The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications

Jakob E. Bardram

Centre for Pervasive Computing
Department of Computer Science, University of Aarhus
Aabogade 34, DK–8200 Aarhus N., Denmark
bardram@daimi.au.dk

Abstract. Context-awareness is a key concept in ubiquitous computing. But to avoid developing dedicated context-awareness sub-systems for specific application areas there is a need for more generic programming frameworks. Such frameworks can help the programmer to develop and deploy context-aware applications faster. This paper describes the *Java Context-Awareness Framework* – JCAF, which is a Java-based context-awareness infrastructure and programming API for creating context-aware computer applications. The paper presents the design principles behind JCAF, its runtime architecture, and its programming API. The paper presents some applications of using JCAF in three different applications and discusses lessons learned from using JCAF.

1 Introduction

The idea of *context-aware computing* was one of the early concepts introduced in some of the pioneering work on ubiquitous computing research [20, 19, 8] and has been subject to extensive research since. 'Context' refers to the physical and social situation in which computational devices are embedded. The goal of context-aware computing is to acquire and utilize information about this context of a device to provide services that are appropriate to the particular setting. For example, a cell phone will always vibrate and newer ring in a concert, if it somehow has knowledge about its current location and the activity going on (i.e. the concert) [16].

In this paper we present the *Java Context-Awareness Framework* – JCAF. The goal of JCAF is to provide a Java-based, lightweight framework with a expressive, compact and small set of interfaces. The purpose is to have a simple and robust framework, which programmers can extend to more specialized context-awareness support in the creation of context-aware applications. Hence, to a large degree the framework is intended for researchers, programmers, and students for experimental purposes. In line with [7] we believe that experimental prototyping of context-aware applications are important in order to understand the whole concept of 'context-awareness' and its applicability in ubiquitous computing 'beyond the desktop'. JCAF is primarily primarily developed for research and teaching purposes and several projects have already been undertaken using JCAF, as discussed in section 6. The contributions of this paper can be summarized as follows:

– It introduces JCAF, a service-oriented, distributed, event-based, secure infrastructure suitable for the deployment and development of a wide range of context-aware applications.
– It suggest a compact Java API for context-awareness, which can be implemented and extended in special-purpose context-awareness systems. This is in line with other Java API for e.g. database connectivity (JDBC), messaging services (JMS), etc.
– It presents and discusses three cases in which JCAF has been applied to develop non-trivial context-aware applications in research and student projects.

The paper starts by outlining the central design principles behind JCAF. Section 3 presents the JCAF Runtime Architecture and section 4 presents the JCAF Application Programmer Interface, which are the two core parts of the JCAF framework. Section 5 discuss the current implementation status of JCAF and the ongoing work based on the lessons learned so far. Section 6 presents how JCAF have been used and evaluated and presents three specific projects, discussing in detail how JCAF was used in these specific cases. Section 7 discuss related work and section 8 concludes the paper.

2 Design Principles for JCAF

The goal of JCAF is to create a general-purpose, robust, event-based, service-oriented infrastructure and a generic, expressive Java programming framework for the deployment and development of context-aware applications. Requirements for context-awareness systems and/or frameworks have been widely discussed and described (see e.g. [7, 11, 9, 10, 1, 5, 2]). JCAF incorporates many of these concerns and we shall here merely highlight the core design principles of JCAF.

Basically, JCAF is divided into two parts: a *Context-awareness Runtime Infrastructure* and a *Context-awareness Programming Framework* (or Application Programmer Interface (API)). The core design principles of the runtime infrastructure are:

– *Distributed and Cooperating Services* – A context service may be dedicated to a specific purpose, like handling context information in a private home. Most context management is specific for this home, but occasionally it might become relevant to contact services running in other homes. Therefore, a context-awareness infrastructure should be distributed and loosely coupled, while maintaining ways of cooperating in a peer-to-peer or hierarchical fashion.
– *Event-based Infrastructure* – The core quality of context-aware applications is their ability to react to changes in their environment. Hence, applications should be able to subscribe to relevant context events and be notified when such events occur.
– *Security and Privacy* – Context data, used e.g. in a medical setting, should be protected, subject to access control, and not revealed to unauthorized clients [4, 15]. Furthermore, establishing the credibility and origin of context information is key for some type of context-aware applications. Such cases may require an authentication mechanism for clients, and even a secure communication link between clients and services. However, in line with [14] we argue for supporting *adequate security* in an ubicomp environment. Hence, eaves-dropping sensor information like

temperature and location is seldom a major security issue – often is is easier to measure the temperature than listening in on low-power radio communication.
- *Extensible* – The infrastructure should be extensible in several ways, without the need for restarting it. First, it should be possible to deploy, modify, and remove context services. Second, the infrastructure should support evolvement of supported types of context by dynamically load context definitions, functionality, and acquisition mechanisms, like new context sensors.

The goal of the JCAF API is to make it easy to design and develop context-aware application for specific purposes and usage settings. This leads to the following key design principles for the programming API:

- *Semantic-free modeling abstractions* – The type of context information that is relevant to model and handle varies across application settings. For example, in a hospital, items like beds, pill-container, and medicine are important context information for the work of clinicians, but this is specific to hospitals. Hence the application programmer should be able to model and handle context data specific for various settings.
- *Context Quality* – Applications are concerned with the quality of context information, including uncertainty [10]. The clinical application trying to find relevant patient data during an operation might suggest to show more than one piece of medical data, if the uncertainty is too high. Quality measures for context information must hence be preserved from its measurement, through any transformation, and to its use by applications.
- *Support for Activities* – The reason for capturing location and other context information is typically not for direct use in applications but to enable the reasoning at the level of *user activities* [10, 6]. For example, we want an EPR to show the correct medicine schema when the nurse is giving medicine to the patient. The framework must hence provide handles for writing application-specific code, which can 'translate' changes in the context into suggestions for user activities.

3 The JCAF Runtime Architecture

The JCAF Runtime Infrastructure is illustrated in figure 1. It consists of a range of *Context Services* which are connected in a Peer-to-Peer setup, each responsible for handling context in a specific environment. For example, a context service might run in an operating room, handling specific context information in this setting, like who is there, what are they doing, who is the patient, and what is the status of the operation. A network of services can cooperate by querying each other for context information.

Each *Context Service* is a long-lived process analog to a J2EE Application Server[1]. An *Entity* with its *Context* information is managed by the service's *Entity Container*. An entity is a small Java program that runs within the Context Service and responds

[1] The JCAF framework uses the J2EE specification as an architectural 'pattern' and inspiration. The JCAF runtime is analog to an Application Server running J2EE applications, and the JCAF API is analog to the J2EE API for creating e.g. servlets.

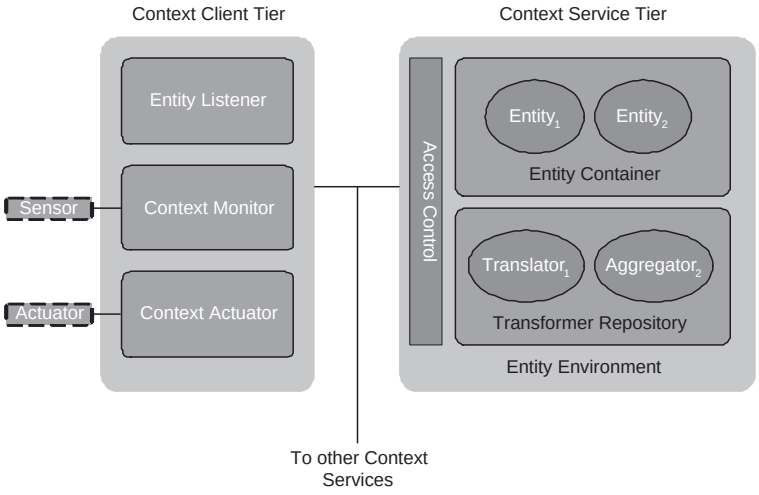

Fig. 1. The Runtime Architecture of the JCAF Framework

to changes in its context. The life cycle of an entity is controlled by the container in which the entity has been added. The entity container handles subscribers to context events and notifies relevant clients on changes to entities. An entity, its context and its life cycle is further discussed in section 4.

The Entity components in a Context Service work together and with other components to accomplish their tasks. Hence they must have ways to access each other and to access shared resources, like database connections or RMI stubs to other processes. This is accomplished through the *Entity Environment*, which all Entities has a handle to when executing [2]. Besides access to general resources like initialization parameters and logging facilities, the Entity Environment provides methods for accessing *Key-Value Attributes* and *Context Transformers*. Context transformers are small application-specific Java programs that a developer can write and add to the *Transformer Repository*. The Transformer Repository can be queried for appropriate transformers on runtime (more on transformers in section 4.5).

Access to a Context Service is controlled through the *Access Control* component, which ensures correct authentication of client requests. This component consists basically of two parts, namely an access control list, specifying what the requesting clients can access, and mechanisms for authenticating the client.

Context Clients can access entities and their context information in two ways. Either following a request-response schema, requesting entities and their context data, or by subscribing as an *Entity Listener*, listening for changes to specific entities. JCAF also support *type-based* subscriptions of entity listeners, allowing a client to subscribe to changes to all entities of a specific type, e.g. patients. There are two special kinds of context clients: the *Context Monitor* and the *Context Actuator*. A monitor is a client

[2] The Entity Environment is analog to the Web Context in a J2EE Application Server, where handles to databases, shared objects, and other resources are maintained across servlets.

specially designed for acquiring context information in the environment by cooperating with some kind of sensor equipment, and associate it properly with an Entity. A context actuator is a client designed to work together with one or more actuators to 'change' the context. In JCAF the Monitor and Actuator interfaces are generic and can be used to create a wide range of monitors and actuators, which monitor and affect the physical and digital context. Monitors (sensors) and actuators might not necessarily be hardware components.

4 The JCAF Application Programmer Interface

The JCAF API enables the programmer to create context-aware applications that are deployable in the JCAF infrastructure. An UML diagram for the core interfaces and classes in the JCAF API is illustrated in figure 2. These are the `ContextService`, managing `Entity` objects, `EntityListener`s, and `ContextClient`s. Two examples of context clients currently made in JCAF are the `ContextMonitor`s and `ContextActuator`s. Each context service has an `EntityEnvironment`, where entities can access and store application-specific attributes using a key-value data structure. Let us consider some of these core JCAF interfaces and classes in details.

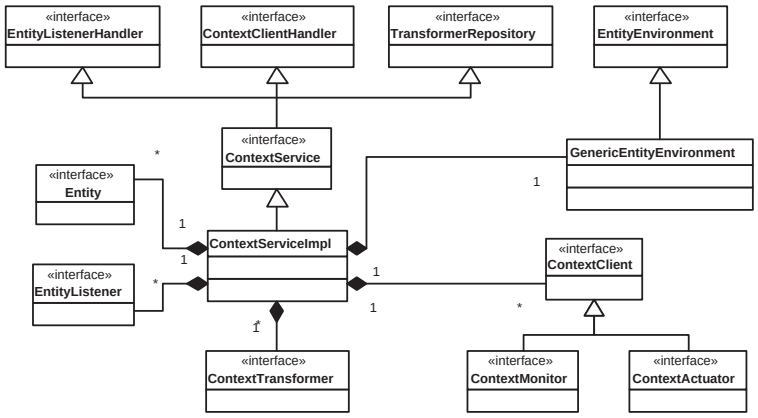

Fig. 2. The overall UML diagram for the JCAF Framework showing the important interfaces and classes available in the programmer's API. The UML for `Entity` is further specified in figure 3.

4.1 Context Service and Entity Environment

A Context Service enables clients to access entities and to set, get, and subscribe to changes in context information for entities. The `ContextService` interface is shown below.

```
package dk.pervasive.jcaf;

public interface ContextService
       extends TransformerRepository,
               ContextClientHandler,
               EntityListenerHandler {

  public String getServerInfo() throws RemoteException;
  public Entity getEntity(String id) throws RemoteException;
  public void lookupEntity(String id, int hops, RemoteEntityListener l) throws RemoteException;
  public String[] getAllEntityIds() throws RemoteException;
  public Entity[] getAllEntities() throws RemoteException;
  public Entity[] getAllEntitiesByType(Class type) throws RemoteException;
  public void addEntity(Entity entity) throws RemoteException;
  public void setEntity(Entity entity) throws RemoteException;
  public void removeEntity(Entity entity) throws RemoteException;
  public void removeEntity(String entity_id) throws RemoteException;
  public Context getContext(String entity_id) throws RemoteException;
  public void setContextItem(String entity_id, ContextItem item) throws RemoteException;

}
```

There are methods for adding, removing, getting and setting entities. The `getEntity()` method returns this Context Service's copy of the Entity object, whereas the `lookupEntity()` method contacts other known Context Services trying to locate the Entity object. The `lookupEntity()` method takes as argument the id of the Entity to look for, the number of steps away from this context service in order to find the entity, and an EntityListener which is called when the entity is found. The method is non-blocking and relies on notifying the entity listener if a matching entity is found. The Context Service inherits from three interfaces. The `TransformerRepository` contains methods for adding and getting transformers (see section 4.5). The `ContextClientHandler` interface contains methods for adding and authenticating a context client, which might be a `ContextMonitor` or a `ContextActuator` (see section 4.4). The `EntityListenerHandler` interface contains methods for adding, removing, and accessing entity listeners(see section 4.3).

The `EntityEnvironment` is shared by all entities in a Context Service. The Entity Environment have methods for setting and getting attributes, accessing information about the Context Service, and accessing the `TransformerRepository`, which holds all the `ContextTransformer`s (see section 4.5).

4.2 Entity and Context

The basic modeling concepts in the JCAF API are the `Entity`, which has a `Context` with a set of `ContextItem`s. These are illustrated in figure 3. An entity, a context, and a context item are all Java interfaces, which developers of context-aware applications must implement[3]. Examples of entities are persons, places, things, patients, beds, pill containers, etc. Examples of context are a Hospital Context and a Office Context, each knowing specific aspects about a hospital and an office, respectively. Examples of context items are physical location, activity as revealed by a user's calendar, and the patient in a hospital bed. Context items are added to an entity's context typically by context monitors or other clients. Hence, the context item `Location`, which models a physical location, can be added to an entity, thereby registering the location of this

[3] JCAF provides default implementations of these core interfaces. For example the `GenericEntity` class implements the `Entity` interface and can be used to create concrete entities using specialization.

entity. The `ContextItem` interface is shown below. It is important to be able to judge the quality of a context item [10]. For example, how accurate is the location estimate. The `getAccuracy()` method is used for this purpose. Implementations of a context items returns a probability between zero and one. The `isSecure()` method is used to establish whether this context information orginates from a trusted and authenticated context monitor.

```
public interface ContextItem extends Serializable {
  public long getSequenceID();
  public boolean isSecure();
  public double getAccuracy();
  public boolean equals(ContextItem anotherItem);
}
```

A subtle, but rather important aspect of entities is that they themselves are context items. Hence, if I have a patient (an entity) I can add a pill container (also an entity and hence a context item) into the context of this patient, thereby indicating that this pill container is used for this patient. The modeling mechanisms in the framework are semantic-free – in this example with the pill container having a patient in its context might just as well be interpreted differently. For example, having the patient in the container's context might mean that the patient is located close to it[4].

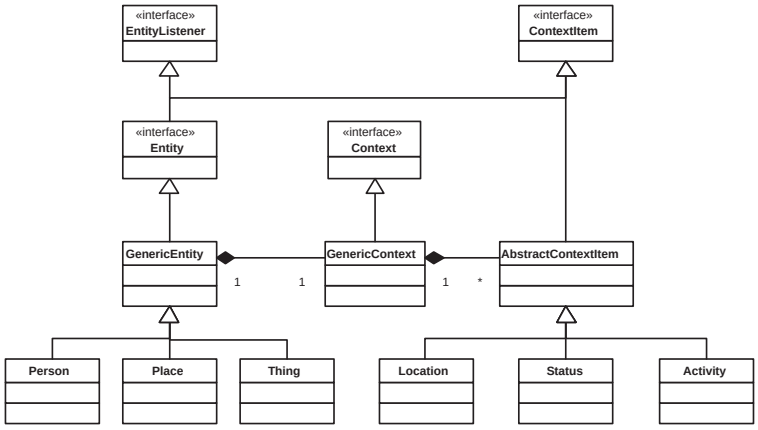

Fig. 3. The UML model of an `Entity` with a `Context` containing a range of `ContextItem`s. Note that the an Entity also is a Context Item.

4.3 EntityListeners and ContextEvent

Central to JCAF is its event-based infrastructure, which propagates events about changes in Entities' Context to interested clients. `EntityListener`s are handled by

[4] Even though this interpretation is quite valid our experience of building systems with JCAF has taught us not to model location in this way. Hence the JCAF API contains pre-made classes for modeling different kinds of location, like GPS or in-door location in an office environment.

the `ContextService`, which inherits from the `EntityListenerHandler` interface. This interface contains methods for adding and removing `EntityListener`s. Entity Listeners can be added on specific entities as well as type-based by specifying a entity class type. For example, an entity listener can listen to all person entities. Clients interested in listening to context changes can implement the `EntityListener` interface shown below.

```
public interface EntityListener {
   public void contextChanged(ContextEvent event);
}
```

Entities themselves are aware of changes to their context by implementing this interface. The central processing part of an Entity is hence its `contextChanged()` method. This method is guaranteed to be called by the entity container whenever this entity's context is changed. This is a very powerful way to implement functionality handling changes in the entity's context and thereby create logic, which translates such changes into meaningful activities for users of the application. The `ContextEvent` object is a standard `java.util.EventObject` that gives access to the Entity and the Context Item, which caused the change. A `RemoteEntityListener` interface exists as well, enabling clients to listen on changes to Entities in a remote ContextService process.

4.4 Context Clients – Monitors and Actuators

The JCAF framework can handle the acqucition and transformation of context information in two ways – synchronously and asynchronously. A context monitor can continuously supply context information (i.e. items) to an entity by using the `setContextItem()` method on the Context Service interface. For example, a location monitor can update the location of an entity when it sees it. A client requesting the context information for an entity will received the latest location information. This is called *asynchronous context management*, because the client and the monitors (in general all clients) work independent of each other. The asynchronous mode is the prevalent mode in the JCAF framework. However, some context-aware applications might want to have up-to-date context information. Therefore, the JCAF framework also supports the *synchronous mode*, where a client requests the context for an entity, and the entity asks its context to refresh itself. A user's current activity according to his calendar is an example where the activity monitor asks the calendar about the activity at the time of calling. The `ContextMonitor` interface is:

```
public interface ContextMonitor extends Remote {
   public ContextItem getContextItem(String id) throws RemoteException;
}
```

A monitor can register itself at a `ContextMonitorHandler` by using the `addContextMonitor()` method. When clients ask for context information, by using the `getContext()` method, then relevant registered ContextMonitors are called to acquire context information by calling their `getContextItem()` method. To avoid deadlocks (e.g. if the calendar system does not answer), the `getContext()` method starts a separate thread to handle monitors and returns immediately with whatever context information is available currently. When the Context Monitors starts

reporting back (which might take some time), then clients are notified using the `contextChanged()` method in the `EntityListener` interface. This is illustrated in the interaction diagram shown in figure 4.

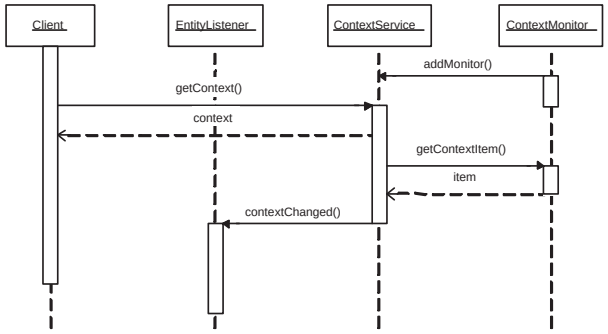

Fig. 4. Interaction Diagram for asynchronous context acqucition using Context Monitors registered at the Context Service.

Context Actuators are used to set context information. The interface for the `ContextActuator` is shown below:

```
public interface ContextActuator {
  public void contextItemChanged(ContextEvent event);
}
```

Context Actuators can register at a `ContextActuatorHandler` (i.e. at a context service) by specifying what type of Context Items it is an actuator for. When a Context Item is changed in the context service (i.e. the `contextChanged()` method is triggered), all Context Actuator registered as interested in this type of Context Items are notified using the `contextItemChanged()` method on the actuator. This can be used to keep context information synchronized in several place.

4.5 Context Transformers

The API for the `ContextTransformer` interface is shown below:

```
public interface ContextTransformer {
  public Class[] getInType();
  public Class getOutType();
  public ContextItem translate(ContextItem[] in) throws ContextTransformerException;
}
```

The `getInType()` method tells which types of `ContextItem`s that this transformer can take as input and the `getOutType()` tells what type of Context Item it returns. The `translate()` method takes an array of context items as input and returns a translated context item. If there is only one element in the array of input types, then a transformer works as a *translator* by translating from one type of Context Item into another. If there is more than one element, the transformer works as an *aggregator*, by aggregating several Context Items into one Context Item.

Context Transformers are located in the `TransformerRepository` shown below:

```
public interface TransformerRepository {
   public void addContextTransformer(ContextTransformer t);
   public void removeContextTransformer(ContextTransformer t);
   public ContextTransformer getContextTransformer(Class type);
   public ContextTransformer getContextTransformer(Class[] in, Class out);
}
```

A Transformer Repository holds a range of transformers. A client can use the `getContextTransformer()` method to get a transformer, which can transform an array of Context Items into another Context Item. Transformers can be added to the Transformer Repository on runtime and it can be queried for appropriate transformers. These transformers can subsequently be put together in a pipeline of transformers to obtain the desired transformation.

5 Implementation and Future Work

JCAF is currently in a version 1.5 and is implemented using J2SE 1.4. The core functionality of JCAF as described above is implemented and working. Remote communication is currently implemented using Java RMI. A Context Service is looked up using the Java RMI Registry and accessed using RMI invocation. The lookup of entities in associated context services (using the `lookupEntity()` method) is also done using RMI. A configuration file contains information about known peers. Hence, there is no automatic discovery of other context services. Nor is there a 'super-peer' who knows of all running instances of context services.

As argued in the introduction, JCAF is designed to support different implementations. Currently, we have engaged in two projects that are implementing the distribution mechanisms in JCAF differently. One project is to create a Simple Context Protocol (SCP), which is a text-based protocol running over TCP/IP sockets. This protocol is to be used to access context services from non-java programs and provide a more decoupled protocol than RMI. A side-result from this project might be a SOAP interface, although we find SOAP too resource consuming for some of the applications we want to run on e.g. mobile phones and embedded devices. Another project is looking into creating a robust peer-to-peer infrastructure for JCAF, where distributed context services can discover each other and cooperate directly.

Security is implemented using an authentication mechanism based on a digital signature using the Java Security API. This is currently used for Context Clients (i.e. Context Monitors and Actuators) and the authentication mechanisms is part of the `ContextClientHandler` interface. Context information from authenticated monitors are labeled 'secure'. This security mechanism could be extended to include other types of Context Clients, like Entity Listeners and Transformers added to the JCAF while running. Finally, security might be enhanced used encrypted communication between a context service and some clients, especially if sensitive (medical) data is transmitted. However, as discussed in section 2 we are very cautious about providing 'adequate security' and we are not sure (yet) if these latter security mechanisms are necessary. We plan to implement them, if we come across a case which requires them. As for access control, a simple role-based access control mechanisms is used currently:

monitors can add context items (secure monitors can add secure items), and clients can query context information. This access control mechanism could be extended to real access control lists, which have a fine-grained specification of the rights of each client.

The modeling mechanisms for context as presented above have been implemented and applied in different projects, as discussed in the next section. Currently, however, we are investigating how to enhance the modeling capabilities of JCAF, especially by adding methods for modeling 'associations' between context information [9]. However, caution must be applied in not creating too generic modeling abstractions which makes the framework hard to use.

In the current version of JCAF we have implemented a range of monitors for monitoring location based on RFID, WLAN, Bluetooth, and IrDA. Furthermore, monitors for monitoring activity in an online calendar and status information in an Instance Messaging system have been implemented. JCAF also contains several implementations of common entities (person, place, thing) and context items (location, status, activity, network capacity) as well as generic implementations of context clients and monitors.

6 Application of JCAF

JCAF have been used in different research and educational projects at our university. Table 1 contains an overview of these projects. In this section we will discuss how JCAF was used in three of these projects: (i) Proximity-Based User Authentication, (ii) the Context-Aware Hospital Bed, and (iii) the AWARE Framework. Each of these projects highlights different parts of the JCAF framework[5].

6.1 Proximity-Based User Authentication

Proximity-based user authentication [3] is a mechanism that allows users to log in to a computer just by approaching it and start using it. The system consists of two independent mechanisms. The first mechanism is a personal token with enough processing power to do public key cryptography. This token can be some jewelry (e.g. a ring, necklace, or earring), or it can be a personal pen used on the various touch screen embedded in a hospital. Currently we are using Java Smartcard technology as the personal token. When the user approaches a computer, this token can authenticate the user using public key cryptography. This is however not secure enough for use in hospitals – this token might be lost or stolen. Hence, when using e.g. smartcards in hospitals today, users are also required to enter a password or a PIN code. To avoid this, the second mechanism in our setup is to track the user's location via the context-awareness infrastructure. If the infrastructure can verify the location of the user in the same place as the token (and hence the computer) s/he is authorized. The location of the user can apply various methods based on e.g. something the user wear or trying to recognize the voice. Currently we monitor RFID tags woven into the clinicians whitecoats (see [3] for details).

[5] The design and evaluation of this technology have been done in cooperation with a range of clinicians, applying user-centered design methods like observations, design workshops, and prototyping.

Table 1. The use of JCAF in different projects, ranging from research projects (R) to students projects (S) in class.

Project Title	Type	Description
Proximity-Based User Authentication	R	Enables a user to log in to a computer by physically approaching it.
Context-Aware Hospital Bed	R	A hospital bed that adjust itself and react according to entities in its physical environment, like patient, medicine, and medical equipment.
Bang & Olufsen AV Home	S	Using context-awareness to make B&O AV appliances adjust themselves according to the location of people and things.
AWARE Framework	R	A system that distributes context information about users, thereby facilitating a social, peripheral awareness, which helps users coordinate their cooperation.
Wearable Computers for Emergency Personnel	S	A wearable system for emergency workers, like ambulance personnel. Helps them react to changes in the work context.

In this application of the JCAF framework, two aspects becomes important. The first one is the security of the framework. If the context-awareness framework is used to verify the location of the users, it is of crucial importance that any adversary trying to gain illegal access cannot send a false "I'm here" message to the systems. Hence, we need to trust and hence authenticate the Context Monitors reporting on the location of users. A context monitor is authenticated to a Context Service by using the `authenticate()` method providing an id, some data, and a signature on this data. This signature is verified by the authenticate method using the monitor's public key, which has been added to the context service's key ring at an earlier point. An example of a secure context monitor is shown below:

```
public class SecureLocationMonitor extends AbstractContextClient {

  SecureContextService secureCS;

  public SecureLocationMonitor() {
    super();

    try {
      PrivateKey key = ... // holds 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();

      // tries to authenticate at the server.
      secureCS = getContextServer().authenticate(this.getClass().getName(), data, signature);

      // If successful, then a secure context service is returned.
      if (secureCS != null) {
        System.out.println("Got a secure connection to the server : " + secureCS);

        // Now use this secure service to provide some location information
        secureCS.setContextItem("1732745-3872", new Location("loc://daimi.au.dk/hopper.333"));
      }
    } catch (Exception e) {...}
  }
}
```

The only way to access a secure context service is through the authenticate method. When the `setContextItem()` method on the secure service is used, the context item is marked as secure. Hence, a client using this context information can ask if this item is secure by using the `isSecure()` method on the `ContextItem` interface.

The second aspects concerns the quality of the context data. It is of equal importance that the user authentication mechanism can judge the quality of the location data and decide whether the quality is sufficient to trust as a verification of the location of the user. Hence, the aggregation of quality (or uncertainty) measures is important in this application of context-aware computing, and thus relies on the `getAccuracy()` methods of a Context Item. In our current implementation of the `Location` context item, accuracy decreases by 1% pr. minute since the last measurement. In the User Authentication protocol there is a threshold, which determines how accurate the location estimation needs to be to verify the user.

6.2 The Context-Aware Hospital Bed

Another example of a context-aware medical application is the *Interactive Hospital Bed* [2]. The bed has an integrated computer and a touch sensitive display, and is equipped with various sensors that can identify the patient lying in the bed, the clinician standing beside the bed, and various medical equipment with RFID tags. In this way the computer can adapt the computer screen to the users in its vicinity. For example, when the nurse arrives with the patient's medicine, the bed is able to log in the nurse (using 'Proximity-Based Login'), check if the nurse is carrying the right medicine for this patient, and it can display the relevant information on the screen, typically the medicine schema from the EPR system.

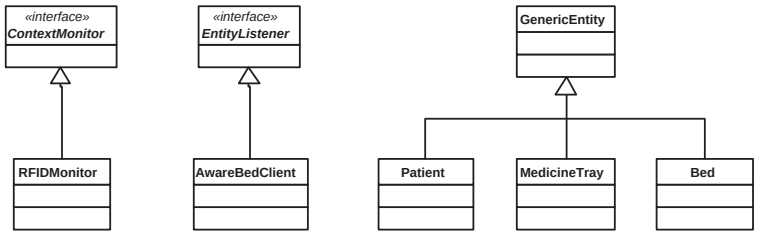

Fig. 5. UML class diagram for the Context-Aware Hospital Bed as implemented by extending the JCAF framework.

Figure 5 shows how this bed application is created by extending the JCAF framework. The `AwareBedClient` is an Entity Listener running on the bed and is listening to changes to the Entity `Bed`, which models the bed in JCAF. In addition, JCAF models `Patient` and `MedicineTray` entities. A medicine tray knows which patient it belongs to. When an `RFIDMonitor` registers an RFID tag nearby the bed, it notifies the context service about a new entity in the bed's context. This triggers a `contextChanged()` event in the `AwareBedClient`, which is an entity listener.

In the `contextChanged()` method we can implement what should happen when entities come close to the bed. An example is shown below:

```
public class AwareBedClient implements EntityListener {

  Patient myPatient = null;
  ...

  public void contextChanged(ContextEvent event) {
    if (event.getType().equals(Patient.class)) {
      System.out.println("Patient " + event.getItem() + " has been put into this bed (" + bed.getId() + ")");
      myPatient = (Patient) event.getItem(); // Keep a reference to 'my' patient
    }

    if (event.getType().equals(MedicineTray.class)) {
      System.out.println("MedicineTray " + event.getItem() + " is at this bed (" + bed.getId() + ")");
      MedicineTray tray = (MedicineTray) event.getItem();
      if (tray.getPatientId().equalsIgnoreCase(myPatient.getId())) {
        /*
         * The medicine tray is a the RIGHT patient.
         */
      } else {
        /*
         * The medicine tray is a the WRONG patient.
         */
      }
    }
  }
}
```

In this example, both the patient and the medicine tray is equipped with RFID tags. Hence, when a patient comes close to the bed, we put him in this bed. When a medicine tray comes close to the bed, we check if this is the right tray for the patient in this bed. For simplicity reasons, the listing above does not contain the details of what happens when a medicine tray comes close to the bed. In the context-aware bed application these lines contains code that access the EPR system running on the bed-client computer and shows relevant medical information in the three different cases.

The Context-Aware Hospital Bed also contains an example of a simple Context Transformer, which can resolve an entity's id from an RFID tag id. For example, resolving a patient or a medicine tray based on an RFID id. When the RFIDMonitor scans an RFID tag it calls the `setContextItem()` method on the Context Service with the id of the bed and a new `RFIDItem` context item as arguments. A `Place` entity (the superclass for `Bed`) can translate an RFID item into its coresponding Entity by using an 'RFID to Entity ID transformer'. This transformation is triggered whenever an RFID item is added to a place's context, i.e. in the place's `contextChanged()` method, as shown below. Note that this method is called by the entity container managing the runtime part of an entity and hence have access to resources in the service holding this entity.

```
public class Place extends LocatableEntity {

  protected ContextTransformer transformer = null;
  ...

  public void contextHasChanged(ContextEvent event) {
    if (transformer == null) {
      transformer = (RFIDToEntityIDTransformer) getEntityEnvironment().
                                                  getTransformerRepository().
                                                  getContextTransformer(RFIDToEntityIDTransformer.class);
    }

    if (event.getType().equals(RFIDItem.class)) {
      ContextItem[] in = new ContextItem[1];
      in[0] = event.getItem();
      try {
        EntityIDItem id = (EntityIDItem) transformer.translate(in);
        Entity entity = getContextService().getEntity(id.getEntityId());
        getContextService().setContextItem(this.getId(), entity);
      } catch (ContextTransformerException e) {...}
    }
```

```
  }
}
```

First we get an instance of a `RFIDToEntityIDTransformer`. Then we check if this is a new `RFIDItem` added to the context of this place. Then we translate from the RFID to the Entity ID using the transformer, look up the entity based on the resolved id, and adds this entity (e.g. a patient) to the context of this place. Note that the `AwareBedClient` listed above also receives notifications about RFID items being added to the bed but does not react on it.

6.3 The AWARE Framework for Social Awareness

When people need to engage in a cooperative effort there is a risk of interrupting each other. For example, when calling people using a mobile phone or accessing them directly in their offices. People hence often tries to maintain a 'social awareness' of each other in order to align their cooperation to the work context of their colleagues. This social awareness relies on having access to the work context and when people are not co-located this access can be mediated using networked computers (including very small portable ones). The main purpose of the AWARE platform is to provide such social awareness by notifying and informing users about the working context of their fellow colleagues. For this purpose JCAF is used to monitor the context of people.

In the AWARE platform a context service is running, which monitors the surroundings in a workplace, like an office or a hospital. In our current implementation these monitors include location monitors using Bluetooth beacons, status monitors, and calendar monitors. The bluetooth location monitor uses the discovery protocol in the bluetooth stack to look for bluetooth location beacons. These beacons are named according to their location using a location URL (see also [13]). For example, an office in the computer science department at our university would be `loc://daimi.au.dk/hopper.333`. This simple solution was chosen in order not to have the monitor and the beacon to connect using bluetooth, which takes some time. The calendar monitor is an example of an asynchronous monitor. The calendar monitor is able to extract the content of a user's online calendar at the time of request. This, however, might take some time. Hence, when a client asks for context information about a user, then the calendar monitor is started in a separated thread and the context service returns with the current available context information about this user. When the calendar monitor has finished talking to the online calendar, this context information is added to the user's context, and entity listeners are notified.

We have build two types of AWARE clients – an simple browser interface where a users can see context information about his fellow colleagues, and the AWAREPhone, which is a mobile phone clients. The AWAREPhone implements a location monitor, using its in-build bluetooth capabilities. Via the list of contact persons, a user can see the working context of a colleague and based on this information choose an appropriate cooperation strategy, like calling, sending a message, or not to disturb.

7 Related Work

Numerous related work within context-awareness exists. We shall hence concentrate on work specifically related to the core design principles in JCAF as described in section 2.

Creating support for context-awareness by having a server or infrastructure component is common in many context-awareness systems, like Schilit's mobile application customization system [18], the Contextual Information Service (CIS) [17], the Trivial Context System (TCoS) [11], and the Secure Context Service (SCS) [4, 15]. All of these act as the middleware that acquires raw contextual information from sensors and provides interpreted context to applications via a standard API. They can also monitor the context changes and send events to interested applications. All of these architectures, however, work in a strict client-server fashion and provides no support for distributed and loosely coupled context services and clients. Even though the Context Toolkit [7] can be viewed as a loosely coupled infrastructure, the intention of it was to provide a toolkit similarly to a GUI toolkit, which is used in the development of an application and 'linked' into the application. The intention of the JCAF framework is to have a context-awareness infrastructure deploying in an organization (e.g. a hospital) and applications can discover and utilize this infrastructure when needed. The Rome system developed at Stanford [12] is based on the concept of a context trigger, much like context events in JCAF. However, Rome's decentralized evaluation of triggers embedded in end devices does not allow context sharing and requires the end device to have the capability to sense and process all of the necessary raw contextual information. In JCAF entities residing in a context service are notified on context events and can access each other locally and look up remotely located entities, without involving any clients. Hence, the JCAF event structure is not only used for client notification but also for triggering actions in the entities residing in the context services' entity container.

Despite the importance of security and privacy in ubiquitous computing [14] little work have been done here, with the Secure Context Service (SCS) [4, 15] as a notable exception. However, SCS is based on a Role-Based Access Control (RBAC) mechanism and is a closed system where the identity of all clients must be known to the system a priori. JCAF, on the other hand, supports a more relaxed security strategy where unknown context clients can access and provide context information, but this context information is labelled insecure. This strategy in more aligned with the basic purpose of JCAF, i.e. to provide the basic building blocks for experimenting with context-awareness.

Even though Java has been used as a programming language in many context-awareness systems, there is to our knowledge no Java Framework or API available for context-awareness. As a toolkit to be programmed in Java, the Context Toolkit [7] is what come closest. However, in the Context Toolkit, Java's basic abstractions for TCP/IP networking and hashtables of string-based context information is used. There is no object-oriented modeling of context information, nor any use of Java serialization of complex context data or the use of Java RMI. JCAF is an attempt to suggest a Java API for context-awareness, analogue to the Java APIs for e.g. database access (JDBC), service discovery (JINI), and messaging (JMS).

8 Conclusion

This paper have presented the Java Context-Awareness Framework (JCAF), including its core design principles, its runtime infrastructure, and its programming API. The application of JCAF in three different cases within ubiquitous computing support in a hospital setting was presented and lessons from using JCAF in these cases was discussed. When looking at related work, JCAF shares similarities with much of the research already done within creating generic support for the creation of context-aware applications. Distinct features of JCAF are, however, its support for distributed cooperating context services, its event-based middleware architecture, its support for a relaxed security model for authenticating context clients, and its support for semantic-free modeling of context information in Java.

JCAF should be viewed as a proposed Java API specification of support for context-awareness much in line with other Java API specifications, like the Java Database Connectivity (JDBC), the Java Service Discovery (JINI), and the Java Messaging Service (JMS) API. Hence, JCAF do not claim to be radically new in its support for creating context-aware applications. It merely provides a comprehensive set of Java APIs and generic implementations which allows researchers, students, and programmers to start extending the framework and begin experimenting with context-awareness as a concept and as a technology.

Acknowledgments

The Danish Center of Information Technology (CIT) and ISIS Katrinebjerg funded this research. Henrik Bærbak Christensen was much involved in the early discussion on contet-awareness in hospitals.

References

1. G. D. Abowd. Software engineering issues for ubiquitous computing. In *Proceedings of the 21st international conference on Software engineering*, pages 75–84. IEEE Computer Society Press, 1999.
2. J. E. Bardram. Applications of ContextAware Computing in Hospital Work – Examples and Design Principles. In *ACM Symposium on Applied Computing (ACM SAC2004)*, pages xx–xx. ACM Press, 2004.
3. J. E. Bardram, R. E. Kjær, and M. . Pedersen. Context-Aware User Authentication – Supporting Proximity-Based Login in Pervasive Computing. In A. Dey, J. McCarthy, and A. Schmidt, editors, *Proceedings of Ubicomp 2003: Ubiquitous Computing*, volume 2864 of *Lecture Notes in Computer Science*, pages 107–123, Seattle, Washington, USA, Oct. 2003. Springer Verlag.
4. C. Bisdikian, J. Christensen, J. Davis, II, M. R. Ebling, G. Hunt, W. Jerome, H. Lei, S. Maes, and D. Sow. Enabling location-based applications. In *Proceedings of the 1st international workshop on Mobile commerce*, pages 38–42. ACM Press, 2001.
5. L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):921–945, Oct. 2003.

6. H. B. Christensen. Using Logic Programming to Detect Activities in Pervasive Healthcare. In *International Conference on Logic Programming, ICLP 2002*, Copenhagen, Denmark, Aug. 2002. Springer Verlag.
7. A. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16:97–166, 2001.
8. A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. *Wireless Networks*, 8(2/3):187–197, 2002.
9. K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling context information in pervasive computing systems. In M. Naghshineh and F. Mattern, editors, *Proceedings of Pervasive 2002: Pervasive Computing : First International Conference*, volume 2414 of *Lecture Notes in Computer Science*, pages 167–180, Zürich, Switzerland, Aug. 2002. Springer Verlag.
10. J. Hightower, B. Brumitt, and G. Borriello. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)*. IEEE Computer Society Press, 2002.
11. F. Hohl, L. Mehrmann, and A. Hamdan. A context system for a mobile service platform. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *Proceedings of ARCS 2002: Trends in Network and Pervasive Computing*, volume 2299 of *Lecture Notes in Computer Science*, pages 21–33, Karslruhe, Germany, Mar. 2002. Springer Verlag.
12. A. C. Huang, B. C. Ling, S. Ponnekanti, and A. Fox. Pervasive computing: What is it good for? In *In Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 84–91. ACM Press, Aug. 1999.
13. C. Jiang and P. Steenkiste. A hybrid location model with a computable location identifier for ubiquitous computing. In G. Borriello and L. E. Holmquist, editors, *Proceedings of Ubicomp 2002: Ubiquitous Computing*, volume 2498 of *Lecture Notes in Computer Science*, pages 246–263, Göteborg, Sweden, Sept. 2002. Springer Verlag.
14. M. Langheinrich. Privacy by Design – Principles of Privacy-Aware Ubiquitous Systems. In G. D. Abowd, B. Brumitt, and S. Shafer, editors, *Proceedings of Ubicomp 2001: Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291, Atlanta, Georgia, USA, Sept. 2001. Springer Verlag.
15. H. Lei, D. M. Sow, I. John S. Davis, G. Banavar, and M. R. Ebling. The design and applications of a context service. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):45–55, 2002.
16. T. Moran and P. Dourish. Introduction to this speical issue on context-aware computing. *Human-Computer Interaction*, 16:87–95, 2001.
17. J. Pascoe. Adding generic contextual capabilities to wearable computers. In *In Proceedings of the Second International Symposium on Wearable Computers*, pages 129–138. IEEE Computer Society Press, Oct. 1998.
18. B. N. Schilit, M. M. Theimer, and B. B. Welch. Customizing mobile applications. In *Proceedings of USENIX Mobile and Location-Independent Computing Symposium*, pages 129–138. USENIX Association, Aug. 1993.
19. R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the parctab ubiquitous computing environment. *IEEE Personal Communications*, 2(6):28–43, 1995.
20. M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, September 1991.