Tutorial for the Java Context Awareness Framework (JCAF), version 1.5

Jakob E. Bardram
Centre for Pervasive Healthcare
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
bardram@daimi.au.dk

DRAFT – April 2005
Date: 2005/06/20 21:07:14 | RCSfile: jcaf.tutorial.v15.tex,v | Revision: 1.7

Contents

Chapter 1

Introduction

This tutorial contains basic information on how to run and program JCAF version 1.5. The theoretical thoughts behind JCAF is described in the PERVASIVE 2005 paper [3]. More technical details of the design of JCAF is described in the technical report on JCAF [1]. Please read these documents as a background for this tutorial. JCAF has been used to implement different context-aware applications and architectures [5, 2, 4].

JCAF version 1.5 is build on Java 1.4. JCAF relies on Java RMI for its distribution. Therefore, knowlegde and programming experience in Java RMI is needed before using JCAF.

Chapter 2

Installing and Running JCAF

JCAF is deployed in a zip file called `jcaf.v15.zip`. This archive contains the following files:

File	Description
java.policy	The Java policy file. Currently this file sets permission to `java.security.AllPermission`. But this file can be edited for more restrictive permission settings.
jcaf.v15.jar	The JCAF version 1.5 jar file with all the core JCAF classes.
jcaf.examples.jar	Examples for the JCAF version 1.5 distribution.
ContextService.bat	Executable bat file for MS Windows starting a context service.
ContextService	Executable SH file for Unix.
runRegistry.bat	Executable bat file for MS Windows starting the RMI Registry. This is optional. Under normal circumstance the ContextService will start the RMI Registry if not already running on this machine.
runRegistry	Executable SH file for Unix.

To deploy JCAF simply extract this zip file. A context service is started by using the `ContextService` command with the following parameters:

```
ContextService name [peer]
```

where,

- `name` – the name of this context service
- `peer` – the URI of a peer context service (optional)

The peer context service should be running when starting a context service. An example is:

```
ContextService aware_service rmi://snaps.daimi.au.dk/base_service
```

2.1 Deploying new Classes

Under normal circumstances, the JCAF ContextService should be able to dynamically load new classes on runtime without restarting the service. This is part of the standard dynamic classloading in Java using the `codebase` property.

For some unknown reason, however, I have not been able to make this work in connection with JCAF. Therefore, in order to deploy and load you own classes – like your own implementations of new kinds of entities and context items – you need to add these classes to the classpath of the Context Service. The most convinient way to do this is to make a jar file, place it in the `deploy` directory, and add this jar file to the classpath property in the startup script (ContextService or ContextService.bat) before restarting the context service. This is done by the `jcaf.examples.jar` file.

Chapter 3

Context Modelling

JCAF uses a pure Java-based object-oriented approach for modelling context information. The core modelling abstraction in JCAF is the interfaces and classes: `Entity`, `Context`, `Relationship`, and `ContextItem`. An UML diagram of the relationships between these classes and interface can be found in the JCAF Technical Report [1].

3.1 The `Entity`

The basic modelling concept in JCAF is the `Entity` interface. This interface defines an entity in the real world that we want to model in order to keep track of its context. Objects implementing this interface is hence the main object in the context service. An entity is a small Java program that runs within a context service on some host. Entities receive and respond to requests from clients, usually using Java RMI. Furthermore they are notified if their context change, because an `Entity` extends the `EntityListener` interface.

This interface defines methods to initialize an entity, to service requests, and to remove an entity from the context service. These are known as life-cycle methods and are called in the following sequence:

- The entity is constructed, then initialized with the `init()` method.
- When the entity's context is changed the `contextChanged()` method is called.
- The entity is taken out of service, then destroyed with the `destroy` method, then garbage collected and finalized.

In addition to the life-cycle methods, this interface provides the `getEntityConfig()` method, which the entity can use to get any startup information, and the `getEntityInfo()` method, which allows the entity to return basic information about itself, such as author, version, and copyright.

To implement this interface, you can write a generic entity that extends `dk.pervasive.jcaf.GenericEntity`. A simple example of the implementation of a `Person` entity is listed below:

```
public class Person extends GenericEntity {

    private String name, location, note;
    private byte[] publicKey;

    public Person() {
        super();
    }

    public Person(String id) {
        super(id);
    }

    public Person(String id, String name) {
        super(id);
        this.name = name;
    }

    public String getEntityInfo() {
        return "Person entity";
    }

    public String getName() {
        return name;
    }

    // other setters and getters
}
```

3.2 The `Context`

Once you have an entity object you can acces its context using the `getContext()` method. An entity has one `Context`, which contains a set of `ContextItem` objects indexed by `Relationship` objects as keys[1]. Please note that an `Entity` also implements the `ContextItem` interface. Hence, one entity can be in the context of another. For example, if a person A and a place P both are entities, then P can be in A's context.

Because context information is communicated using Java RMI, entities and their context need to be serializable. Hence, when you implement entities, relationships, and context items you must make sure that they are serializable. This also includes to make no-argument constructors to be used in the de-serialization. All of these classes also implements the `XMLSerializable` interface and you hence need to provide a `toXML()` method for each of them. Serialization to XML is very convinient in many situations and is incorporated for future use where a SOAP-like remote method invocation is planned.

[1] A `Context` is implemented using the `Hashtable` class in Java.

3.3 The `contextChanged()` method

Entities in a context service's `EntityContainer` is notified when changes to their context happens. The entity container calls the `contextChanged()` method on the entity. For example, the following code could be part of the `Person` entity above:

```
public class Person extends GenericEntity {
  ...
  public void contextChanged(ContextEvent event) {
    String new_location = null;
    if (event.getRelationship() instanceof Located) {
      if (event.getItem() instanceof Location) {
        new_location = ((Location) event.getItem()).getLocation();
      }
      if (event.getItem() instanceof Place) {
        new_location = ((Place) event.getItem()).getId();
      }
    }
    if (new_location != null) {
      if (event.getEventType() == ContextEvent.RELATIONSHIP_ADDED)
        this.setLocation(new_location);
      else
        this.setLocation("Unknown");
    }
  }
  ...
}
```

This method keeps an entity `location` variale up-to-date when events occur. The method checks if the relationship is of type `Located` in which case the new location should be extracted from the context information. There is two ways in which location can be changed – either by adding a `Location` context item to the context or to add a `Place` entity in the context. Both cases are handled in the example above.

Chapter 4

The Context Service

4.1 Contacting a Context Service

The `ContextService` interface is the main interface to a running context service. A context service is a RMI `Remote` interface and a stub can hence be accessed using the static `lookup()` method on the `java.rmi.Naming` class. For example, the following method looks up a context service in the utility class `AbstractContextClient` located in the `dk.pervasive.jcaf.util` package:

```
protected ContextService getContextService() {
  if (service == null) {
    try {
      System.out.println("Connecting to Context Service...");
      System.out.println("   uri : " + getURI());
      service = (ContextService) Naming.lookup(getURI());
    } catch (NotBoundException e) {

      ...

    }
  }

  return service;
}
```

The `getURI()` method return the URI for the context service to lookup. The fomat is:

```
rmi://<hostname>:<port>/<context_service_name>
```

4.2 Getting and Setting Context Information

To add an `Entity` to a context service use the `addContextEntity()` method. For example, the code

```
Person person = new Person("ole@acm.org", "Ole Hansen");
getContextService().addContextEntity(person);
```

adds a `Person` to the context service. This method asumes that an entity does not exist already, creates a new `Context` for this entity, and calls the `init()` method on the entity, thereby providing the entity with a `EntityConfig` object.

In contrast, the `setEntity()` method only changes the entity stored in the context service and not its context. This methods does no initiate a new life cycle. Hence the call

```
person.setName("Ole B. Hansen");
getContextService().setContextEntity(person);
```

merely updates the entity information and do not change the context information already in the context service.

Various methods for getting `Entity` objects exists. The `getEntity()` method gets an entity based on its id. The `getAllEntityIds()` method returns an array of all entity ids. The `getAllEntities()` method returns an array of all entities. This may be a rather large data set and this method should hence be used with causion. The `getAllEntitiesByType()` method returns an array of all entities of a specific type, e.g. `Person`.

When the `removeEntity()` method is called, the `destroy()` method on the `Entity` is called before it is removed from the context service.

Context information for entities are added and removed by using the `addContextItem()` and `removeContextItem` methods on the `ContextService` interface. For exampel, the following code updates location context information for a person:

```
Location office = new Location("hopper.333");
Located test = new Located("tutorial_code");
getContextService().addContextItem(person.getId(), test, office);
```

The variable `office` is a `Location`, which is a context item containing location information. The variable `test` is a `Located`, which is a relationship containing information on the source (the name `"tutorial_code"`) of this relationship[1].

Calling the `addContextItem()` and `removeContextItem` methods will cause the context service to notify the relevant entity using the `contextChanged()` method. If several entities are in each other's context the whole chain will be notified. In version 1.5 there is no detection of cycles in event chains – hence care should be taken no to create complex networks of entities in each others context in order to avoid potential cycles. Later version of JCAF will incorporate detection of cycles.

[1] Better examples of relationship modelling is providen in chapter 5 when discussing context clients.

Chapter 5

Context Clients

All programs accessing a context service is in principle a context client. However, two special types of clients exists, namely a context monitor and a context actuator.

5.1 The `AbstractContextClient` class

The `AbstractContextClient` class in the `dk.pervasive.jcaf.util` package provide a simple abstract class to be used when developing context clients. It main method is the `getContextService()` which helps lookup a context service based on a specified URI.

5.2 Context Monitors

Two types of context monitors can be created: asynchronous and synchronous monitors. Asynchronous monitors reports context items to a context service as the monitor sense it. Synchronous monitors provide context information when asked by the context service to acquire (sense) up-to-date information.

A simple asynchronous monitor would be a location monitor that reports location based on RFID tags. The implementation would look something like this:

```
public class RFIDMonitor extends AbstractMonitor
                         implements RFIDScannerListener {

  private Located rfid_located = null;

  public RFIDMonitor(String service_uri) {
    super(service_uri);
    rfid_located = new Located(this.getClass().getName());
  }

  ...

  public void tag(RFIDScanEvent event) {
```

```
    ...

    if (event.getMethod() == RFIDScanEvent.TAG_SCANNED) {
      rfid_located.resetTime();
      getContextService().addContextItem(
                      event.getId(),
                      rfid_located,
                      new Location(event.getScannerName()));
    }
    if (event.getMethod() == RFIDScanEvent.TAG_LEFT) {
      getContextService().removeContextItem(
                      event.getId(),
                      rfid_located);
    }
    ...
  }
}
```

The method `tag()` from the `RFIDScannerListener` interface is called by an RFID scanner – the details are omitted here. When an RFID tag is scanned, this tag is added as a `Location` context item to the entity with the same id as the tag. When the RFID tag has left the scanner, this relationship is removed again. Note the use of the `Located` relationship. It is initialized with some locator identification (the RFID scanner) and the time of location is reset everytime the tag is scanned.

A synchronous monitor would implement the `RemoteContextMonitor` interface, which consists of one method, `monitor()`. Hence, a monitor looking into an online calendar to check what a user is doing currently would look something like:

```
public class MeetingMakerMonitor extends AbstractMonitor {
  public synchronized void monitor(String entity_id)
              throws RemoteException {

    // Only using the id before the '@' in an email address
    final String cid = id.substring(0, id.indexOf('@'));
    System.out.println("Trying to get MM appointment for '" + cid + "'");
    MMThread thread = new MMThread(cid);
    thread.start();
  }
}
```

This example extends the `AbstractMonitor` utility class which has convinient methods for accessing a context service, etc. Note that the `monitor()` method returns nothing – it starts a new thread, which tries to access the online calendar, and then returns. Later the `MMThread` thread will report back the calendar information once this has been retreived from the MeetingMaker server.

A synchronous monitor can be registered to a context service using the `addContextClient()` method on the `ContextClientHandler` interface, which is implemented by the context services. The signature of this methods is:

```
public void addContextClient(int type,
                             Class relation_type,
                             RemoteContextClient client);
```

The `type` indicates if this is a monitor or an actuator. The `relation_type` described what kind of `Relationship` types this monitor or actuator can handle. For example, `Located` for location monitors or `Booked` for online calendar monitors. Hence, our MeetingMaker monitor above would register using the following lines of code:

```
getContextService().addContextClient(RemoteContextClient.TYPE_MONITOR,
                                     Booked.class,
                                     this);
```

This monitor is called when an entity's context is to be refreshed (which happens when somebody tries to get it from the service) and this context contains a relationship of the relationship type specified. Hence, in this example: if a person has a relationship of type `Booked` and this person's context is requested, then the MeetingMaker monitor would be called to refresh its information on this person. Please note, however, that the context information returned by using the `getContext()` method has not been refreshed by invoking monitors. This happens afterwards and the client hence needs to listen to the changes to the entity. Details are in the technical report [1].

5.3 Context Actuators

Context actuators are made by implementing the `RemoteContextActuator` interface. This interface has only one method:

```
public interface RemoteContextActuator extends RemoteContextClient {
  public void contextItemChanged(ContextEvent event);
}
```

Context actuators register at a context service using the same method as the context monitors described above. A context actuator is called when a context information type it has registred for is changed in the context service. For example, if a context actuator listen for location events by registrering interest in the `Located` relationship type, then this actuator would be notified everytime some entity's located relationship is changed. This can be used to maintain an active map, for example.

5.4 Entity Listeners

Entity listeners listen to changes in an entity. As already described entity listerners contain the `contextChanged()` method. Most (remote) context clients need to implement the `RemoteContextListener` interface. JCAF has a small utility class to wrap such remote context listeners into local ones. Hence an entitylistener can be made as:

```
public class ContextTester implements EntityListener {
  ...
  private RemoteEntityListenerImpl listener;

  public ContextTester() {
    super();
    try {
      listener = new RemoteEntityListenerImpl();
      listener.addEntityListener(this);
    } catch (RemoteException ignored) {}
  }

  private void test() {
    getContextService().addEntityListener(listener, Person.class);
    ...
  }

  public void contextChanged(ContextEvent event) {
    System.out.println(event.getEntity().toXML());
  }
}
```

The `RemoteEntityListenerImpl` wraps remote entity listeners The purpose of this small exercise is to allow you to implement the `EntityListener` interface instead of the `RemoteEntityListener` interface. In this way you do not need to RMI compile you class.

Chapter 6

Security

Version 1.5 of JCAF incorporates limited support for security. More specificly, it supports context monitors to authenticate themselves to a context service. Such authenticated monitors are called 'secure context monitors' and can provide 'secure' context information. The `ContextItem` interface has a `isSecure()` method which returns true if this item has been provide by a secure context monitor.

The following code gives an example:

```
...
PrivateKey key = // get this client's private key
byte[] data = this.getClass().getName().getBytes();
Signature sig = Signature.getInstance("DSA");
sig.initSign(key);
sig.update(data);
byte[] signature = sig.sign();

// Authenticate to the context service
secureCS = getContextService().authenticate(
                this.getClass().getName(),
                data,
                signature);
System.out.println("Got a secure connection to the server : " + secureCS);

// Adding a secure location context item to the person's context
secureCS.addContextItem(person,
                        located,
                        new Location("loc://daimi.au.dk/hopper.333"));
```

JCAF uses the standard `java.security` packages and the security mechanisms in JCAF can hence be extended. How the PKI infrastructure is managed – e.g. how public and private keys are generated and distributed – is not part of JCAF.

Chapter 7

Peer-to-Peer Context Services

JCAF enables context services to cooperate in a federated network of peers. This is typically used to divide responsibility amongst different context services. For example, in the AWARE architecture [4] a special-purpose context service is used to manage context information for users subscribed to its awareness service. A context service can have multiple peers.

7.1 Setting up P2P Context Services

As described in chapter 2 one peer context service can be specified on startup. On runtime, peer context services can be added and removed from a context service by using the `addContextService()` and `removeContextService()` methods on the `PeerHandler` inteface. Peer context services are indexed by their URI.

7.2 Looking up Entities in a P2P Network

Entities can be searched in a peer-to-peer network of context services by using the `lookupEntity()` method on the `PeerHandler` inteface. The signature for this method is:

```
public void lookupEntity(String id,
                         int hops,
                         RemoteDiscoveryListener dl);
```

The call is non-blocking. Hence a `RemoteDiscoveryListener` callback object must be specified. The method searches for the entity specified by '`id`' and hops across '`hops`' peers.

The `RemoteDiscoveryListenerImpl` is a convinient wrapper implementation for a `RemoteDiscoveryListener`. This is similar to the `RemoteEntityListenerImpl` wrapper discussed in section 5.4. The following piece of code illustrates how entities can be searched:

```
public class PeerTester implements DiscoveryListener {
  ...
  private RemoteDiscoveryListenerImpl dl;

  public PeerTester() {
    super();
    try {
      dl = new RemoteDiscoveryListenerImpl();
      dl.addDiscoveryListener(this);
    } catch (RemoteException e) {}
    ...
  }

  private void test() {
    ...
    getContextService().lookupEntity("ole@acm.org", 3, dl);
  }

  public void discovered(DiscoveryEvent event) {
    System.out.println("Entity found at " + event.getURI());
    System.out.println(event.getEntity().toXML());
  }

}
```

In this example, the entity with id '`ole@acm.org`' is searched in default context service and 3 hops away from it. If found, the methods `discovered()` is called back. If the entity is in more than one context service, this call returns the first which is found. The `DiscoveryEvent` contains the serialized entity object and an URI to the context service in which it was found.

Chapter 8

Examples

The zip file comes with some examples in the `jcaf.examples.jar` file. This file also contain an ANT build script (`build.xml`) which can be used to build the examples and can be a basis for creating a build script for other JCAF applications.

Bibliography

[1] Jakob E. Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. Technical Report CfPC Technical Report 2004–PB–61, Centre for Pervasive Computing, Aarhus, Denmark, 2003. Available from http://www.pervasive.dk/publications.

[2] Jakob E. Bardram. Applications of ContextAware Computing in Hospital Work – Examples and Design Principles. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1574–1579. ACM Press, 2004.

[3] Jakob E. Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In Hans Gellersen, Roy Want, and Albrecht Schmidt, editors, *Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive 2005)*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115, Munich, Germany, May 2005. Springer Verlag.

[4] Jakob E. Bardram and Thomas R. Hansen. The AWARE architecture: supporting context-mediated social awareness in mobile cooperation. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 192–201. ACM Press, 2004.

[5] Jakob E. Bardram, Rasmus E. Kjær, and Michael . Pedersen. Context-Aware User Authentication – Supporting Proximity-Based Login in Pervasive Computing. In Anind Dey, Joe McCarthy, and Albrecht Schmidt, editors, *Proceedings of UbiComp 2003*, volume 2864 of *Lecture Notes in Computer Science*, pages 107–123, Seattle, Washington, USA, October 2003. Springer Verlag.