APPENDIX 1

The conceptual model of the principles proposed in Dynamic Aspect Weaving approach are described using Web Ontology Language (OWL) to enhance the understandability and to easily transforming them to any platform specific model. The OWL document describing the conceptual model of the proposed Dynamic Change Aspect shown in Figure 3.1 of Chapter 3.1.3 is listed below:

```xml
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#" >

<owl:Class rdf:ID = "Dynamic Change Aspect">
</owl:Class>

<owl:Class rdf:ID = "Software Changes">
</owl:Class>

<owl:Class rdf:ID = "Software Functional Unit">
</owl:Class>

<owl:ObjectProperty rdf:ID = "encapsulates">
    <rdfs:domain rdf:resource = "&owl; FunctionalProperty/>
    <rdfs:range rdf:resource = "&owl; Software Changes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID = "associate with">
    <rdfs:domain rdf:resource = "&owl; Software Functional Unit"/>
    <rdfs:range rdf:resource = "&owl; Software Changes"/>
</owl:ObjectProperty>
```

<owl:ObjectProperty rdf:ID = “binds with”>
  <rdf:type rdf:resource = “&owl; FunctionalProperty” />
  <rdfs:domain rdf:resource = “#Dynamic Change Request” />
  <rdfs:range rdf:resource = “#Software Functional Unit”>
    <owl:Restriction>
      <owl:onProperty rdf:resource = “associate with” />
      <owl:allValuesFrom
        rdf:resource = “#Software Functional Unit” />
    </owl:Restriction>
  </owl:ObjectProperty>

The ontology specification of the proposed Dynamic Structural Introspection described in Figure 3.2 of Chapter 3.2.2 is given below:

<rdf:RDF

  xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”
  xmlns:rdfs=”http://www.w3.org/2000/01/rdf-schema#”
  xmlns:owl=”http://www.w3.org/2002/07/owl#”>

  <owl:class rdf:ID = “Dynamic Meta Model”>
  </owl:class>

  <owl:class rdf:ID = “Meta Interface”>
  </owl:class>

  <owl:class rdf:ID = “Meta Implementation Class”>
  </owl:class>

  <owl:class rdf:ID = “Meta Method”>
  </owl:class>

  <owl:class rdf:ID = “Application Component”>
  </owl:class>

  <owl:ObjectProperty rdf:ID = “associate with”>
    <rdf:type rdf:resource = “&owl; FunctionalProperty” />
    <rdfs:domain rdf:resource = “#Application Component” />
    <rdfs:range rdf:resource = “#Software Changes” />
  </owl:ObjectProperty>

  <owl:ObjectProperty ref:ID = “has”>
    <rdf:type rdf:resource = “&owl; FunctionalProperty” />
    <rdfs:domain rdf:resource = “#Dynamic Meta Model” />
    <rdfs:range rdf:resource = “#Meta Interface” />
  </owl:ObjectProperty>

</rdf:RDF>
<owl:Restriction>
  <owl:onProperty rdf:resource = “associate with” />
  <owl:allValuesFrom ref:resource = “#Application Component” />
</owl:Restriction>
</owl:ObjectProperty>

<owl:ObjectProperty ref:ID = “has” >
<rdfs:type rdf:resource = “&owl; FunctionalProperty” />
<rdfs:domain rdf:resource = “#Dynamic Meta Model” />
<rdfs:range rdf:resource = “#Meta Implementation Class” />
  <owl:Restriction>
    <owl:onProperty rdf:resource = “associate with” />
    <owl:allValuesFrom ref:resource = “#Application Component” />
  </owl:Restriction>
</owl:ObjectProperty>

<owl:ObjectProperty ref:ID = “has” >
<rdfs:type rdf:resource = “&owl; FunctionalProperty” />
<rdfs:domain rdf:resource = “#Dynamic Meta Model” />
<rdfs:range rdf:resource = “#Meta Method” />
  <owl:Restriction>
    <owl:onProperty rdf:resource = “associate with” />
    <owl:allValuesFrom ref:resource = “#Application Component” />
  </owl:Restriction>
</owl:ObjectProperty>

The ontology specification of dynamic function invocation concept described in Figure 3.3 of Chapter 3.3.2 is stated below using Web Ontology Language.
<owl:class rdf:ID = “Dynamic Function Adapter” >
</owl:class>

<owl:class rdf:ID = “Application Functional Component” >
</owl:class>

<owl:class rdf:ID = “Change Implementer” >
</owl:class>

<owl:ObjectProperty rdf:ID = “uses” >
<rdfs:type rdf:resource = “&owl; FunctionalProperty” />
</owl:ObjectProperty>
<rdfs:domain rdf:resource="#Dynamic Function Adapter" />
<rdfs:range rdf:resource="#Dynamic Meta Model" />
</owl:ObjectProperty>

<owl:ObjectProperty ref:ID = " is a subclass of" >
<rdfs:type rdf:resource="#&owl; FunctionalProperty" />
<rdfs:domain rdf:resource="#Dynamic Function Adapter" />
<rdfs:range rdf:resource="#Application Functional Component" />
</owl:ObjectProperty>

<owl:ObjectProperty ref:ID = " binds with" >
<rdfs:type rdf:resource="#&owl; FunctionalProperty" />
<rdfs:domain rdf:resource="#Change Implementer" />
<rdfs:range rdf:resource="#Dynamic Function Adapter" />
</owl:ObjectProperty>

The components of the proposed adaptable middleware model are implemented using AdaptManager, ChangeRequestParser, ComponentMeta, CFWrapper, ChangeRequest and AspectWeaver classes. These classes (except ChangeRequest) are developed using Java and ChangeRequest class is implemented in AspectJ. The list of codes associated with the implementation of these classes is described below:

AdaptManager class: The receiveRequestXml method is defined to receive the request for changes in the requirements by passing the xml formatted request in string. The method adaptController invokes the operations such as xmlParser defined in ChangeRequestParser class to parse the change request xmlstring and returns the details such as component name, service name and change specifications, aspectgen implemented in AspectGenerator class by passing the change specification and service details required to generate aspects, initiateWrap defined in CFWrapper class to create wrapper for the function associated with the change by providing component and service details and aspectweave to bind the advice defined in the ChangeRequest
aspect class with the wrapper generated in CFWrapper. The attributes defined in the class are representing xml formatted change request, component name, service (method) name and change request specification. The above mentioned details are specified in the AdaptManager java class that is shown below.

```java
public class AdaptManager {
    static String changespec;
    static String requestxml;
    static String componentname;
    static String methodname;
    static String[] changerequestparam;
    static ChangeRequestParser crp = new ChangeRequestParser();
    static AspectGenerator ag = new AspectGenerator();
    static CFWrapper cfw = new CFWrapper();
    static AspectWeaver asw = new AspectWeaver();

    /* receive the request for changes in functionality of the system */
    public void receiveRequestXml(String changerequest){
        requestxml= changerequest;
    }

    /* sequencing the activities involved in the adaptation process */
    public void adaptController(){
        changerequestparam = crp.xmlParser(requestxml);
        componentname = changerequestparam[0];
        methodname = changerequestparam[0];
        ag.aspectgen(changespec, methodname);
    }
}
```
```java
    cfw.initiateWrap(componentname, methodname);
    asw.aspectweave();

    }

public static void main(String args[]){
    AdaptManager am = new AdaptManager();
    String changerequest = null;
    am.receiveRequestXml(changerequest);
    am.adaptController();

    }
}
```

**ChangeRequestParser class:** This class receives the request for changes in the requirements in the xml structure and extracts the details specified in the xml file by parsing it. The method xmlParser is implemented to parse the xml file using document object model parsing technique. The file to be parsed is sent by the AdaptManager class and the information extracted during parsing is returned to AdaptManager class. The implementation of parsing procedure in java is shown below.

```java
public class ChangeRequestParser {
    /* parses the change request represented in the prescribed xml format */
    public String[] xmlParser(String fileName) {
        String[] changerequest = null;
        try {
            File fXmlFile = new File(fileName);
            DocumentBuilderFactory dbFactory =
```
DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder =
dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(xmlFile);
doc.getDocumentElement().normalize();
doc.getDocumentElement().getNodeType());
NodeList nList =
doc.getElementsByTagName("CHANGEREREQUEST");

for (int temp = 0; temp < nList.getLength();
temp++) {
    Node nNode = nList.item(temp);
    if (nNode.getNodeType() ==
        Node.ELEMENT_NODE) {
        Element eElement = (Element) nNode;
        changerequest[0] =
            getTagValue("COMPONENTNAME",
                eElement);
        changerequest[1] =
            getTagValue("OPERATIONNAME", eElement);
        changerequest[2] =
            getTagValue("CHANGESPECIFICATION",
                eElement);
    }
}

} catch (Exception e) {
    e.printStackTrace();
}

return changerequest;
private String getTagValue(String sTag, Element eElement) {
    NodeList nList =
    Element.getElementsByTagName(sTag).item(0).getChildNodes
    ();
    Node nValue = (Node) nList.item(0);
    return nValue.getNodeValue();
}

ComponentMeta: The implementation of extracting the structural details of
the component that is associated with the changes at run-time is shown in the
ComponentMeta java class. The reflection service defined in java is used to
extract the details of interfaces, implementation classes and operations
defined in the application component. The objectLookup, classLookup,
interfaceLookup and methodLookup methods defined in the ComponentMeta
class are implementing the dynamic structural introspection procedure
proposed in the present work.

public class ComponentMeta {
    /* returns the remote object reference */
    public Object objectLookup(String objname) {
        Object obj = new Object();
        /*try {
            String url = "rmi://localhost/";
            obj = Naming.lookup(url + objname);
        } catch (Throwable e) { */
        System.err.println(e);
        /*}
        return obj;*/
    }
}
/* extracts the class associated with the object */

public Class<?>[] classLookUp(String objname) {
    Object objt = objectLookUp(objname);
    @SuppressWarnings("rawtypes")
    Class c = objt.getClass();
    System.out.println(c);
    return c;
}

/* extracts the interfaces implemented in the class */

public Class<?>[] interfaceLookUp(String objname) {
    Class<?> cl = classLookUp(objname);
    Class<?>[] intfame = cl.getInterfaces();
    System.out.println(intfname);
    return intfame;
}

/* extract the methods defined in the interface */

public Method [] methodLookUp(String objname) {
    Class<?>[] ifname = interfaceLookUp(objname);
    Method [] m = ifname[0].getDeclaredMethods();
    return m;
}

CFWrapper class: The dynamic function invocation technique proposed in the present work is implemented in the CFWrapper class. The details required to perform the wrapping process is passed by AdaptManager class through initiateWrap method. The methodWrapper method uses the ComponentMeta class to fetch the methods associated with the changes by sending the name of
the interfaces of the components specified in the change request. It generates wrapper for each method associated with the changes by invoking that method at run-time.

```java
public class CFWrapper {
    String methodname;
    String comobjectname;
    /* invoke the wrapper generator */
    public void initiateWrap(String componentname, String operationname) {
        comobjectname = componentname;
        methodname = operationname;
        CFWrapper wrap = new CFWrapper();
        wrap.methodWrapper();
    }
    /* generating wrapper for invoking the method */
    public void methodWrapper() {
        int i=0;
        ComponentMeta comp = new ComponentMeta();
        Object[] objt1 = comp.interfaceLookup(comobjectname);
        try {
            Method[] meth = comp.methodLookup(comobjectname);
            int len = meth.length;
            for (i=1; i<= len; i++) {
                String mname = meth[i].getName();
                if (mname == methodname){
                    Object[] objectParameters = {new String[] {}};
                    objectParameters[0]="";
                    meth[i].invoke(objt1[0],objectParameters);
                }
            }
        }
    }
```
```java
    } } }
    catch (Throwable e) {
        System.err.println(e);
    }
}

**AspectGenerator class:** This class receives the information required to generate the aspect class to implement the changes. The aspectgen method is invoked by the AdaptManager class to provide the details about the change specifications. The service of providing the change specification details is implemented in getCspec method. The process for initiating the aspect generation is implemented in this class.

```java
    public class AspectGenerator {
        String changespec;
        String methodname;

        // initiates the aspect generation process */
        public void aspectgen(String cspec, String mname){
            changespec = cspec;
            methodname = mname;
        }
        /*provides the change specification details*/
        public String getCspec(){
            return changespec;
        }
    }
```

**ChangeRequest aspect class:** The getChangespect method is defined to get the details about the changes specified in the requirements from
AspectGenerator class. The methodWrapper method defined in CFWrapper is specified as pointcut dynRequest in this class. This pointcut is common for all changes. The dynRequest advice template is given in this class to implement the details specified in the change request.

```java
public aspect ChangeRequest {
    String changespecs = "version1";
    ChangeRequest();
    /* gets the change specification details*/
    public void getChangespec() {
        AspectGenerator asg = new AspectGenerator();
        changespecs = asg.getSpec();
    }
    /* define the pointcut as methodWrapper */
    pointcut dynRequest() : execution (void
        CFWrapper.methodWrapper());
    before(): dynRequest();
    // after(): dynRequest();
    // around(): dynRequest();
    /* generate the file path to store this aspect class */
    public void constructFileNamePath() {
        String filepath = File.separatorChar
            + "java" + File.separatorChar + changespecs + "ChangeRequest";
        System.out.println("The path of the file is : " + filepath);
    }
}
```

AspectWeaver class: The dynamic weaving process is implemented through aspectweave method defined in this class.
public class AspectWeaver {
    /* executes the commands associated with aspect weaving */
    public void aspectWeave() {
        try {
            Runtime rt = Runtime.getRuntime();
            rt.exec("ajc ChangeRequest.aj -outxml -outjar timing.jar");
            rt.exec("java -javaagent:c:\aspectj1.6\lib\aspectjweaver.jar -classpath .;timing.jar;c:\aspectj1.6\lib\aspectjrt.jar CFWrapper");
        }
        catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
APPENDIX 2

Case Study on Sales Processing System

Nowadays, the sales transactions are being performed in online. The online sale enables the customers and the sellers to perform their transaction in an efficient way. Following are the requirements stated for the sales processing system:

- Requirement # S01: Check the validity of the sales order
- Requirement # S02: Order confirmation and delivery scheduling
- Requirement # S03: Update order status to the customer
- Requirement # S04: Bill generation and Payment
- Requirement # S05: Shipment process

These requirements are implemented in the software system through Order Processor, Order Status Publisher, Bill Generator, Payment Handler and Shipment Processor classes and the set C is defined with these classes. The Order Processor class is defined with the methods such as orderValidation to validate the information specified in the order quoted by the customer and orderConfirmation to check the feasibility of accepting the order. The methods of Order Status Publisher class are eMail and sMs, which are used to notify the status of the order in process. The BillGenerator comprises the salesInvoice method. The Payment Handler class handles the
cash and credit payment through the methods namely cash and credit respectively. The trackNotify method is defined in the Shipment Process class to notify the status of the shipping and delivery details. The structure of the sales processing system is shown in Figure A2.1.

This online sales processing system should be more reliable and securable. Hence, the processes specified in the system could require accommodating the changes in their strategies. This leads to have changes in validating the order using the additional information such as customer bank details (bank IFSC code) and product code that includes bar code information. Similarly, the changes in the order confirmation policy are specified based on the details of the previous transactions done by the customer. The order status only specifies the information about the intermediate states of the order processing task, whereas the information about the expected time of completion of each operation in the order process to be specified. Hence, such changes are to be incorporated into the order status template. The changes in the payment process are required to ensure the security and reliability of the process. The template for delivery status specifications should be defined with the provision for accommodating the variations in the shipment details based on the shipping organization.
Figure A2.1 Sales processing system structure
The changes that occur in the sales transaction scenario are incorporated into the system through the proposed adaptable middleware model. The automation of the adaptation process through realizing the changes using dynamic change aspect defined for each change and weaving the changes into the functions using dynamic wrapper of that function. The Order Confirmation Policy Change, Order Status Spec Change, Payment Strategy Change and Delivery Status Spec Change aspect classes are created to implement the changes discussed in the sales transaction process. The implementation of the changes is specified in the advice of the appropriate aspect class. Each advice is weaved with the methods of sales processing application classes that are associated with the changes. For instance, change in the order confirmation policy is realized through Order Confirmation Policy Change aspect class and the advice implements the order confirmation policy change is weaved with orderConfirmation method of Order Processor class in the system. Similarly, the realization and binding of the sales transaction changes are achieved. Thus using proposed adaptable middleware, the sales processing system is adapting to the changes in the requirements that occur at run-time.

The adaptability efficiency of the sales transaction system is evaluated using Conceptual Binding between Aspect and Classes (CBAC) and Functional Change Adaptation Time (FCAT) metrics. The conceptual binding between aspect and classes represents the association between the changes implemented in the aspect and the methods of the classes that adapting to the changes in the requirements. This association is shown through the realization of the requirements in the methods of the classes. Hence, Requirement Class Association Matrix has been generated to show the requirements realization details.
The methods defined in each class and their association with the requirements is shown in Table A2.1. Requirement #SO1 is realized in the orderValidation method of Order Processor class. Hence, changes in the order validation should be reflected on orderValidation method and the corresponding CBAC value is 1/5, where 5 is number of classes in the system. Order confirmation process is implemented in orderConfirmation method of Order Processor class and the corresponding notification is specified using eMail and sMs methods of Order Status Publisher class. Hence, the association between process of order confirmation and sales processing system is derived as 3 and adaptability of changes in this process is specified as 3/5. Similarly, order status notification process is realized in the methods of Order Status Publisher and Shipment Process classes and CBAC value of adapting to the changes in this process is 3/5. The coupling between the Requirement #SO4 with the classes in the system is measured as 3, because it is realized in the salesInvoice, cash and credit methods of Bill Generator and Payment Handler classes. Hence, adapting to the changes in the bill generation and payment process is measured as 3/5. The adaptability measures determined for incorporating the dynamic changes stated in the sales processing system is shown in Table A2.2.
Table A2.1 Requirements class association matrix for sales processing system

<table>
<thead>
<tr>
<th></th>
<th>Order Processor</th>
<th>Order Status Publisher</th>
<th>Bill Generator</th>
<th>Payment Handler</th>
<th>Shipment Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Validation</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Order Confirmation</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>eMail</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sMs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sales Invoice</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cash</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>credit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Track Notify</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table A2.2 CBAC factor for sales processing system

<table>
<thead>
<tr>
<th>Dynamic Changes</th>
<th>CBAC Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Validation Strategy Changes (Single Reflection)</td>
<td>0.2</td>
</tr>
<tr>
<td>Order Confirmation Policy Changes (Multiple Reflection)</td>
<td>0.6</td>
</tr>
<tr>
<td>Changes in the Format/ Mode of Order Status Notification (Multiple Reflection)</td>
<td>0.6</td>
</tr>
<tr>
<td>Payment Process Changes (Multiple Reflection)</td>
<td>0.6</td>
</tr>
<tr>
<td>Changes in the Shipment Procedure (Single Reflection)</td>
<td>0.2</td>
</tr>
</tbody>
</table>
The above discussion shows the ways in which the adaptability of changes in the requirements is measured using the number of methods/classes in the system involved in the realization of those requirements. Also it is derived that the value of CBAC associated with the changes in the requirement is high when more number of methods and classes are associated with the changes. Changes in the requirements are classified into two types namely, changes to be reflected on one method/class (Single Reflection) and changes to be reflected on multiple methods / classes (Multiple Reflection). The CBAC values observed for adapting these categories of changes are shown in Table A2.2. From this observation, it is concluded that the CBAC value for single reflection is \( \frac{1}{5} \), where \( \frac{1}{5} \) represents number of classes in the system; CBAC value for two reflections is \( \frac{2}{5} \); CBAC value for three reflections is \( \frac{3}{5} \); and CBAC value for \( n \) number of reflections is \( \frac{n}{5} \). Adaptability values observed for the changes stated in the sales transaction clearly shows that „The adaptability of the system (AOS) is increased with increasing the CBAC factor”. This result is clearly shown in the adaptability chart depicted in Figure A2.2.

![Figure A2.2 Adaptability chart](image-url)
The evaluation of adaptability of a sales transaction processing system is performed using the structural design of the system, which comprises the classes, components and interfaces along with their internal structure. Hence, the CBAC metric is used to evaluate the adaptability at the architectural level. For evaluating the adaptability at execution level is performed through measuring the time taken for executing the changes after incorporating them into the appropriate methods of the classes. The Functional Change Adaptation Time (FCAT) metric has been proposed to perform the adaptability evaluation at execution time. The activities involved in evaluating the adaptability of a sales processing system using FCAT metric is described below.

The classes in the sales transaction system are implemented as Java classes. They are interpreted as distributed classes through defining remote interfaces for each class using RMI framework. The changes implemented as Aspect classes using AspectJ. The dynamic weaving is performed through load time mechanism defined in AspectJ along with the dynamic wrapper generated for each method associated with the change. The time taken for weaving the advice defined in the Aspect class with the wrapper and the time taken for executing the wrapper is measured during the invocation of the changes specified at run-time. The total time consumed towards the change adaptation process performed while adapting to the changes specified in the sales transaction scenario is measured as shown in Table A2.3. The data shown in the table is the average of the time taken for adapting to the changes while executing the adaptation process as described in the proposed adaptable middleware for each change in many cycles.
Table A2.3  FCAT for sales processing system

<table>
<thead>
<tr>
<th>Dynamic Changes</th>
<th>FCAT Value (Milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Validation Strategy Changes</td>
<td>362</td>
</tr>
<tr>
<td>Order Confirmation Policy Changes</td>
<td>375</td>
</tr>
<tr>
<td>Changes in the Format/ Mode of Order Status Notification</td>
<td>341</td>
</tr>
<tr>
<td>Payment Process Changes</td>
<td>406</td>
</tr>
<tr>
<td>Changes in the Shipment Procedure</td>
<td>314</td>
</tr>
</tbody>
</table>

According to the FCAT metric definition, the time taken for executing the function along with the changes is only considered to show the adaptability efficiency of a system. The data shown in Table A2.3 is also derived using FCAT metric. Hence, it is concluded that the adaptability to the changes using proposed adaptable middleware is performed without reconfiguration, which implies that the time taken for adapting to the changes is considerably less, when compare with other adaptability approaches.