Unified Incident Command and Decision Support (UICDS)

TechNote – Adapter Development 101

For UICDS Software Version 1.2.x

18 June 2013

UICDS-TECHNOTE-AG101-R01C00

Prepared for EAGLE Task Order HSHQDC-12-J-00110

Science Applications International Corporation
4001 North Fairfax Drive, Suite 250
Arlington, VA 22203

Approvals

The signatures below constitute approval of this document.

Contributors

Daphne Hurrell
UICDS Technical Lead

Submitter

Daphne Hurrell
UICDS Technical Lead

Approver(s)

Chip Mahoney
UICDS Project Manager

Dr. James Morentz
UICDS Outreach Director

Mark Boriack
Quality Assurance

Contents

1.0 DOCUMENT PURPOSE 1

2.0 SUMMARY OF ARCHITECTURE 1

3.0 OVERVIEW OF INTERACTIONS WITH UICDS CORE 2

4.0 GENERAL ADAPTER CONCEPTS 3

5.0 HOW AN ADAPTER INTERACTS WITH UICDS 4

6.0 UICDS EXAMPLE CODE 6

7.0 JAVA EXAMPLE CODE 6

7.1 SPRING WEB SERVICE CLIENT 6
7.2 SUMMARIZATION OF CODE 7
7.3 AXIS2 13

8.0 C# EXAMPLE CODE 13

8.1 SUMMARIZATION OF CODE 14

Revisions

Revision Number	Date	Description
R01C00	18 June 2013	Initial document.

1.0 *Document Purpose*

This document accompanies an online, on-demand UICDS Tutorial that describes the development of an adapter using UICDS Example Code which is available in both Java and C# examples. While the tutorial highlights the Dot Net environment, the same principles pertain to the Java environment, both of which are described below.

To view the tutorial, go to https://uicds.kzoinnovations.com/swf/player/1309 with your UICDS Collaboration Portal credentials. In addition, Section 9 contains links to other tutorials of use in adapter development.

2.0 *Summary of Architecture*

The UICDS architecture, as illustrated in Figure 1, is a partial mesh network of UICDS servers that allow clients to collaboratively assemble and share emergency management information about an incident using web services provided by a UICDS server. Collaboration occurs between clients on one UICDS server based on subscription profiles and among clients on different UICDS servers based on information sharing agreements between the UICDS servers. Creation or modification of shared information is communicated to clients via a notification service. Clients can be directly addressed for targeted notification delivery through a combination of unique client instance and core names.

The UICDS architecture is built on service-oriented principles using open standards. Each UICDS server, named a UICDS Core, serves as a local point of integration for technology providers and agency services. UICDS Cores consist of three varieties of services: infrastructure, domain, and external.

- Infrastructure services enable the sharing of information between cores and are based on existing, established industry standards. UICDS Infrastructure Services provide the fundamental UICDS capabilities such as work product management, information sharing agreements, external notifications, UICDS resource management and event/error logging.
- UICDS Domain Services provide the interfaces to external applications and enable the exchange of UICDS Work Products. Domain services provide for the management of information specific to emergency management; such as incidents, command hierarchies, tasking, and the common operating picture. These services rely on existing and developing standards in the emergency management domain such as those from NIEM and the OASIS EM Technical Committees.
- Finally, in addition to the standard UICDS services provided by a core, each core provides the ability to register external services for connections, thus allowing for unanticipated data formats and developing and future standards.

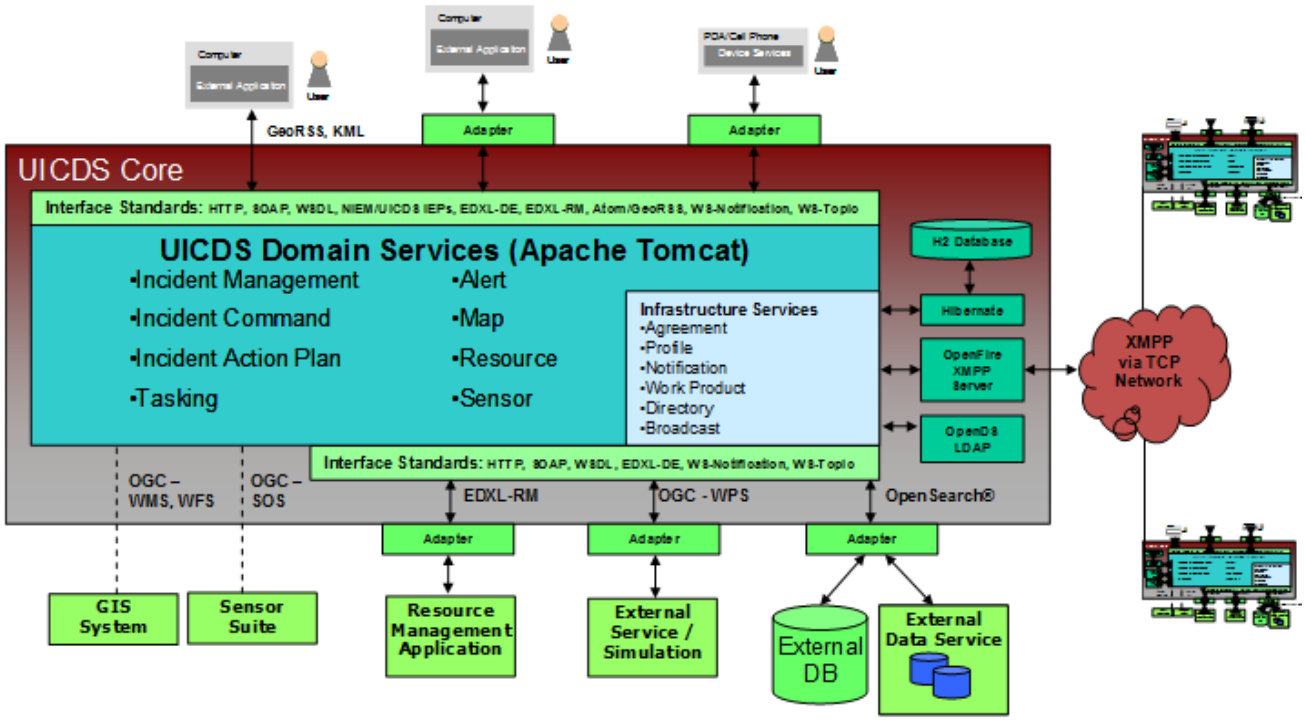


Figure 1, UICDS System Architecture

3.0 *Overview of Interactions with UICDS Core*

The following figures illustrate the various UICDS functions.

Figure 1shows a simplified version how an incident is created in the core and how the adapters are being notified about that incident.

An Incident Management Application with a UICDS adapter (1) creates an incident on the UICDS core and the core automatically creates (2) three default work products: Incident, Map and Incident Command. At this point, the UICDS core puts notifications (3) in the queue of any Resource Instances that have registered to be notified for any of the three types of work products.

In the diagram, the Incident Management Application is receiving the Incident and Map Work Product notification while the Planning Application is getting notifications for all the three types of work products. The important point is that the applications receive notifications based on what is in their Resource Profile. Both Resource Instance and Resource Profile are discussed in section 4.

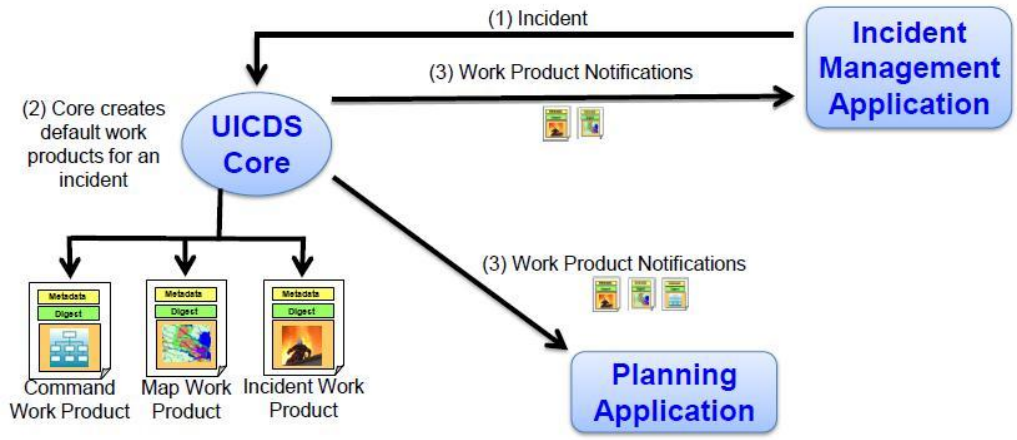


Figure 2, Incident creation and client notification

Figure 3 shows an application update of a work product.

The Planning Application receives (2) the Map Work Product created notification. It adds a layer of an evacuation route (1) to the Map Work Product and updates the Map Work Product to the UICDS Core. The Incident Management Application then receives the notification (2) of the Map Work Product update.

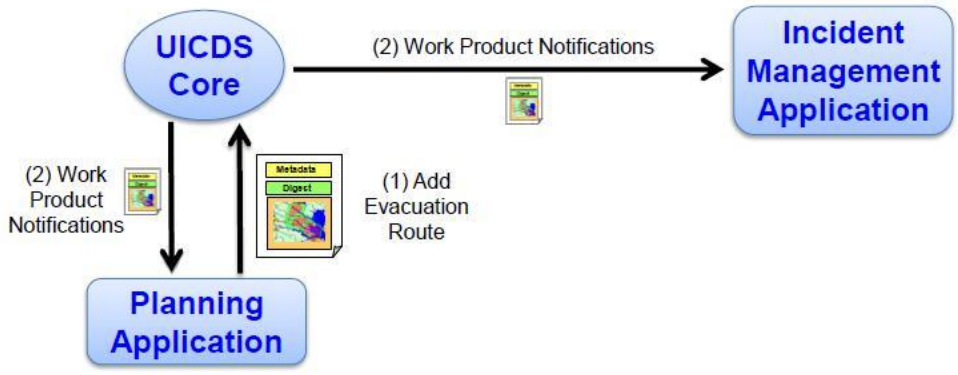


Figure 3, Work product update

4.0 *General Adapter Concepts*

The main functions of an adapter are to create and update a UICDS Work Product. Adapters accomplish this by:

1. Translating application data contained in the application into a UICDS Work Product.
2. Posting the work product to the UICDS Core.
3. Consuming a work product from the UICDS Core.
4. Translating work product data into application specific data model for use in the application and/or visualize in the application user interface.

The application is now getting data from an outside source which it may never have done before. An adapter may need to differentiate UICDS incidents from its own application incidents. There are several ways adapter can integrate with an application. The adapter can be integrated directly into the

application, for example as single the war file, or if the application already has an external API, the adapter can be programmed to interface with the API as a separate process as shown in Figure 4.

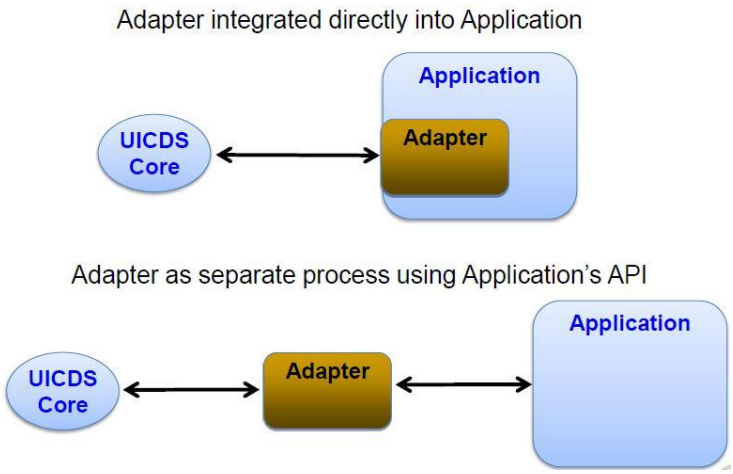


Figure 4, Adapter deployment concepts

Whether an application receives external data, exposes its data externally or conducts bidirectional data exchange, an adapter can be implemented to achieve the goal. In addition, an adapter is also the front line of data dissemination control to filter out data that data owners do not want to expose or is not needed by the application. Figure 5 illustrates these concepts.

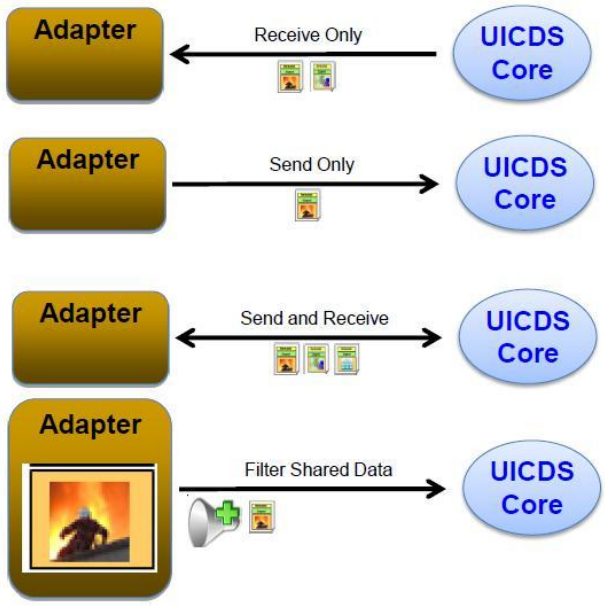


Figure 5, Adapter information flow concepts

5.0 *How an Adapter Interacts with UICDS*

Figure 6 takes the concept described in section 3 and shows how the adapter interacts with UICDS.

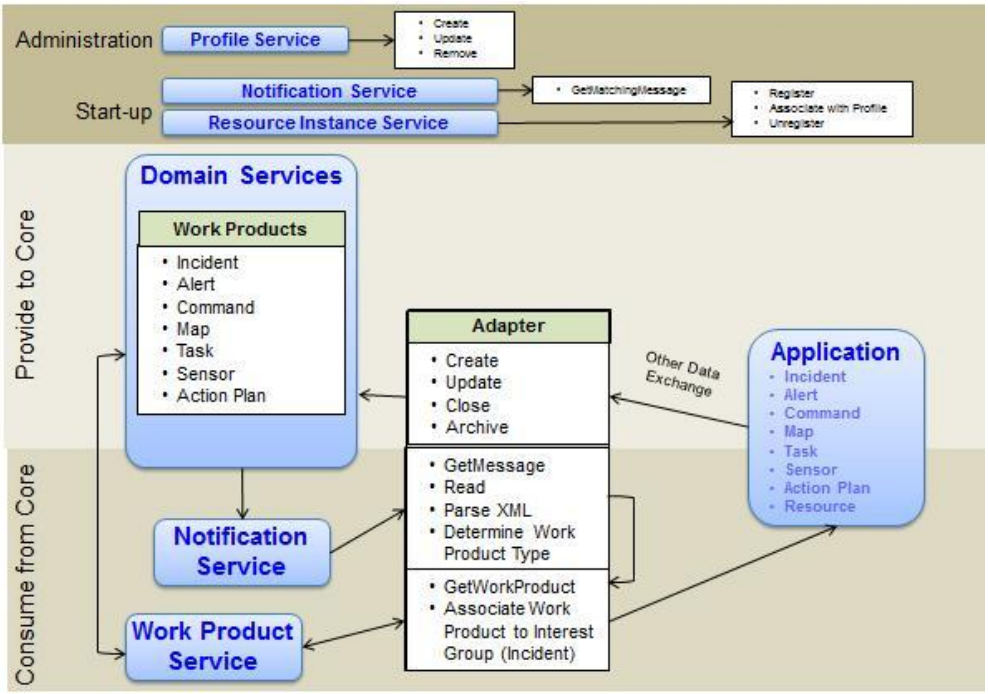


Figure 6, Adapter interactions with UICDS

The basics are on the very top of the diagram. The first thing the adapter should do is to create a Resource Profile to specify which type of work products it is interested in and wants to be notified about. After the Resource Profile has been created, you need a place to get notifications. In order to achieve that, you now create a Resource Instance. The Resource Instance will allow you to ask for notifications.

The Application on the right hand side of the diagram now can create, update, close and archive incidents and work products through the adapter. The adapter uses one of the appropriate UICDS Domain Services to achieve this goal. To create, update, close and archive incidents, the adapter will use the Incident Management Service.

An Application sends incident data to its adapter; the adapter then translates the data and creates an incident on UICDS core by posting a CreateIncident SOAP message to the UICDS Incident Management Service. The XML data is then parsed, digested, attached to the incident work product and stored in the Work Product Service. The Notification Service then adds a notification in the queues of all the Resource Instances that are registered to receive Incident Work Products. The next time the adapter calls GetMessages of the Notification Service, the adapter will receive a list of notifications since last time it called GetMessages. Note that the digest of the incident work product, included in the notification, may contain enough information to update the Application. If the digest information is not enough, the adapter can use the GetWorkProduct of the Work Product Service to retrieve the full incident work product to update the Application.

The sequence diagram illustrated in Figure 7 shows the overall flow as work products are created or updated. Adapters receive notifications and the adapters can get new or updated work products. This diagram uses the example of the Planning Application and Incident Management Application in section 2.0

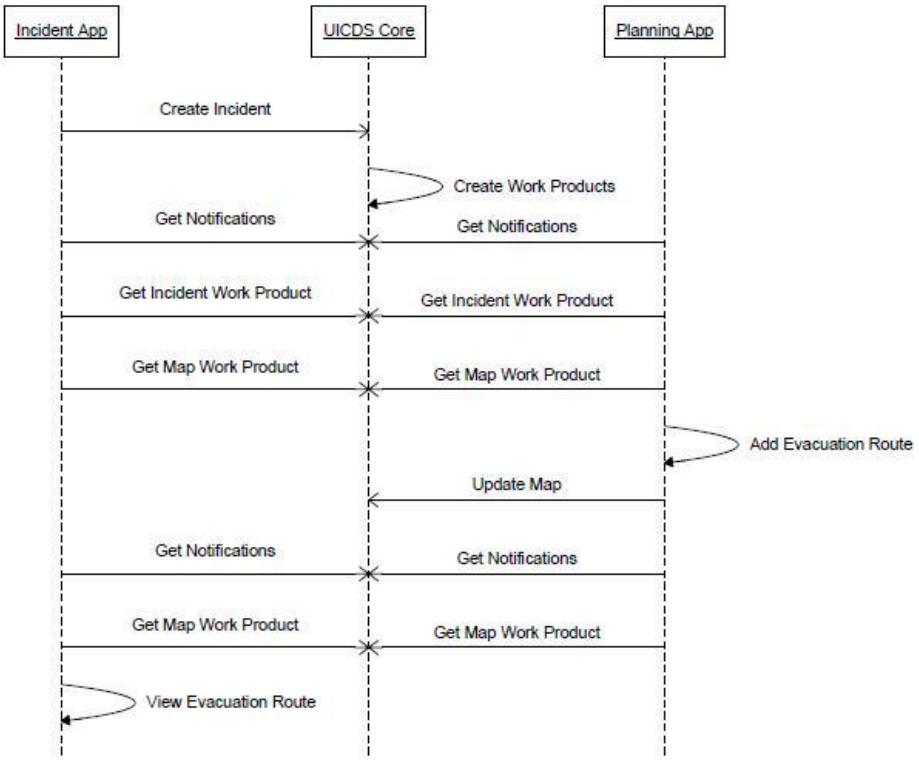


Figure 7, Work product create/update sequence

6.0 *UICDS Example Code*

UICDS example code is intended to show basic interactions with a UICDS Core. Java and C# example code are available to download on the UICDS download site https://uicds-test5.saic.com and demonstrate how to use UICDS Web Services. A UICDS client developer can potentially reuse the framework that is layout in the example code to develop his or her own UICDS adapter. The example code shows the basic incident lifecycle:

a. Create incident

b. Update incident

c. Close incident

d. Archive incident.

7.0 *Java Example Code*

7.1 Spring Web Service Client

The async client example code maintains a cache of incidents and associated work products and illustrates how to handle asynchronous responses from the core. The code does the following:

1. Creates a Resource Profile if one does not exist.

2. Registers a Resource Instance if one does not exist.

3. Gets Matching Messages to get cache of incidents and work products up to date with the core.

4. Processes notifications and get any updated work products to update the local cache.
5. Creates, updates, closes and archives an incident.
6. Checks status on updating an incident and wait if the update was PENDING.

The sample code uses an XmlBeans representation of the SOAP messages. You can use the jar files included in the UICDS InstallKit to build with. See the Readme file in the java sample code to understand how to extract the jar files into your local maven repository.

Figure 8 illustrates the async client class diagram. the basic framework of the UICDS core. The class that manages all the incidents is the WorkProductListener. Whenever the UICDS core processes incoming notifications, it will post events to each of the listener and the listener will perform whatever is necessary. In this example, when a work product is updated, the UicdsIncidentManager finds the correct instance of the UicdsIncident class and updates the work product. The UicdsResourceProfile and the UicdsApplicationInstance classes show how to build up a request to create a profile and register an application as a resource instance. The UicdsIncident class shows how to create, update, close and archive an incident.

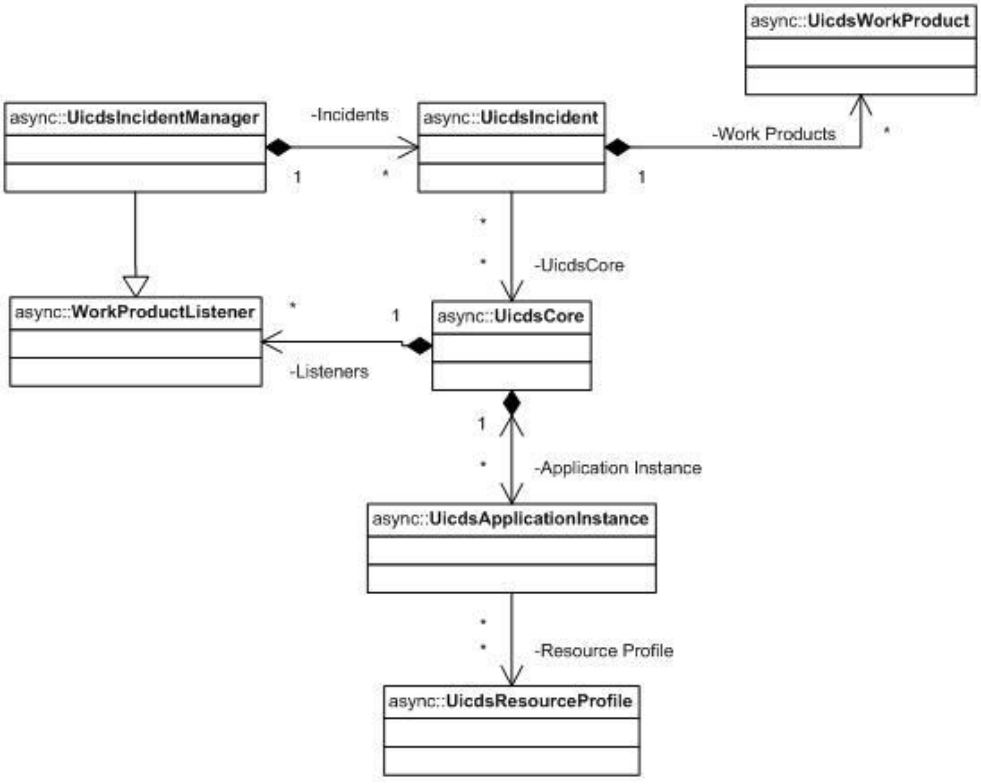
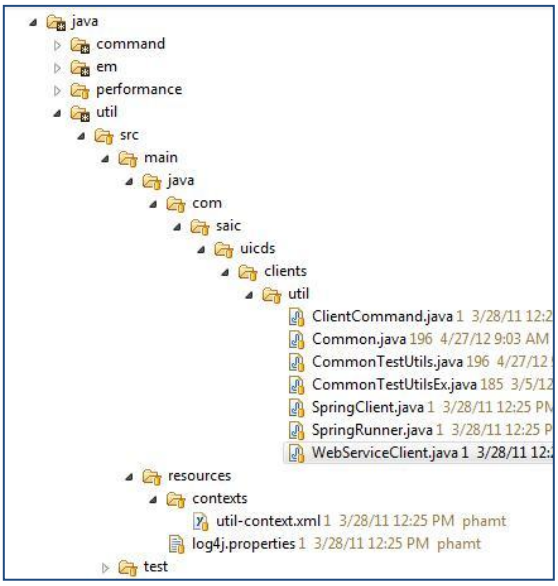


Figure 8, Async client class diagram

7.2 Summarization of Code

The UICDS Java example code directory tree structure is illustrated in Figure 9 and shows how to find the code summarized in the following sections.



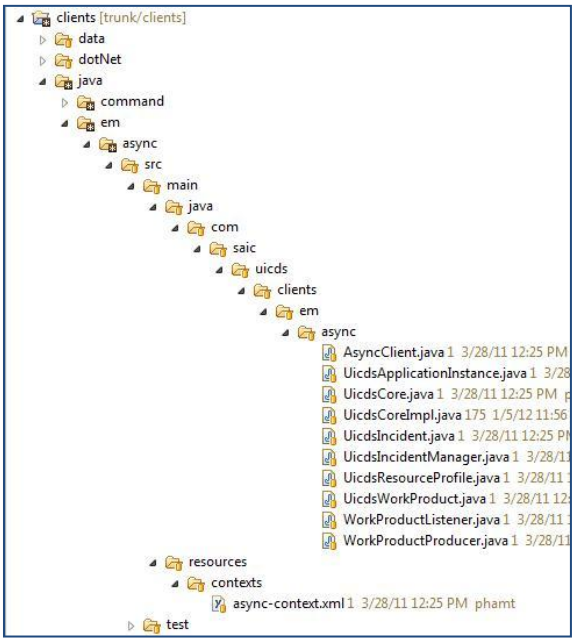


Figure 9, UICDS Java example code directory tree

A summary of the key functions performed by the example code and available to incorporate into UICDS Incident adapters follows:

Context File

Spring application context is used to configure the async client. The context file is async-context.xml and it can be found in the clients\java\em\async\src\main\resources\contexts\ directory of the sample code. Creating the Java proxy to access UICDS web services is done here in order to easily initialize the Spring Web Template.

```
<bean id="asyncClient" class="com.saic.uicds.clients.em.async.AsyncClient">
    <property name="uicdsCore" ref="uicdsCore" />
    <property name="webServiceClient" ref="springWebServiceClient" />
</bean>

<bean id="uicdsCore" class="com.saic.uicds.clients.em.async.UicdsCoreImpl">
     <property name="webServiceClient" ref="springWebServiceClient"/>
     <property name="applicationID" value="AsyncExample"/>
     <property name="applicationProfileID" value="AsyncProfile"/>
     <property name="localID" value="Async Example Client"/>
     <property name="applicationProfileInterests">
       <set>
           <value>Incident</value>
       </set>
     </property>
</bean>

<bean id="springWebServiceClient" class="com.saic.uicds.clients.util.SpringClient">
     <property name="webServiceTemplate" ref="webServiceTemplate"/>
</bean>
```

```
    <bean id="uicdsResourceProfile"
class="com.saic.uicds.clients.em.async.UicdsResourceProfile">
          <property name="webServiceClient" ref="springWebServiceClient"/>
    </bean>

    <bean id="uicdsIncident" class="com.saic.uicds.clients.em.async.UicdsIncident">
         <property name="uicdsCore" ref="uicdsCore"/>
    </bean>
```

Java Proxy

The proxy for the core is then available in this variable defined in AsyncClient.java:

```
private UicdsCore uicdsCore;
```

Initialize UicdsCore Object

The UicdsCore object is created by the Spring Framework in async-context.xml. This call in the AsyncClient.java sets the application specific data for the object.

```
private void initializeUicdsCore() {
```

UICDS Incident Class Instance

This call in the AsyncClient.java creates an instance of a class that represents a UICDS Incident and has the proxy for communicating with the core:

```
// Create an incident (gets all the associated work products)
        log.info("Creating Incident");
        UicdsIncident uicdsIncident = new UicdsIncident();
        uicdsIncident.setUicdsCore(uicdsCore);
```

Create UICDS Incident Document

In AsyncClient.java this line creates a UICDS Incident document from a file that is specified by the INCIDENT_EXAMPLE_FILE variable in AsyncClient.java. This file has XML where the data has already been inserted into the XML template:

```
// Create an IncidentDocument to describe the incident
        IncidentDocument incidentDoc = getIncidentSample();
```

Here is where your adapter potentially can replace this with your incident data.

Create Incident on UICDS Core

This instance of a UICDS Incident document can then be used in the following code in AsyncClient.java to have the uicdsIncident object create the incident on the core.

```
    // Create the incident on the core
        uicdsIncident.createOnCore(incidentDoc.getIncident());
```

In UicdsIncident.java, createOnCore puts the incident data into a CreateIncidentRequestDocument and which is then sent by the uicdsCore object and receives a response back.

```
public String createOnCore(UICDSIncidentType incident) {

        String wpid = null;
        CreateIncidentRequestDocument request =
CreateIncidentRequestDocument.Factory.newInstance();
        request.addNewCreateIncidentRequest().setIncident(incident);

        try {
            CreateIncidentResponseDocument response = (CreateIncidentResponseDocument)
uicdsCore.marshalSendAndReceive(request);

            Enum status =
response.getCreateIncidentResponse().getWorkProductPublicationResponse().getWorkProductPr
ocessingStatus().getStatus();
```

UICDS Core Interface

UicdsCore.java represents an interface to a UICDS Core. It allows a client to invoke a web service request using XmlBeans marshaling. It also handles asynchronous operations by keeping track of pending responses and watching for notifications that return the status of pending requests.

In the UicdsCoreImpl.java, marshalSendAndRecieve calls the webServiceClient in the clients\util directory to send the request.

```
      public XmlObject marshalSendAndReceive(XmlObject request) {

            XmlObject response = XmlObject.Factory.newInstance();
            try {
                  // log.debug("sending request to uicds core");
                  response = webServiceClient.sendRequest(request);
```

WebServiceClient

The Spring Client class in the clients\util directory implements the WebServiceClient. The sendRequest in SpringClient.java calls the marshalSendAndReceive Web Service Operations from the Spring framework to send a web service message.

```
public class SpringClient implements WebServiceClient {

    private static final String INCIDENT_SERVICE_NS = "http://uicds.org/incident";
    private static final String INCIDENT_ELEMENT_NAME = "Incident";

    protected WebServiceOperations webServiceTemplate;

   public XmlObject sendRequest(XmlObject request) {
        if (webServiceTemplate == null) {
            System.err.println("webServiceTemplate is null");
            return null;
        }
```

```
        if (request == null) {
            System.err.println("sendRequest failed : sendRequest is null");
            return null;
        }

        XmlObject response = (XmlObject)
webServiceTemplate.marshalSendAndReceive(request);

        return response;
    }
```

Update Incident

In this example code, the incident XML is updated in the UICDS Incident document in these lines in AsyncClient.java:

```
        // Get the current incident document
        UICDSIncidentType incidentType = uicdsIncident.getIncidentDocument();

        // Update the incident
        System.out.println("Updating incident: "
            +
incidentType.getActivityIdentificationArray(0).getIdentificationIDArray(0).getStringValue
());

        // Change the type of incident
        if (incidentType.sizeOfActivityCategoryTextArray() < 1) {
            incidentType.addNewActivityCategoryText();
        }
        incidentType.getActivityCategoryTextArray(0).setStringValue("CHANGED");
```

The updated work product is then used to update the incident on the core in this line:

```
    // Update the incident on the core
        ProcessingStatusType status = uicdsIncident.updateIncident(incidentType);
```

Again, the code also handles the pending status of the update (asynchronous) in the following lines:

```
        // If the request is pending then process requests until the request is
        // accepted or rejected
        // Get the asynchronous completion token
        if (status != null && status.getStatus() == ProcessingStateType.PENDING) {
            IdentifierType incidentUpdateACT = status.getACT();
            System.out.println("Incident update is PENDING");

            // Process notifications from the core until the update request is
            // completed
            // This loop should also process other incoming notification
            // messages such
            // as updates for other work products
            while (!uicdsCore.requestCompleted(incidentUpdateACT)) {
                // Process messages from the core
```

```
            uicdsCore.processNotifications();

            // Get the status of the request we are waiting on
            status = uicdsCore.getRequestStatus(incidentUpdateACT);
        }

        // Check the final status of the request
        status = uicdsCore.getRequestStatus(incidentUpdateACT);
        if (status.getStatus() == ProcessingStateType.REJECTED) {
            log.error("UpdateIncident request was rejected: " + status.getMessage());
        }
    } else if (status == null) {
        System.err.println("Processing status for incident update was null");
    } else {
        System.out.println("Incident update was ACCEPTED");
```

Close and Archive Incident

In the example code, the incident is closed and archived with the following lines in the AsyncClient.java:

```
System.out.println("Close the incident");
 uicdsIncident.closeIncident(uicdsIncident.getIdentification());

 System.out.println("Archive the incident");
 uicdsIncident.archiveIncident(uicdsIncident.getIdentification());
```

Credentials to Access the UICDS Core

The URL, username and password to enable access to UICDS web services is set in the clients\java\em\async\src\main\resources\contexts\async-context.xml file in this section of XML. The defaultUri needs to be changed from http://localhost/uicds/core/ws/services to https://YourFullyQualifiedDomainName/uicds/core/ws/services. Note http is used for localhost and https is used if you use FQDN. Two values of the UsernamePasswordCredentials need to be changed to valid credentials in the UICDS Core LDAP .

```
<bean id="webServiceTemplate" class="org.springframework.ws.client.core.WebServiceTemplate">
    <constructor-arg ref="messageFactory" />
    <property name="marshaller" ref="xmlbeansMarshaller" />
    <property name="unmarshaller" ref="xmlbeansMarshaller" />
            <property name="defaultUri" value="http://localhost/uicds/core/ws/services"/>
    <property name="messageSender">
      <bean class="org.springframework.ws.transport.http.CommonsHttpMessageSender">
        <property name="credentials">
          <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
            <constructor-arg value="user1" />
            <constructor-arg value="user1password" />
          </bean>
        </property>
      </bean>
```

```
    </property>
  </bean>
```

7.3 Axis2

This example code walks through the same incident life cycle in Axis2 framework. This is a file driven code that also uses the xml bean representations of the Web Service SOAP messages by fetching the WSDL from a core running on a localhost.

8.0 *C# Example Code*

The C# example code utilizes LINQ for xml parsing and processing the SOAP messages. The example code also maintains a cache of incidents and associated work products. The example code basically does the same thing as the java example code with couple of extra functionalities. The C# example code has the following functionality:

1. Registers a Resource Instance if one does not exist.
2. Gets a list of the current incidents.
3. Creates a local cache of all the current incidents and their associated work products.
4. Updates the ICS work product.
5. Processes notifications, gets any updated work products and updates the local cache.
6. Creates, updates, closes and archives an incident.
7. Adds a layer to the Map work product and posts the update.

Figure 10 illustrates the C# example code class diagram.

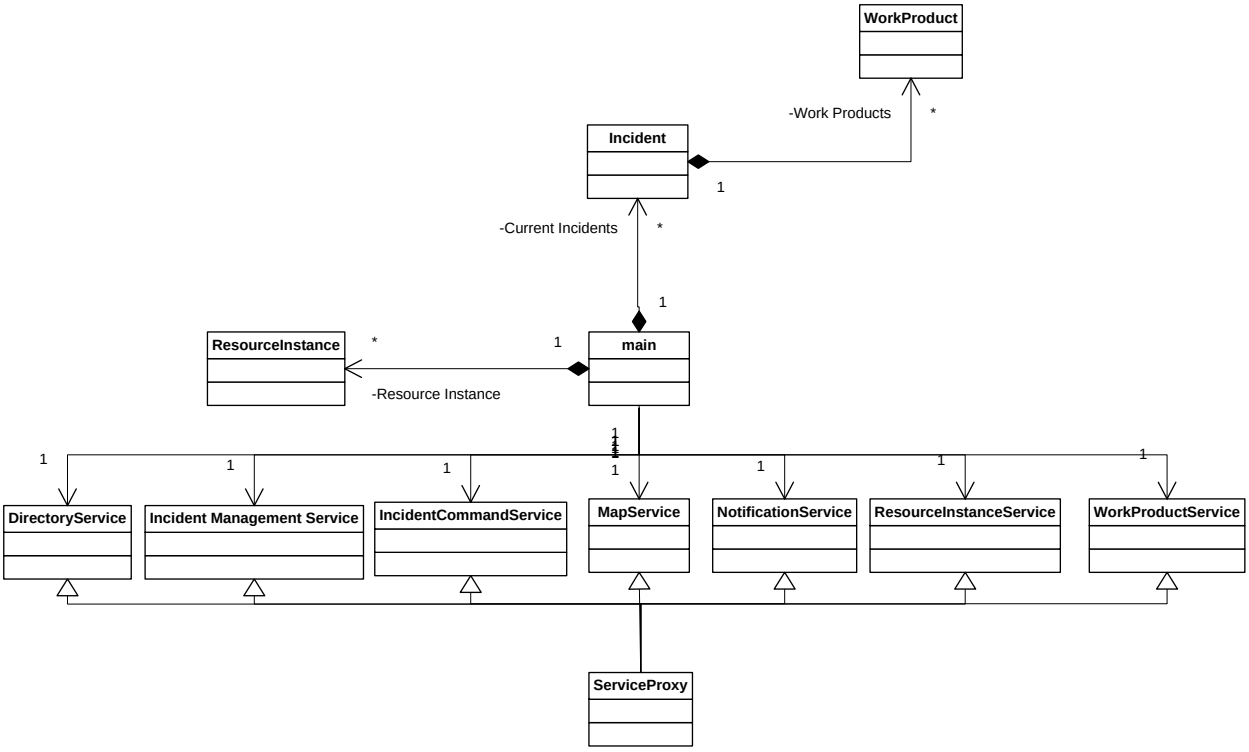


Figure 10, C# example code class diagram

Off the main there is a list of current incidents and ResourceInstance that presents the Resource Instance that the application is using. For handling the Services' service calls, classes that represent each of the services are built and derived from a ServiceProxy, which handles the lowest level interaction with the core. Each of the services allows you to build up the correct messages and uses the ServiceProxy call to proxy back and forth between the core.

8.1 Summarization of Code

The UICDS C# example code directory tree structure is illustrated in Figure 11 and shows how to find the various pieces of code.

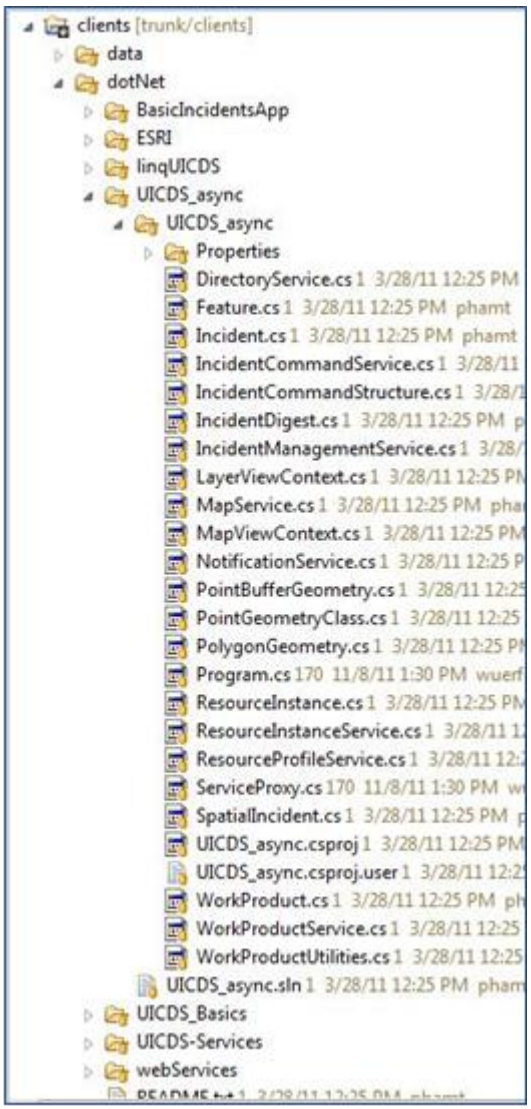


Figure 11, C# example code directory tree

A summary of the key functions performed by the example code is as follows:

1. Creating the .NET proxy for accessing the Incident Management Service is shown in this call:

```
// Create a proxy for the incident management service
IncidentManagementService incidentManagementService =
   CreateIncidentManagementServiceProxy(incidentManagementServiceURL, username,
password, workProductServiceProxy);
```

2. In Program.cs this call creates an instance of a class that represents an Incident work product:

```
// Create a local representation of the incident
Incident incident = createIncident();
```

3. The createIncident method in Program.cs sets the values for the necessary data:

```
private static Incident createIncident()
```

```
{
    Incident incident = new Incident();
    incident.name = "Flood";
    incident.dateTime = "2009-10-14T09:55:22";
    incident.description = "A flood on the river";
    incident.latitudeDegrees = 38;
    incident.latitudeMinutes = 17;
    incident.latitudeSeconds = 50.00;
    incident.longitudeDegrees = -77;
    incident.longitudeMinutes = 17;
    incident.longitudeSeconds = 50.00;
    incident.type = "Env";
    return incident;
}
```

4. This instance of an Incident work product can then be used in the following code to send to the Incident Management Service Proxy CreateIncident method to create an incident on the core.

```
// Create the incident on the core
incidentManagementService.CreateIncident(incident);
```

5. To update the Incident work product, use this line which as an example uses the Incident Description data element:

```
// Update the local representation of the incident
  incident.description = "The flood on the river has covered downtown";
```

6. That updated work product is then used to update the incident on the core in this line:

```
  // Post the update to the core
  incidentManagementService.UpdateIncident(incident);
```

7. The incident can be closed and archived with the following lines:

```
  // Close the incident
  incidentManagementService.CloseIncident(incident);

  // Archive the incident (remove it from the core)
  incidentManagementService.ArchiveIncident(incident);
```

8. The internal class AcceptAllCertificatePolicy at the top of the Program.cs file is used to allow the HTTP calls to accept any certificates.

9. These variables in the Program class must be set for the Incident Management Service proxy to work correctly:

```
private static String host = "https://localhost/uicds/core/ws/services/";
private static String workProductServiceURL = host + "WorkProductService";
private static String incidentManagementServiceURL = host + "IncidentManagementService";
```

10. In the Main function in Program.cs the username and password for a user on the core must be set with these two lines:

```
// Credentials to access the core
String username = "user1";
String password = "user1";
```

9.0 *Tutorials to Help You Get Started With UICDS Adapter Development*

All tutorials can be found on www.UICDS.us, Login - Technology Providers

Topic: Getting Started with UICDS Parts 1 and 2 - Oct 7, 2010 and Oct 21, 2010

Part: Getting Started With UICDS Training - October 7 2010 Conference Call – **Focus on Profiles, Notifications, and Incidents**

Contents:

- Overview of UICDS Interactions
- General Adapter Concepts
- How Applications Interact with UICDS
- Sequence of Creating Resource Profiles and Receiving Notifications
- Resource Profile Service
- Notification Service - Receiving Notifications
- Incident Service - Create Incident
- Incident Service - Update Incident
- Incident Service - Close and Archive Incident

Link: http://uicds.kzoplatform.com/swf/player/1103/chapter:486

Topic: Working with Example Code Tutorial and Biweekly Call - Nov 4, 2010

Part: **Java Example, C# Code, Axis2 Code**

Contents:

- Create incident
- Update incident
- Close incident
- Archive incident

Link: http://uicds.kzoplatform.com/swf/player/1103/chapter:486

Topic: Getting Started with UICDS Parts 1 and 2 - Oct 7, 2010 and Oct 21, 2010

Part: Getting Started with UICDS Part 2 - October 21, 2010 – **Focus on Resources**

Contents:

- Overview of interactions with the UICDS core
- EDXL-DE Message Header for RMS
- Sending Resource Requests using UICDS
- Sending Resource Commits using UICDS
- Receiving Notifications

Link: http://uicds.kzoplatform.com/swf/player/1102/chapter:480