47.
client.WindowManager

-Hashtable hashtable=null
-JDesktopPane desktopPane=null
-Vector vector=null
-IToolFetcher toolFetcher=null
-IClassListProvider classListProvider=null
-IRequestDispatcher requestDispatcher=null
-ImageRequestListener imageRequestListener=null

+ WindowManager(JDesktopPane desktopPane)
+Graphics2D getGraphics(long objectID, String title)
+JComponent getComponent(long objectID, String title)
+void displayObject(Object object2DDisplayed)
+void internalFrameActivated(InternalFrameEvent internalFrameEvent)
+void internalFrameClosed(InternalFrameEvent internalFrameEvent)
+void internalFrameClosing(InternalFrameEvent internalFrameEvent)
+void internalFrameDeactivated(InternalFrameEvent internalFrameEvent)
+void internalFrameDeiconified(InternalFrameEvent internalFrameEvent)
+void internalFrameIconified(InternalFrameEvent internalFrameEvent)
+void internalFrameOpened(InternalFrameEvent internalFrameEvent)
+void actionPerformed(ActionEvent actionEvent)
+void displayDiagram(Object object)
+void setRegistry(Registry registry)
+void addDiagramChangeListener(DiagramChangeListener diagramChangeListener)
+void removeDiagramChangeListener(DiagramChangeListener diagramChangeListener)
-void fireDiagramChangeEvent(String diagramType)
+void setToolFetcher(IToolFetcher toolFetcher)
+String fetchToolName(String actionCommand)
+void setClassListProvider(IClassListProvider classListProvider)
+void setRequestDispatcher(IRequestDispatcher requestDispatcher)
+void processRequestEvent(IRequest request)
+void processResponse(IResponse response)
+void processClassDiagramResponseEvent(ClassDiagramResponse classDiagramResponse)
+void setImageRequestListener(ImageRequestListener imageRequestListener)

Figure 2.47 WindowManager
Graphical User Interface

The main components of VIDEO Client's Graphical User Interface are:

- Menu
- Toolbar
- VideoTOC
- Desktop
- Status Bar

Menu Building. Menu customization is done using XML files. The file "client/config/MainMenu.xml" is used for building the menu. The class MenuBuilder shown in Figure 2.48 reads the file "client/config/MainMenu.xml" and constructs the menu.
Figure 2.48 MenuBuilder

ToolBar. Toolbar customization is done using XML files. The main toolbar is constructed using the file "client/config/MainToolBar.xml". The tool bar construction is delegated to ToolBarBuilder and class diagram of ToolBarBuilder is shown in Figure 2.49.
Figure 2.49 ToolBar Builder

VideoTOC. VideoTOC is used for the hierarchical display of various components of the application. VideoTOC is implemented using JTree. VideoTOC displays all objects related to the application in use. Each tree node is a representation of an object in the application. Every tree node supports a custom context menu depending on the object being represented. VideoTOCRenderer is used for rendering. VideoTOC is shown in Figure 2.50.
Flyweight pattern is used for rendering the tree nodes. The images are read from the file "images.prop"
file. The file contains key value pairs of the TreeNode's class name and the image to be used. An icon with a red bounding rectangle indicates that the object is locked. VideoTOCRenderer is shown in Figure 2.51.
client.VideoTOCRenderer

- String imageFileName="images.prop"
- IImageRequestLister imageRequestLister=null
- IPropertyRequestLister propertyRequestLister=null
- ILockRegistry lockRegistry=null
- Map mapImageIcons=null

+ void setStreamRequestLister(IStreamRequestListener streamRequestListener)
+ void setmageRequestLister(IImageRequestLister imageRequestLister)
+ void setPropertyRequestLister(IPropertyRequestListener propertyRequestLister)
+ void setlmageFileName(String imageFileName)
+ String getlmageFileName()
+ Component getTreeCellRendererComponent(JTree tree, Object value, boolean isSelected, boolean isExpanded, boolean isLeaf, int nRow, boolean hasFocus)
+ void setLockRegistry(final ILockRegistry lockRegistry)
- Image getlmage(String imageName)
+ ImageIcon getImageIcon(String string)
+ ImageIcon getImageIcon(DefaultMutableTreeNode defaultMutableTreeNode)
+ ImageIcon getImageIcon(LogicalViewNode logicalViewNode)
+ ImageIcon getImageIcon(DeviceNode deviceNode)
+ ImageIcon getImageIcon(DeploymentViewNode deploymentViewNode)
+ ImageIcon getImageIcon(ProcessorNode processorNode)
+ ImageIcon getImageIcon(ComponentViewNode componentViewNode)
+ ImageIcon getImageIcon(UseCaseViewNode useCaseViewNode)
+ ImageIcon getImageIcon(ComponentNode componentNode)
+ ImageIcon getImageIcon(Attribute attribute)
+ ImageIcon getImageIcon(Operation operation)
+ ImageIcon getImageIcon(VideoClass videoClass)
+ ImageIcon getImageIcon(ClassDiagram classDiagram)

Figure 2.51 VideoTOCRenderer
Desktop. A Multiple Document Interface (MDI) is supported by using JDesktopPane. The Desktop displays the UML diagrams. JInternalFrame is used for the displaying diagrams. Each UML diagram is represented in a separate window with a related toolbar. Toolbar negotiation is supported i.e. depending on the type of the UML diagram having focus the toolbar in the desktop would change to a related toolbar.

StatusBar. StatusBar displays the client status, whether working online or offline, the status of sending a request to server, and the status of incoming messages from the server. The StatusBar is designed as an Observer of RequestDispatcher.

Creating Requests

The classes capable of generating a request event are:

- Attribute Node
- Operation Node
- Class Node
- Classdiagram Node
- LogicalViewNode

Even though there are multiple sources of creating requests there is only one RequestDispatcher class to
dispatch a request to the server. To channel all requests through the RequestDispatcher class the design pattern "Chain of Responsibility" is used. The event notification chain is shown in Figure 2.52.

Figure 2.52 Event Notification Chain
Response Processing

Every response object from the server is received by ClientListener first. ClientListener publishes the response and the registered response subscribers would process the response further. Response publishing by ClientListener is shown in Figure 2.53 and the collaboration diagram showing the response processing is shown in Figure 2.54.
Figure 2.54 Response Processing Collaboration Diagram
As a registered response processor of ClientListener, VideoClient processes the response by invoking a dynamic response processor. The properties file "client/config/ResponseProcessors.prop" contains the mapping between ResponseClass and the ResponseProcessor to be dynamically invoked. Again for the dynamic invocation of the response processor the response processor façade is used. The interface IResponseProcessor shown in Figure 2.55 defines the base interface and the response processor façade is shown in Figure 2.56. Noting that IResponseProcessor is a response publisher, after creating the response processor Registry is registered as a response subscriber of IResponseProcessor.

<table>
<thead>
<tr>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientIResponseProcessor</td>
</tr>
<tr>
<td>+void processResponse(IResponse response)</td>
</tr>
<tr>
<td>+void addResponseListener(IResponseListener responseListener)</td>
</tr>
<tr>
<td>+void removeResponseListener(IResponseListener responseListener)</td>
</tr>
</tbody>
</table>

Figure 2.55 IResponseProcessor
Upon receiving a notification from the response processor Registry synchronizes the application state based on the response received and fires a new ResponseEvent based on the response it received. VideoTOC and WindowManager as registered response event listeners would
receive the ResponseEvent. While WindowManager processes a ClassdiagramResponseEvent VideoTOC delegates the response event processing to LogicalViewNode. LogicalViewNode processes the ClassdiagramResponseEvent and ClassResponseEvent and delegates AttributeResponseEvent and OperationResponseEvent processing to ClassNode.

Processing ClassdiagramResponse (new class diagram) by LogicalViewNode is shown in Figure 2.57.
Figure 2.57 ClassDiagramResponse Processing
Processing ClassResponse (add new class) by LogicalViewNode is shown in Figure 2.58

Figure 2.58 ClassResponse Processing
Processing Attribute response (new attribute) is shown in Figure 2.59.
Processing operation response (new operation) is shown in Figure 2.60

Figure 2.60 OperationResponse Processing
Summary

In this chapter VIDEO architecture, video package structure, video components, and use case realization with the help of sequence diagrams were explained. In the next chapter test cases explaining the supported functionality is discussed.
CHAPTER THREE

TESTING

Overview

In this chapter, test cases for the supported functionality describing the necessary steps are explained. Screen shots showing the test cases are also shown.

Test Cases

To demonstrate the functionality within the scope of the project the following test cases are described:

- Class Requests
  - Create a new class
  - Checkout a class
  - Check in a class
  - Rename a class
  - Delete a class
  - Add a new attribute
  - Add a new operation

- Attribute Requests
  - Rename an attribute
  - Delete an attribute
- Operation Requests
  - Rename an operation
  - Delete an operation

- Class Diagram Requests
  - Create a new classdiagram
  - Delete a classdiagram
  - Rename a Classdiagram
  - Checkout a classdiagram
  - Check-In a classdiagram
  - Add a Class

Class Requests

All requests related to a class are created by using the class node's context menu except for creating a new class which is done through the logical view node's context menu.

Create a New Class. A new class is created from the logical view node's context menu. Creating a new class is shown in Figure 3.1 and the steps required to create a new class are:

- From the TOC select Logical View node
- Right click to pop up the context menu
- Select New->Class
Figure 3.1 Creating New Class
Checkout a Class. Checking out a class is shown in Figure 3.2. The necessary steps to check out a class are:

- Select the class node required to check out
- Right click to pop up the context menu
- Select Checkout

Figure 3.2 Check-Out a Class
Check-In a Class. The process of checking in a class is shown in Figure 3.3. The steps involved in check-in a class are:

- Select the class node to check in
- Right click to bring up the context menu
- Select Check In
Rename a Class. The process of renaming a class is shown in Figure 3.4. The steps required to rename a class are:

- Select the class node to rename
- Right click to popup the context menu
- Select Rename
- A dialog box prompting to enter the new class name would be displayed. Enter the new class name and select OK to rename or Cancel to cancel the operation.
Figure 3.4 Renaming a Class

Delete a Class. The process of deleting a class is shown in Figure 3.5. The necessary steps to delete a class are:

- Select the class node
- Right click to popup the context menu
- Select delete
Figure 3.5  Deleting a Class

Add New Attribute. The process of adding a new class is shown in Figure 3.6. The steps to add a new attribute are:

- Select the class node
- Right click to popup the context menu
- Select New->Attribute
Add an Operation. The process of adding a new operation is shown in Figure 3.7. The steps to add a new operation are:

- Select the class node
- Right click to popup the context menu
- Select New->Operation
Figure 3.7 Adding a New Operation
Attribute Requests

Attribute requests are generated by using attribute node's context menu.

Rename an Attribute. The process of renaming an attribute is shown in Figure 3.8. The steps to rename an attribute are:

- Select the attribute node to be renamed
- Right click to popup the context menu
- Select Rename
Delete an Attribute. The process of deleting an attribute is shown in Figure 3.9. The necessary steps to delete an attribute are:

- Select the attribute node
- Right click to popup the context menu
- Select delete
Operation Requests

Operation requests are generated by using the operation node's context menu.

Rename an Operation. The process of renaming an operation is shown in Figure 3.10. The steps to rename an operation are:

- Select the operation node to be renamed
- Right click to popup the context menu
- Select rename.
- A dialog box prompting to enter the new name for the attribute would be displayed. Enter the new name and select OK to rename the operation or select Cancel to cancel the renaming operation.

Figure 3.10 Renaming an Operation
Delete an Operation. The process of deleting an operation is shown in Figure 3.11. The steps required to delete an operation are:

- Select the operation node to be deleted
- Right click to popup the context menu
- Select Delete

Figure 3.11 Delete an Operation
Class Diagram Requests

Request for creating a class diagram is generated from logical view node's context menu. Checkout, check-in, rename, and delete requests are generated from the class diagram node's context menu.

Create a New Classdiagram. The process of creating a new Classdiagram is shown in Figure 3.12. The steps required to create a new Classdiagram are:

- Select the logical view node
- Right click to popup the context menu
- Select New->Class Diagram
Figure 3.12 Creating a New Class Diagram

Delete a Classdiagram. The process of deleting a Classdiagram is shown in Figure 3.13. The detailed steps to delete a class diagram are:

- Select the class diagram node that needs to be deleted
- Right click to popup the context menu
- Select Delete
Figure 3.13 Deleting a Class Diagram

Rename a Class Diagram. Renaming a class diagram is shown in Figure 3.14. The detailed steps to rename a class diagram are:

- Select the class diagram to be renamed
- Right click to popup the context menu
- Select Rename
- A dialog box prompting to enter the new name for the class diagram is displayed. Enter the new
name for the class diagram and select OK to rename the class diagram or Cancel to cancel the class diagram renaming.

Figure 3.14 Renaming a Class Diagram

Checkout a Class diagram. Checking out a class diagram is shown in Figure 3.15. The steps required to checking out a class diagram are:

- Select the class diagram node to be checked out
Right click to popup the context menu
Select Checkout

Figure 3.15 Checking-Out a Class Diagram

Check-In a Classdiagram. Checking in a class diagram is shown in Figure 3.16. The steps required to check-in a class diagram are:

Select the class diagram to be checked in
- Right click to popup the context menu
- Select Check In

Figure 3.16 Check-In a Class Diagram

Add a Class. A class to a class diagram is added interactively. First the tool for adding a class should be selected. The tool for adding a class to an active class diagram is shown in Figure 3.17. After selecting the tool
click on the class diagram window to add a class at the location. The class diagram manager dialog would be displayed prompting the user to select a class to add to the class diagram. The class diagram manager is shown in Figure 3.18. From the class diagram manager select the classes to be added to the class diagram and select Apply to add the classes or Cancel to cancel the operation.

![Add Class Tool](image)

Figure 3.17 Add Class Tool
Summary

In this chapter various test cases were explained with the help of screenshots. In the next chapter the build environment and maintenance manual are described.
Overview

In this chapter, server, and client configuration files are described. The build environment is described. Also configuring the application in a web environment and starting the server is described.

Video Server Configuration

The Server component of VIDEO consists of VideoRMIServer and VideoServer. VideoRMIServer queues incoming requests from clients while VideoServer manages clients' status and publishes the response. The VideoRMIServer and the VideoServer are configured using a properties file "server\config\serverinit.prop". The properties defined in the configuration file are discussed below.

Application Properties

Properties described in this section are related to the application. These properties include title of the application, the file names for stdout, and stderr redirection.

- **title**. This property defines the application title.
**stdout.** This property defines the name of the file to which the standard output stream is redirected.

**stderr.** This property defines the name of the file to which the standard error stream is redirected.

**ServerSocket Properties**

Properties described in this section are related to the server socket.

**port.** The port to which the ServerSocket would be bound and listening for client connections. This port is used for broadcasting the response generated by the server.

**VideoRMIServer Properties**

The properties described in this section are related to the VideoRMIServer.

**bindName.** "bindname" is a string by which VideoRMIServer can be identified by the network, and clients would be using during lookup process.

**bindAddress.** "bindAddress" is the IPAddress to which the rmiregsitry is bound, used in the case of a server with multiple network cards.

**bindPort.** The RMIFregistry by default runs on port 1099. This parameter would allow using a port other than the default port.
Client Configuration

Various configuration files are used for client configuration. Client configuration includes menu customization, customization of various tool bars, response processor customization, images and others. All client configuration files are located in the directory "client\config". Configuration files used in the project are described in this section.

Client Initialization

"clientinit.prop" lists various initialization properties used during client initialization as key value pairs. The following properties are used:

- **port**. This property defines server socket port, and the server is listening for incoming requests.
- **bindPort**. This property defines the rmiregistry port.
- **bindName**. This property describes the bound name of the VideoServerImpl. This property is used for look up by the clients.
Images

"images.prop" lists the key value pairs of various objects and the images used.

MainMenu

"MainMenu.xml" is used for the construction of client's menu.

MainToolBar

"MainToolBar.xml" is used for the construction of client's toolbar.

ClassDiaqToolBar

"ClassDiaqToolBar.xml" is used for the construction of Classdiagram toolbar.

ResponseProcessors

"ResponseProcessors.prop" lists the key value pairs of the response object and response processor to be invoked.

Tool

"Tools.prop" lists the key value pairs of the commands and the associated invoked tools.
Build Environment

The file video.zip included in the enclosed compact disc contains the source code and the build file. The following directory structure is employed for VIDEO:

- server: Contains all classes related to server
- client: Contains all classes related to client
- video: Contains all classes common to video
- event: Contains all event classes used by the client
- drawingtools: Contains all classes used for client side drawing
- utilities: Contains all utility classes

Ant is used for compiling the source code. The file "build.xml" is used for building the application. The compiled classes would be placed in the destination directory specified in the build.xml file, directory "build".

Server Setup

The server set up consists of copying the class files, images and html files to the web server. The class files should be placed in a directory, which is accessible by a web client.
Web Server Configuration

Copy all class files preserving the package structure to the web server's directory from where the application would be served.

Starting Server

The batch file "RemoteServer.bat" starts the rmiregistry and binds to the rmiregistry to the port specified. The batch file "server.bat" starts the server. Run "RemoteServer.bat" to run rmiregistry and execute "server.bat" to start the server.

Connecting to Server

The batch file client.bat invokes the appletviewer and connects to the server.

Summary

In this chapter the build environment was described. Client, and server configuration files and details were described. Web server set up and the batch files to start the server, and connecting to the server were described.
CHAPTER FIVE

CONCLUSIONS AND FUTURE DIRECTIONS

Overview

A real-time conferencing collaborative application framework required a real-time multi-client server synchronization unit. The synchronization unit was developed using a combination of RMI and sockets. Supporting user interactivity and resulting multitude of requests and responses a configurable collaborative framework was developed.

Within the scope of the project the framework was extended to create a collaborative unified modeling tool supporting limited functionality. VIDEO client was developed as an applet supporting multiple document interface. Client's graphical user interface was similar to Rational Rose, a commercial modeling tool.

The UML functionality not covered in this project can be extended by defining new request, response, request processor, and response processor classes. In addition the framework can be extended for any collaborative applications.
To make the framework useful in enterprise solutions it is necessary that robust authorization and authentication, logging, session management, load balancing and safe fail-over, support for multiple applications, and other enterprise features should be supported. For implementing the enterprise features, some recommendations are made in this chapter.

**Future Directions**

**Authentication and Authorization**

Light Weight Directory Access Protocol (LDAP) is a widely recognized protocol for user authentication and authorization. There are several free and commercial products available supporting LDAP with support for different programming languages. LDAP services could be integrated with the framework for user authentication and authorization.

**Load Balancing and Session Management**

Load balancing with session fail-over support is very important for scalability. This includes installing the application on different servers, and making session information persistent.
Support for Multiple Channels

The framework should support different projects simultaneously. The projects could be any collaborative work such as multi-client chat server, stock ticker application, on line news broadcasting, and many more.

Support for Non-Web Applications

The framework should be modified to support standalone applications in addition to the applets as a web application.

Support for Different Protocols

At present, the request and response objects are implemented as serializable objects. The framework should be modified to support different objects such as XML strings in addition to the serializable objects.

Design as Thin Client

Applets are relatively fat clients. Scalable Vector Graphics (SVG) can be used for rendering graphics objects in the web browser. By using SVG the application could be designed as a thin client.

Including Activation Framework

Activation framework provides automation procedures for starting the Server. Enterprise solutions require the
support for activation framework. The framework should be modified to include activation framework.

**Concurrent Editing of Diagrams**

Concurrent editing of UML diagrams could be supported by locking interested areas of the diagram.

**Summary**

In this chapter, some of the requirements of enterprise solutions were discussed. Some future suggestions were made to make the framework suitable for enterprise solutions.
REFERENCES


[4] "Java Platform1.2 Specification" by Sun Microsystems


