SHREYANSH SHUKLA
BE COMP B 24
SA – Exp 1

Experiment No.: 1

Modeling using xADL

Learning Objective: Student should be able to do Modeling using xADL

Tool:- Apigen , Data Binding Library

Theory:

xADL is a highly-extensible software architecture description language (ADL). It is used to describe various aspects of the architecture of a software system. The architecture of a software system is its high-level design; design at the level of components, connectors, and their configurations. Like other ADLs such as Rapide, Darwin, and Wright, xADL's core models the four most common architectural constructs, namely:

- **Components** : (the loci of computation),
- **Connectors** : (the loci of communication),
- **Interfaces :** (the exposed entry and exit points for components and connectors), and
- **Configurations** : (topological arrangements of components and connectors as realized by **links**).

- xADL is an XML-based language. XML, the eXtensible Markup Language, was originally created to annotate, or "mark up" text documents with semantic information. Elements of text are marked up using tags, or special strings, that delimit a section of text. Tags begin with an open angle-bracket (<) and end with a closing angle-bracket (>). In XML documents, tags generally come in pairs, signifying the start and end of a text element. Start tags contain a tag name immediately after the opening angle-bracket, and end-tags contain the same name, prefaced by a forward slash (/) immediately after the opening angle-bracket. Elements may be nested as necessary, but may not overlap. An example of some marked up text in XML might be:

  <name><first>Herb</first> <last>Mahler</last></name>

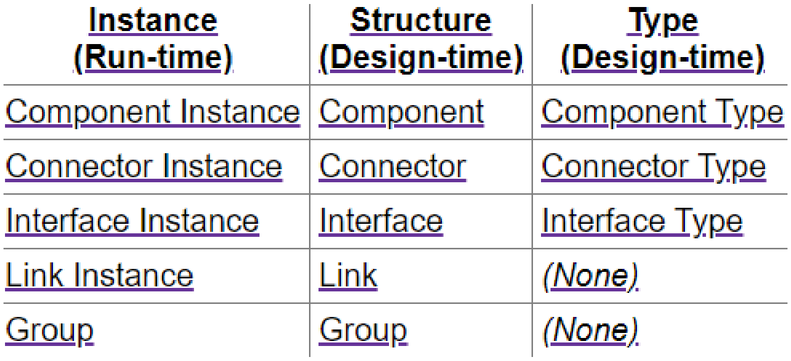
- To HTML users, this format may look familiar. This is because XML and HTML both share a common historical ancestor, SGML (the Standard Generalized Markup Language). In HTML, however, there is a finite set of allowed tags, each of which has a specific meaning dedicated to screen layout. So, tags like <H1> in HTML indicate that a text element is to be laid out in the Heading 1 style (usually large and bold, although this varies depending on the layout engine used), but do not indicate any other semantics about the element--is it a story headline? Is it someone's name? This information is not present in HTML.

- This lack of a static set of allowed tags introduces a new problem into XML applications. What tags are allowed and what do they mean? How do two parties sharing marked-up documents come to an agreement on what elements are allowed, and where? How can they ensure that their information is marked up in a consistent way that is meaningful to both of them?
- The answer to this problem is to introduce a *meta-language*, a language for defining languages. In XML, meta-languages define what elements are allowed, where they are allowed to occur (and what their cardinality is), and what data may be part of each element. The XML 1.0 standard included such a meta-language, called the DTD (Document Type Definition) language. The DTD meta-language was sufficient for expressing XML-based languages as a set of production rules, much like a BNF (Backus-Naur Form) grammar, but proved insufficient for certain types of applications. To remedy some issues that users had with DTDs, the W3C (World Wide Web Committee) drafted a new meta-language for XML called XML Schema. XML schemas are more expressive than DTDs, they have an XML-like syntax (DTDs do not), and they allow type relationships among element types much like object-oriented inheritance.
- The development of XML schemas made it much easier for developers to create and evolve XML-based languages, and fostered the development of modular languages like xADL. A full treatment of XML is far out of scope for this guide.
- xADL's XML basis means that data in xADL is arranged hierarchically. Connections between data elements that are not hierarchically arranged are managed using simple XML links; xADL's tool support facilitates navigating these links.
- As an XML-based language, xADL documents are readable and writable by hand, as simple text documents. However, because xADL is defined in multiple schemas, each schema having its own XML namespace, the actual code can get quite complicated. For example, this is a real component description in xADL:

The xADL Type System

xADL adopts the more traditional types-and-instances model found in many programming languages. In this model, components, connectors, and interfaces all have types (called *component types*, *connector types*, and *interface types*, respectively).

Links do not have types because links do not have any architectural semantics. The relationships between types, structure, and instances are shown in the following table:



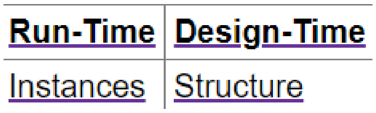
Instance (Run-time)	Structure (Design-time)	Type (Design-time)
Component Instance	Component	Component Type
Connector Instance	Connector	Connector Type
Interface Instance	Interface	Interface Type
Link Instance	Link	*(None)*
Group	Group	*(None)*

- Component Types
  - Sub architectures for Component Types
  - Signatures
- Connector Types*
  - Sub architectures for Component Types
  - Signatures
- Interface Types

Modeling of xADL

The instances schema gives xADL the ability to model running instances of architectural constructs like components, connectors, interfaces, and links. However, much work on software architecture is centered around the *design* of the architecture, rather than capturing properties of a running one.

For the purposes of this discussion, we make a distinction between architecture instances, which exist at run-time, and structural elements, which exist at design-time.



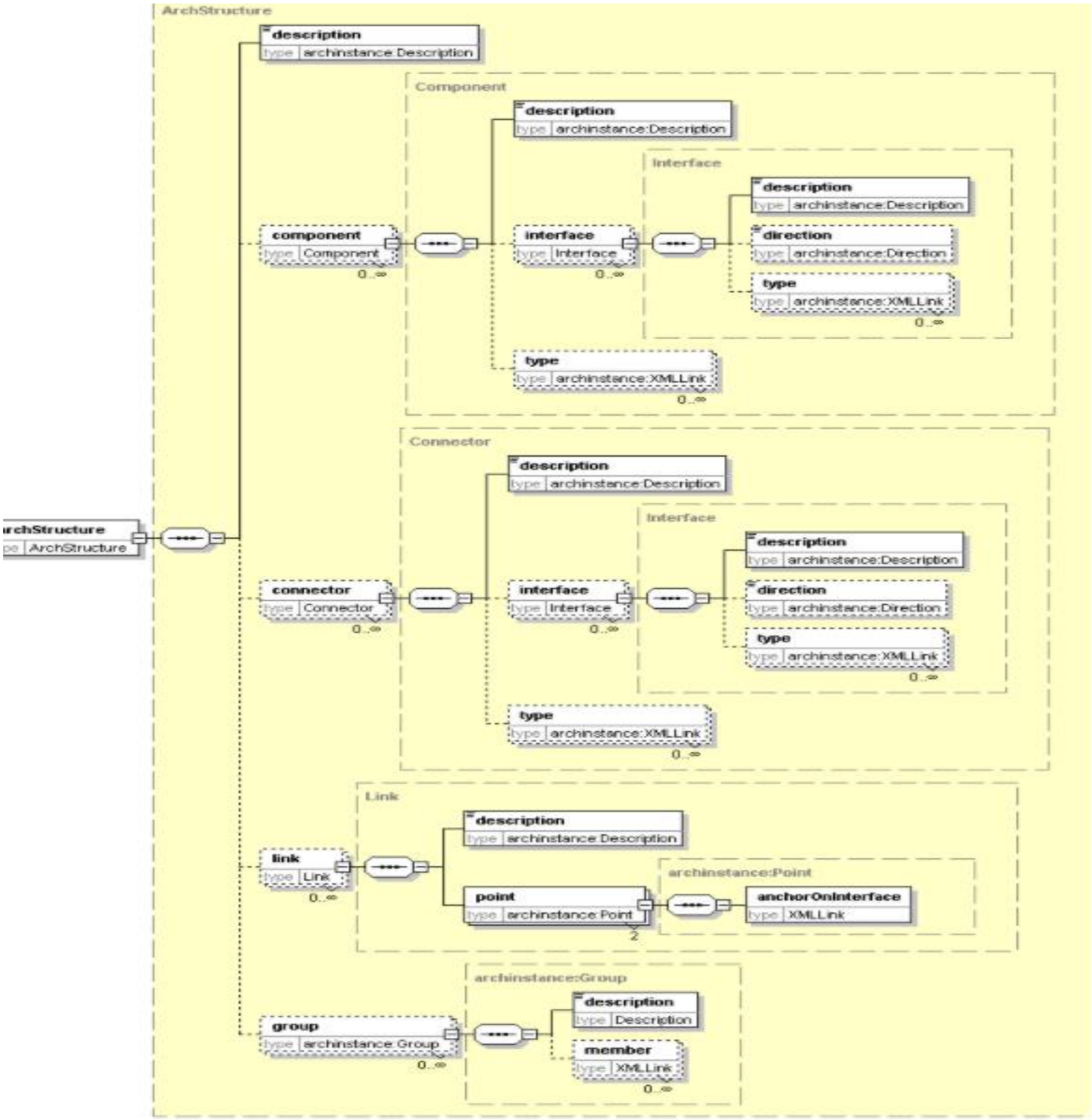
Run-Time	Design-Time
Instances	Structure

The xADL constructs available for modeling architectural structure mirror almost exactly those available for modeling architecture instances. The constructs defined in the instance schema are:

- Components
- Connectors

- Interfaces
- Links
- General Groups

Structure of design-time architecture



"component" is used as a structural construct, as opposed to "component instance," which is used to describe a run-time instance. Similarly, "connector," "interface," and "link" are used instead of "connector instance, "interface instance," and "link instance."

```
connector{
  (attr) id   = "conn1"
  description = "Connector 1"

  interface{
    (attr) id = "conn1.IFACE_TOP"
    description = "Connector 1 Top Interface"
    direction = "inout"
  }

  interface{
    (attr) id = "conn1.IFACE_BOTTOM"
    description = "Connector 1 Bottom Interface"
    direction = "inout"
  }
}

component{
  (attr) id   = "comp2"
  description = "Component 2"

  interface{
    (attr) id = "comp2.IFACE_TOP"
    description = "Component 2 Top Interface"
    direction = "inout"
  }

  interface{
    (attr) id = "comp2.IFACE_BOTTOM"
    description = "Component 2 Bottom Interface"
```

```
      direction = "inout"
    }
  }


  link{
    (attr) id = "link1"
    description = "Comp1 to Conn1 Link"
    point{
      (link) anchorOnInterface = "#comp1.IFACE_BOTTOM"
    }
    point{
      (link) anchorOnInterface = "#conn1.IFACE_TOP"
    }
  }
```

Result and Discussion:

1. Understood the xADL – The Software architecture Design Language.
2. It has similarities to XML.

Learning Outcomes: Students should be able to

LO1: Define xADL.

LO2: Identify xADL Command.

LO3: Apply xADL Command for desired Output.

Course Outcomes: Upon completion of the course students will be able to do Modeling using xADL.

Conclusion:

1. xADL has Components, Connectors, Interfaces.
2. It is used to Design Software Architecture

Viva Questions:

1. Define xADL.
2. Explain xADL.
3. Explain syntax of xADL.
4. Explain modeling using xADL
5. Explain xADL type system.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	

SHREYANSH SHUKLA
BE COMP B 24
SA - EXP 2

Experiment No.: 2

Visualization using xADL 2.0

Learning Objective: Student should understand and able to do Visualization using xADL 2.0

Tool:- Apigen or Data binding Library.

Theory:

What is xADL 2.0?

xADL 2.0 is a software architecture description language (ADL) developed by the University of California, Irvine for modeling the architecture of software systems. Unlike many other ADLs, xADL 2.0 is defined as a set of XML schemas. This gives xADL 2.0 unprecedented extensibility and flexibility, as well as basic support from the many available commercial XML tools.

The current set of xADL 2.0 schemas includes modeling support for:

- run-time and design-time elements of a system;
- support for architectural types;
- advanced configuration management concepts such as versions, options, and variants;
- product family architectures; and
- architecture "diff"ing (initial support).

xADL 2.0 is also an application of xArch, a core XML schema defined jointly by UCI and Carnegie Mellon University.

xADL 2.0 includes constructs that permit modeling of:

- Architecture structure and types,
- Product families (architectural versions, options, and variants), and
- Implementation mappings (mappings from architecture types to their implementations).

xADL 2.0's modules are defined as XML Schemas, making xADL 2.0 an XML-based language. All xADL 2.0 documents i.e. architecture descriptions are XML documents that are valid with respect to the xADL 2.0 schemas.

xADL 2.0 can be extended by end-users to optimize the language for particular domains. Tools are available that provide users with support for both using existing modules (schemas) and creating and manipulating their own extensions to xADL 2.0.

xADL 2.0 is an XML-based language. XML, the extensible Markup Language, was originally created to annotate, or "mark up" text documents with semantic information. Elements of text are marked up using tags, or special strings, that delimit a section of text. Tags begin with an open angle-bracket (<) and end with a closing angle-bracket (>). In XML documents, tags generally come in pairs, signifying the start and end of a text element. Start tags contain a tag name immediately after the opening angle-bracket, and end-tags contain the same name, prefaced by a forward slash (/) immediately after the opening angle-bracket. Elements may be nested as necessary, but may not overlap. An example of some marked up text in XML might be:

```
<name><first>Herb</first> <last>Mahler</last></name>
```

Constructs Defined in the Instances Schema :

A common theme throughout xADL 2.0, defined first in the instance schema, is that of IDs and Descriptions. Many elements in xADL 2.0 have an ID and/or a Description. Identifiers are assumed to be unique to a particular document. They do not necessarily have to be human-readable, although it helps if they are. Descriptions are intended to be human-readable identifiers of the described constructs.

Furthermore, the instance schema defines an element type called an XMLLink that is used over and over again in other xADL 2.0 schemas. XMLLinks are links to other XML elements. xADL 2.0 borrows the linking strategy from the XLink standard. However, because of poor support for the XLink standard in terms of real tools, xADL 2.0 document authors are advised to follow the following simplified convention for specifying XMLLinks.

In xADL 2.0, anything with an ID can be the target (i.e. the thing being pointed to) for an XMLLink. Every XMLLink has two parts (implemented as XML attributes), type and href; these are defined by the XLink standard. For xADL 2.0 XMLLinks, the type field should always be the string simple, indicating a simple XLink. The href field should be filled out with a URL such as:

http://*server*/*directory*/*document*.xml#*id*

This is a fairly standard fully-specified URL, linking to a document, but using the *anchor* part of the URL (i.e. the part after the pound sign ('#')) to indicate the identifier of the specific target element. Of course, if you are linking to an element in the same document, it is often preferable to link using a relative URL, such as:

#*id*

Which would be the element with ID *id* in the current document. So, two examples of valid hrefs might be:

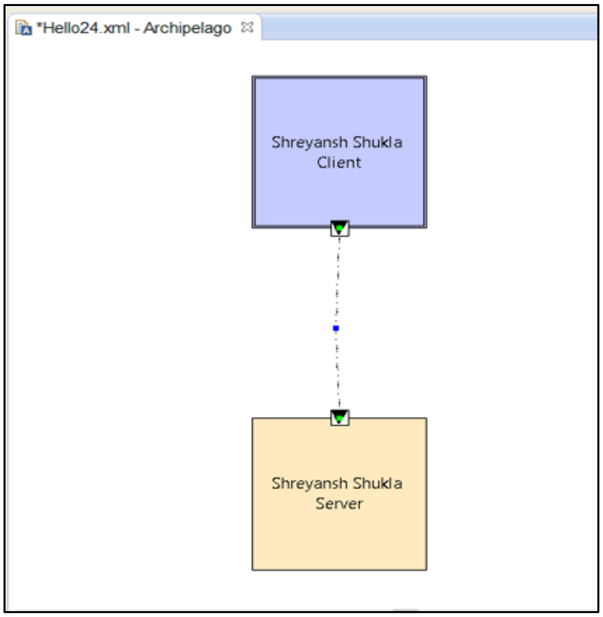
http://www.isr.uci.edu/foo/bar/archstudio.xml#ArchEdit
#ArchEdit *(from within the file archstudio.xml)*

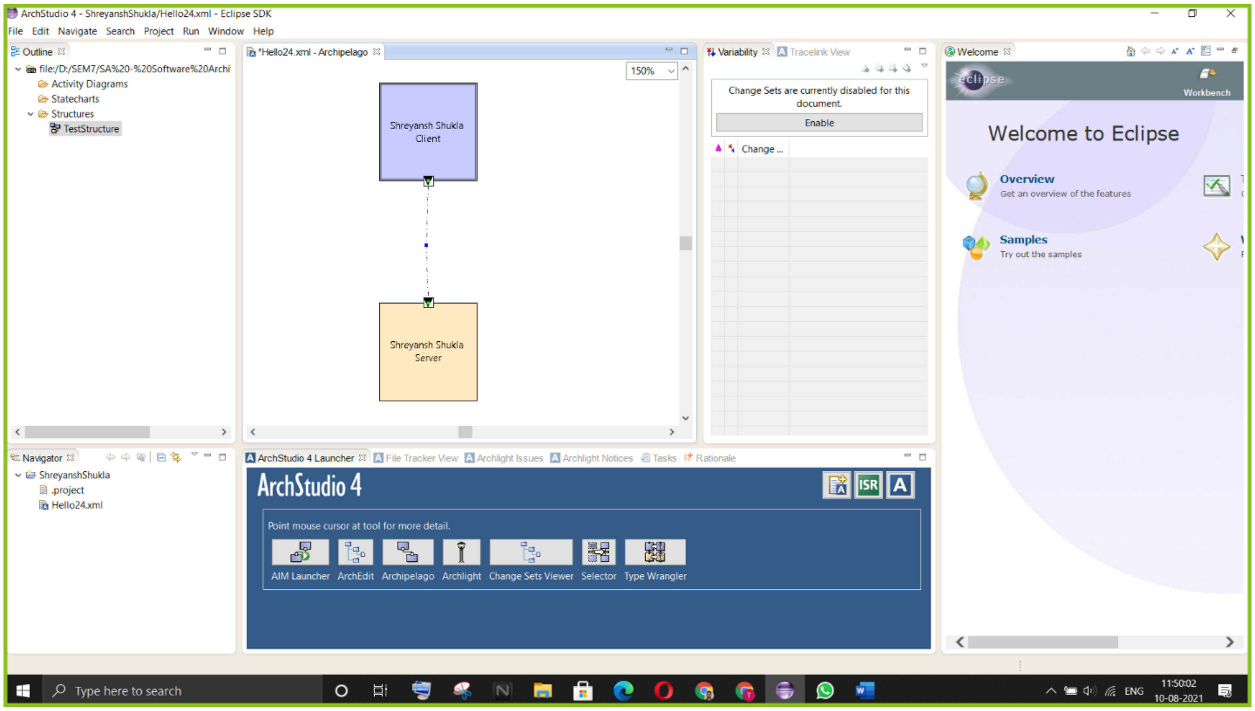
Visualization using xADL 2.0 :

```
interface{
   (attr) id = "conn1.IFACE_TOP"
   description = "Connector 1 Top Interface"
   direction = "inout"
  }

  interface{
   (attr) id = "conn1.IFACE_BOTTOM"
   description = "Connector 1 Bottom Interface"
   direction = "inout"
  }
 }

 component{
  (attr) id   = "comp2"
  description = "Component 2"

  interface{
   (attr) id = "comp2.IFACE_TOP"
   description = "Component 2 Top Interface"
   direction = "inout"
  }

  interface{
   (attr) id = "comp2.IFACE_BOTTOM"
   description = "Component 2 Bottom Interface"
   direction = "inout"
  }
 }

 link{
  (attr) id = "link1"
```

```
      description = "Comp1 to Conn1 Link"
      point{
        (link) anchorOnInterface = "#comp1.IFACE_BOTTOM"
      }
      point{
        (link) anchorOnInterface = "#conn1.IFACE_TOP"
      }
    }

    link{
      (attr) id = "link2"
      description = "Conn1 to Comp2 Link"
      point{
        (link) anchorOnInterface = "#conn1.IFACE_BOTTOM"
      }
      point{
        (link) anchorOnInterface = "#comp2.IFACE_TOP"
      }
    }
  }
```

Result and Discussion:





Sample Component – Connector - Link

Learning Outcomes: Students should have be able to understand

LO1: Define xADL 2.0.

LO2: Identify xADL Command.

LO3: Apply xADL Command for desired Output.

Course Outcomes: Upon completion of the course students will be able to do visualization using xADL 2.0.

Conclusion:

In this Experiment, we have Installed JDK , Eclipse & Archstudio

Following which, we have created a sample Component Connector Link

Viva Questions:

1. Define xADL 2.0.
2. Explain xADL 2.0.
3. Explain syntax of xADL 2.0.
4. Explain syntax of Constructs Defined in the Instances Schema.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

SHREYANSH SHUKLA
BE COMP B 24
SA – Exp 3

Experiment No.: 3

Creating Web service

Learning Objective: Student should be able to understand creating web services using and also different web services components.

Theory: A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service. For example, a client invokes a web service by sending an XML message, then waits for a corresponding XML response. As all communication is in XML, web services are not tied to any one operating system or programming language—Java can talk with Perl; Windows applications can talk with Unix applications.

Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents.

The architecture of web service interacts among three roles: service provider, service requester, and service registry. The interaction involves the three operations: publish, find, and bind. These operations and roles act upon the web services artifacts. The web service artifacts are the web service software module and its description.

Web Service Architecture

There are three roles in web service architecture:

i. Service Provider: From an architectural perspective, it is the platform that hosts the services.

ii. Service Requestor: Service requestor is the application that is looking for and invoking or initiating an interaction with a service. The browser plays the requester role, driven by a consumer or a program without a user interface.

iii. Service Registry: Service requestors find service and obtain binding information for services during development.

Different process in web service Architecture:

- Publication of service descriptions **(Publish)**
- Finding of services descriptions **(Find)**
- Invoking of service based on service descriptions **(Bind)**

Publish: In the publish operation, a service description must be published so that a service requester can find the service.

Find: In the find operation, the service requestor retrieves the service description directly. It can be involved in two different lifecycle phases for the service requestor:

- At design, time to retrieve the service's interface description for program development.
- And, at the runtime to retrieve the service's binding and location description for invocation.

Bind: In the bind operation, the service requestor invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact, and invoke the service.

Artifacts of the web service

There are two artifacts of web services:

- Service
- Service Registry

Service: A service is an **interface** described by a service description. The service description is the implementation of the service. A service is a software module deployed on network-accessible platforms provided by the service provider. It interacts with a service requestor. Sometimes it also functions as a requestor, using other Web Services in its implementation.

Service Description: The service description comprises the details of the interface and implementation of the service. It includes its data types, operations, binding information, and network location. It can also categorize other metadata to enable discovery and utilize by service requestors. It can be published to a service requestor or a service registry.

Web Service Implementation:

Requirements Phase: The objective of the requirements phase is to understand the business requirement and translate them into the web services requirement. The requirement analyst should do requirement elicitation (it is the practice of researching and discovering the requirements of the system from the user, customer, and other stakeholders). The analyst should interpret, consolidate, and communicate these requirements to the development team. The requirements should be grouped in a centralized repository where they can be viewed, prioritized, and mined for interactive features.

Analysis Phase: The purpose of the analysis phase is to refine and translate the web service into conceptual models by which the technical development team can understand. It also defines the high-level structure and identifies the web service interface contracts.

Design Phase: In this phase, the detailed design of web services is done. The designers define web service interface contract that has been identified in the analysis phase. The defined web service interface contract identifies the elements and the corresponding data types as well as mode of interaction between web services and client.

Coding Phase: Coding and debugging phase is quite similar to other software component-based coding and debugging phase. The main difference lies in the creation of additional web service interface wrappers, generation of WSDL, and client stubs.

Test Phase: In this phase, the tester performs interoperability testing between the platform and the client's program. Testing to be conducted is to ensure that web services can bear the maximum load and stress. Other tasks like profiling of the web service application and inspection of the SOAP message should also perform in the test phase.

Deployment Phase: The purpose of the deployment phase is to ensure that the web service is properly deployed in the distributed system. It executes after the testing phase. The primary task of deployer is to ensure that the web service has been properly configured and managed. Other optional tasks like specifying and registering the web service with a UDDI registry also done in this phase.

Web Service Protocol Stack:

A second option for viewing the web service architecture is to examine the emerging web service protocol stack. The stack is still evolving, but currently has four main layers.

Service Transport

This layer is responsible for transporting messages between applications. Currently, this layer includes Hyper Text Transport Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), and newer protocols such as Blocks Extensible Exchange Protocol (BEEP).

XML Messaging

This layer is responsible for encoding messages in a common XML format so that messages can be understood at either end. Currently, this layer includes XML-RPC and SOAP.

Service Description

This layer is responsible for describing the public interface to a specific web service. Currently, service description is handled via the Web Service Description Language (WSDL).

Service Discovery

This layer is responsible for centralizing services into a common registry and providing easy publish/find functionality. Currently, service discovery is handled via Universal Description, Discovery, and Integration (UDDI).

- **Service transport** is responsible for actually transporting XML messages between two computers.

Hyper Text Transfer Protocol (HTTP)

Currently, HTTP is the most popular option for service transport. HTTP is simple, stable, and widely deployed. Furthermore, most firewalls allow HTTP traffic. This allows XMLRPC or SOAP messages to masquerade as HTTP messages. This is good if you want to integrate remote applications, but it does raise a number of security concerns is a number of security concerns.

Blocks Extensible Exchange Protocol (BEEP)

This is a promising alternative to HTTP. BEEP is a new Internet Engineering Task Force (IETF) framework for building new protocols. BEEP is layered directly on TCP and includes a number of built-in features, including an initial handshake protocol, authentication, security, and error handling. Using BEEP, one can create new protocols for a variety of applications, including instant messaging, file transfer, content syndication, and network management.

SOAP is not tied to any specific transport protocol. In fact, you can use SOAP via HTTP, SMTP, or FTP. One promising idea is therefore to use SOAP over BEEP.

Result and Discussion:

1) A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system.
2) Phases of Web Services
   A. Requirements
   B. Analysis
   C. Design
   D. Coding
   E. Test
   F. Deployment
3) A second option for viewing the web service architecture is to examine the emerging web service protocol stack.

Learning Outcomes: Students should have the ability to

LO1: Define Web services.

LO2: Identify different phases of web services.

LO3: Explain web service protocol.

Course Outcomes: Upon completion of the course students will be able to know about web services and its implementation.

Conclusion:

In this Experiment, we have understood Web services, its artifacts and different phases.

Viva Questions:

1. Define web services.
2. Explain artifacts of web services.
3. Explain different phases in web service Implementation.
4. Explain Service Transport in web services.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

SHREYANSH SHUKLA
BE COMP B 24
SA EXP 4

Experiment No.: 4

Integrate Software Components using middleware

Learning Objective: Student should be able to integrate software components using middleware

Tool:- Java RMI

Theory:

Software system integration is essential where communication between different applications running on different platform is needed. Suppose a system designed for payroll running with Human Resource System. In that case employees' data need to be inserted in both systems. The system integration benefits a lot in these cases where data and services needed to be shared.

Web services are becoming very popular to share data between systems over the network and over the internet as well. In software industry the software integration carried same steps as software development and hence demands same kind of development procedures and testing.

This ensures the meaningful and clear communication between the systems. Systems integration becomes inevitable in Enterprise Systems where the whole organization needed to share data and services and give the feel to user as one system. The core purpose of integration is to make the systems communicate and also to make the whole system flexible and expandable.

The integration of different softwares written in different language and based on different platforms can be tricky. In that situation a middleware is necessary to enable the communication between different softwares. The middleware enables the software system not only to share data but also share the services

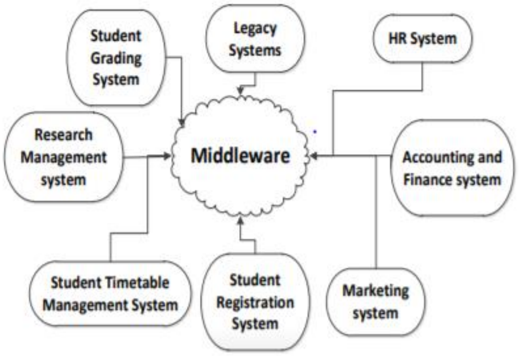

Student Grading System
Legacy Systems
HR System
Research Management system
Middleware
Accounting and Finance system
Student Timetable Management System
Student Registration System
Marketing system

Figure 1: Middleware

A . Middleware

Independently written software systems need to be integrated in large system for example industry, institution, etc. These systems need to agree on common method for integration. To get them communicate there is need to have something in between them. The middle thing is termed commonly as middleware.

B .Service Oriented Architecture

Service Oriented Architecture (SOA) is the architectural design and pattern which is used to provide services to different applications. Its goal is to achieve loose coupling between interacting components of applications. Web Services, Corba, Jini, etc are the technologies used to implement SOA.

Main benefit of SOA is that it can provide the means of communication between completely different applications (built in different technologies). The services are also completely independent and reusable and the nature of reusability provide the less time to market. All the services technologies needs to be implemented using the SOA design pattern and need to be designed on the basis of SOA to get maximum benefit.

C .Web Services

Web service is the SOA technology with additional requirements of using internet protocols (HTTP, FTP, SMTP, etc.) and using XML (Extensible Markup Language) for message transmission. [7] These are application components which communicate between different applications using open protocol. Open protocol is the web protocol for querying and updating information. These components can be used by different kinds of applications to exchange information. HTML (HyperText Markup Language) and XML are the basics of web service implementation.

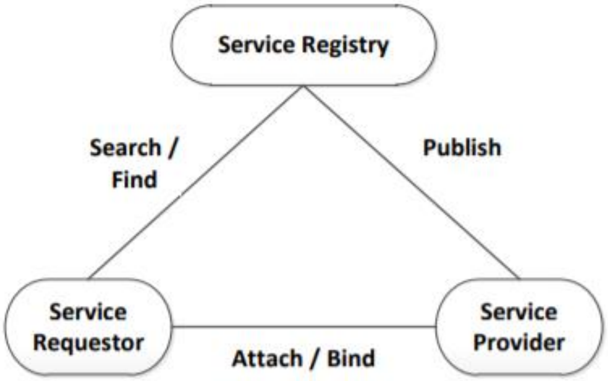


Figure 2: Web-Service Architecture

D.WSDL (Web Service Descriptive Language)

WSDL is a language to describe web services and providing the link to access these web services. It is acting as a publisher in web service architecture. It is the XML document which is recommended by W3C (World Wide Web Consortium) in 2007. In WSDL XML describes the service and the address from where applications can access the service. [8] WSDL predecessors were COM and CORBA

E. UDDI (Universal Description, Discovery and Integration)

It is directory service / registry service where applications can register and look for web services. It is Platform independent framework. It is acting like service register in web service architecture. It uses HTML, XML and DNS (Domain Name Server) protocols which enables it to become the directory service. It defines the keyword search, categories and classification for an application and registers it into business directory. In that way it is making the application easier to be approached by the customers online

F. SOAP (Simple Object Access Protocol)

It is a messaging / invoker in web service architecture. It is use to send messages in between the service and service consumer; and in between service consumer and service registry. It used to communicate with web service. It is a message framework which transfers the information between sender and receiver. SOAP doesn’t define the service but defines the mechanisms for messaging. It binds the client to the service

G.SOA - A solution to spaghetti architecture

In a fairly medium scale to large software architecture where there is need to integrate or communicate between different kinds of applications the introduction of links can make the architecture messy and it is called spaghetti architecture [9]. Figure 4 shows that problem in detail. It makes the system less flexible and expandability is the nightmare. Service Oriented Architecture (SOA) makes the architecture flexible and expandable.

Result and Discussion:

Different types of Web service Integration were understood

Learning Outcomes: Students should have be able to

LO1: Define Middleware.

LO2: Identify different components in middleware.

LO3: Explain Software Components using middleware.

Course Outcomes: Upon completion of the course students will be able to understand middleware and its components.

Conclusion:

Independently written software systems need to be integrated in large system for example industry, institution, etc. These systems need to agree on common method for integration. To get them communicate there is need to have something in between them. The middle thing is termed commonly as middleware.

Viva Questions:

1. Define Middleware.
2. Explain Service Oriented Architecture.
3. Explain Web Services.
4. Explain Universal Description, Discovery and Integration.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

SHREYANSH SHUKLA
BE COMP B 24
SA EXP - 5

Experiment No.: 5

Use middleware to implement connectors

Learning Objective: Student should be able to understand use of middleware to implement connectors.

Theory:

What is Middleware ?

Middleware is a more effective program that acts as bridge in between various applications and other databases otherwise tools. It is placed in between operating system and other applications which run on it. Middleware allows making better communication, application services, messaging, authentication, API management and management of data between different kinds of applications which help to exchange data.

The connectors sit between the two APIs or you can say and the ends of the connectors are APIs. The connectors receive data from one app/solution and process it to make it understandable and accessible in the other app/solution, regardless of whether any direct form of integration was available in the two apps.

Role of Middleware is :-

Middleware is a potentially useful tool when building software connectors. First, it can be used to bridge thread, process and network boundaries. Second, it can provide pre-built protocols for exchanging data among software components or connectors. Finally, some middleware packages include features of software connectors such as filtering, routing, and broadcast of messages or other data.

A signal interaction is a one-way interaction between an initiating object, called a client, and a responding object, called a server. An operation interaction is an interaction between a client object and server object that is either an interrogation or an announcement. An interrogation is composed of two one-way interactions: a request and a response. An announcement is a one-way request from a client object to a server object in which the client object expects no response, and the server object does not respond. A flow interaction is an ordered set of one or more one-way communications from a producer object to a consumer object. These interactions are a generalized metamodel for describing communication styles between objects that hat can be implemented using a variety of middleware such as Remote Procedure Call (RPC) and Remote Method Invocation (RMI) and message queues (as selected and specified in the technology view).

can be implemented using a variety of middleware such as Remote Procedure Call (RPC) and Remote Method Invocation (RMI) and message queues.

Connectors as a primary vehicle for interprocess communication. A single conceptual connector can be “broken up” vertically (a) or horizontally (b) for this purpose.

Vertical Connectors :

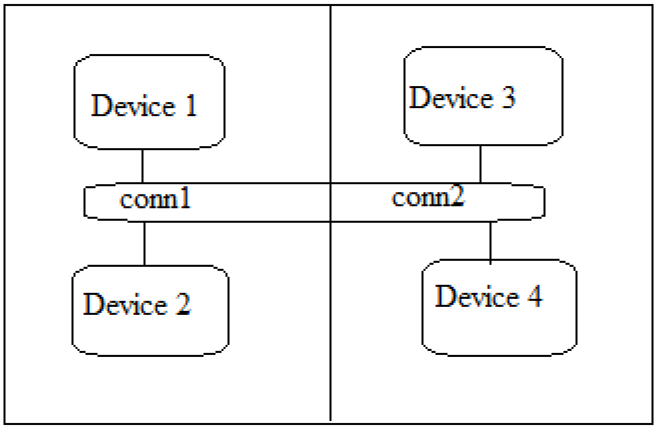


Figure (a)

Horizontal Connectors :

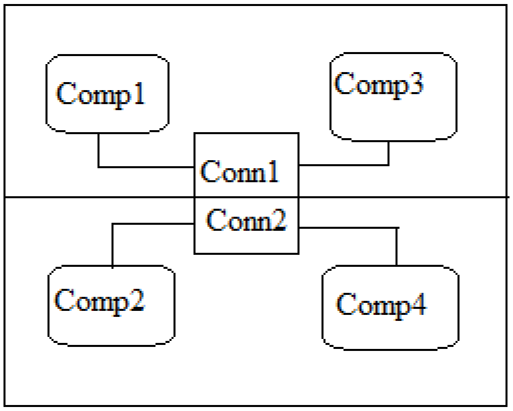


Figure (b)

Linking Ports Across Process Boundaries :

The ports can call methods on each other, sending messages as method parameters. Our intent was to simply use the middleware to exchange port references across process boundaries and use the existing technique for message passing.

the middleware technology would be entirely encapsulated within the port entity and would not be visible to architects or developers. The singleprocess implementation of a C2 connector links two ports together by having each port contain a reference to the other one.

Linking Connectors Across Process Boundaries :

Sharing communication ports across process boundaries gave us fine-grained control over implementing an architecture as a multi-process application. However, it required additional functionality in the C2 implementation framework and did not isolate the change to the appropriate abstraction: the connector. In order to remedy this, we devised two connectorbased approaches. Both of these approaches consist of implementing a single conceptual software connector using two or more actual connectors that are linked across process or network boundaries. Each actual connector thus becomes a segment of a single “ virtual connector.” All access to the underlying middleware technology is encapsulated entirely within the abstraction of a connector, meaning that it is unseen by both architects and developers. We call the first approach “ lateral welding,” depicted in Fig. 2a. Messages sent to any segment of the multi-process connector are broadcast to all other segments. Upon receiving a message, each segment has the responsibility of filtering and forwarding it to components in its process as appropriate. Only messages are sent across process boundaries. While the lateral welding approach allowed us to “ vertically slice” a C2 application, we also developed an approach to “ horizontally slice” an application, as shown in Fig. 2b. This approach is similar to the idea of lateral welding: a conceptual connector is broken up into top and bottom segments, each of which exhibits the same properties as a single-process connector to the components attached above and below it, respectively. However, the segments themselves are joined using the appropriate middleware. When used with a middleware technology that supports dynamic change at run-time, all of these approaches, both using ports and connectors, can be used to build applications where processes can join and leave a running application.

Using Middleware Technologies :

To explore the use of OTS middleware with software connectors, we chose four representative technologies from the field of available middleware packages. These were Q, an RPC system, Polylith, a message bus, RMI, a connection mechanism for Java objects, and ILU, a distributed objects package.A description of one of our efforts involving integrating two middleware technologies simultaneously in the same application is given here. With each middleware package, we were able to encapsulate all the middleware functionality within the connectors themselves. This means that architects and developers can use the middleware-enhanced connectors thus created just as they would use normal, in-process C2 connectors.

Simultaneous Use of Multiple Middleware Packages :

Each middleware technology we evaluated has unique benefits. By combining multiple such technologies in a single application, the application can potentially obtain the benefits of all of them. For instance, a middleware technology that supports multiple platforms but only a single language, such as RMI, could be combined with one that supports multiple languages but a single platform, such as Q, to create an application that supports both multiple languages and multiple platforms. The advantages of combining multiple middleware technologies within software connectors are manifold. In the absence of a single panacea solution that supports all required platforms, languages, and network protocols, the ability to leverage the capabilities of several different middleware technologies significantly widens the range of applications that can be implemented within an architectural style such as C2. We combined our implementations of ILU-C2 and RMI-C2 connectors in a version of the KLAX application, a real time video game application built as an experimental platform for work on the C2 architecture. We were able to do so with no modification to the middleware-enhanced C2 framework or the connectors themselves by combining the lateral welding technique shown in Fig. 2a with the horizontal slicing technique shown in Fig. 2b. This approach works for any combination of OTS connectors that use the lateral welding technique. An alternative approach would have been to create a single connector that supported both ILU and RMI, but this would have required changes to the framework.

Result and Discussion: Because software connectors provide a uniform interface to other connectors and components within an architecture, architects need not be concerned with the properties of different middleware technologies as long as the technology can be encapsulated within a software connector.

AppMain.java

```java
package com.airhacks;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;
import java.util.Scanner;

public class AppMain {

    public static void main(String[] args) throws Exception {

        Calculator calculator = new Calculator();
        registerWithJmxAgent(calculator);
        startConsoleApp(calculator);
    }

    private static void startConsoleApp(Calculator calculator) {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("-----------------");
            String input1 = getUserInput(scanner, "enter first number");
            double d1 = toDouble(input1);

            String input2 = getUserInput(scanner, "enter second number");
            double d2 = toDouble(input2);

            double sum = calculator.add(d1, d2);
            System.out.printf("sum = %s (rounded to %s decimal places)%n",
                    sum, calculator.getDecimalPlaces());
        }
    }

    private static double toDouble(String input) {
        try {
            return Double.parseDouble(input);
        } catch (NumberFormatException e) {
            System.out.println("Not a valid number, defaulting to 0");
            return 0;
        }
    }

    private static void registerWithJmxAgent(Calculator calculator) throws Exception {
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        ObjectName name = new ObjectName("com.airhacks:type=calculator");
        mbs.registerMBean(calculator, name);
    }

    public static String getUserInput(Scanner scanner, String msg) {
        System.out.print(msg + ">");
        String s = scanner.nextLine();
        if ("exit".equals(s)) {
            System.exit(0);
        }
        return s;
    }
}
```

Calculator.java

```java
package com.airhacks;

import java.math.BigDecimal;
import java.math.RoundingMode;

public class Calculator implements CalculatorMBean {
    private int decimalPlaces = 2;

    public double add(double d1, double d2) {
        BigDecimal bd1 = new BigDecimal(d1);
        BigDecimal bd2 = new BigDecimal(d2);

        BigDecimal sum = bd1.add(bd2);
        return sum.setScale(decimalPlaces, RoundingMode.HALF_UP)
                  .doubleValue();
    }

    public void setDecimalPlaces(int decimalPlaces) {
        this.decimalPlaces = decimalPlaces;
    }

    public int getDecimalPlaces() {
        return decimalPlaces;
    }
}
```

CalculatorMBean.java

```
package com.airhacks;

public interface CalculatorMBean {

    void setDecimalPlaces(int decimalPlaces);

    int getDecimalPlaces();
}
```

Learning Outcomes: Students should have be able to

LO1: Define Middleware.

LO2: Identify different connectors in middleware.

LO3: Explain middleware implements in connectors.

Course Outcomes: Upon completion of the course students will be able to understand middleware and its connectors.

Conclusion:

Middleware is a more effective program that acts as bridge in between various applications and other databases otherwise tools. It is placed in between operating system and other applications which run on it.

Viva Questions:

1. Define Middleware.
2. Explain use of middleware.
3. Explain implementation of connectors.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

SHREYANSH SHUKLA
BE COMP B 24
SA – Exp 6

Experiment No.: 6

Wrapper to connect two applications with different architectures

Learning Objective: Student should be able to understand wrapper to connect two applications with different architectures.

Theory:

Wrapper Architecture:

In wrapper, there is a clear separation of concerns: the mediator deals with data source distribution while the wrappers deal with data source heterogeneity and autonomy.

This is achieved by using a common language between mediator and wrappers, and the translation to the data source language is done by the wrappers (as shown in the diagram).

Each data source has an associated wrapper that exports information about the source schema, data and query processing capabilities. To deal with the heterogeneous nature of data sources, wrappers transform queries received from the mediator, expressed in a common query language, to the particular query language of the source. A wrapper supports the functionality of translating queries appropriate to the particular server, and reformatting answers (data) appropriate to the mediator. One of the major practical uses of wrappers has been to allow an SQL-based DBMS to access non-SQL databases.

The mediator centralizes the information provided by the the wrappers in a unified view of all available data. This unified view can be of two fundamental types local-as-view (LAV) and global-as-view (GAV). In LAV, the global schema definition exists, and each data source schema is treated as a view definition over it. In GAV on the other hand, the global schema is defined as a set of views over the data source schemas. These views indicate how the elements of the global schema can be derived, when needed, from the elements of the data source schemas. The main functionality of the mediator is to provide uniform access to multiple data sources and perform query decomposition and processing using the wrappers to access the data sources.

Software encapsulation is based on the technology of wrapping. When asked what to do with the existing legacy software, in a new object-oriented architecture, his answer was to wrap it. Since then, there have been dozens of papers written on the subject in the object-oriented literature, but hardly any in the field of reverse- and re-engineering. One of the best technical discussions of the subject is to be found in the book “The Essential CORBA” by Mowbray and Zahari. According to these authors, an object wrapper provides access to a legacy system through an encapsulation layer. The encapsulation exposes only the attributes and operations desired by the software architect. The same authors go on to describe seven techniques for implementing a wrapper:

– remote procedure calls,

– file transfers,

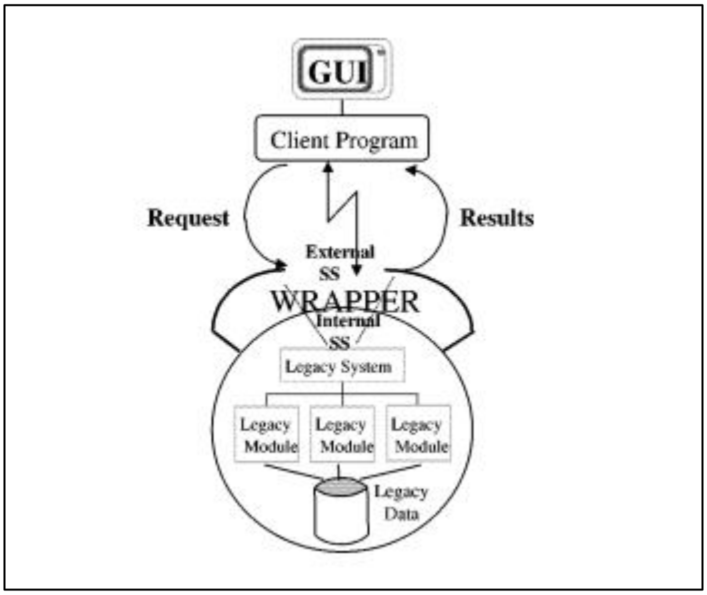
– sockets or docking,

– application program interfaces,

– script procedures,

– macros,

and – common headers.



Wrapper class between Client and server Architecture

These techniques can be implemented independently or in combination with one another to build a connection between the requester of a service and the service provider.

Database wrappers are gateways to existing databases. They allow client applications implemented in a modern object-oriented language to access data stored in a legacy database. System service wrappers provide a customized access to standard system services such as printing, sorting, routing and queuing. It is possible for a user program to invoke such services without knowledge of their internal interfaces. Application wrappers encapsulate batch processes or online transactions. They allow new client applications to include the legacy components as objects which can be called to perform certain tasks such as producing a report or updating a file. Function wrappers offer an interface to invoke individual functions within a wrapped program. Not the program as a whole but only certain parts of it can be invoked from the client application. This amounts to a limited access.

All wrapper uses some kind of message passing mechanism to connect themselves to their clients. As a rule, the wrapper is in the same address space as the wrapped object. On the input side, it receives incoming requests, re-formats them, loads the object and invokes it with the reformatted arguments. On the output side, it takes the results from the wrapped object, re-formats them and sends them back to the requester. This is, in essence, what wrapping is all about.

Adapting a program for wrapping: It would be ideal if programs could be encapsulated without making any changes to them whatsoever. However, even in the case of module wrapping some change has to be made. Since the programs to be wrapped should also continue to operate in the normal mode, it is unlikely that they can be adapted manually. The risk of error is too high and the adaptation has to be repeated after every change to the program. So, if wrapping is to be done on a wide scale, the program adaptation has to be automated.

This will require four tools:

– a transaction wrapper,

– a program wrapper,

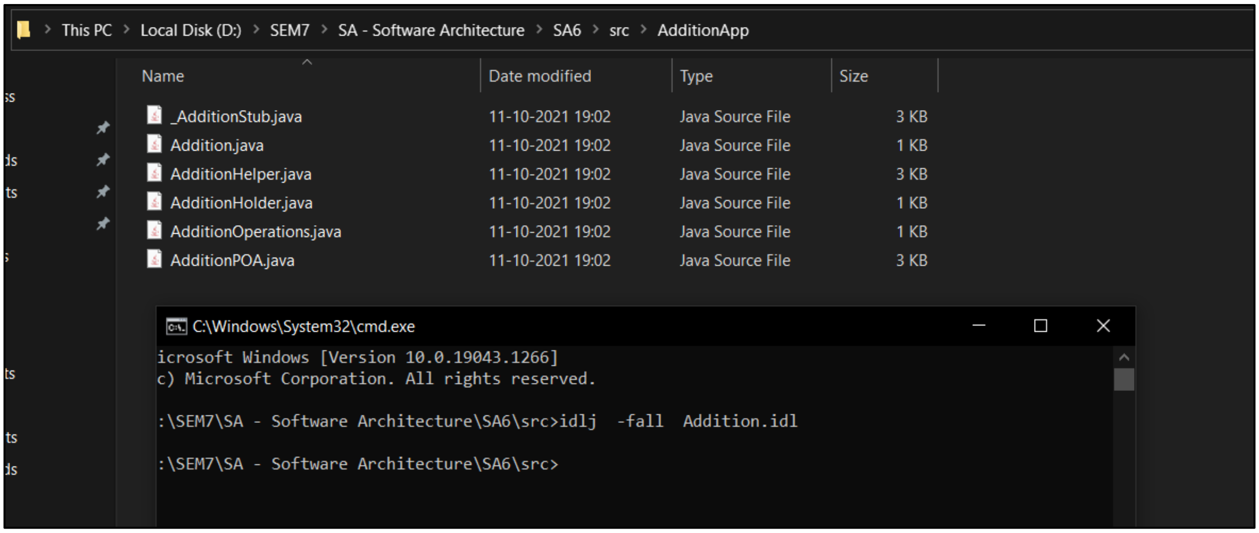
– a module wrapper, and

– a procedure wrapper.

Result and Discussion:

In this Experiment, we studied Wrapper Architecture and types of Wrappers.
In wrapper, there is a clear separation of concerns: the mediator deals with data source distribution while the wrappers deal with data source heterogeneity and autonomy.

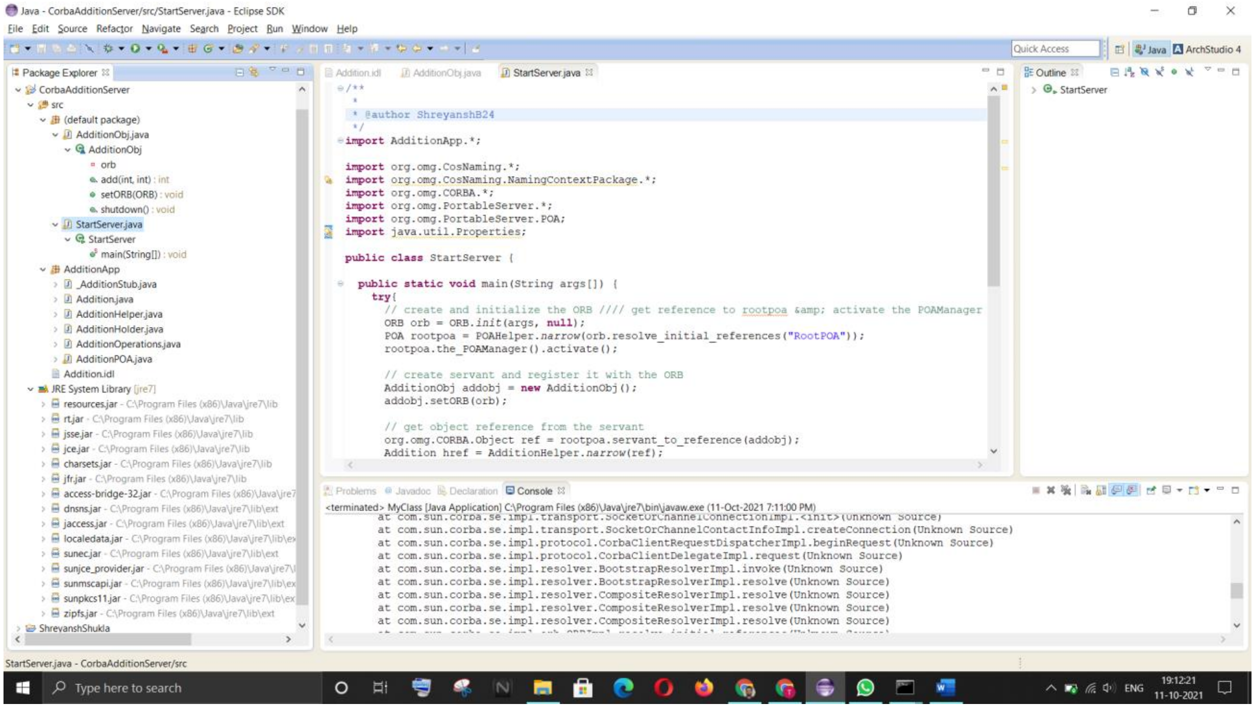
Database wrappers are gateways to existing databases. They allow client applications implemented in a modern object-oriented language to access data stored in a legacy database.


This PC > Local Disk (D:) > SEM7 > SA - Software Architecture > SA6 > src > AdditionApp
C:\Windows\System32\cmd.exe

Name	Date modified	Type	Size
_AdditionStub.java	11-10-2021 19:02	Java Source File	3 KB
Addition.java	11-10-2021 19:02	Java Source File	1 KB
AdditionHelper.java	11-10-2021 19:02	Java Source File	3 KB
AdditionHolder.java	11-10-2021 19:02	Java Source File	1 KB
AdditionOperations.java	11-10-2021 19:02	Java Source File	1 KB
AdditionPOA.java	11-10-2021 19:02	Java Source File	3 KB

```
icrosoft Windows [Version 10.0.19043.1266]
c) Microsoft Corporation. All rights reserved.

:\SEM7\SA - Software Architecture\SA6\src>idlj  -fall  Addition.idl

:\SEM7\SA - Software Architecture\SA6\src>
```

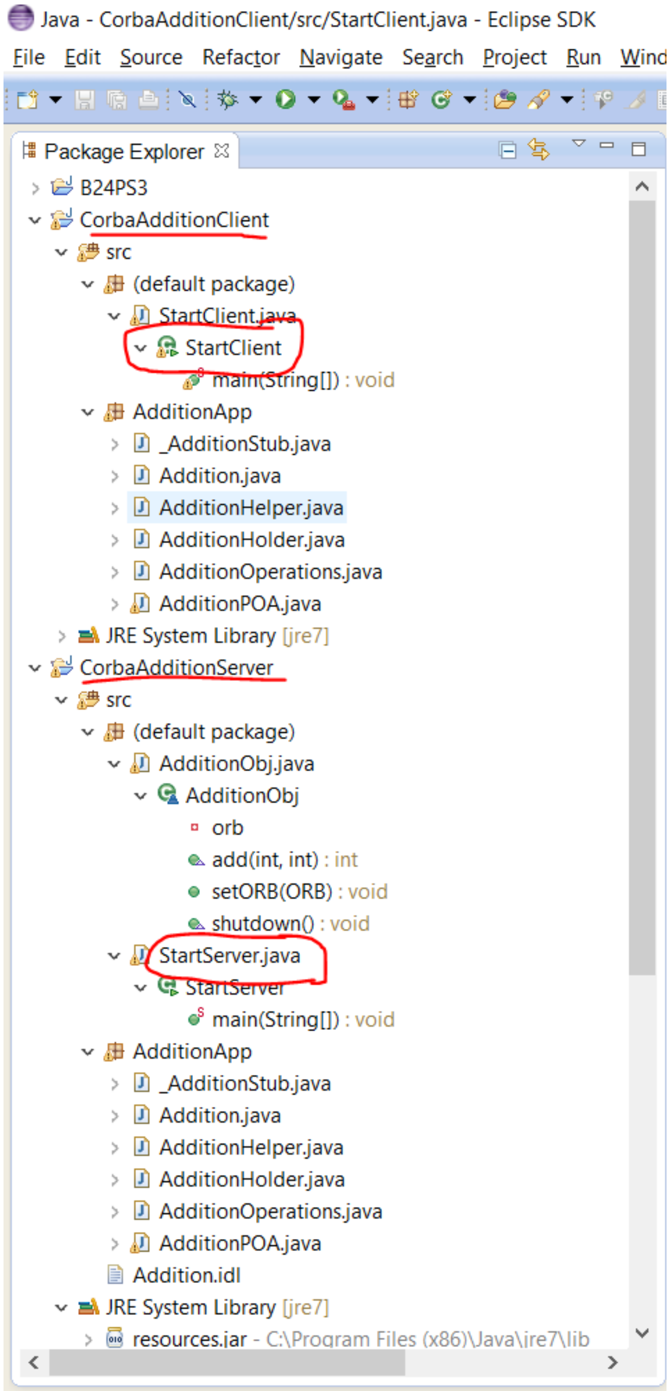
Creation of Stubs & ORB via idlj command


Java - CorbaAdditionServer/src/StartServer.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
ArchStudio 4
Package Explorer
CorbaAdditionServer
src
(default package)
AdditionObj.java
AdditionObj
orb
add(int, int) : int
setORB(ORB) : void
shutdown() : void
StartServer.java
StartServer
main(String[]) : void
AdditionApp
_AdditionStub.java
Addition.java
AdditionHelper.java
AdditionHolder.java
AdditionOperations.java
AdditionPOA.java
Addition.idl
JRE System Library [jre7]
resources.jar - C:\Program Files (x86)\Java\jre7\lib
rt.jar - C:\Program Files (x86)\Java\jre7\lib
jsse.jar - C:\Program Files (x86)\Java\jre7\lib
jce.jar - C:\Program Files (x86)\Java\jre7\lib
charsets.jar - C:\Program Files (x86)\Java\jre7\lib
jfr.jar - C:\Program Files (x86)\Java\jre7\lib
access-bridge-32.jar - C:\Program Files (x86)\Java\jre7
dnsns.jar - C:\Program Files (x86)\Java\jre7\lib\ext
jaccess.jar - C:\Program Files (x86)\Java\jre7\lib\ext
localedata.jar - C:\Program Files (x86)\Java\jre7\lib\ex
sunec.jar - C:\Program Files (x86)\Java\jre7\lib\ext
sunjce_provider.jar - C:\Program Files (x86)\Java\jre7\l
sunmscapi.jar - C:\Program Files (x86)\Java\jre7\lib\ex
sunpkcs11.jar - C:\Program Files (x86)\Java\jre7\lib\ex
zipfs.jar - C:\Program Files (x86)\Java\jre7\lib\ext
ShreyanshShukla
Addition.idl
AdditionObj.java
StartServer.java
Outline
StartServer
Problems
Javadoc
Declaration
Console
StartServer.java - CorbaAdditionServer/src
Type here to search
ENG
19:12:21
11-10-2021

```
/**
 *
 * @author ShreyanshB24
 */
import AdditionApp.*;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

public class StartServer {

  public static void main(String args[]) {
    try{
      // create and initialize the ORB //// get reference to rootpoa &amp; activate the POAManager
      ORB orb = ORB.init(args, null);
      POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();

      // create servant and register it with the ORB
      AdditionObj addobj = new AdditionObj();
      addobj.setORB(orb);

      // get object reference from the servant
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(addobj);
      Addition href = AdditionHelper.narrow(ref);
```

```
<terminated> MyClass [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (11-Oct-2021 7:11:00 PM)
        at com.sun.corba.se.impl.transport.SocketOrChannelConnectionImpl.<init>(Unknown Source)
        at com.sun.corba.se.impl.transport.SocketOrChannelContactInfoImpl.createConnection(Unknown Source)
        at com.sun.corba.se.impl.protocol.CorbaClientRequestDispatcherImpl.beginRequest(Unknown Source)
        at com.sun.corba.se.impl.protocol.CorbaClientDelegateImpl.request(Unknown Source)
        at com.sun.corba.se.impl.resolver.BootstrapResolverImpl.invoke(Unknown Source)
        at com.sun.corba.se.impl.resolver.BootstrapResolverImpl.resolve(Unknown Source)
        at com.sun.corba.se.impl.resolver.CompositeResolverImpl.resolve(Unknown Source)
        at com.sun.corba.se.impl.resolver.CompositeResolverImpl.resolve(Unknown Source)
        at com.sun.corba.se.impl.resolver.CompositeResolverImpl.resolve(Unknown Source)
```

Server Running


Java - CorbaAdditionClient/src/StartClient.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Wind
Package Explorer

```
> B24PS3
v CorbaAdditionClient
  v src
    v (default package)
      v StartClient.java
        v StartClient
            main(String[]) : void
    v AdditionApp
      > _AdditionStub.java
      > Addition.java
      > AdditionHelper.java
      > AdditionHolder.java
      > AdditionOperations.java
      > AdditionPOA.java
  > JRE System Library [jre7]
v CorbaAdditionServer
  v src
    v (default package)
      v AdditionObj.java
        v AdditionObj
            orb
            add(int, int) : int
            setORB(ORB) : void
            shutdown() : void
      v StartServer.java
        v StartServer
            main(String[]) : void
    v AdditionApp
      > _AdditionStub.java
      > Addition.java
      > AdditionHelper.java
      > AdditionHolder.java
      > AdditionOperations.java
      > AdditionPOA.java
      Addition.idl
  v JRE System Library [jre7]
    > resources.jar - C:\Program Files (x86)\Java\jre7\lib
```

Creation of Server & Client

Learning Outcomes: Students should have been able to

LO1: Define Wrapper.

LO2: Identify different types of wrappers.

LO3: Explain adapting a program for wrapping.

Course Outcomes: Upon completion of the course students will be able to understand wrapper to connect two applications with different architectures.

Conclusion:

- Adapting a program for wrapping: It would be ideal if programs could be encapsulated without making any changes to them whatsoever. However, even in the case of module wrapping some change has to be made. Since the programs to be wrapped should also continue to operate in the normal mode, it is unlikely that they can be adapted manually. The risk of error is too high and the adaptation has to be repeated after every change to the program. So, if wrapping is to be done on a wide scale, the program adaptation has to be automated.

Viva Questions:

1. Define wrapping.
2. Explain implementation of wrapping.
3. Explain implementation of wrapping of two different architectures.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

SHREYANSH SHUKLA
BE COMP B 24
SA – Exp 7

Experiment No.: 7

Identifying Design requirements for an Architecture for any specific domain.

Learning Objective: Student should be able to understand Design requirements for an Architecture for any specific domain.

Theory:

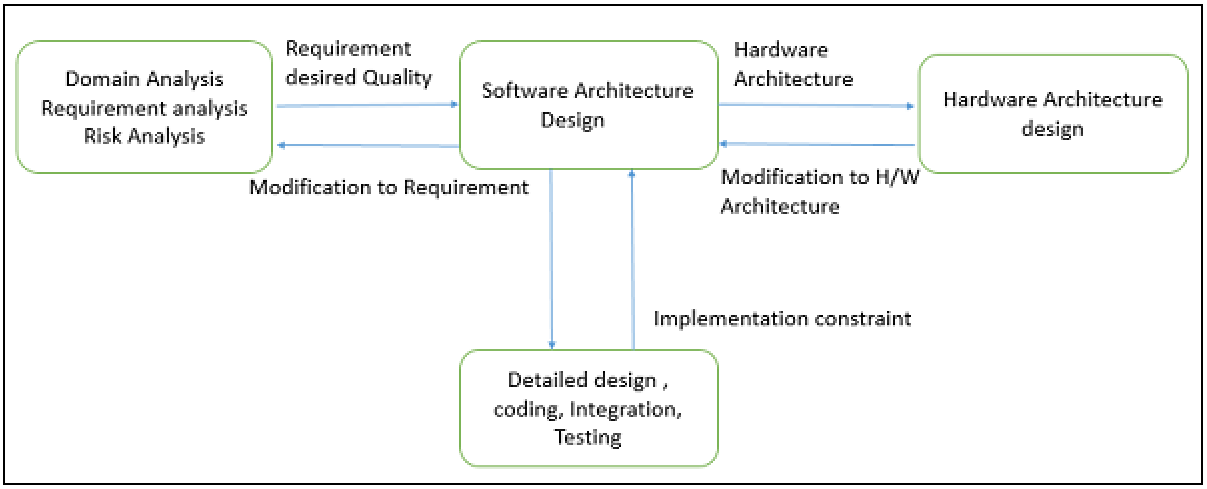
Software design:

Software design provides a design plan that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows −

To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.

Act as a blueprint during the development process.

Guide the implementation tasks, including detailed design, coding, integration, and testing.



It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.

Software design is responsible for the code level design such as, what each module is doing, the classes scope, and the functions purposes, etc. When used strategically, they can make a programmer significantly more efficient by allowing them to avoid reinventing the wheel, instead using methods refined by others already. They also provide a useful common language to conceptualize repeated problems and solutions when discussing with others or managing code in larger teams.

Major artifacts of the software design process include:

- **Software requirements specification.** This document describes the expected behavior of the system in the form of functional and non-functional requirements. These requirements should be clear, actionable, measurable, and traceable to business requirements. Requirements should also define how the software should interact with humans, hardware, and other systems.
- **High-level design.** The high-level design breaks the system's architectural design into a less-abstracted view of sub-systems and modules and depicts their interaction with each other. This high-level design perspective focuses on how the system, along with all its components, implements in the form of modules. It recognizes the modular structure of each sub-system and their interaction among one another.
- **Detailed design.** Detailed design involves the implementation of what is visible as a system and its sub-systems in a high-level design. This activity is more detailed towards modules and their implementations. It defines a logical structure of each module and their interfaces to communicate with other modules.

The purpose of an architecture schema or design record is to serve as a vehicle for software understanding by functioning as a collection point for knowledge about the components that make up a DSSA. In particular, the design record organizes

- domain- specific knowledge about components or design alter natives and
- Implementation in knowledge about alternate implementations,

The primary goal of a design record is to adequately describe the components in a reference architecture such that the application engineer can make design decisions and component selections without looking at inn implementations. The secondary goal of a design record is to provide information that the tools in the supporting environment can use.

The design record data elements used by Loral Federal Systems

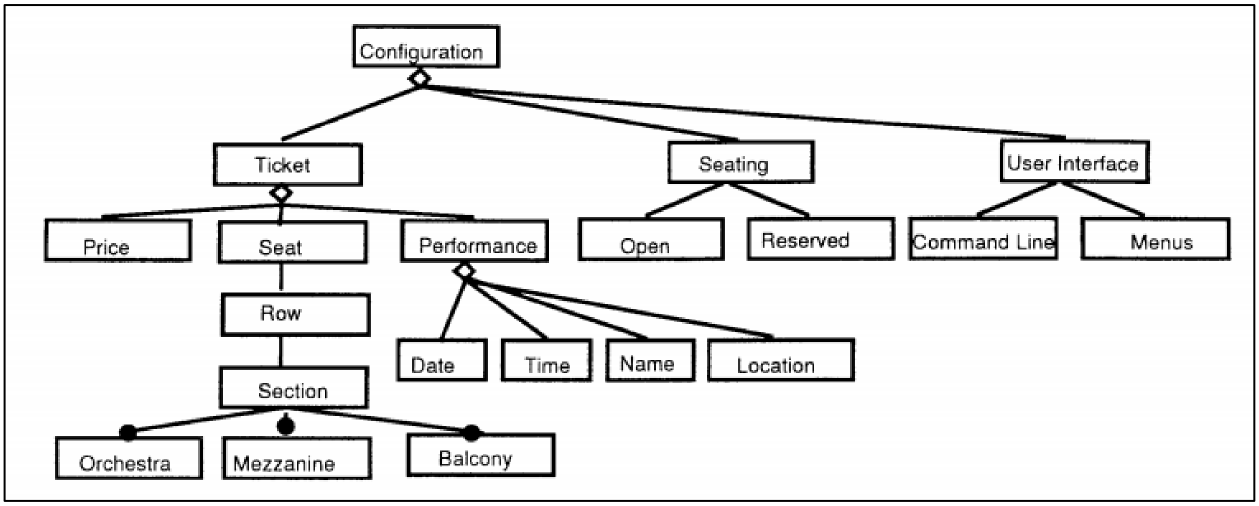
phases in the software life cycle, include:

1. Name/type

2. Description

3. reference requirements satisfied,
4. design structure (data flow and control flow diagrams),
5. design rationale,
6. interface and architecture specifications and dependencies,
7. P D L (program Design Language) text,
8. implementation,
9. configuration and version data, and

10. test cases.
11. metric data,
12. access rights,
13. search points,
14. catalog information,
15. library and DSSA links, and hypertext paths

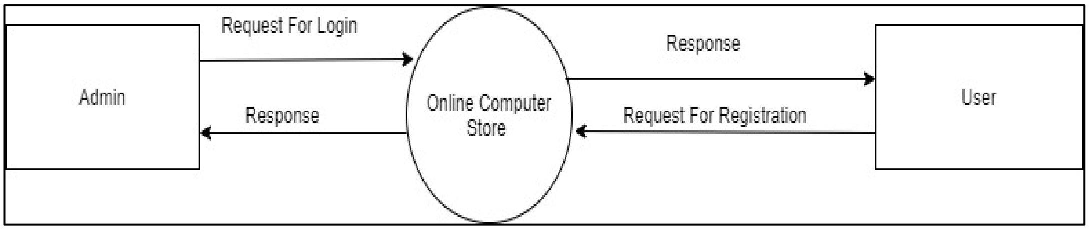


Many different programming formats incorporate the same essential elements. In all cases, the design programming fits within a larger context of planning efforts which can also be programmed. For design programming for a building, we propose a six-step process as follows:

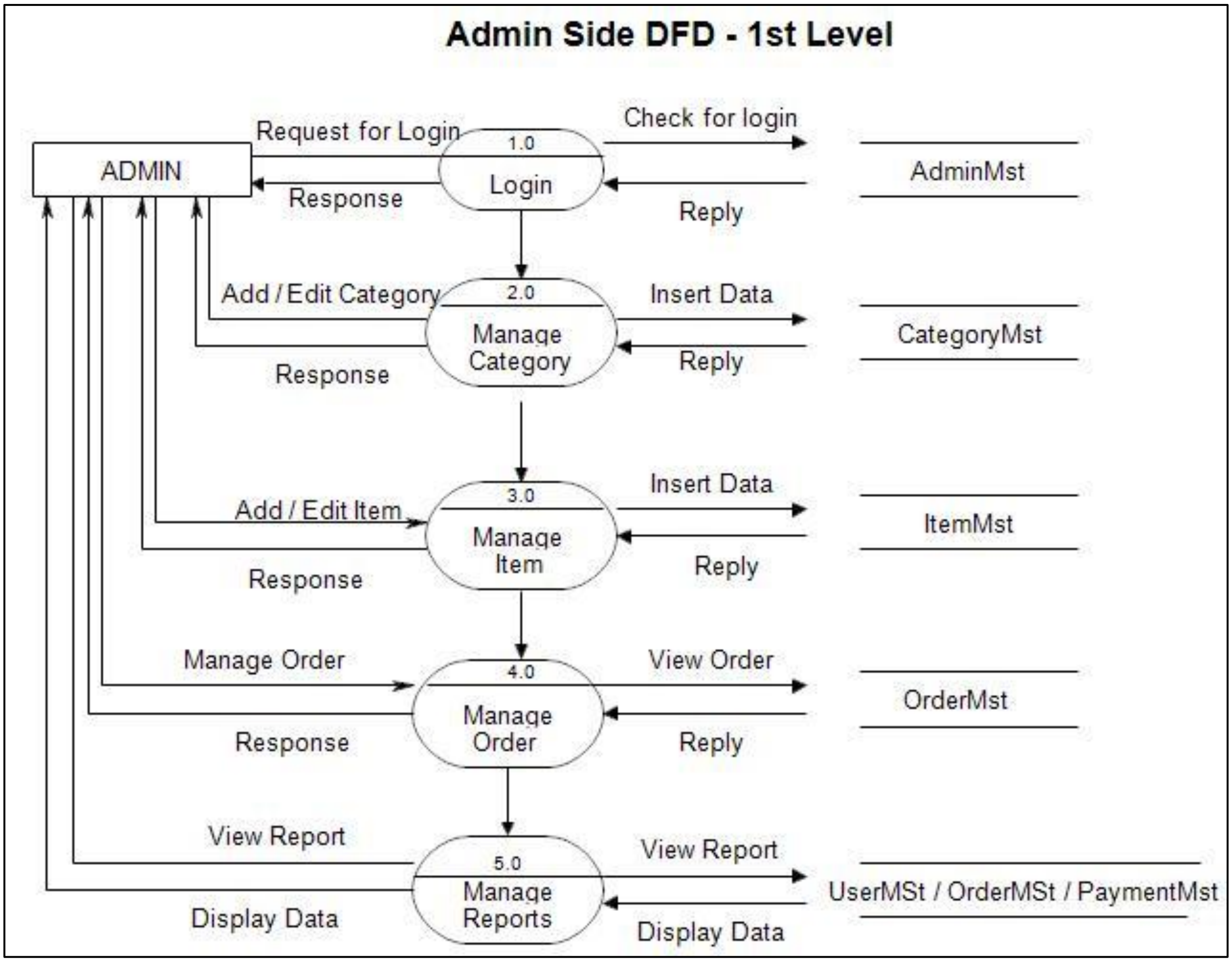
1. Research the project type
2. Establish goals and objectives
3. Gather relevant information
4. Identify strategies
5. Determine quantitative requirements
6. Summarize the program

Result and Discussion:

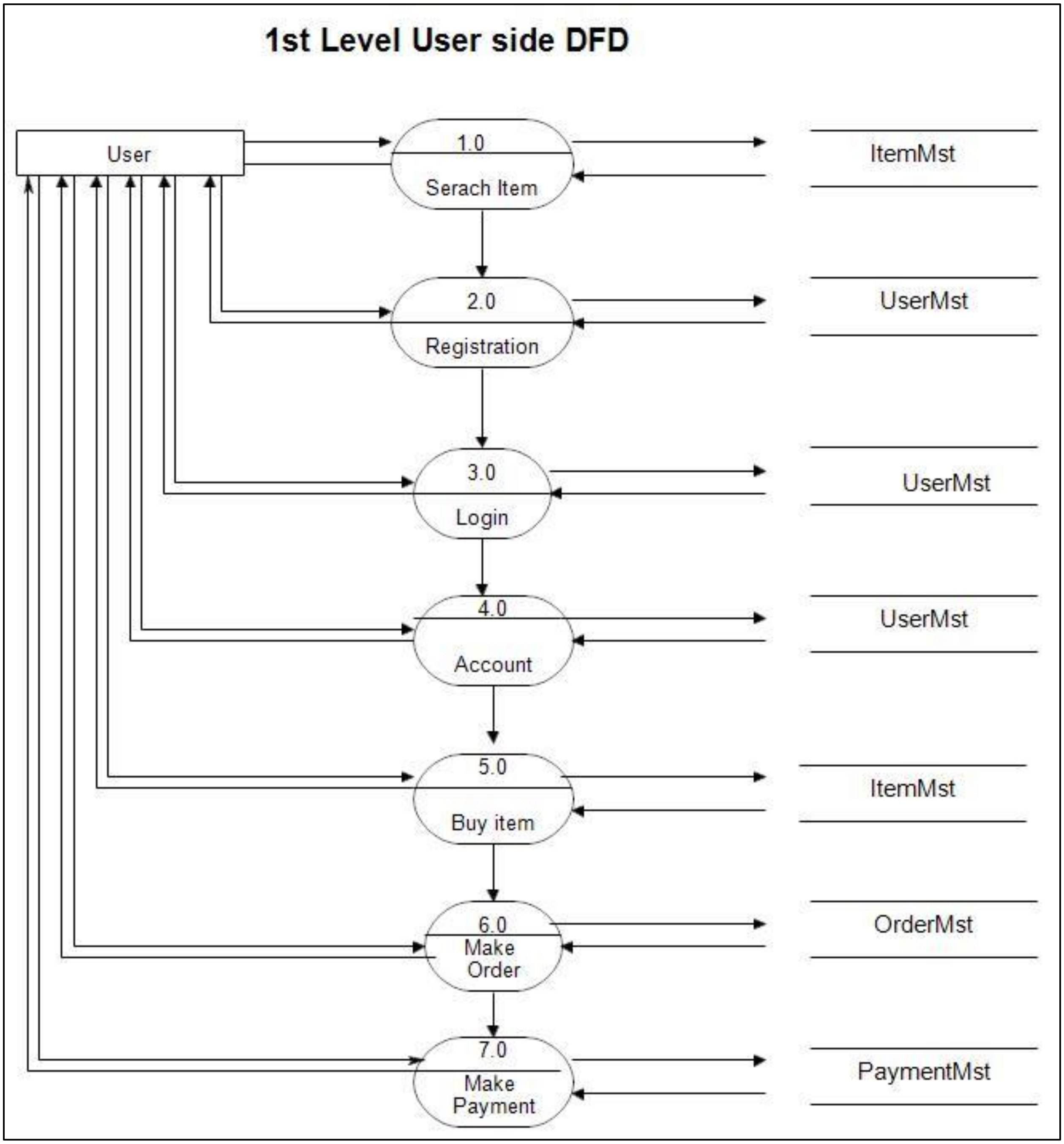
Project Title: Online Computer Store (E- Commerce)



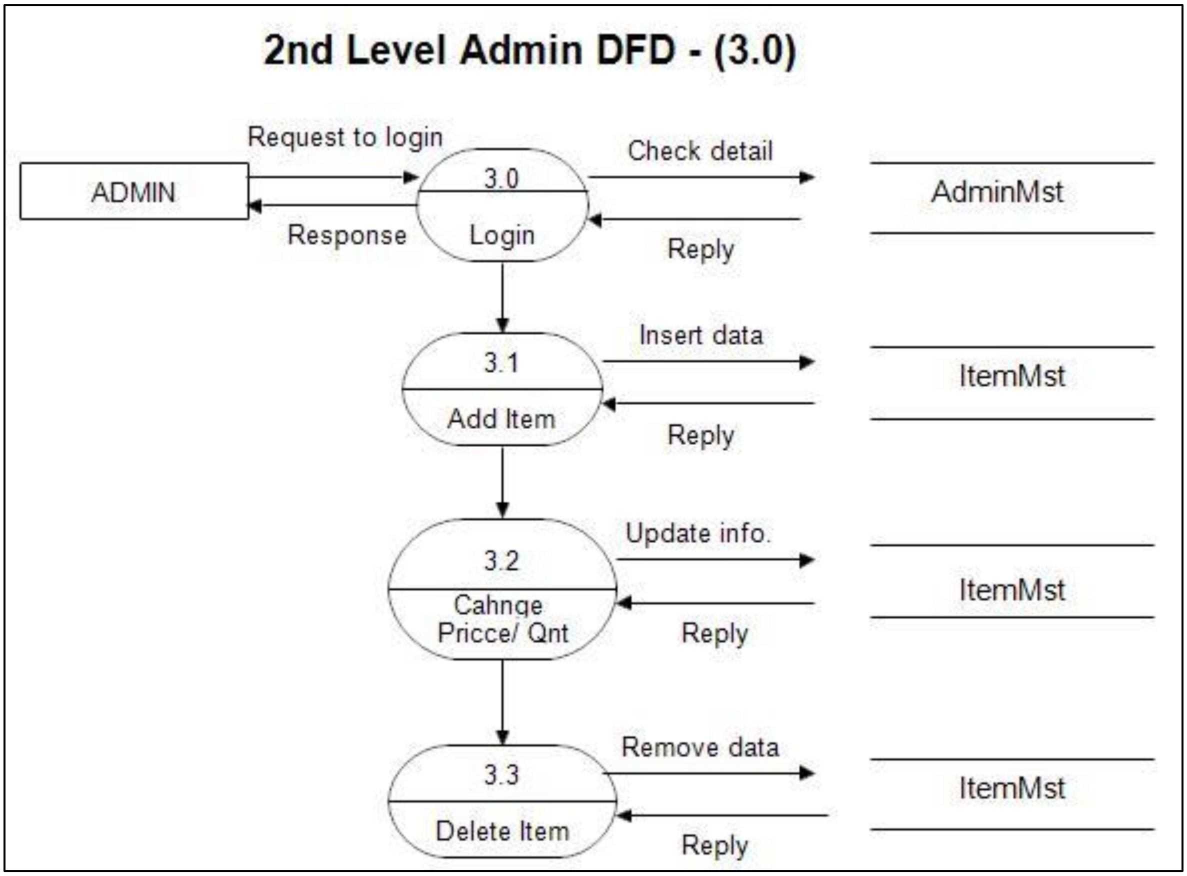
Context Diagram / Level – 0 DFD



1st Level – Admin Side Data Flow Diagram



<u>1st Level - User side DFD</u>



2nd Level – Admin side DFD (3.0)

Learning Outcomes: Students should have been able to understand

LO1: Define software design.

LO2: Identify different design requirements for software.

LO3: Explain implementation of design requirements of software.

Course Outcomes: Upon completion of the course students will be able to understand design requirement of software Architecture.

Conclusion:

1. In this experiment we created a DFD for Online Computer Store.
2. Data Flow Diagram (DFD) provides a visual representation of the flow of information (i.e., data) within a system. By drawing a Data Flow Diagram, you can tell the information provided by and delivered to someone who takes part in system processes, the information needed to complete the processes and the information needed to be stored and accessed.

Viva Questions:

1. Define design requirement in software.
2. Explain different design requirements in software Architecture.
3. Explain the phases of programming design.
4. Explain any three design requirements.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	

SHREYANSH SHUKLA
BE COMP B 24
SA – Exp 8

Experiment No.: 8

Identifying System requirements for an Architecture for any specific domain.

Learning Objective: Student should be able to understand System requirements for an Architecture for any specific domain**.**

Theory:

The purpose of a requirements architecture is to structure and organize requirements in such a way that the requirements are stable, usable, adapt to changes, and are elegant. When a requirements architecture is sound, it helps facilitate better design of the system it attempts to describe. When a requirements architecture is faulty, it can cause problems. When the requirements architecture is poor, the following problems result:

i. No one knows why a requirement was changed
ii. Requirements cannot be reused
iii. Traceability is superficial or unused by other teams
iv. Requirements reviews involve irrelevant information
v. Big picture of the system being built and reasons for building it are not well-understood
vi. It is important to keep in mind that the purpose of a good requirements architecture is to build working software that meets business objectives.

Software requirements must be testable, unambiguous, and concise, a requirements architecture must also possess certain attributes. The above blueprint provides some general guidelines for how to structure requirements, but keeping in mind the following attributes:

a. **Maintainable**: Whatever choices you make in organizing requirements, ensure that you create a structure that can adapt to changes in requirements.
b. **Traceable**: Do you know which requirements any given process flow step is traced to?
c. **Usable**: Consider the stakeholders in the org chart—are the requirements architected in such a way that you could either produce output for each of them or such that they could navigate to the requirements in the tool and find the requirements objects that are relevant to them? The hierarchies and traces you create should be consistent: Don’t create one hierarchy where the FRs are children of the models and another hierarchy for the same project where

FRs are not children of the models but are traced to them. The absolute worst thing to do is to list all requirements objects in a flat list or to manage your requirements in word or excel.

d. **Scalable**: Imagine your requirements architecture with 10 times the number of requirements it has. Now imagine it with 100 times the number of requirements. Architectures should be able to support the addition of new requirements with minimal overhead.

e. **Elegant**: Are there just enough hierarchies to facilitate use? Are you repeating hierarchies just to make traceability easier? Does your architecture contain duplicate models or requirements?

f. **Generalizable**: The architecture approach should be repeatable. You ought to be able to go into any project and no matter the domain uses the same approach to requirements architecture.

All architectures are tradeoffs – like in software architecture, you may need to sometimes sacrifice aesthetics for robust traceability or reuse. Or you may sacrifice usability for ease of exporting to external formats. Understand the tradeoffs you are making with your requirements architecture.

Domain-specific Software Development Various mechanisms of domain-specific software development are under investigation within the projects. Compositional mechanisms facilitate reuse of existing artifacts, including software. Generative mechanisms are used when needed components are not available.1 Constraint-based reasoning systems and module interconnection languages are critical underlying technologies for software composition. Prototyping technologies underlie generation. The TRW project serves as a technology conduit from the prototyping community into the DSSA program.

Following are the system requirement of Domain-specific Software Development:

- Performance
    - How quickly must the system respond to interactive operations of different kinds?
    - Are there different classes of interactive operations that users have different tolerances / expectations for?
    - Is there a batch window? What runs in it?
    - Do the batches have their own performance constraints, e.g., to clear the batch window before it closes?
    - Does the batch load influence any interactive users running at the same time?

    - Is there data with a high read/write access ratio that can be cached in memory at different tiers in the architecture?
    - What are the expected performance bottlenecks?
        - CPU?
        - Memory on client, server or intermediate nodes?
        - Hard drive space on each node?
        - Communications links?
        - DB
            - Access
            - Searching
            - Complex joins
        - Interaction with other internal systems?
        - Interaction with systems in other departments?
        - Interaction with partner systems?
        - Interactions with public systems?
- Scalability

    - Peak load of how many users doing what kinds of operations?
    - Ability to grow to how many records in which critical database tables without slowing down related operations by more than X
    - Avoiding saturating a communication link that cannot be upgraded to a higher speed?
    - What dimensions can be scaled, e.g., more CPUs, more memory, more servers, geographical distribution?
    - Is the primary scaling strategy to "scale up" or to "scale out" -- that is, to upgrade the nodes in a fixed topology, or to add nodes?
- Availability

    - What is the required uptime percentage?
    - Does this vary by time of day or location?
    - What is the current schedule of controlled outages? Is this acceptable, or is there a goal to improve it?
- Reliability

    - Are there components with re-liabilities that are known to be less than the required reliability of the system?
    - What strategies are currently in place to build more reliable capabilities out of less reliable capabilities?
    - What is the expected mean time to failure-by-failure severity by operation?
    - How will reliability be assessed prior to deployment?

- Security
  - What operations need to be secured?
  - How will users be administered?
  - How will users be given permissions to access secured operations?
  - What are the different levels of security and how do these maps?
    - Security by operation
    - Security by type of object
    -
    - Security by instance of object
- Maintainability
  - Are there concerns about the ability to hire appropriate technology skills, attract them to the area at reasonable prices?
  - What kinds of changes are anticipated in the first rounds of maintenance? What is their relative priority?
  - What sort of regression testing is required to ensure that maintenance changes do not degrade existing functionality?
  - What sort of maintenance documentation is expected to be produced? When?
- Flexibility
  - Is there system behavior that needs to be changed regularly without program changes?
    - Can this be encoded in the database?
    - Are there run-time rules that can be handled using a rules interpretation engine?
    - Are there functions that should be user scripted? If so, how will these be QA-ed?
- Configurability
  - What parameters need to be set on a machine-by-machine basis?
- Personalizability
  - What aspects of the system can be customized on a per-user basis?
  - How does the user change these settings?
  - What is the strategy for defaults?
- Usability
  - Are there operations that need to be done as quickly as possible, so that user gestures should be minimized?>

- Are there difficult or occasional-user operations that require non-standard presentations to help the user perform correctly?
- What is the balance between data integrity and the ability to stop in a "work in progress" state?
- What styles of validation are used in what situations?
- What metaphors from existing or parallel systems should be used?
- What sort of training deliverables are expected?
- What sort of on-board help system is expected?

- Portability
    - Data portability between this system and other systems?
    - Portability across different versions of a single vendor's DB?
    - Ability to port to a different vendor's DB? Which one(s)? When?
    - Browser portability? What browser versions? Historical and future?
    - Operating system portability?
- Conformance to standards
    - What legal standards apply?
    - What technical standards apply?
    - Other standards, e.g., 508.1 for disabled users?
    - What development standards apply?
        - Database naming standards
        - Existing internal architectural standards (e.g., everything goes in an Oracle database)
        - Language and coding standards
        - Testing and review standards
        - Presentation standards, e.g., use of standard colors, controls or other affordances?
        - Lifecycle models or methodologies
- Internationalizability
    - What languages?
    - In what order?
    - How translated?
    - Single or multi byte character sets?
- Efficiency -- space and time
- Responsiveness
    - What are the expected and upper limit response times per operation in the system?

  - What is the trade-off between lower averages and wider variations in response time?
- Interoperability
  - What systems will this system interoperate with immediately?
  - What other systems are anticipated?
  - What classes of internal and external systems might later be needed to interoperate with?
  - What functionality from this system needs to be exposed as a service in a service-oriented architecture?
  - What functionality from this system needs to be exposed as a Web service or via a portal?
- Upgradeability
  - Do the servers need to be upgraded while running?
  - How many client stations need to be upgraded, and what are the costs and mechanisms for upgrading them?
  - How often do different kind of fixes need to be distributed? Are there "hot fixes" that have to go out right away, but others that can wait? How often do each kind occur?
- Auditability / traceability
  - What record of who did what when must be maintained?
  - For how long?
  - Who accesses the audit trails?
  - How?
  - Is archive to tape or other off-site storage media required?
  - Is "effective dating" required?
- Transactionality
  - What are the important database and application transaction boundaries?
  - Is standard "optimistic" locking appropriate, or is something more complex required in some or all cases>
  - Is disconnected operation required by any node?
- Administrability
  - What live usage information needs to be displayed?
  - To who? How? When?
  - What "live" interventions are required?
  - What ability to handle remote configurations are required?

- Are there existing application management consoles that will be used to manage this application?

Result and Discussion:

In this Experiment, I have created an SRS document for Online Computer Store.
It is attached at the end of this Practical.

Learning Outcomes: Students should have been able to understand

LO1: Define software System.

LO2: Identify different system requirements of software Architecture.

LO3: Explain system requirement of Domain-specific Software Development.

Course Outcomes: Upon completion of the course students will be able to understand System requirements for an Architecture for any specific domain.

Conclusion:

A **software requirements specification** (SRS) is a document that describes what the software will do and how it will be expected to perform. It also describes the functionality the product needs to fulfill all stakeholders (business, users) needs.

Viva Questions:

1. Define Software System.
2. Explain different requirements for creating a software Architecture.
3. Explain any five-system requirement.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

Software Requirements Specification

for

Online Computer Store

Version 1.0 approved

Prepared by

Shreyansh Shukla (BE COMP B 24)

Thakur College of Engineering and Technology

4th October 2021

Table of Contents

Revision History

Name	Date	Reason for Changes	Version

1. Introduction

1.1 Purpose

This document is meant to delineate the features of OSS, so as to serve as a guide to the developers on one hand and a software validation document for the prospective client on the other. The Online Shopping System (OSS) for online computer shop web applications is intended to provide complete solutions for vendors as well as customers through a single gateway using the internet. It will enable vendors to setup online shops, customer to browse through the shop and purchase them online without having to visit the shop physically. The administration module will enable a system administrator to approve and reject requests for new features and maintain various lists of shop category.

1.2 Intended Audience and Reading Suggestions

This document is intended for Users, Project Team Members and Project Evaluators. This document contains the requirements of the project "OSS(online shopping system)", starting with an Introduction of the Project, followed by Description of the Project, External Interface Requirements, System Features and Non-Functional Requirements. Document Readers are suggested to follow the inherent sequence, for better comprehension and easy understanding of this document.

1.3 Product Scope

This system allows the customers to maintain their cart for add or remove the product over the internet.

1.4 References

This document refers https://dipeshagrawal.files.wordpress.com/2018/07/docuri-com_online-project.pdf for identifying Requirements from Problem Statement.

2. Overall Description

2.1 Product Perspective

This product aimed toward a person who don't want to visit the shop as he might don't get time for that or might not interested in visiting there and dealing with lot of formalities.

2.2 Product Functions

✓CATEGORY MANAGEMENT

✓ITEMS MANAGEMENT

✓USER MANAGEMENT

✓ORDER MANAGEMENT

2.3 User Classes and Characteristics

User should be familiar with the terms like login, register, order system etc.

2.3.1 Customer

Through a web browser the customers can search for a Computer and accessories online by its name or manufacturer later can add to the shopping cart. The user can login using his account details or new customers can set up an account very quickly. They should give the details of their full name, email account, username and password.

2.4 Operating Environment

The system operates with the following software components and applications,
A full internet connection is required for OSS and any Operating system and a Browser like Mozilla Firefox, internet explorer and chrome etc.

2.5 Design and Implementation Constraints

Memory: device will have 2GB internal hard drive. Software and database cannot exceed this amount.
Internet: A full internet connection required.
Operating System: software does not require any specific Operating system

2.6 Assumptions and Dependencies

It is assumed that the hardware designed will work correctly with the third-party operating system and the developed software.
The customer has a computer with a browser and have Internet.

3. External Interface Requirements

3.1 User Interfaces

The online computer Store user interface has been specifically designed with their customers in mind, allows to customer to buy laptops and accessories without going to shop.

The home screen offers a menu with a list of functions that the device performs. The user can select one of the options on the menu, and is taken to the respective screen. Every screen displays the menu on the bottom. The user can click on any one of the options and is taken to the screen of their choice. In addition, clicking on the button displays the home screen with the menu options.

3.2 Hardware Interfaces

Hardware requirements for insurance on internet will be same for both parties which are as follows:

Processor: Dual Core

RAM: 2 GB

NIC: For each party Communication Interface

The two parties should be connected by LAN or WAN for the communication purpose.

Sender
Communication
Receiver

3.3 Software Interfaces

It is compatible with Windows, Linux & Mac operating systems. Software is web based so software needs a web browser and internet connection

3.4 Communications Interfaces

Users can connect with system using browser and internet once user login user can easily buy computers.

4. System Features

4.1 System Feature 1

This section provides a requirement overview of the system. Various functional modules that can be implemented by the system will be –

Description:

4.1.1 Registration:- If a customer wants to buy the product then he/she must be registered, unregistered users can't go to the shopping cart.

4.1.2 Login:- Customer logins to the system by entering valid user id and password for the shopping.

4.1.3 Changes to Cart:- Changes to cart means the customer after login or registration can make order or cancel order of the product from the shopping cart.

4.1.4 Payment:- In this system we are dealing with the mode of payment by Cash. We will extend this to credit card, debit card in the future.

4.1.5 Logout:- After ordering or surfing for the product customer has to logout.

4.1.6 Report Generation:- After ordering for the product, the system will send one copy of the bill to the customer's Email-address and another one for the system database.

5. Other Nonfunctional Requirements

5.1 Performance Requirements

In order to maintain an acceptable speed at maximum number of uploads allowed from a particular customer as any number of users can access to the system at any time.
Also, the connections to the servers will be based on the attributes of the user like his location and server will be working 24X7 times.

5.2 Safety Requirements

The application must not be used to reveal the user information in public. Actions like security check of confidential documents like Credit-Card/Debit-Card, etc. should be prevented.

5.3 Security Requirements

All passwords of users must be encrypted before storing them in databases. Only authorized users should have access to databases storing personal user information.

5.4 Software Quality Attributes

The application should be available on all Android Devices. It should be scalable, allowing for low and medium traffic. It should be flexible, allowing developers to add new updates, without having to change prior code. It should be maintained and should be checked for errors, weekly. It should be robust and easily portable. All modules should be testable and reusable

6. Other Requirements

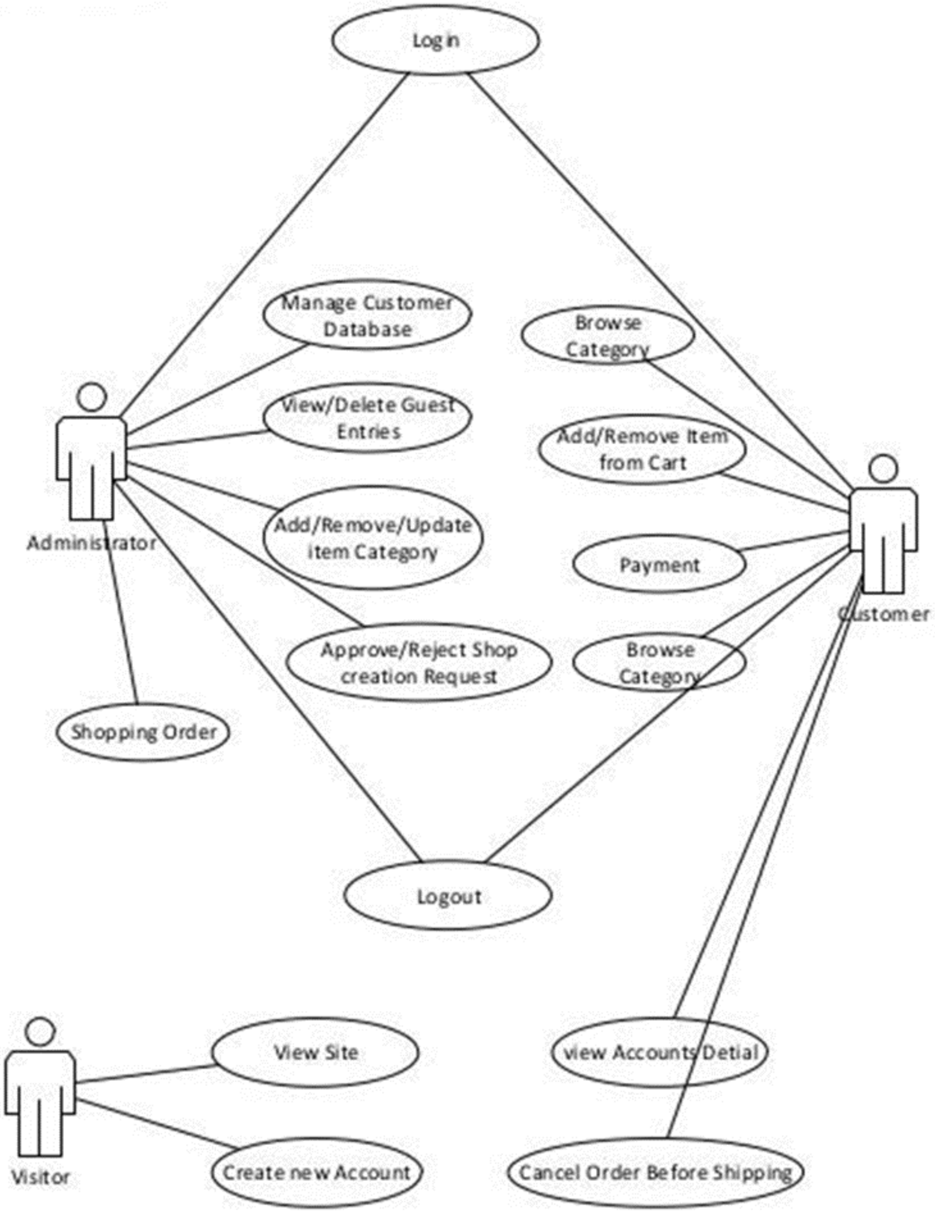
Following Non-Functional Requirements will be there in the insurance to the internet:
Secure access to consumers confidential data.
24X7 availability
Better component design to get better performance at peak time
Flexible service-based architecture will be highly desirable for future extension. Non-Functional Requirements define system properties and constraints.
Various other Non-Functional Requirements are:

>Security
>Reliability
>Maintainability
>Portability
> Extensibility
> Reusability

Appendix A: Glossary

- OSS: Online shopping system
- UI: User Interface
- API: Application Program Interface
- JSON: JavaScript Object Notation
- FTP: File Transfer Protocol
- HTTP: Hypertext Transfer Protocol

Appendix B: Analysis Models



SHREYANSH SHUKLA
BE COMP B 24
SA – EXP - 9

Experiment No.: 9

Mapping of non-functional components with system requirements.

Learning Objective: Student should be able to understand Mapping of non-functional components with system requirements.

Theory:

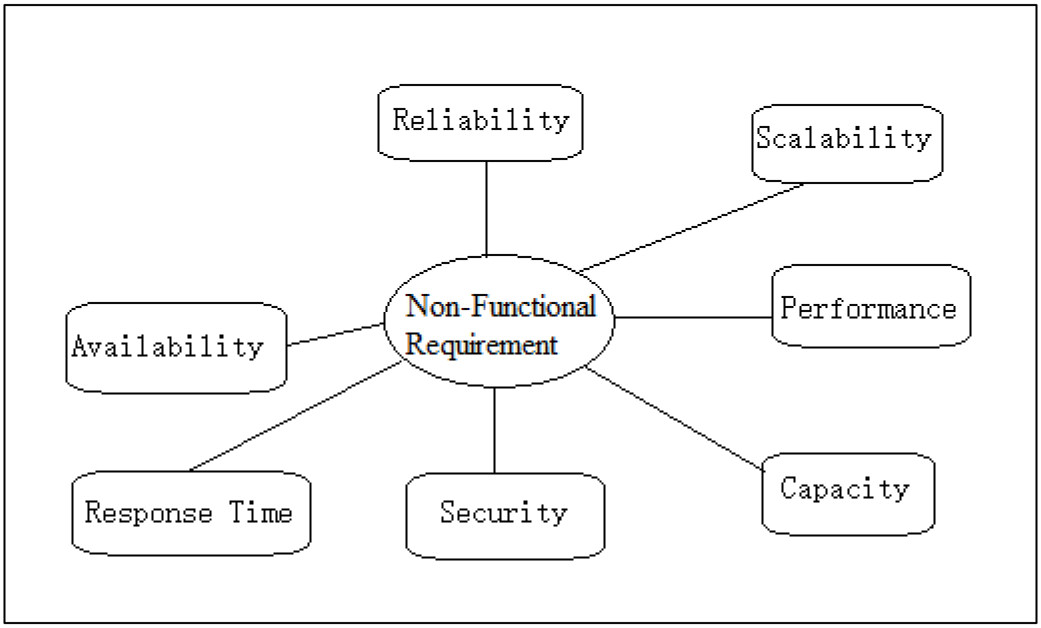
What is Non-Functional Requirement?

NON-FUNCTIONAL REQUIREMENT (NFR) specifies the quality attribute of a software system. They judge the software system based on Responsiveness, Usability, Security, Portability and other non-functional standards that are critical to the success of the software system.

NFRs define the system properties and specify the behavioral pattern under various operating conditions. The various estimation methods help in sizing the application based on the functional requirements. However, most of these methods have overlooked the influence of non-functional requirements.

The key NFRs that can be attributed to an application and their mapping as follows.

1) Reliability	Operation Ease
2) Response Time	No mapping given
3) Performance	Performance, Online Update, Online Date Entry
4) Security	No mapping given
5) Availability	No mapping given
6) Scalability	Transaction rate
7) Capacity	No mapping given



Middleware is a more effective program that acts as bridge in between various applications and other databases otherwise tools. It is placed in between operating system and other applications which run on it. Middleware allows making better communication, application services, messaging, authentication, API management and management of data between different kinds of applications which help to exchange data.

The connectors sit between the two APIs or you can say and the ends of the connectors are APIs. The connectors receive data from one app/solution and process it to make it understandable and accessible in the other app/solution, regardless of whether any direct form of integration was available in the two apps.

Mapping of NFRs:

Operation Ease to Reliability:

An application or the software system once installed and configured on a given platform should require no manual intervention, except for starting and shutting down. The system should be able to maintain a specified level of performance in case of software faults. It should also be able to re-establish its level of performance and to recover all the data directly affected in case of a failure in the minimum time and effort. This is mapped on to the reliability NFR. It may be defined as “a system which is capable of reestablishing its level of performance and recovering the data directly affected in case of a failure and on the time and effort needed for it. The design criteria for reliability can be defined as self-contained the system should have all the features necessary for all its operations including recovering it by itself; completeness- it should be complete in itself and not dependent on anything else; robustness/integrity- it should not easily breakdown; error tolerance- it should be able to tolerate errors and rectify them and continue in its operation. There are "numerous metrics for determining reliability: mean time to failure, defect reports and counts, resource consumption, stability, uptime percentage and even customer perception."

Performance:

Real time systems have strict performance parameters like performing at the same level even during peak user times, producing high throughput, serving a huge user base, etc. The DI varies from no special performance requirements to response time being critical during all business hours and till performance analysis tools being used in the design. System should meet the desired performance expectation. Also, if online update has to take place, then the performance expectations to be met are very high – fast response, low processing time and high throughput rates. The performance NFR is also based on the Online Data Entry requirements of an application. The present-day trend is to have interactive and real-time data entry. The GUI development requires a lot of effort as help has to be provided, validation to be implemented, reference information for faster data entry operations, etc. Performance when related to this can be defined as "attributes of software that bear on response and processing times and on throughput rates in performing its function.

Transaction Rate to Scalability:

In many business applications the transaction rate increases to high peak levels once in a day or once in a week with the requirement remaining so that there has to be no dramatic increase in transaction time. This issue has to be looked into in the design, development and/or installation phases of a project. This GSC is mapped on to the scalability NFR. The term scalability implies "the ability to scale up to peak transaction loads. In order to achieve this the application has to be designed in such a way so that it should cater to the highest possible figures thus wasting resources when the transaction rate is low. The architecture should be designed in a multi-layered manner in complex algorithm-based applications to scale up to peak transaction rates. In today's systems, this GSC does not contribute much to the DI as present-day hardware and operating systems provide built-in features such as high bandwidth network, high speed storage disks with high-speed disk access timings and CPUs with high MHZ processing speed which when combined leads to build in high transaction rates.

Result and Discussion: Because software connectors provide a uniform interface to other connectors and components within an architecture, architects need not be concerned with the properties of different middleware technologies as long as the technology can be encapsulated within a software connector.

Learning Outcomes: Students should have been able to

LO1: Define nonfunctional Requirement.

LO2: Identify different component of nonfunctional Requirement.

LO3: Explain mapping.

Course Outcomes: Upon completion of the course students will be able to understand mapping of non-functional components with system requirements.

Conclusion:

Mapping of non-functional components with system requirements **for Online Computer Store**

Non-Functional Requirements	**System Requirements**
Response Time	Data Communications, Distributed data processing, Performance
Security	Multiple sites, Online Update
Availability	Online data entry, operation ease, Purchase of Items
Capacity	Transaction rate, Multiple sites

Viva Questions:

1. Define nonfunctional Requirement.
2. Explain nonfunctional Requirement component.
3. Explain Transaction Rate to Scalability mapping of NFR.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	

Experiment No.: 10

SHREYANSH SHUKLA
BE COMP B 24
SA – EXP 10

Implementation of Software Architecture for identified system/application.

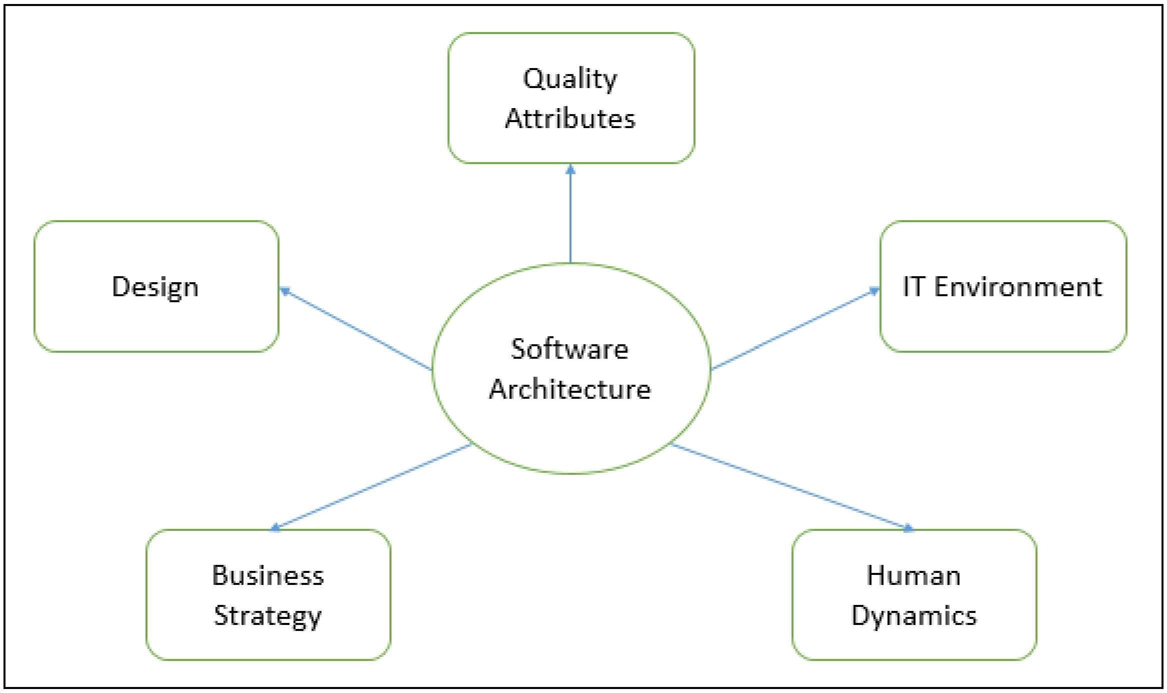
Learning Objective: Student should be able to understand wrapper the implementation of Software Architecture for a system/application.

Theory:

Software architectures describe how a system is decomposed into components, how these components are interconnected, and how they communicate and interact with each other. When poorly understood, these aspects of design are major sources of errors.

Software architecture of a system describes its major components, their relationships, and how they interact with each other.

It essentially serves as a blueprint. It provides an abstraction to manage the system complexity and establish communication and coordination among components.

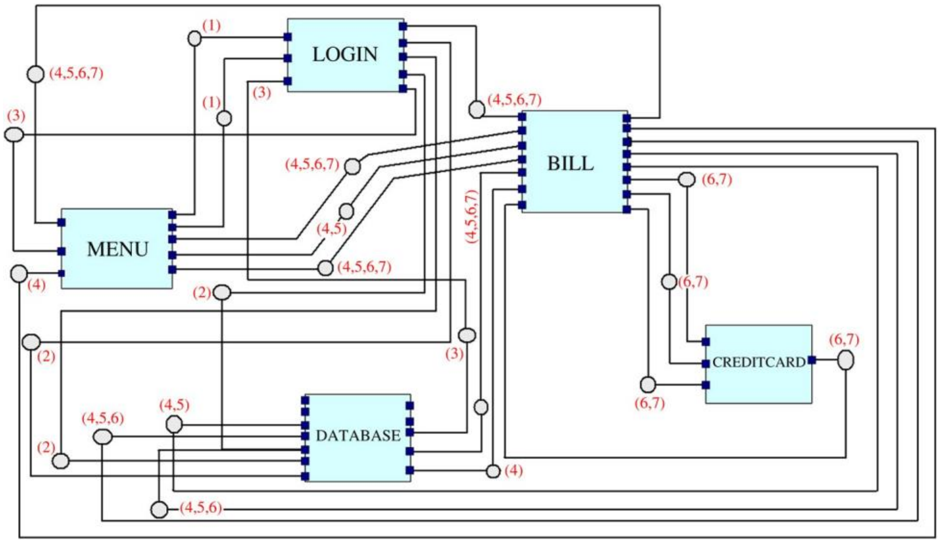


It defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

- The architecture helps define a solution to meet all the technical and operational requirements, with the common goal of optimizing for performance and security.
- Designing the architecture involves the intersection of the organization's needs as well as the needs of the development team. Each decision can have a considerable impact on quality, maintainability, performance, etc.

Implementation of Application:

The system aims at addressing the needs of a typical hospital accounting service, using a component-based approach. The system supports account keeping for each patient. Each patient has his private account, and each user has an authorization level. The accounts can be charged or paid only by users with high authorization, while users with normal authorization are only able to print statements for the patients' accounts. For the payment of an account, a number of alternative payment methods such as credit card payment, cash payment and banking account transfers are provided.



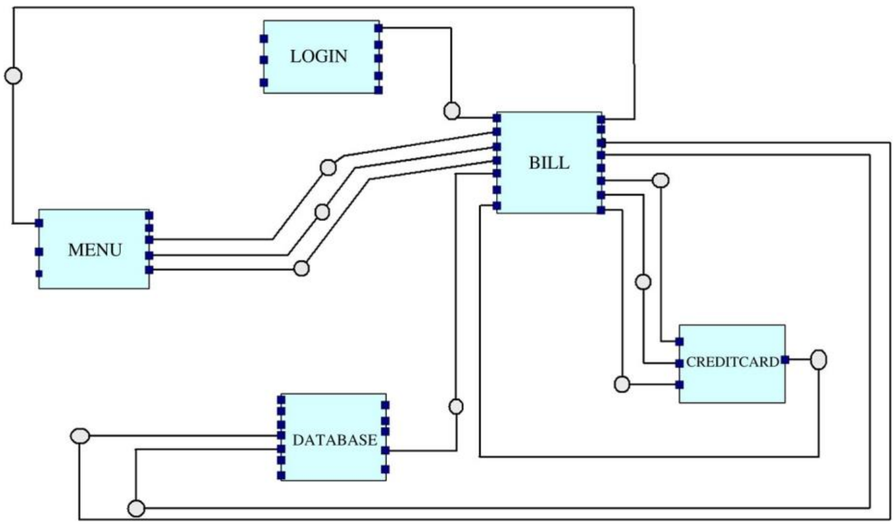
The system described here can be viewed as part of a more complex system, connected to other components and services. The latter more complex system is effectively an amalgamation of a couple of other systems, designed for tele-medicine applications within the context of research projects. These systems have been developed over a period of a year by 2–3 programmers. However, it is interesting to note that the approach developed in this paper allowed for the high-level modeling of the software architecture much faster and in a more natural way. Furthermore, the production of executable code was also a faster process, as much of the code was automatically generated from the tool. For implementation, we use Application as the coordination language and the components are written. ACME is used for the system's architectural design. For the transition from the ACME code to the Application code, Deriving the high-level architecture is a stepwise approach:

a. First, the components and their ports are identified. The only interaction of a component with its environment is through the ports. A port in the IWIM model as well as in most ADLs can be realized as a buffer, temporarily storing information, until it is read by another component (to be accurate, the connectors in ACME — the entities that connect two ports The architecture is designed in AcmeStudio (Eclipse plug-in). For clarification reasons, we enhanced the diagram with state transition information. When a component needs to output a specific value-result to another component, it just pushes this value into the appropriate output port.

b. Second, the connectors are detected, that is, the actual interactions between two components through their ports. More specifically, we detect which components interact using which ports. We also detect which of these connectors are executed concurrently and annotate them with the same Active on number. The idea is to detect all the expected states in a given component configuration. Such a state definition could include the following: e.g., in state 6, component Bill.Out_port_6 sends to component CreditCard.In_port_1 some data.

c. Third, the ADL design is enhanced with the properties defined earlier. This allows the generation of the better part of the coordination code, and leaves very few tasks for the programmer to complete manually.

d. Finally, the ACME code is parsed and the equivalent Application code is generated.



e. The application consists of the following five components:

(i) Login: Verifies username and password and keeps the authorization level of the user.

(ii) Bill: Implements the basic accounting functions: insert/delete customer, pay/charge bill, print statement.

(iii) Credit Card: Communicates with appropriate bank systems to charge the account of a customer that pays using a credit card. This component is used by the Bill component in case the payment method selected by a customer is "credit card payment".

(iv) Database: Receives requests for data handling from the other components and executes them.

(v) Menu: Implements the interaction between the user and the system.

Each of the above components was implemented as a manager process coordinating an atomic process, which in turn implements the functionality of the component. The five manager processes are coordinated by the Main Application. It is interesting to note here that although the system could be implemented using five atomic processes coordinated by the Main Application, we have chosen to wrap every atomic process in a manager process in order to keep computational units unchanged in case of a system change or evolution. In other words, we have separated the computational from the coordination concerns and in that respect, we have generated code that can potentially have a higher degree of reusability than it would have had otherwise, had we generated only an equivalent fragment of executable code.

In the current state of the system, each of the manager processes simply passes data from its input ports to the input ports of the atomic processes that it coordinates and reversely from the output ports of the atomic processes to its own output ports. However, more sophisticated coordination scenarios can be supported. The connectors represent Application streams. Each connector can be active in one or more states.

Our example includes the following states:

(1) Menu_Requests_Login: The MENU component sends the username and password to the LOGIN component, so that the user's authorization is checked.

(2) Login_Requests_Verification: The LOGIN component asks from the DATABASE component to perform a query for the user's authorization.

(3) Login_Gets_Verification: The LOGIN component gets the authorization level from the DATABASE component, and returns the answer to the MENU component.

(4) Menu_Requests_Bill_Info: The MENU component requests information for a given customer from the BILL component, which in turn asks for some information from the DATABASE component and returns the answer to the MENU component.

(5) Menu_Requests_Bill_Create: The MENU component requests the creation of a bill from the BILL component for a new customer. The BILL component prepares and sends the related data

to the DATABASE component. The DATABASE component sends back a confirmation to the BILL component, which returns the confirmation to the MENU component.

(6) Menu_Requests_Bill_Insert/Update: The MENU component requests the insertion or updating of a bill from the BILL component. The BILL component prepares and sends the related data to the DATABASE component. Also, if credit card payment is included, BILL sends the relevant data to the CREDITCARD component, which makes the transaction and sends the result back.

(7) Menu_Requests_Bill_Delete: The MENU component requests the deletion of a bill from the BILL component. The BILL component sends the relevant data to the DATABASE component, and waits for deletion confirmation. It then notifies the MENU component that the deletion is successful.

the methodology is able to implement a complete Application, i.e., the state and all the streams.

Result and Discussion:

Code

```
package helloworld2;
import edu.uci.isr.myx.fw.AbstractMyxSimpleBrick;
import edu.uci.isr.myx.fw.IMyxName;
import edu.uci.isr.myx.fw.MyxUtils;
public class MessageSender extends AbstractMyxSimpleBrick {
public static final IMyxName MESSAGES_NAME = MyxUtils.createName("messages");
@Override public void begin() {
super.begin();
MessageIntf messages = (MessageIntf)MyxUtils.getFirstRequiredServiceObject(this,
MESSAGES_NAME);
messages.processMessage("Hello World!");
}
public Object getServiceObject(IMyxName interfaceName) {
return null;
```

Learning Outcomes: Students should have been able to understand

LO1: Define software Architecture.

LO2: Identify different phases in software Architecture.

LO3: Explain Implementation of software Architecture.

Course Outcomes: Upon completion of the course students will be able to understand Implementation of software Architecture.

Conclusion:

1) In this Experiment, Software Architecture was defined.

2) Different Phases of Software Architecture were identified.

3) Software Architecture was implemented.

4) In this Experiment, implementation of Software Architecture was performed in Eclipse Arch Studio.

5) The implementation was evaluated and archived.

Viva Questions:

1. Define Software Architecture.
2. Explain difference between software Architecture and software design.
3. Explain Implementation of software Architecture with an example.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	

SHREYANSH SHUKLA
BE COMP B 24
SA – EXP 11

Experiment No.: 11

Learning Objective: Student should be able to understand Case Studies on following terms: -

1. Architecture evaluation, analysis and design
2. Architecture Tradeoff Analysis Method (ATAM)
3. Quality Attribute Workshops (QAW)
4. Architecture reconstruction

Theory:

1.Architecture Evaluation, analysis and design:

Architecture evaluation is a development life-cycle activity whereby several stakeholders analyze the software architecture together in a formal or informal process using an assessment technique such as scenarios. Evaluations can utilize the same architecture assessment techniques described. In some methods, such as ATAM, the stakeholders generate the utility tree and the scenarios as part of the evaluation process.

The characteristics of the development organization affect when and how evaluations are conducted. In small organizations that have a less mature engineering culture, a formal evaluation proceeding may not be appropriate. However, many of the principles and patterns of the evaluation techniques can still be applied. The architect may document a handful of critical scenarios that are used to guide the software design and that serve as input into the testing process.

Formal evaluations should be done as soon as the architectural design concept is stable enough to be assessed, but before any real commitment to development has been taken. This allows the team to discover problems with the architecture at a time when such problems are easier and cheaper to address. However, evaluations *can* be performed after the system is under development or even when it is complete. In some cases, I've had to reverse-analyze a system's architecture to determine why some undocumented design decisions were made when no one else on the team had any recollection of why. Although these were not formal evaluation proceedings, I used the same techniques of generating scenarios based on what stakeholders believed characterized the critical quality attributes and evaluated the existing system against them. Quite often the system that was built had quite different characteristics than what the stakeholders believed. In one case, the stakeholders believed that the system was flexible and could easily adapt to end-user information architecture and business process needs. The original designers, who were no longer on the project, had assured the other stakeholders that the system was flexible and adaptable. Because these qualities were not articulated specifically, there was no way to assess that the system satisfied the acquirers' concepts of flexible and adaptable.

This situation is very common and is not a deliberate attempt to build something other than what the stakeholders are asking for. However, in the context of an *ad-hoc* development process and without proper specification of the system, acquirers are not able to articulate their specific needs and the developers are not capturing these specific quality requirements in any useful way. You can probably think of at least one such characteristic of the system that you have worked on. The acquirers want the user interface to have an acceptable response time or the system must be flexible enough to integrate into a customer's existing information architecture and Information Technology (IT) architecture. But these desires, if left at this level of specification, can rarely be satisfied completely or accurately. Remember, understand the right problem and solve the right problem. This is our imperative as software architects.

There are several software architecture evaluation methods:

Scenario-based Architecture Analysis Method (SAAM). This was probably the first documented software architecture analysis method and was originally developed to analyze an architecture for modifiability. However, it is useful for analyzing any nonfunctional aspect of an architecture. It is founded on the use of stakeholder-generated scenarios to assess an architecture.

Architecture Trade-off Analysis Method (ATAM). This is a successor of SAAM and is also gaining widespread use. This method incorporates quality attribute utility trees and quality attribute categories in the analysis of an architecture. Whereas SAAM does not explicitly address the interactions between quality attributes, ATAM does. Thus, the trade-offs are with respect to competing quality attributes. ATAM is a specialization of SAAM, specifically focusing on modifiability, performance, availability, and security.

SAAM Founded on Complex Scenarios (SAAMCS). This method considers the complexity of evaluation scenarios as the most important risk assessment factor.

Extending SAAM by Integration in the Domain (ESAAMI). This method integrates SAAM with domain-specific and reuse-based software development processes.

Software Architecture Analysis Method for Evolution and Reusability (SAAMER). This method focuses specifically on the quality attributes of evolution and reusability.

Scenario-Based Architecture Reengineering (SBAR). This method utilizes scenarios, simulation, mathematical modeling, and experience-based reasoning for assessing quality attributes. This method also incorporates an architecture design method.

Architecture Level Prediction of Software Maintenance (ALPSM). This is another method for analyzing maintainability using scenarios, called change scenarios, which represent maintenance tasks.

Software Architecture Evaluation Model (SAEM). This method is based on formal and rigorous quality requirements.

Analysis Model: Analysis model is the first step towards design. This model describes the structure of the system or application. It consists of class diagram and sequence diagrams that describes the logical implementation of the functional requirements. In this model, functionalities are modeled in terms of boundary, control and entity classes. Sequence diagrams realize use cases by describing the flow of events in the use cases when they are executed. These artifacts are categorized as High-Level Design artifacts for assessment.

Design Model: Design model builds on analysis model by describing, in greater detail, the structure of the system and how the system will be implemented. A Design model consists of design classes structured into packages and subsystems with well-defined interfaces. This model describes the clear relationship between the design classes, and implementation elements. Implementation elements are directories and files including source code, data and executable files. These artifacts are analyzed for to the Detailed Design.

- **Implementation Model:** The Implementation Model represents the physical composition of the implementation in terms of Implementation Subsystems, and Implementation Elements (directories and files, including source code, data, and executable files). This artifact is considered in the assessment of the Detailed Design.

- **Deployment Model:** Deployment model describes the deployment units of a system and one or more physical network (hardware configurations on which these units are deployed. This artifact belongs to the Detailed Design part of an assignment.

2. Architecture Tradeoff Analysis Method (ATAM)

The Architecture Tradeoff Analysis Method (ATAM) is a method for evaluating software architectures relative to quality attribute goals. ATAM evaluations expose architectural risks that potentially inhibit the achievement of an organization's business goals. The ATAM gets its name because it not only reveals how well an architecture satisfies particular quality goals, but it also provides insight into how those quality goals interact with each other—how they trade off against each other.

The ATAM is the leading method in the area of software architecture evaluation. An evaluation using the ATAM typically takes three to four days and gathers together a trained evaluation team, architects, and representatives of the architecture's various stakeholders.

Challenges:

Most complex software systems are required to be modifiable and have good performance. They may also need to be secure, interoperable, portable, and reliable. But for any particular system

- What precisely do these quality attributes such as modifiability, security, performance, and reliability mean?
- Can a system be analyzed to determine these desired qualities?
- How soon can such an analysis occur?
- How do you know if a software architecture for a system is suitable without having to build the system first?

Business drivers and the software architecture are elicited from project decision makers. These are refined into scenarios and the architectural decisions made in support of each one. Analysis of scenarios and decisions results in identification of risks, non-risks, sensitivity points, and tradeoff points in the architecture. Risks are synthesized into a set of risk themes, showing how each one threatens a business driver.

The ATAM consists of nine steps:

1. **Present the ATAM**. The evaluation leader describes the evaluation method to the assembled participants, tries to set their expectations, and answers questions they may have.
2. **Present business drivers**. A project spokesperson (ideally the project manager or system customer) describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (e.g., high availability or time to market or high security).
3. **Present architecture**. The architect will describe the architecture, focusing on how it addresses the business drivers.
4. **Identify architectural approaches**. Architectural approaches are identified by the architect, but are not analyzed.
5. **Generate quality attribute utility tree**. The quality factors that comprise system "utility" (performance, availability, security, modifiability, usability, etc.) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized.
6. **Analyze architectural approaches**. Based on the high-priority factors identified in Step 5, the architectural approaches that address those factors are elicited and analyzed (for example, an architectural approach aimed at meeting performance goals will be subjected to a performance analysis). During this step, architectural risks, sensitivity points, and tradeoff points are identified.
7. **Brainstorm and prioritize scenarios**. A larger set of scenarios is elicited from the entire group of stakeholders. This set of scenarios is prioritized via a voting process involving the entire stakeholder group.

8. **Analyze architectural approaches**. This step reiterates the activities of Step 6, but using the highly ranked scenarios from Step 7. Those scenarios are considered to be test cases to confirm the analysis performed thus far. This analysis may uncover additional architectural approaches, risks, sensitivity points, and tradeoff points, which are then documented.
9. **Present results**. Based on the information collected in the ATAM (approaches, scenarios, attribute-specific questions, the utility tree, risks, non-risks, sensitivity points, tradeoffs), the ATAM team presents the findings to the assembled stakeholders.

The most important results are improved architectures. The output of an ATAM is an out brief presentation and/or a written report that includes the major findings of the evaluation. These are typically

- a set of architectural approaches identified
- a "utility tree"—a hierarchic model of the driving architectural requirements
- the set of scenarios generated and the subset that were mapped onto the architecture
- a set of quality-attribute-specific questions that were applied to the architecture and the responses to these questions
- a set of identified risks
- a set of identified non-risks
- a synthesis of the risks into a set of risk themes that threaten to undermine the business goals for the system

Benefits

- identified risks early in the life cycle
- increased communication among stakeholders
- clarified quality attribute requirements
- improved architecture documentation
- documented basis for architectural decisions

The most important results are improved architectures. The ATAM aids in eliciting sets of quality requirements along multiple dimensions, analyzing the effects of each requirement in isolation, and then understanding the interactions of these requirements.

Who Would Benefit?

Many people have a stake in a system's architecture, and all of them exert whatever influence they can on the architect to make sure that their goals are addressed. For example, the users want a system that is easy to use and has rich functionality. The maintenance organization wants a system that is easy to modify. The developing organization wants a system that is easy to build and that will employ the existing work force to good advantage. The customer wants the system to be built on time and within budget. All of these stakeholders will benefit from applying the ATAM. And needless to say, the architect is also a primary beneficiary.

**3. Quality Attribute Workshops (QAW) **

The QAW is a facilitated, early intervention method used to generate, prioritize, and refine quality attribute scenarios before the software architecture is completed. The QAW is focused on system-level concerns and specifically the role that software will play in the system. The QAW is dependent on the participation of system stakeholders—individuals on whom the system has significant impact, such as end users, installers, administrators, trainers, architects, acquirers, system and software engineers, and others. The QAW is an intense and demanding activity. It is very important that all participants stay focused, are on time, and limit side discussions throughout the day.

The QAW involves the following steps:

1. QAW Presentation and Introductions
2. Business/Mission Presentation
3. Architectural Plan Presentation
4. Identification of Architectural Drivers
5. Scenario Brainstorming
6. Scenario Consolidation
7. Scenario Prioritization
8. Scenario Refinement

The following sections describe each step of the QAW in detail.

Step 1: QAW Presentation and Introductions

In this step, QAW facilitators describe the motivation for the QAW and explain each step of the method. We recommend using a standard slide presentation that can be customized depending on the needs of the sponsor. Next, the facilitators introduce themselves and the stakeholders do likewise, briefly stating their background, their role in the organization, and their relationship to the system being built.

Step 2: Business/Mission Presentation.

After Step 1, a representative of the stakeholder community presents the business and/or mission drivers for the system. The term “business and/or mission drivers” is used carefully here. Some organizations are clearly motivated by business concerns such as profitability, while others, such as governmental organizations, are motivated by mission concerns and find profitability meaningless. The stakeholder representing the business and/or mission concerns (typically a manager or management representative) spends about one hour presenting

- the system’s business/mission context

- high-level functional requirements, constraints, and quality attribute requirements During the presentation, the facilitators listen carefully and capture any relevant information that may shed light on the quality attribute drivers. The quality attributes that will be refined in later steps will be derived largely from the business/mission needs presented in this step.

Step 3: Architectural Plan Presentation

While a detailed system architecture might not exist, it is possible that high-level system descriptions, context drawings, or other artifacts have been created that describe some of the system’s technical details. At this point in the workshop, a technical stakeholder will present the system architectural plans as they stand with respect to these early documents. Information in this presentation may include

- plans and strategies for how key business/mission requirements will be satisfied
- key technical requirements and constraints—such as mandated operating systems, hardware, middleware, and standards—that will drive architectural decisions
- presentation of existing context diagrams, high-level system diagrams, and other written descriptions.

Step 4: Identification of Architectural Drivers

In steps 2 and 3, the facilitators capture information regarding architectural drivers that are key to realizing quality attribute goals in the system. These drivers often include high-level requirements, business/mission concerns, goals and objectives, and various quality attributes. Before undertaking this step, the facilitators should excuse the group for a 15-minute break, during which they will caucus to compare and consolidate notes taken during steps 2 and 3. When the stakeholders reconvene, the facilitators will share their list of key architectural drivers and ask the stakeholders for clarifications, additions, deletions, and corrections. The idea is to reach a consensus on a distilled list of architectural drivers that include high-level requirements, business drivers, constraints, and quality attributes. The final list of architectural drivers will help focus the stakeholders during scenario brainstorming to ensure that these concerns are represented by the scenarios collected.

Step 5: Scenario Brainstorming

After the architectural drivers have been identified, the facilitators initiate the brainstorming process in which stakeholders generate scenarios. The facilitators review the parts of a good scenario (stimulus, environment, and response) and ensure that each scenario is well formed during the workshop. Each stakeholder expresses a scenario representing his or her concerns with respect to the system in round-robin fashion. During a nominal QAW, at least two round-robin passes are made so that each stakeholder can contribute at least two scenarios. The facilitators ensure that at least one representative scenario exists for each architectural driver listed in Step 4.

Scenario generation is a key step in the QAW method and must be carried out with care. We suggest the following guidance to help QAW facilitators during this step:

1. Facilitators should help stakeholders create well-formed scenarios. It is tempting for stakeholders to recite requirements such as "The system shall produce reports for users." While this is an important requirement, facilitators need to ensure that the quality attribute aspects of this requirement are explored further. For example, the following scenario sheds lighter on the performance aspect of this requirement: "A remote user requests a database report via the Web during peak usage and receives the report within five seconds." Note that the initial requirement hasn't been lost, but the scenario further explores the performance aspect of this requirement. Facilitators should note that quality attribute names by themselves are not enough. Rather than say "the system shall be modifiable," the scenario should describe what it means to be modifiable by providing a specific example of a modification to the system.

2. The vocabulary used to describe quality attributes varies widely. Heated debates often revolve around to which quality attribute a particular system property belongs. It doesn't matter what we call a particular quality attribute, as long as there's a scenario that describes what it means.

3. Facilitators need to remember that there are three general types of scenarios and to ensure that each type is covered during the QAW:

a. use case scenarios - involving anticipated uses of the system

b. growth scenarios - involving anticipated changes to the system.

c. exploratory scenarios - involving unanticipated stresses to the system that can include uses and/or changes

4. Facilitators should refer to the list of architectural drivers generated in Step 4 from time to time during scenario brainstorming to ensure that representative scenarios exist for each one.

Step 6: Scenario Consolidation

After the scenario brainstorming, similar scenarios are consolidated when reasonable. To do that, facilitators ask stakeholders to identify those scenarios that are very similar in content.

Scenarios that are similar are merged, as long as the people who proposed them agree and feels that their scenarios will not be diluted in the process. Consolidation is an important step because it helps to prevent a "dilution" of votes during the prioritization of scenarios (Step 7). Such a dilution occurs when stakeholders split their votes between two very similar scenarios. As a result, neither scenario rises to importance and is therefore never refined (Step 8). However, if the two scenarios are similar enough to be merged into one, the votes might be concentrated, and the merged scenario may then rise to the appropriate level of importance and be refined further. Facilitators should make every attempt to reach a majority consensus with the stakeholders before merging scenarios. Though stakeholders may be tempted to merge scenarios with abandon, they should not do so. In actuality, very few scenarios are merged.

Step 7: Scenario Prioritization

Prioritization of the scenarios is accomplished by allocating each stakeholder a number of votes equal to 30% of the total number of scenarios generated after consolidation. The actual number of votes allocated to stakeholders is rounded to an even number of votes at the discretion of the facilitators. For example, if 30 scenarios were generated, each stakeholder gets 30 x 0.3, or 9, votes rounded up to 10. Voting is done in round-robin fashion, in two passes. During CMU/SEI-2003-TR-016 11 each pass, stakeholders allocate half of their votes. Stakeholders can allocate any number of their votes to any scenario or combination of scenarios. The votes are counted, and the scenarios are prioritized accordingly.

Step 8: Scenario Refinement

After the prioritization, depending on the amount of time remaining, the top four or five scenarios are refined in more detail. Facilitators further elaborate each one, documenting the following:

- Further clarify the scenario by clearly describing the following six things:

1. stimulus - the condition that affects the system
2. response - the activity that results from the stimulus
3. source of stimulus - the entity that generated the stimulus
4. environment - the condition under which the stimulus occurred
5. artifact stimulated - the artifact that was stimulated
6. response measure - the measure by which the system's response will be evaluated

- Describe the business/mission goals that are affected by the scenario.

- Describe the relevant quality attributes associated with the scenario.
- Allow the stakeholders to pose questions and raise any issues regarding the scenario. Such questions should concentrate on the quality attribute aspects of the scenario and any concerns that the stakeholders might have in achieving the response called for in the scenario.

4 QAW Benefits:

The QAW provides a forum for a wide variety of stakeholders to gather in one room at one time very early in the development process. It is often the first time such a meeting takes place and generally leads to the identification of conflicting assumptions about system requirements. In addition to clarifying quality attribute requirements, the QAW provides increased stakeholder communication, an informed basis for architectural decisions, improved architectural documentation, and support for analysis and testing throughout the life of the system.

The results of a QAW include

- a list of architectural drivers
- the raw scenarios
- the prioritized list of raw scenarios
- the refined scenarios

This information can be used to:

- update the organization’s architectural vision
- refine system and software requirements
- guide the development of prototypes
- exercise simulations
- understand and clarify the system’s architectural drivers
- influence the order in which the architecture is developed
- describe the operation of a system.

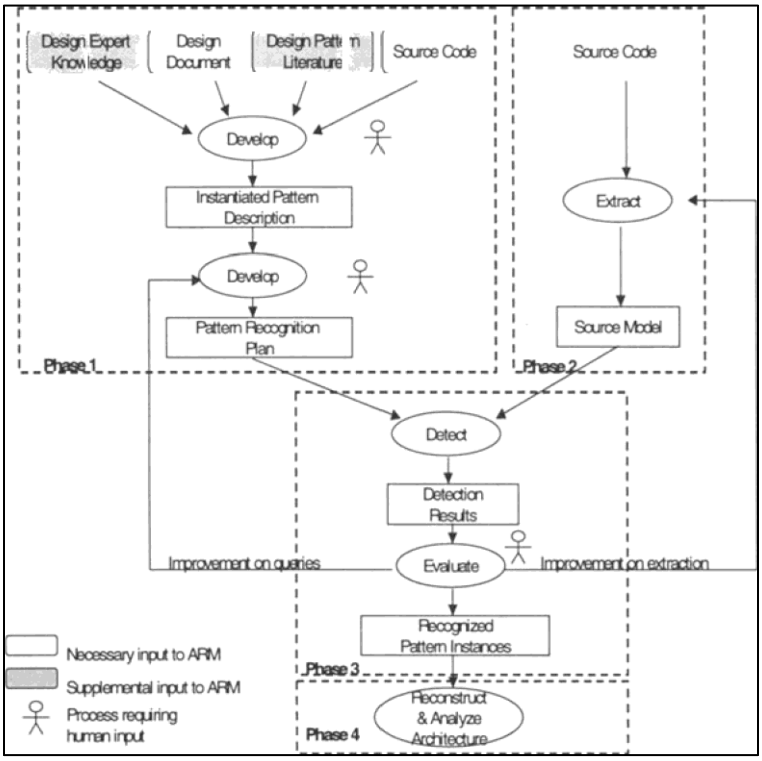
4. ARCHITECTURE RECONSTRUCTION METHOD

To assist software architecture recovery of systems designed and developed with patterns, we developed the Architecture Reconstruction Method (ARM)—a semi-automatic analysis method for reconstructing architectures based on the recognition of architectural patterns.

ARM is depicted in Figure 2. As indicated by the dashed boxes in this

figure, ARM consists of four major phases:

1. Developing a concrete pattern recognition plan.
2. Extracting a source model.
3. Detecting and evaluating pattern instances.
4. Reconstructing and analyzing the architecture.



Constructing a pattern recognition plan consists of three steps. The first is to develop an instantiated pattern description. By instantiation, we mean a concrete pattern description, with all the pattern elements and their relations described in terms of the constructs available from the chosen implementation language. Starting with a design document, one can manually determine the patterns used in the design and can extract the *abstract pattern rules—the* design rules that define a pattern's structural and behavioral properties. Pattern descriptions found in the design pattern literature, e.g., Buschmann, et al., 1996, or obtained from humans who are familiar with the system design can be used to supplement these rules. Using these abstract pattern rules as a guide, one can then examine the source code of several potential pattern instances to derive the corresponding *concrete pattern rules—the* implementation rules that realize abstract pattern rules using data structures, coding conventions, coding methods and algorithms. Such concrete pattern rules can be recognized via syntactic cues, such as naming conventions and programming language keywords, or an analysis of data access and control flow. An instantiated pattern description is a specification of the concrete pattern rules written in Rigi Standard Format (RSF) (Wong, et al., 1994). A clause in RSF is a tuple (relation, entity1, entity2), which represents the relationship *entity relates to entity2*. For example, in the *Mediator design pattern* (see Figure 3), a *Mediator* component serves as the communication hub for all the *Colleague* components. An abstract pattern rule for this pattern is

"The Mediator component mediates communications between colleague components."

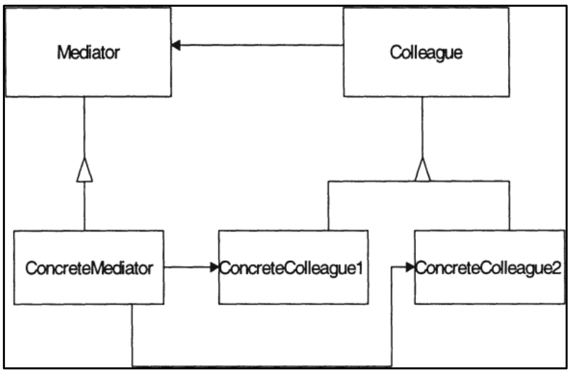


Figure 3: Mediator design pattern

k one of our case studies, the *Mediator pattern* is implemented in a C++ class where mediator and colleague components are member functions.

Extracting a source model

The second phase of ARM is to extract a source model that represents a system's source elements and the relations between them. The output of this phase is a source model that contains the information that is used for detecting *necessary* pattern rules. For example, Table 1 shows some of the relations that Dali currently extracts from C++ programs (Kazman & Carriere, 1999). The relations needed for detecting the necessary pattern rules of the Presentation-Abstraction-Control (PAC) pattem[2] (Buschmann, et al., 1996) in our case studies are denoted by *

Table 1: Typical set of source relations extracted by Dali.

Relation	From	To
calls *	function	function
contains	file	function
defines	file	class
has_subclass *	class	class
has_friend	class	class
defines_fn *	class	function
has_member *	class	variable

defines_var *	function	variable
has_instance *	class	variable
defines_global *	file	variable
var_access *	function	variable

A complication is that patterns are revealed at different levels of abstraction (e.g., the function vs. the class level), thus different pa8s of the recognition plan may need to be applied to a source model at different levels of abstraction. Using abstraction techniques, such as the *aggregation* technique provided by Dali (Kazman & Carriere, 1999), lower-level source model elements can be grouped into a higher-level element without loss of information. Thus, one can use it to bring the source model to appropriate levels of abstraction for pattern detection and architecture analysis.

Detecting and evaluating pattern instances

Detecting pattern instances using Dali is an automatic process in which one uses query tools to execute a recognition plan with respect to a source model. After running the recognition plan on the source model using the query tools, the detection output consists of all the pattern instance candidates. Human evaluation of these candidates is required to compare them with the designed pattern instances and determine which candidates are intended, which are false positives and false negatives. A false positive is a candidate which is not designed as a pattern instance, but is “detected” falsely as an instance. A false negative is a candidate which is designed as an instance, but is not detected as one.

One can try to improve the results (i.e., remove false positives and negatives) by modifying either the recognition plan or the source model and reiterating through ARM method. To improve the pattern recognition plan, one may choose another component of the pattern as the anchor and reorder the queries to form a new plan, or refine the query constraints for some of the pattern elements. If the source model extraction caused the deficiencies, an analyst needs to try to improve the extraction process by refining the existing extraction tools to catch the defects and/or incorporating other extraction tools to enhance the accuracy of source model, as described in (Kazman & Carriere, 1998).

However, if the source code is incomplete or if the pattern is defined by complex dynamic attributes, it may be impossible for the recognition technique to precisely detect all pattern instances. The evaluation process ends when Dali can detect the maximal set of true pattern instances, and the human analyst can explain the presence of false positive and the absence of false negative instances. The output is the set of validated pattern instances.

Reconstructing and analyzing the architecture

In the final step, the analyst uses a visualization tool, such as Rigi, to align the recognized architectural pattern instances with the designed pattern instances, organizing the other elements in the source model around the detected instances. The resultant architecture can be analyzed for deviations from the designed architecture.

Result and Discussion:

1) Architecture evaluation is the analysis of a system's capability to satisfy the most important stakeholder concerns, based on its large-scale design, or architecture. In other words, areas of further development in the system are identified.
2) Architecture tradeoff analysis method is a risk-mitigation process used early in the software development life cycle. ATAM was developed by the Software Engineering Institute at the Carnegie Mellon University
3) Quality Attribute Workshops (QAWs) provide a method for identifying a system's architecture-critical quality attributes, such as availability, performance, security, interoperability, and modifiability, that are derived from mission or business goals.
4) Architecture reconstruction is the process of obtaining the "as-built" architecture of an implemented system from the existing legacy system. For this process, tools are used to extract information about the system that will assist in building successive levels of abstraction.

Learning Outcomes: Students should have the ability to

LO1: Define Architecture evaluation.

LO2: Identify Architecture Tradeoff Analysis Method.

LO3: Learn Quality Attribute Workshops.

LO4: Learn Architecture reconstruction.

Course Outcomes: Upon completion of the course students will be able to learn

1.Architecture evaluation, analysis and design.
2.Architecture Tradeoff Analysis Method (ATAM).
3.Quality Attribute Workshops (QAW). □
4.Architecture reconstruction.

Conclusion:

This experiment was performed as a case study for - Architecture evaluation, analysis and design; Architecture Tradeoff Analysis Method (ATAM); Quality Attribute Workshops (QAW); Architecture reconstruction

Viva Questions:

1. Define Architecture Evaluation, analysis and design.
2. Explain Architecture Tradeoff Analysis Method.
3. Explain Quality Attribute Workshops.
4. Explain Architecture reconstruction.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	