A Software Framework for Developing Distributed Cooperative Decision Support Systems – Construction Phase

Alexandre Gachet

Working Paper – March 2002
Decision Support Systems Group – Prof. Dr. P. Hättenschwiler
Department of Computer Science – University of Fribourg, Switzerland

e-mail: alexandre.gachet@unifr.ch

Abstract

This paper describes the *construction phase* of the development process of a Software Framework for Developing Distributed Cooperative Decision Support Systems. This construction phase is built upon the two-layered architecture designed during the *elaboration phase* of this development process (Gachet, 2001*b*). The construction of the two layers of the framework is presented with UML class diagrams, UML sequence diagrams, code extracts, and comprehensive explanations.
The *inception phase* of this development process is described in Gachet (2001*a*). The last phase of this framework (*transition*) will be described in forthcoming papers.

Keywords: cooperative DSS, distributed computing, Java, Jini, JavaSpaces

Table of Contents

1. Introduction

1.1. Inception

The inception phase of this project analyzed the reasons why the broad use of DSSs has not occurred yet and made propositions to improve this situation (Gachet, 2001*a*). It showed that, for the most part, modern, distributed computing architectures could solve many of the presented issues.

1.2. Elaboration

The elaboration of our software framework for developing distributed cooperative DSSs could be split in two phases: (1) the elaboration of a Jini-based framework for developing *distributed systems*, and (2) the elaboration of a framework for developing *cooperative DSSs* (Figure 1). The results of the first phase provide an empty, distributed shell offering low-level foundations for the second phase (Gachet, 2001*b*).

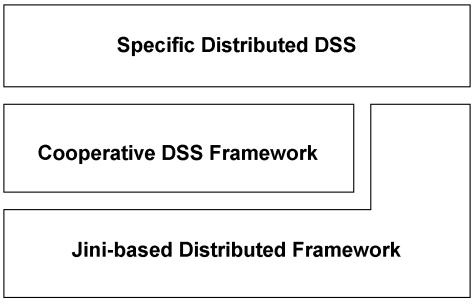

Figure 1. The three different parts of a complete DSS. The bottom two layers build the overall framework.

The *specific distributed DSS* can use, extend, and adapt the functionalities provided by the *cooperative DSS framework* to design its own DSS environment, and the functionalities provided by the *Jini-based distributed framework* to solve low-level, technical issues. Furthermore, the *cooperative DSS framework* is itself built on top of the *Jini-based distributed framework*. Thus, this basic architecture defines a clear separation between the two main families of functionalities of the framework.

1.3. Organization

This paper is definitely *not* organized as a tutorial explaining how to develop a distributed DSS with the proposed framework. It is mostly organized as a description of the construction phase of the framework. After this introductory part, the second part deals with the construction phase of the bottom layer of the overall framework (the *Jini-based distributed framework*). This layer breaks down into various pieces, which are first described individually, and then described in relation to the other pieces. The third part deals with the construction phase of the top layer of the overall framework (the *cooperative DSS framework*). Again, the layer is broken down into various pieces described both individually and in relation to each other. The last part concludes the paper and presents future research directions. An appendice giving an across-the-board view of all the Java packages and classes of the framework wraps up the description of this construction phase.

1.4. Notation

This paper presents many figures to illustrate the details of the framework. Most diagrams are designed as *UML class diagrams*, or as *UML sequence diagrams*. More information about the UML notation can be found in Fowler (2000).

Furthermore, this paper also uses a notation extending the UML class diagrams to clearly separate *(a)* the classes provided by the Java language and the Jini technology, *(b)* the classes of the framework (that is, the classes of the bottom two layers of Figure 1), and *(c)* the classes that the DSS implementor has to create to build a specific DSS (the top layer of Figure 1). Figure 2 exemplifies this notation.

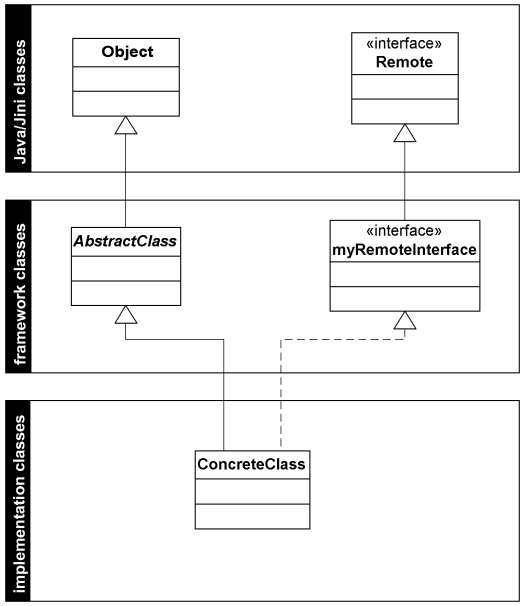

Figure 2. An example of a typical class diagram

Figure 2 shows a **ConcreteClass** extending **AbstractClass** and implementing **myRemoteInterface**. **myRemoteInterface** extends the **java.rmi.Remote** interface and, as with all Java classes, **AbstractClass** is a subclass of **java.lang.Object**. The surrounding boxes clearly indicate which classes belong to the *Java/Jini SDK*, which classes belong to the *framework*, and which classes belong to the *implementation*. These boxes will be used throughout this paper. The Java/Jini classes box is omitted if it is not absolutely needed. Moreover, to avoid cluttering the figures and befuddling the reader's mind, note that the arguments of the methods are never displayed.

1.5. Unit Testing

The different steps of the construction phase have been validated against repeatable, unit tests. According to Griffiths (2001), "a unit test exercises *a unit* of production code in isolation from the full system and checks that the results are as expected. The size of *a unit* to be tested depends on the size of a set of coherent functionality and in practice varies between a class and a package. The purpose is to identify bugs in the code being tested prior to integrating the code into the rest of the system."

These repeatable tests have been implemented using the *JUnit* framework and are provided in the **ch.unifr.dicodess.junit** package of the sources. For more information about *JUnit*, visit the URL *http://www.junit.org*. Moreover, the code has been optimized using the *Borland Optimize*it *Suite (http://www.optimizeit.com)*.

2. Jini-based Distributed Framework

2.1. Basic Requirement

2.1.1. Presentation / Logic Separation

A good framework should make no assumption about the user interface (UI) of the specific systems based on it. In other words, the classes provided by the framework should be usable for developing a simple text-based system, as well as for developing a system using a complex graphical user interface, or even for developing systems with more exotic UI (cell phones, web-based applications, etc.)

To achieve this requirement, the framework must provide a neat and tidy separation between the *presentation* and the *logic* of the system components. After analyzing different well-known design patterns (*Model-View-Controller*, *Presentation-Abstraction-Control*) (see Buschmann et al., 1996), we chose a simple pattern based on aggregation, well suited for the Java programming language. This basic pattern is illustrated in Figure 3. For more information about the elaboration of the pattern, see Gachet (2001*b*).

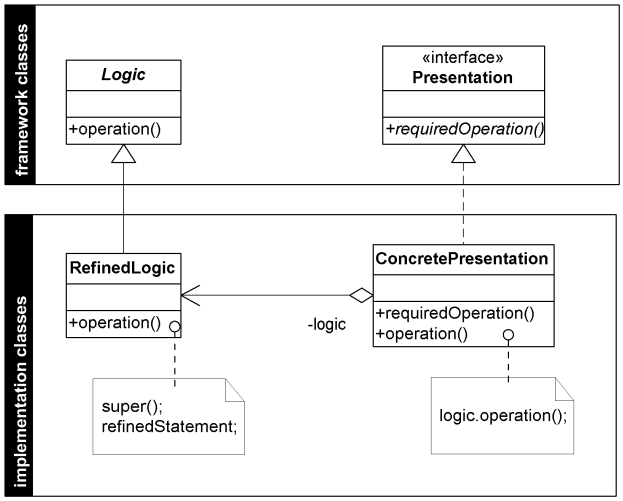

Figure 3. Presentation/Logic Separation

2.1.2. Extension and Implementation

According to Buschmann et al. (1996), a *framework* is "a partially complete software (sub-) system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made."

In our software framework for building distributed cooperative DSSs, we mainly provide two means of adaptation for specific functionality: *(a)* adaptation *by extension*, and *(b)* adaptation *by implementation*. Figure 4 helps to explain these concepts.

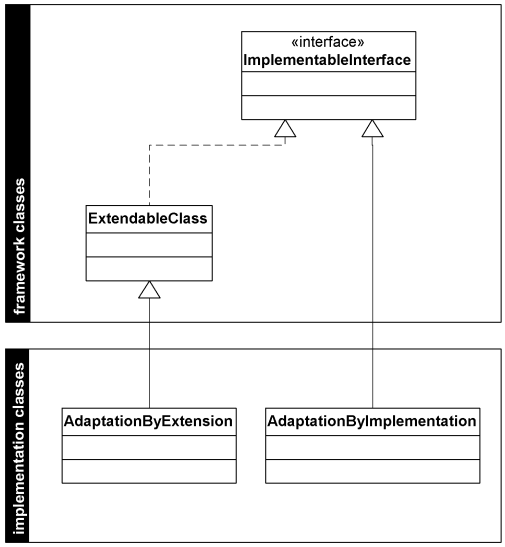

Figure 4. Adaptation by extension *or* by implementation

In many situations, the framework will offer one Java interface (**ImplementableInterface**), and one Java class (**ExtendableClass**) providing a default implementation of this interface. If this default implementation is adapted to the DSS implementor's needs, or if she wants to develop a prototype very quickly, she can easily build a trivial subclass of **ExtendableClass** and use it in her specific DSS (**AdaptationByExtension**). However, if the default implementation is not adapted to the DSS implementor's needs, or if she has a different mindset about the implementation of the **ImplementableInterface**, she can directly write her own implementation of the interface (**AdaptationByImplementation**), which will need more development time, but will eventually produce a class better suited to her needs.

2.1.3. Event Model

The design pattern of Figure 3 showed a clean separation between the *presentation* and the *application logic*. However, it did not provide a two-way communication channel between these two parts. The aggregation relationship between the **ConcretePresentation** class and the **RefinedLogic** class provides a one-way communication channel (from the UI to the logic), but the *logic* part cannot send information back to the UI. Figure 5 illustrates the architecture of the *event model* designed to address this issue for the Jini-based distributed framework. It is a simple *store-and-forward* event adapter that suits the needs of the framework better than other event adapters, such as *notification filter agents* or *notification mailbox agents*.

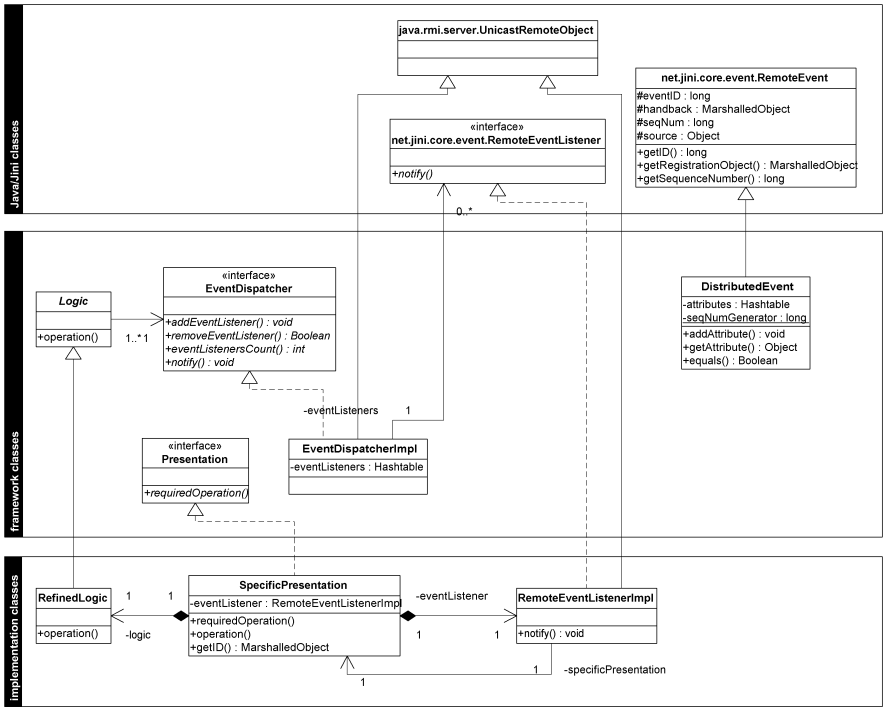

Figure 5. The event model

In Figure 5, the **SpecificPresentation** class instantiates the **RemoteEventListenerImpl** class (which itself extends the **net.jini.core.event.RemoteEventListener** interface that

any object that wants to receive a notification of a remote event from some other object needs to implement) and passes a reference to itself to the listener. **Logic** classes contact an **EventDispatcher** Jini service, whose implementation (**EventDispatcherImpl**) manages an **Hashtable** of interested **RemoteEventListener**s. Note that both **EventDispatcherImpl** and **RemoteEventListenerImpl** extend **java.rmi.server.UnicastRemoteObject**, as they have a remote behavior. Thus, when a **Logic** class needs to send information back to the UI, it asks the **EventDispatcher** service to notify the interested event listeners by calling their respective `notify()` method with a **DistributedEvent** as a parameter.

This pattern borrows some concepts from the *Jini Event Model*. Indeed, the **RemoteEventListenerImpl** class implements the **net.jini.core.event.RemoteEventListener** interface, and the **DistributedEvent** class extends the **net.jini.core.event.RemoteEvent** class. Moreover, the **EventDispatcher** interface defines some familiar method signatures:

```
public void addEventListener(RemoteEventListener el,
                             MarshalledObject id)
                                                 throws RemoteException;

public boolean removeEventListener(RemoteEventListener el)
                                                 throws RemoteException;

public int eventListenersCount() throws RemoteException;

public void notify(RemoteEvent evt) throws RemoteException;
```

The `notify()` method is of particular interest. In this framework, two kinds of events can occur: *unicast* events, and *multicast* events. *Unicast events* are events that should be sent only to the instance of **SpecificPresentation** associated with the event producer (the **RefinedLogic** class). In other words, even if the event dispatcher service manages several remote event listeners, it should notify only one **RemoteEventListener** in response to an unicast event: the remote event listener created by the instance of **SpecificPresentation** associated with the event producer (the **RefinedLogic** class). On the other hand, *multicast events* are events that should be sent to all the **SpecificPresentation** instances that expressed interest in received remote events by registering a **RemoteEventListenerImpl** with the **EventDispatcher**. This distinction is very similar to the distinction between unicast and multicast packets sent over a TCP/IP network.

However, to make this distinction, the event dispatching service needs to identify *(a)* the event producer (the **Logic** class), and *(b)* the **RemoteEventListener** associated with the **SpecificPresentation** of this event producer. To achieve this goal, each instance of **SpecificPresentation** receives a globally unique ID. This unique ID is wrapped in a **MarshalledObject** and is used as a handback in every unicast **DistributedEvent** sent to the **EventDispatcher** service. (For more information about the handback object, refer to the Javadoc of the **net.jini.core.event.RemoteEvent** class). Moreover, an instance of **SpecificPresentation** always passes its unique ID along with its **RemoteEventListenerImpl** during registration (`public void addEventListener(RemoteEventListener el, MarshalledObject id)`). Consequently, the event dispatching service can also store and retrieve the unique ID associated with any given **RemoteEventListener**. The event dispatching service can then compare the handback object with the retrieved ID and see if they are

equivalent. If they are, the **DistributedEvent** is sent to the **RemoteEventListenerImpl**. If they are not, the **DistributedEvent** is not sent to this **RemoteEventListenerImpl**.

The situation is much easier when dealing with multicast **DistributedEvent**s. The event producer just needs to pass a `null` value instead of an unique ID. When receiving such a **DistributedEvent**, the event dispatching service knows that it needs to notify *all* its registered **RemoteEventListener**s, with no regard to their respective unique IDs. Figure 6 and Figure 7 are sequence diagrams dwelling on these operations.

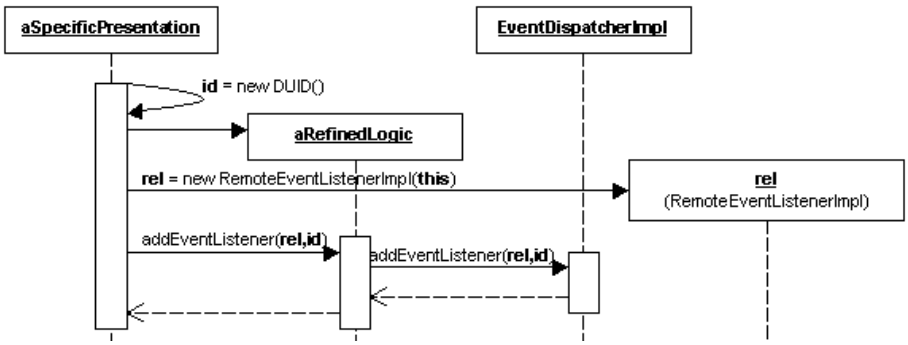

Figure 6. Setup of the event model

In Figure 6, **aSpecificPresentation** first generates its *distributed unique ID* (`new DUID()`). The **DUID** class will be detailed in section 3.3. For the moment, it is enough to know that a *distributed unique ID* is a combination of a **java.rmi.server.UID** (an identifier that is unique with respect to the the host on which it is generated), and of a **java.net.InetAddress** (which makes the UID globally unique). Then, **aSpecificPresentation** creates a new instance of **RefinedLogic** and a new instance of **RemoteEventListenerImpl**. Note that **aSpecificPresentation** passes a reference to itself (`this`) as an argument of the constructor of **RemoteEventListenerImpl**. Then, it delegates the registration of its distributed event listener to **aRefinedLogic**. **aRefinedLogic** contacts the event dispatching Jini service and registers the distributed event listener *along with* the DUID of **aSpecificPresentation** with it.

Figure 7 illustrates two situations. In situation *(a)*, **aRefinedLogic** wants to notify only its associated instance of **SpecificPresentation** (*unicast* event). It does so by calling the `notify()` method on the event dispatching service, with a **DistributedEvent** whose handback object is the distributed unique ID of the corresponding **SpecificPresentation**. **EventDispatcherImpl** first retrieves the non-null ID of the **DistributedEvent**. Then, for each remote event listener that it manages, it retrieves the associated ID (passed during the registration, as explained in Figure 6) and, if both IDs are equals, it calls the `notify()` method of the corresponding **RemoteEventListener**, which in turn executes the appropriate operation on **aSpecificPresentation**. In situation *(b)*, **aRefinedLogic** wants to notify all the instances of **SpecificPresentation** living in the Jini federation, and which have registered a **RemoteEventListener** (*multicast* event). It does so by calling the `notify()` method on the event dispatcher Jini service, with a **DistributedEvent** whose handback object is `null`. In that case, the event dispatching service simply calls the `notify()` method of all the **DistributedEventListener**s it manages.

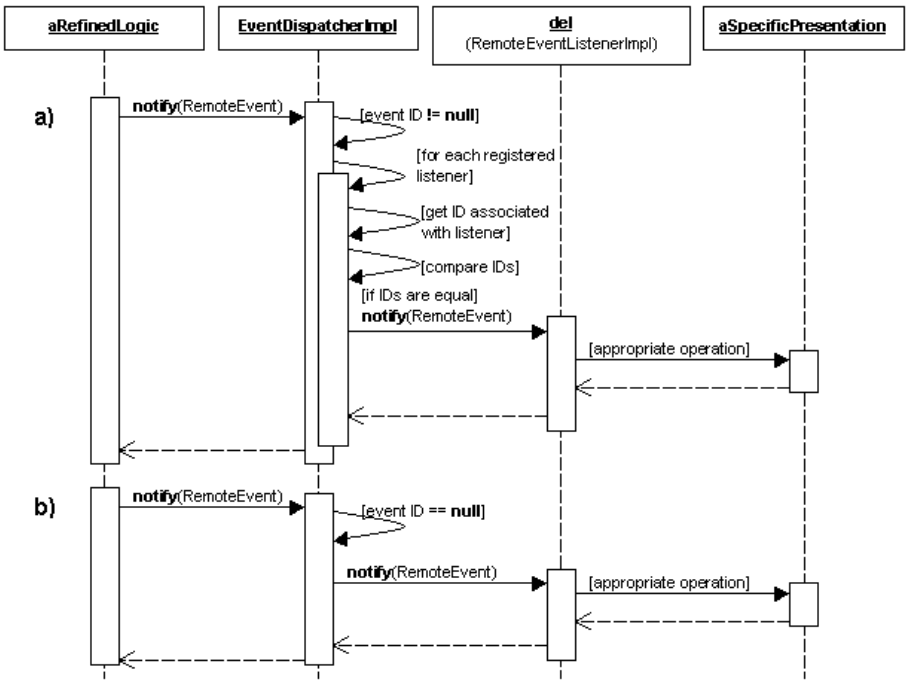

Figure 7. The `notify()` *method*

The **EventDispatcherImpl** class (the implementation of the **EventDispatcher** interface provided by the framework) is used as a Jini service. The current version of this service does not use the leasing model to manage the registered event listeners. It uses a simpler model instead: if a registered remote event listener fails to answer to a `notify()` call, the **EventDispatcherImpl** simply discards it from its list of listeners. The corresponding **SpecificPresentation** class will have to register again its **DistributedEventListener** if it wants to receive newer notifications. A better version of this service, benefitting from Jini's leasing model, should replace the current version in the future. Furthermore, the DSS implementor is free to implement her own version of the **EventDispatcher** interface, according to the "adaptation by implementation" principle (section 2.1.2).

The **DistributedEvent** class is a subclass of **net.jini.core.event.RemoteEvent**, adding the support of generic attributes, with a simple **Hashtable**:

```
private Hashtable att = new Hashtable();

public void addAttribute(Object key, Object value) {
    att.put(key, value);
}

public Object getAttribute(Object key) {
    return att.get(key);
}
```

Moreover, the **DistributedEvent** class ensures that the remote events are ordered on a *per-producer* basis. In other words, if a remote event A from producer X has a sequence number equal to 1, and a remote event B also from producer X has a sequence number equal to 2, then it is sure that event A has been generated *before*

event B. However, had event B been generated by another producer, no assumption could be made about the order of the two events. This *per-producer* basis of the events order is easier to implement and proved to be satisfactory for the framework.

2.1.4. Modular Approach

Our framework is designed to allow the DSS implementor to create a small main class, which in turn calls different *modules* or *services*, each of whichwraps a part of the *application logic*. For example, the code responsible for user authentication can be developed as a separate module, or service.

This distinction between modules and services is due to the use of the distributed Jini technology. If we were developing a framework for developing local applications, we would only talk about modules. In other words, our user authentication module would just be another class associated with the main class (Figure 8).

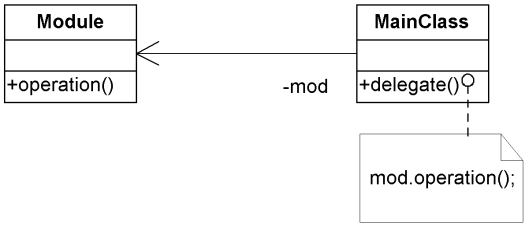

Figure 8. Traditional modular composition

However, with the Jini technology, we can define our modules as *distributed Jini services*, rather than local modules. In our example, it means that a server could launch a user authentication service on the network; then, the main class would access this service at runtime, instead of using a local module. As explained in Gachet (2001*b*), this approach provides several advantages:

- The main client becomes a *thin client*;
- Changes made to the service do not affect the main client, even at runtime;
- The same instance of the service can be accessed by several clients.

Figure 9 shows the class diagram defining modules and services. The three classes **LogicModule**, **SpecificLogicModule** and **SpecificPresentation** on the left side of the figure describe the definition of a *module*. The relationship between **SpecificPresentation** and **SpecificLogicModule** expresses the presentation/logic separation presented in section 2.1.1. The **SpecificLogicModule** class extends the **LogicModule** super-class. **LogicModule** provides some general-purpose functionalities; for example, it manages the **EventDispatcher** Jini service introduced in section 2.1.3, using the **ServicesManager** class that will be presented in a forthcoming section (2.2.1).

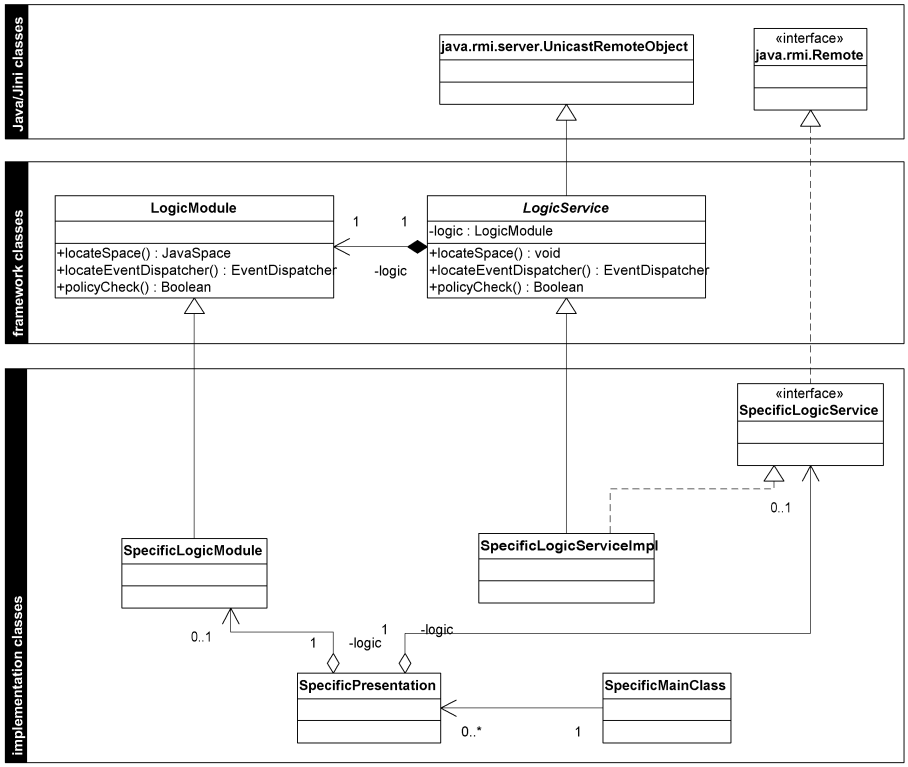

Figure 9. Modules *and* Jini Services

Example 1. *The locateEventDispatcher() method of the LogicModule class*

```
/**
 * This getter method looks in the Jini federation for the
 * EventDispatcher service that different instances of the system use
 * to communicate.
 * @return the EventDispatcher used to communicate
 * @exception NoEventDispatcherException if no event dispatcher
 *                                       service can be found in the
 *                                       federation
 * @exception RemoteException if a communication-related exception
 *                            occurs during the execution of a remote
 *                            method call.
 */
public EventDispatcher locateEventDispatcher()
                  throws NoEventDispatcherException, RemoteException {
    ServiceItem dispatcher = null;
    try {
        dispatcher =
            ServicesManager.getInstance(service.getProjectName()).
                                    lookup(EventDispatcher.class, 1000);
    } catch(IOException ioe) {
        /* in this case, an IOException can be assimilated to a
         * RemoteException */
        throw new RemoteException(ioe.getMessage(), ioe);
    } catch(InterruptedException ie) {
        /* should not happen. If it does, stop the application for
         * easier debug. */
        ie.printStackTrace();
        System.exit(1);
```

```
    }
    if(dispatcher != null)
        return (EventDispatcher)dispatcher.service;
    else
        throw new NoEventDispatcherException();
}
```

The other classes describe the definition of a *service*. A service can be seen as a *remote module*. As a *remote* object, the abstract class named **LogicService** extends the **java.rmi.server.UnicastRemoteObject** class (a Java class that provides support for point-to-point active object references using TCP streams); as a *module*, this same class is bound to the **LogicModule** class (composition relationship). If we choose to implement our framework in a language supporting multiple inheritance, **LogicService** would extend both classes (**UnicastRemoteObject** and **LogicModule**). However, as Java only supports single inheritance, the **LogicService** class forwards all the module-dependant requests to its private instance of the **LogicModule** class, as shown by Example 2:

Example 2. *The locateEventDispatcher() method of the LogicService class*

```
public EventDispatcher locateEventDispatcher()
                    throws NoEventDispatcherException, RemoteException {
    return logic.locateEventDispatcher();
}
```

An alternative solution using interfaces could be possible.

Figure 10 and Figure 11 are two sequence diagrams explaining how a system interacts with modules and with services.

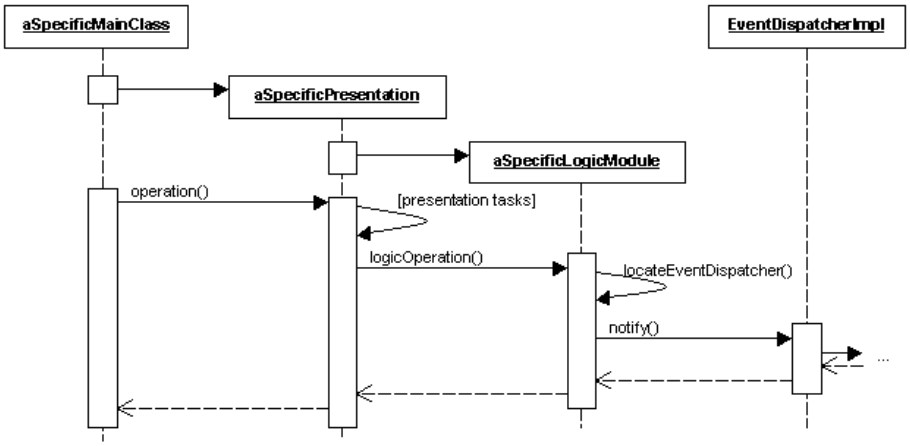

Figure 10. Interaction with a module

Figure 10 is quite self-explanatory. **aSpecificMainClass** creates the presentation class of the module (**aSpecificPresentation**) when needed. According to the presentation/logic separation used throughout the framework, **aSpecificPresentation** creates in turn the logic class of the module (**aSpecificLogicModule**). Then, when **aSpecificMainClass** delegates an operation to **aSpecificPresentation** (`operation()`), **aSpecificPresentation** takes care of the presentation tasks, and delegates the logic work to **aSpecificLogicModule** (`logicOperation()`). Finally, **aSpecificLogicModule** can use the event model presented in section 2.1.3 for notification purposes.

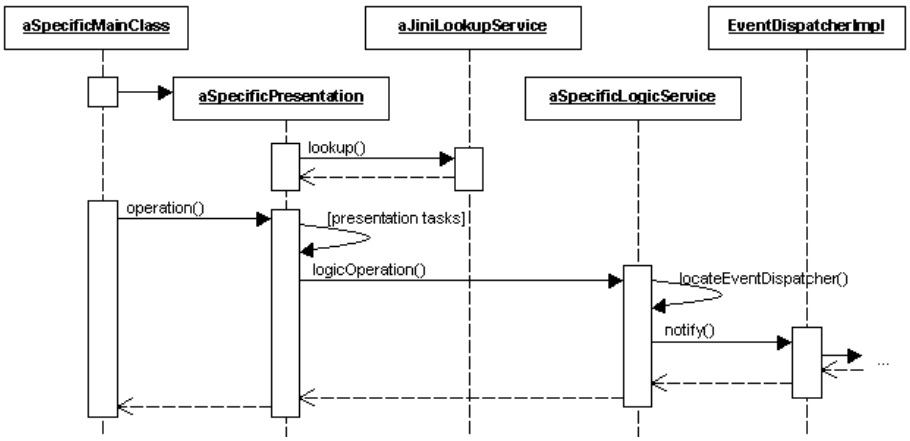

Figure 11. Interaction with a service

Figure 11 is somewhat more interesting, yet not more difficult to understand. As with Figure 10, **aSpecificMainClass** creates the presentation class of the service (**aSpecificPresentation**) when needed. **aSpecificPresentation** will then query Jini lookup services (using the **ServicesManager** class that will be presented in section 2.2.1) to get the *proxy* object of the corresponding logic service. This proxy object allows **aSpecificPresentation** to communicate with **aSpecificLogicService**. (For more information about how Jini lookup services work, refer to Kumaran 2002, or Li et al. 2000). Then, when **aSpecificMainClass** delegates an operation to **aSpecificPresentation** (`operation()`), **aSpecificPresentation** takes care of the presentation tasks, and delegates the logic work to **aSpecificLogicService** (`logicOperation()`). Finally, **aSpecificLogicService** can use the event model presented in section 2.1.3 for notification purposes.

As usual with Java remote objects, we have to define a remote interface. This is the role of the **SpecificLogicService** interface (on the right side of Figure 9), which extends the **java.rmi.Remote** interface (any Java object that is a remote object must directly or indirectly implement this interface). The methods of this interface are eventually implemented by the **SpecificLogicImpl** class.

By virtue of the clean presentation/logic separation previously defined (check back to section 2.1.1), the same instance of **SpecificPresentation** can be bound to a *module* (**SpecificLogicModule**) or to a *service* (**SpecificLogicService**).

Even if the **SpecificLogicModule**, **SpecificLogicService**, **SpecificLogicServiceImpl** and **SpecificPresentation** classes of Figure 9 are considered "implementation classes" because they give a DSS implementor the freedom to create her own DSS modules or services, it is important to stress that the framework itself also uses them to create embedded modules and services, such as properties panel, object editors, model solving services, etc.

2.2. Technology Requirements

2.2.1. Managing the Jini Technology

In a previous paper, Gachet (2001*b*) briefly presented the Jini technology (and its JavaSpaces service) and explained why this technology had been chosen to implement the framework. During the construction phase, it was decided to use the *singleton* design pattern to create a generic class managing all the tasks associated with an existing Jini federation, such as lookup discovery, services discovery and management, caching, etc. Figure 12 shows this central class.

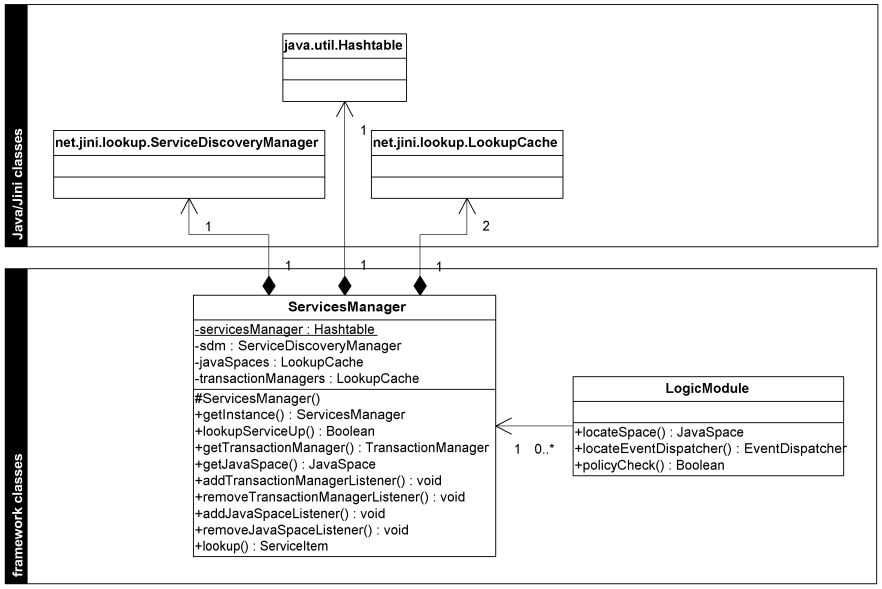

Figure 12. The ServicesManager *class*

The **ServicesManager** class manages one instance of itself per Jini federation. In other words, if a specific DSS combines the knowledge of two Jini federations, the **ServicesManager** class will create one instance of itself for each Jini federation. These instances are stored in the private static `servicesManager` **Hashtable**, and they can be retrieved through the method `getInstance()`, whose single argument is a **String** representing the Jini federation name being sought. Example 1 illustrated the use of this method, with `service.getProjectName()` returning the name of the Jini federation to consider:

```
dispatcher =
            ServicesManager.getInstance(service.getProjectName()).
                                    lookup(EventDispatcher.class, 1000);
```

The protected constructor automatically instantiates a **LookupDiscoveryManager**, as well as a **ServiceDiscoveryManager** (`sdm`) and two **LookupCache**s: one for the JavaSpaces services (`javaSpaces`), and one for the TransactionManager services (`transactionManagers`). All the remaining public methods of this class use the

information of these discovery managers and lookup caches to return the needed information in a quick and efficient way.

Example 3. *The protected constructor of ServicesManager*

```
/**
 * This protected constructor initializes the main
 * ServiceDiscoveryManager for the given federation and creates two
 * LookupCaches for the JavaSpaces services and the Transaction
 * services.
 * @param federation the name of the Jini federation
 * @exception IOException can be thrown when socket allocation occurs
 *                        or during the multicast discovery process.
 * @exception RemoteException typically, this exception occurs as a
 *                            result of an attempt to export a remote
 *                            listener that receives service events
 *                            from the lookup services in the managed
 *                            set of the LookupCaches.
 */
protected ServicesManager(String federation) throws IOException,
                                                    RemoteException {
    // set up a service discovery manager for the specified federation
    String[] groups = new String[] { federation };
    LookupLocator[] locators = null;
    DiscoveryListener listener = null;
    /* throws IOException */
    LookupDiscoveryManager discoveryMgr =
        new LookupDiscoveryManager(groups, locators, listener);
    LeaseRenewalManager leaseMgr = null;
    /* A client that uses the SDM must set java.rmi.server.codebase
     * to something that allows lookup services to download the code
     * for the SDM's remote event listener stub:
     * ServiceDiscoveryManager$LookupCacheImpl$LookupListener_Stub
     * in the jini-ext.jar archive */
    /* throws IOException */
    sdm = new ServiceDiscoveryManager(discoveryMgr, leaseMgr);

    // create a lookup cache for the instances of the JavaSpaces
    // service
    Class[] serviceTypes = new Class[] { JavaSpace.class };
    ServiceTemplate template = new ServiceTemplate(null,
                                                   serviceTypes,
                                                   null);
    /* throws RemoteException */
    javaSpaces = sdm.createLookupCache(template, null, null);

    // create a lookup cache for the instances of the transaction
    // manager service
    serviceTypes = new Class[] { TransactionManager.class };
    template = new ServiceTemplate(null, serviceTypes, null);
    /* throws RemoteException */
    transactionManagers = sdm.createLookupCache(template, null, null);
}
```

The **LogicModule** class (in the middle-right part of Figure 12) uses the **ServicesManager** class to find JavaSpaces (and other Jini services) in a Jini federation. The `getJavaSpace()` method allows the **LogicModule** to contact a specific JavaSpace, if it exists in the system. JavaSpaces in our framework are DSS-dependant and will be handled in detail in the second part of this paper.

Example 4. *The getJavaSpace() method*

```
/**
 * Finds a JavaSpace object that satisfies the given manager name
 * parameter. The JavaSpace returned must have been previously
 * discovered to be both registered with one or more of the lookup
 * services in the managed set, and to match criteria defined by the
 * entity.
 * @param mgrName name of the JavaSpace looked after
 * @return JavaSpace that satisfies the space name, and that was
 *         previously discovered to be registered with one or more
 *         lookup services in the managed set. A null value will be
 *         returned if no JavaSpace is found that matches the criteria
 *         or if the cache is empty.
 */
 public JavaSpace getJavaSpace(final String mgrName) {
    ServiceItem item = null;
    ServiceItemFilter filter = new ServiceItemFilter() {
        public boolean check(ServiceItem item) {
            Manager mgr = null;
            try {
                mgr = Manager.getTemplate(mgrName);
                return (((JavaSpace)item.service).
                           read(mgr, null, JavaSpace.NO_WAIT) != null);
            } catch(UnusableEntryException uee) {
                /* reject all candidates throwing exception */
                return false;
            } catch(InterruptedException ie) {
                /* reject all candidates throwing exception */
                return false;
            } catch(RemoteException re) {
                /* reject all candidates throwing exception */
                return false;
            } catch(TransactionException te) {
             ; /* cannot happen. <i>No</i> transaction is used here */
            }
            return false;
        }
    };
    item = javaSpaces.lookup(filter);
    if(item == null) return null;
    else return (JavaSpace)item.service;
}
```

Moreover, the **ServicesManager** class allows the **LogicModule** class to register instances of the **ServiceDiscoveryListener** interface, and to be notified when JavaSpaces and transaction manager services come and go in a federation. Internally, the **ServicesManager** class uses an instance of the **ServiceDiscoveryManager** utility class (`sdm`) to manage all the operations related to lookup discovery, template matching, and services discovery. As the **ServiceDiscoveryManager** class registers its own remote listeners during lookup discovery, the entity using this class needs to include the `<JINI_HOME>`*/lib/sdm-dl.jar* archive in its codebase, otherwise the discovery process will fail silently, without reporting any error. This could lead to a bug difficult to track down.

Example 5. *The addJavaSpaceListener() method*

```
/**
 * Registers a ServiceDiscoveryListener object with the event
 * mechanism of a LookupCache. The listener object will receive a
 * ServiceDiscoveryEvent  upon the discovery, removal, or modification
 * of one of the cache's services. Once a listener is registered, it
 * will be notified of all service references discovered to date, and
 * will be notified as new services are discovered and existing
```

```
 * services are modified or discarded. If the parameter value
 * duplicates (using equals) another element in the set of listeners,
 * no action is taken. If the parameter value is null, a
 * NullPointerException is thrown.
 * @param listener the ServiceDiscoveryListener object to register.
 */
public void addJavaSpaceListener(ServiceDiscoveryListener listener) {
   javaSpaces.addListener(listener);
}
```

The **ServicesManager** class can also be used to find other Jini services. The `getTransactionManager()` method returns an instance of the transaction manager service (if one exists in the federation) and the generic `lookup()` methods allow to look up any Jini service in any federation. Figure 13 and Figure 14 present two examples of interaction between a logic module class and the **ServicesManager**. Note that these sequence diagrams are simplified for the ease of understanding.

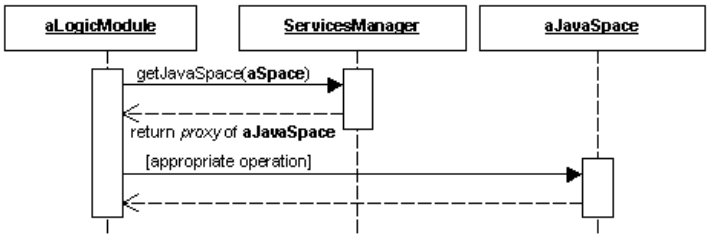

Figure 13. Synchronous interaction with ServicesManager

In Figure 13, **aLogicModule** needs to interact with a JavaSpace whose name is `aSpace`. As **ServicesManager** keeps track of the existing JavaSpaces in the Jini federation, it returns the *proxy* object associated with the JavaSpace of interest (**aJavaSpace**). Then, **aLogicModule** can directly interact with **aJavaSpace**. Things could not be easier for **aLogicModule**.

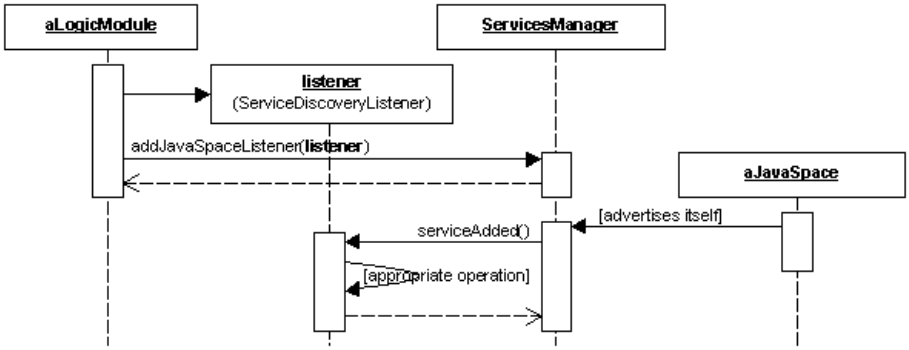

Figure 14. Asynchronous interaction with ServicesManager

In Figure 14, **aLogicModule** wants to be notified every time a JavaSpace enters the Jini federation, leaves the Jini federation, or has its state changed in the Jini federation. To do that, **aLogicModule** first creates a new **ServiceDiscoveryListener** (called `listener` in the figure). Then, it registers this listener with **ServicesManager** (`addJavaSpaceListener()`). As soon as **aJavaSpace** enters the Jini federation, it advertises itself on the network. As the **ServicesManager** class monitors the lookup services in the federation, it gets notified and can in turn notify the registered

ServiceDiscoveryListeners (`serviceAdded()`). The listener can then execute the appropriate operations.

Another utility class frees the DSS implementors from all the tasks associated with the *creation* of a Jini federation, such as the launch of an activation system, the instanciation of a shared virtual machine, the instanciation of HTTP servers, of lookup services, of transaction managers services, of JavaSpaces, etc. All these operations are needed to bootstrap the system and can be tricky to implement for a Java programmer lacking of experience with Jini. This utility class is called **NetworkAdmin** and is presented in Figure 15.

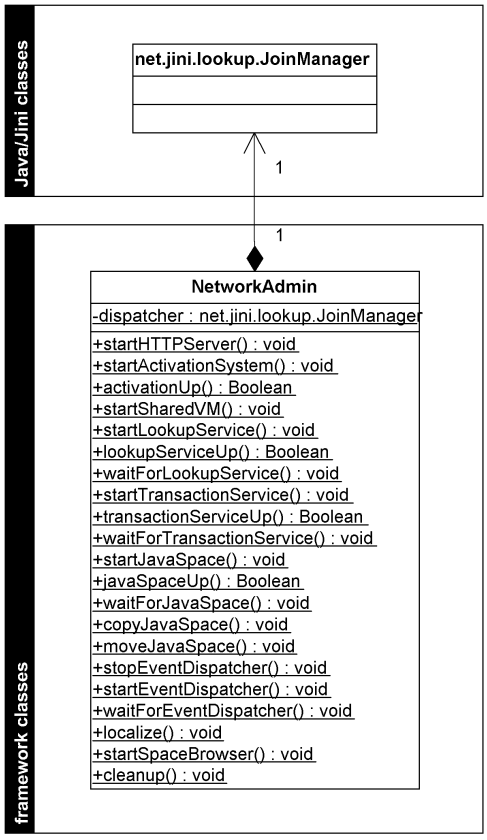

Figure 15. The NetworkAdmin *class*

All the methods of this utility class are static. As explained before, most of them are used during the launch of the system, when a Jini federation is first created. Other methods, such as `copyJavaSpace()`, `moveJavaSpace()` or `localize()` are used at different moments during the lifetime of the system. For more information about this class, consult its Javadoc.

2.2.2. A JavaSpace-oriented Object Model

Objects put in a JavaSpace have to implement the **net.jini.core.entry.Entry** interface. Based on this requirement, we defined a set of classes extending the

net.jini.entry.AbstractEntry class (a specific, generally useful base class for **Entry** types), and modeling the components needed by the framework. Figure 16 illustrates these classes.

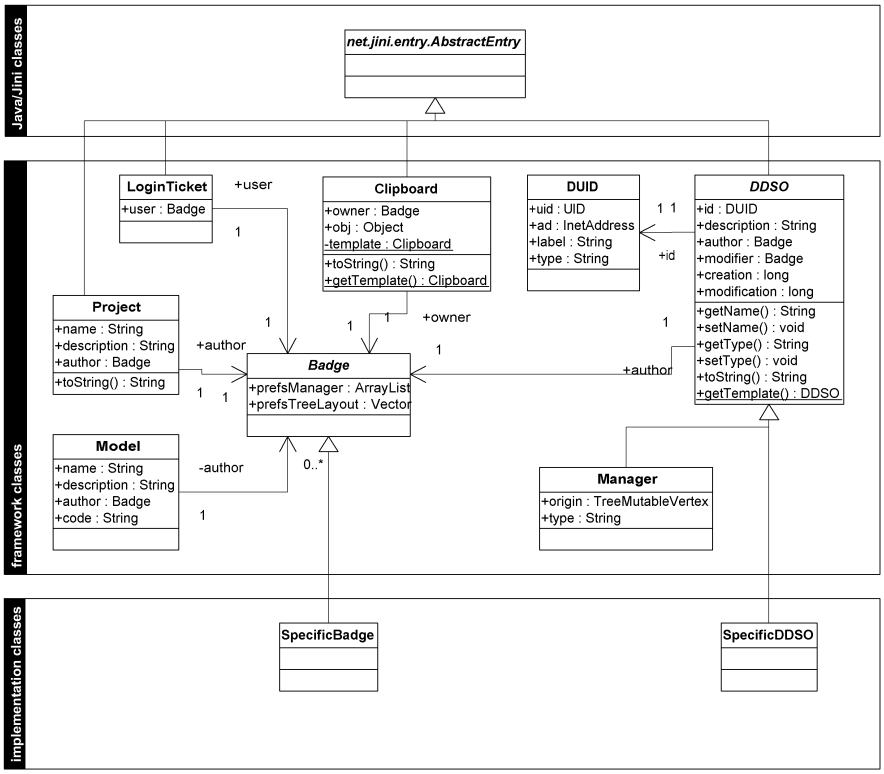

Figure 16. Entry objects

Project obviously stores information needed to define a project. **Badge** is an abstract class used in the system for *user identification* and *authentication*. The exact content of this class is application-dependant. Thus, the DSS implementor has to extend it (**SpecificBadge**). A **LoginTicket** object is created when a user enters the system, and is destroyed when the user leaves the system or disappears for some unexpected reason (network failure, computer crash, session timeout, etc.) This **LoginTicket** can be referenced in order to know when a registered user is connected to the system, and when she is not. This is especially useful if the DSS implementor plans to add a real-time communication service (for more information about such services, see section 4).

The **DDSO** abstract class is the super-class of all DSS-oriented objects in the system. This is a logical extension of the DSO (Decision Support Object) defined by Schroff (1998). This class allows us to extend the works of Schroff in a distributed environment. In the third chapter of this paper, while defining the framework for developing cooperative DDSs (the middle layer of Figure 1), a certain number of classes extending **DDSO** and belonging to the **ch.unifr.dicodess.entry** package will be added to the overall framework. The **Manager** class will also be described in chapter three. Note that the DSS implementor is free to extend **DDSO** in order to define her

own DDSOs (**SpecificDDSO**). Finally, the **Clipboard** class is a special object used for distributed copy/paste operations.

All of these classes are very simple. This represents one of the advantages of the JavaSpaces technology, which allows neat and tidy definitions of objects that will be used in distributed environments. Example 6 shows the implementation of the **DDSO** class.

Example 6. *The DDSO class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.DUID;
import net.jini.entry.AbstractEntry;
import java.net.UnknownHostException;
import java.util.Date;

/**
 * This abstract class defines a general-purpose Distributed Decision
 * Support Object. It is subclassed by all classes defining decision
 * objects stored in a JavaSpace. Its super-class is
 * net.jini.entry.AbstractEntry, which implements Entry as well as the
 * equals(), toString(), and hashCode() methods.
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 **/
public class DDSO extends AbstractEntry {
    /** unique ID */
    public DUID id;
    /** general-purpose fields */
    public String description;
    /** general-purpose fields */
    public Badge author, modifier;
    /** general-purpose fields */
    public Long creation, modification;
    /** specific type of this DDSO */
    /**
     * class variable used as "shell" for all the template requests
     */
    private static DDSO template;

    // static initializer
    static {
        template = new DDSO();
    }

    /**
     * This public constructor is required by Jini, but also used by
     * subclasses (e.g. Badge) when defining templates.
     **/
    public DDSO() {
        // this constructor leaves all the fields of this DDSO as null
        /* super() */
    }

    /**
     * Constructor. Creates a new DDSO. This constructor is never used
     * to create DDSO templates.
     * @param name human-readable name of the DDSO
     * @param description human-readable description of the DDSO
     * @param author the Badge of the author of this DDSO
```

```
 * @param type a String describing the type of the DDSO
 * @exception UnknownHostException if no IP address for the host
 *                                 could be found.
 **/
public DDSO(String name, String description,
          Badge author, String type) throws UnknownHostException {
                  /* throws UnknownHostException */
    this.id = new DUID(name, type); // unique ID
    this.description = description;
    this.author = this.modifier = author;
    this.creation = this.modification =
        new Long(new Date().getTime());
}

/**
 * Constructor. Creates a new DDSO. This constructor should never
 * be used to create DDSO templates.
 * @param name human-readable description of the DDSO
 * @param type a String describing the type of the DDSO
 * @exception UnknownHostException if no IP address for the host
 *                                 could be found.
 **/
public DDSO(String name, String type)
                                    throws UnknownHostException {
    this(name, null, null, type);
}

/**
 * This private constructor is only used by subclasses when
 * defining templates.
 * @param id DUID of the requested entry
 **/
private DDSO(DUID id) {
    this.id = id;
}

/**
 * Creates a DDSO template with the specified information. This
 * method is synchronized because it always uses the same instance
 * as template.
 * @param id DUID of the requested DDSO
 * @return a DDSO template containing the required information
 **/
public synchronized static DDSO getTemplate(DUID id) {
    template.id = id;
    return template;
}

/**
 * Getter method. Retrieves the type field of this DSDO's DUID
 * @return the type field of this DDSO's DUID
 */
public String getType() {
    return id.getType();
}

/**
 * Setter method. Change the type attribute in this DDSO and in
 * the corresponding DUID
 * @param type new type
 **/
public void setType(String type) {
    this.id.setType(type);
}
```

```
    /**
     * Getter method. Retrieves the name field of this DSDO's DUID
     * @return the name field of this DDSO's DUID
     */
    public String getName() {
        return id.getLabel();
    }

    /**
     * Setter method. Change the name attribute in this DDSO and in
     * the corresponding DUID
     * @param name new name
     **/
    public void setName(String name) {
        this.id.setLabel(name);
    }

    /**
     * Trivial implementation of Object.toString()
     * @return the name of this DDSO
     **/
    public String toString() {
        return id.getLabel();
    }
}
```

2.3. Putting Pieces Together

The previous sections presented different facets of the Jini-based distributed framework (presentation/logic separation, event model, modular architecture, Jini technology management, JavaSpaces entries). It is now time to glue these pieces together, in order to consolidate the framework. Figure 17 introduces the necessary gluing classes.

In Figure 17, the **RemoteEventListenerImpl** implementation class is the point of contact with the *event model* (section 2.1.3), **LogicModule** is the point of contact with the *modular architecture* (section 2.1.4) and with the *Jini technology management* (section 2.2.1), and the **LogicDSSImpl** class is the point of contact with the JavaSpaces-oriented object model (section 2.2.2).

The triplet **OME**, **OMEImpl** and **SpecificOME** (on the right side of the figure) and the group of four classes **LogicDSS**, **LogicDSSImpl**, **SpecificLogicDSS** and **SpecificLogicDSSImpl** (on the left side of the figure) build the presentation/logic combination of the kernel of the framework. OME stands for "Object Manager Environment", a DSS concept first put forth by Schroff (1998). In our framework, an *object manager environment* manages all the tasks associated with the presentation of the **DDSO**s to the user. This environment will be detailed in the third part of this paper. The **OME** interface and the **OMEImpl** abstract class comply with the *adaption by extension/implementation* principle presented in section 2.1.2. The **LogicDSS** interface extends **java.rmi.Remote**, because it can be used as a remote point of contact by other Jini services. The **LogicDSSImpl** class implements this interface and extends the **LogicService** class (not displayed here to avoid cluttering the figure; check back to Figure 9 for more information about the **LogicService** class). One of the first tasks of a specific DSS will be to instanciate the **SpecificOME** class.

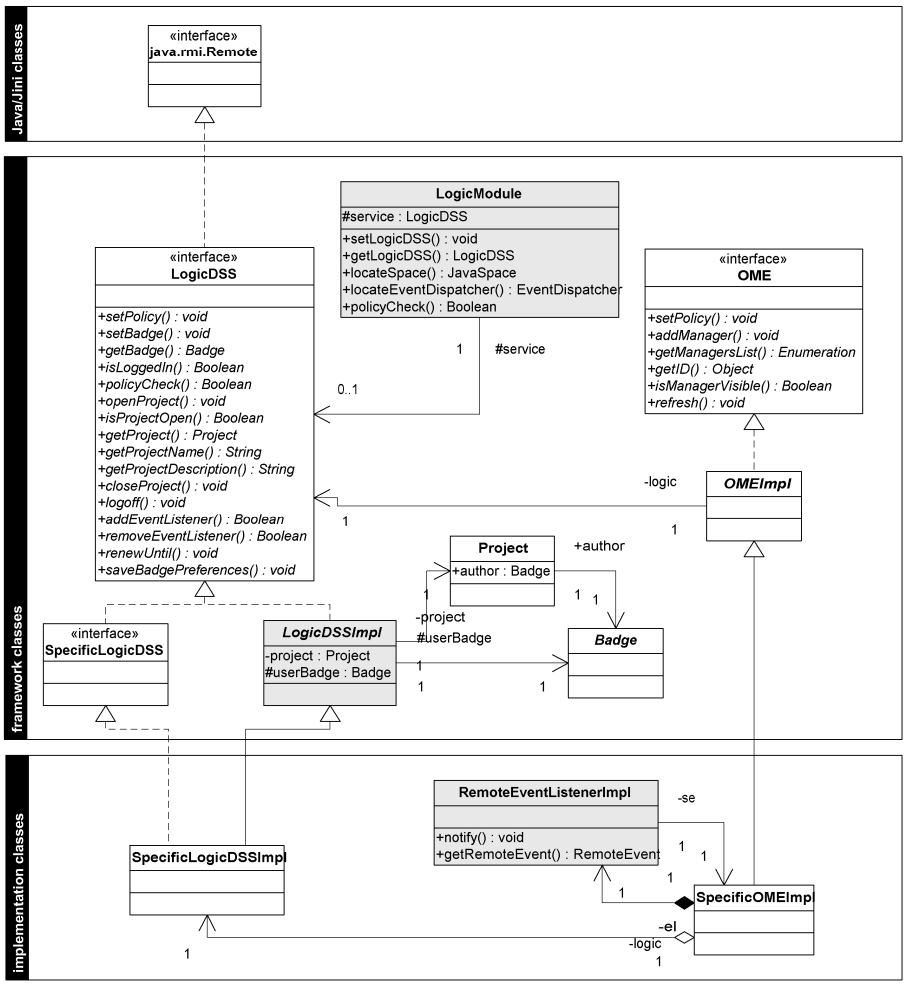

Figure 17. Putting pieces together

3. A Framework for Developing Cooperative DSSs

3.1. Introduction

As explained in the first part of this paper, the *framework for developing cooperative DSSs* is built on top of the *Jini-based distributed framework* introduced in the second part. Check back to Figure 1 to see the relationship between these two layers of the overall framework. The framework for developing cooperative DSSs is inspired by Schroff (1998), whose works have been summarized in Gachet (2001*b*). It is mandatory to be familiar with his research to fully understand the following explanations. The innovative DSS approach used in this framework is illustrated in Figure 18 (Hättenschwiler, 1999). Information can also be found in Hättenschwiler et al. (1998).

The framework for developing cooperative DSSs is based on three main concepts: (1) a *use case model*, (2) *distributed decision support objects*, and (3) *object managers.*

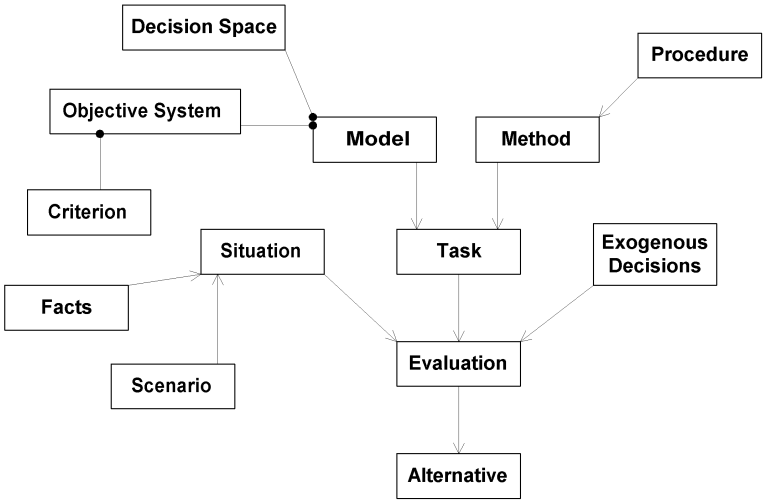

Figure 18. Innovative DSS approach. See Hättenschwiler (1999)

3.2. Use Cases

One of the goals of a distributed system is to allow different *users* to access the system at the same time, from different locations. These different users represent different *roles* in the system. This is especially true for Decision Support Systems as, for example, the role of the *decision-maker* is completely different from the role of the *system administrator.* Thus, one of the first requirements of a distributed framework is to provide an *user identification and authentication* mechanism, in order to identify the role of the user and to offer her a suitable environment (for

example, the *decision-maker* role does not need an administrative console, as administration is the job of the *system administrator* role).

Schroff (1998) presented a *use case model of DSSs*. He identified seven *roles* and proposed a *use case diagram* for each of them. These seven roles are: the *decision assistant*, the *model expert*, the *facts administrator*, the *facts updater*, the *system administrator*, the *consultant*, and the *decision-maker*. The corresponding *use case diagrams* represent the starting point of the elaboration of our framework. Refer to Gachet (2001*b*) for a brief presentation of each role and its corresponding use case diagram.

3.3. Distributed Decision Support Objects (DDSOs)

Figure 16 introduced the **DDSO** class, which extends the concept of *Decision Support Object (DSO)* in a distributed environment. As a subclass of **net.jini.entry.AbstractEntry**, this class defines objects that can be managed in JavaSpaces. Each DDSO is identified by a globally unique identifier. This id is an instance of the **DUID** class (which is the acronym for "Distributed Unique ID"), that we already mentioned in section 2.1.3. A **DUID** combines an instance of **java.server.rmi.UID** (an abstraction for creating identifiers that are unique in regards to their respective hosts), and an instance of **java.net.InetAddress** (a representation of the host's IP address where the DDSO is created). This combination never changes during the lifecycle of the DDSO and this id is the only way to identify a DDSO; it acts like a pointer to the actual DDSO. The **DUID** class contains two more pieces of information: a **label** field, which is a **String** representation of the underlying object, and a **type** field, which is a **String** defining the type of the underlying object (for example, SCENARIO, TASK, INFLUENCE_FACTOR, etc.) The **label** field is used here to provide a human-readable description of the underlying object (for example, as a node label in a tree structure, as a simple name in a list, as a link in a web page, etc.) without querying the JavaSpace for the real object. Otherwise, the display of a project with hundreds of DDSOs would imply a slew of communication overhead between the application and the JavaSpaces, and would oviously raise efficiency issues. However, the `equals()` method of the **DUID** class only checks the **UID** and **InetAddress** objects for equality, as the value of the **label** field can change during the lifetime of the **DUID** (for example after a *rename* operation).

Example 7. *The DUID class*

```
package ch.unifr.dicodess.core;

import ch.unifr.gachet.graph.ApplicationObject;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.rmi.server.UID;

/**
 * Distributed Unique ID. Pair an instance of InetAddress with an
 * instance of UID.
 * Extends ApplicationObject to be compatible with the clone process
 * of TreeMutableVertex
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
```

```
 **/
public class DUID extends ApplicationObject {
    /** Unique ID with respect to the host on which it is generated.
      */
    private UID uid;
    /** IP address of the machine on which this DUID is generated */
    private InetAddress ad;
    /** human-readable String associated with the DUID */
    private String label;
    /** user-defined type of the object represented by this DUID */
    private String type;

    /**
     * Constructor. Creates a new DUID with the specified information
     * @param label human-readable String associated with this DUID
     * @param type user-defined type of the object represented by this
     * DUID.
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     */
    public DUID(String label, String type)
                                        throws UnknownHostException {
        uid = new UID();
        /* throws UnknownHostException */
        ad = InetAddress.getLocalHost();
        this.label = label;
        this.type = type;
    }

    /**
     * Getter method. Retrieves the uid field of this DUID
     * @return the uid field of this DUID
     */
    public UID getUID() {
        return uid;
    }

    /**
     * Getter method. Retrieves the ad field of this DUID
     * @return the ad field of this DUID
     */
    public InetAddress getInetAddress() {
        return ad;
    }

    /**
     * Getter method. Retrieves the label field of this DUID
     * @return the label field of this DUID
     */
    public String getLabel() {
        return label;
    }

    /**
     * Setter method. Set the label field of this DUID
     * @param label the new value for the label
     */
    public void setLabel(String label) {
        this.label = label;
    }

    /**
     * Getter method. Retrieves the type field of this DUID
     * @return the type field of this DUID
```

```
     */
    public String getType() {
        return type;
    }

    /**
     * Setter method. Set the type field of this DUID
     * @param type the new value for the type field
     */
    public void setType(String type) {
        this.type = type;
    }

    /**
     * Simple hashcode method that may sound strange, but that
     * respects criterions defined by Object.hashCode()
     * @return an hashcode as an int
     **/
    public int hashCode() {
        return uid.hashCode();
    }

    /**
     * Check two DUID objects for equality
     * @param d the other object
     * @return true if both objects are equals, false otherwise
     */
    public boolean equals(Object d) {
        return(uid.equals(((DUID)d).getUID()) &&
               ad.equals(((DUID)d).getInetAddress()));
    }

    /**
     * Trivial implementation of Object.toString()
     * @return the label of this DDSO
     **/
    public String toString() {
        return label;
    }

    /**
     * Trivial implementation of Object.clone()
     * @return a clone of this DUID
     * @exception CloneNotSupportedException -
     **/
    public Object clone() throws CloneNotSupportedException {
        return (DUID)super.clone();
    }
}
```

The framework provides the DDSOs necessary to run an evaluation. The DSS implementor is then free to add her own DDSOs according to her needs. The following sections will briefly present the logical relationships between the DDSOs of a same manager. The object managers themselves will be introduced in section 3.4. For the moment, it is enough to say that the framework comes precustomized with six object managers: the *system* manager, the *facts* manager, the *scenario* manager, the *task* manager, the *evaluation* manager, and the *report* manager.

3.3.1. The System Manager's DDSOs

We begin our discussion with the *System Manager*'s DDSOs. The System Manager is responsible for the organisational information of the system. Figure 19 shows the logical relationships between the main **DDSO** class and two of its subclasses: **Badge** and **Group**. As mentioned in section 2.2.2, user roles are represented by **Badge** objects in the framework. A **Badge** is an abstract class that needs to be extended by the DSS implementor (a **Badge** subclass can use a typical username/password combination, or a more exotic authentication structure). A **Badge** is a member of at least one **Group**. Note that the logical structure illustrated in Figure 19 does not represent a recursive association on **Group** (which would allow groups to be nested in other groups). In fact, the framework considers that this recursion is more a *presentation* information than a *logic* information. This distinction between presentation and logic in an object manager will be detailed in section 3.4. Thus, the definition of the recursive association is left to the corresponding presentation structure. However, the DSS implementor is free to extend **Group** if she wants to explicitly represent this recursive association.

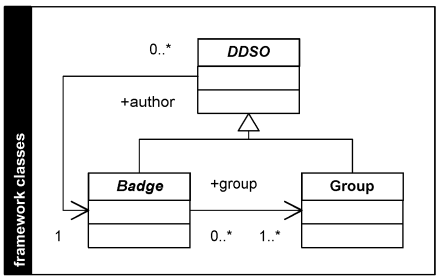

Figure 19. DDSOs needed by the System Manager

Example 8 and Example 9 show the implementations of the **Badge** and **Group** classes. The framework also provides a concrete subclass of **Badge** (called **BadgeNT**, as it mimics the authentification process of a NT workstation).

Example 8. *The Badge class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.Constants;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.Vector;

/**
 * This abstract subclass of DDSO defines a general-purpose entry
 * aimed at identifying users of the DSS and at storing their
 * preferences
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 **/
abstract public class Badge extends DDSO {
    /** preferences: stores the visible managers and their order */
    public ArrayList prefsMgr;
    /** preferences: stores the layout of the JTree when the user logs
     * off */
```

```
    public Vector prefsTreeLayout;

    /**
     * This public constructor is required but is also used by
     * subclasses when defining templates.
     **/
    public Badge() {
        /* super(); */ // implicit call creating an empty DDSO
    }

    /**
     * Constructor. Creates a new Badge. This constructor is never
     * used to create templates.
     * @param name human-readable name of the DDSO
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     **/
    public Badge(String name) throws UnknownHostException {
        super(name, Constants.BADGE);
    }
}
```

Example 9. *The Group class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.Constants;
import ch.unifr.dicodess.core.DUID;
import java.net.UnknownHostException;

/**
 * This subclass of DDSO defines a JavaSpace entry representing a
 * Group DDSO
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 */
public class Group extends DDSO {
    /** Group name */
    public String name;

    /**
     * Required no-arg constructor.  Should not be used directly.
     */
    public Group() {
        /* super(); */ // implicit call
    }

    /**
     * Constructor. Creates a new Group. This constructor should never
     * be used to create Group templates.
     * @param name human-readable name of the group
     * @param description human-readable description of the group
     * @param author the Badge of the author of this group
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     */
    public Group(String name, String description, Badge author)
                                        throws UnknownHostException {
        super(name, description, author, Constants.GROUP);
        this.name = name;
    }

    /**
```

```
     * This private constructor is used internally to create
     * templates. It is needed several times by BadgeServer
     * @param name name of the requested group
     */
    private Group(String name) {
        super(); // call creating an empty DDSO
        this.name = name;
    }

    /**
     * Creates a Group template with the specified information.
     * @param name name of the requested group
     * @return a Group template containing the required information
     */
    public static Group getTemplate(String name) {
        // create a new Group with the specified values
        return new Group(name);
    }

    /**
     * Creates an empty Group template.
     * @return an empty Group template
     */
    public static Group getTemplate() {
        return new Group();
    }
}
```

3.3.2. The Facts Manager's DDSOs

The *Facts Manager* requires only one DDSO called **FactBase** (Example 10). The framework assumes that, in its simplest form, a **FactBase** is nothing more than a pointer to a partial collection of facts (data) about the decision environment. According to the actual data structure used by a specific system, **FactBase** can remain the only class needed by the facts manager, or can become the super-class of a large set of subclasses developed by the DSS implementor. However, this decision is implementation-dependant and should not be influenced by the framework.

Example 10. *The FactsBase class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.Constants;
import ch.unifr.dicodess.core.DUID;
import java.net.UnknownHostException;

/**
 * This subclass of DDSO defines a JavaSpace entry representing a
 * FactBase DDSO
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 **/
public class FactBase extends DDSO {
    /** The data of the facts base are serialized in this String */
    public String snapshot;

    /**
     * Required no-arg constructor. Should not be used directly.
```

```
     **/
    public FactBase() {
        /* super(); */ // implicit call
    }

    /**
     * Constructor. Creates a new FactBase. This constructor should
     * never be used to create FactBase templates.
     * @param name human-readable name of the evaluation
     * @param description human-readable description of the evaluation
     * @param author the Badge of the author of this evaluation
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     */
    public FactBase(String name, String description, Badge author)
                                        throws UnknownHostException {
        super(name, description, author, Constants.FACTS_BASE);
    }
}
```

3.3.3. The Scenario Manager's DDSOs

The *Scenario Manager* is a bit more complicated. As for the facts manager, only one DDSO is absolutely mandatory: the **Scenario** DDSO. The innards of a scenario are then implementation-dependant. In other words, the scenario of a model-driven DSS will not have the same inner structure as the scenario of a data-driven, or of a knowledge-driven DSS. Consequently, the framework provides one core DDSO for the scenario manager: **Scenario**. However, to relieve the DSS implementor of complex programming tasks, we also chose to add in the framework some utility DDSOs, mainly as templates. We chose to implement the model designed by Schroff (1998), well suited for model-driven DSSs. It needs a minimal set of four DDSOs representing **Scenario**, **InfluenceFactor**, **DetailLevel** and **Entry** objects. The DSS implementor is free *(a)* to use them unchanged if they fit her needs, *(b)* to use them as templates that she will adapt to her needs, or *(c)* to discard them and implement her own model. Figure 20 shows the relationships between these specific subclasses of **DDSO**. A **Scenario** is a collection of information about the decision environment for the purpose of supplying missing facts and exchanging or modifying existing facts.

In this model, a **Scenario** is composed of zero, one, or several sub-**Scenario**s and of zero, one, or several **InfluenceFactor**s, which are in turn composed of zero, one, or several **Entry** objects[1]. An **InfluenceFactor** describes one certain influence of the **Scenario** on the decision environment (for example, the influence of a single event causing an anomaly). **Entry** objects contain the actual information of the scenario (*payload*); data is retrieved from these objects to derive a Scenario submodel. Starting from a **Scenario** DDSO, it is easy to find in a recursive way all the **Entry** objects representing the corresponding model. However, an **InfluenceFactor** is also bound to the dimensions of a **DetailLevel**, which is itself implicitly bound to a data domain[2]. To enhance flexibility, both **InfluenceFactor** and **DetailLevel** classes use a

[1] A **Scenario** devoid of influence factor or an **InfluenceFactor** without entry is not really useful, but remains technically valid.

[2] The reader familiar with Schroff's works may be surprised to see that no **DataDomain** class is provided in Figure 20. In fact, the framework considers that the only information stored in a data domain

Vector to store the corresponding dimensions. As this set of dimensions is not supposed to change during the lifecycle of an **InfluenceFactor** and of a specific **DetailLevel**, it is absolutely acceptable.

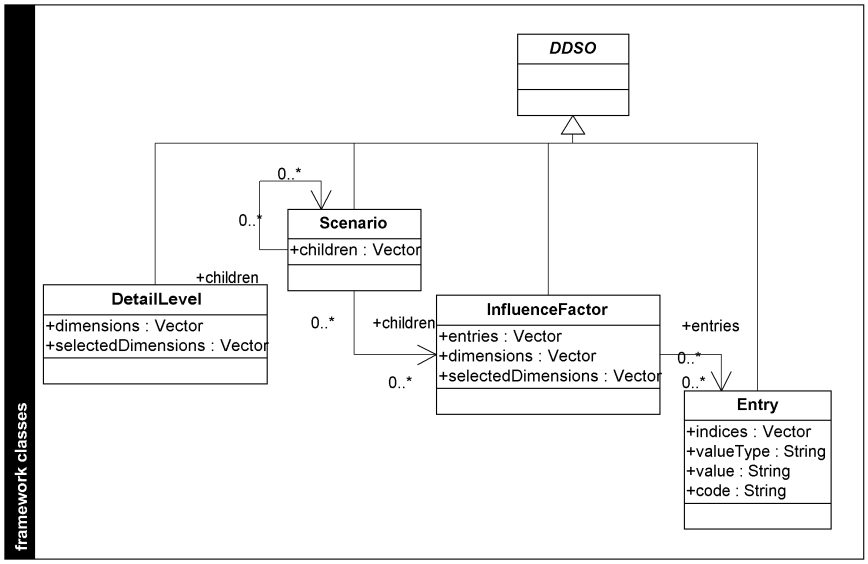

Figure 20. DDSOs of the Scenario Manager

Example 11 shows the **Scenario** class. Note that, according to what was said before, this class has no hard-coded association with the utility classes of Figure 20. For example, the possible associations with instances of **InfluenceFactor** are managed through a generic **Vector** class member (`children`). Thus, this class remains generic enough to be used with other scenario models designed by the DSS implementor.

Example 11. *The Scenario class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.Constants;
import ch.unifr.dicodess.core.DUID;
import java.net.UnknownHostException;
import java.util.Vector;

/**
 * This subclass of DDSO defines a JavaSpace entry representing a
 * Scenario DDSO
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 **/
public class Scenario extends DDSO {
    /** A Scenario knows its children (either InfluenceFactors or
     * Scenarios) */
    public Vector children;

    /**
     * Required no-arg constructor. Should not be used directly.
```

---

is its name, which can be managed by the corresponding representation object and does not require an underlying logic object.

```
     **/
    public Scenario() {
        /* super(); */ // implicit call
    }

    /**
     * Constructor. Creates a new Scenario. This constructor should
     * never be used to create Scenario templates.
     * @param name human-readable name of the scenario
     * @param description human-readable description of the scenario
     * @param author the Badge of the author of this scenario
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     **/
    public Scenario(String name, String description, Badge author)
                                          throws UnknownHostException {
        super(name, description, author, Constants.SCENARIO);
        this.children = new Vector();
    }

    /**
     * Add a child to this scenario
     * @param child DUID of the child InfluenceFactor or child
     *              Scenario
     **/
    public void addChild(DUID child) {
        children.add(child);
    }

    /**
     * Remove a child from this scenario
     * @param child DUID of the child InfluenceFactor of child
     *              Scenario
     **/
    public void removeChild(DUID child) {
        children.remove(child);
    }

    /**
     * Moves the DDSO child one position up in the tree hierarchy.
     * @param child DUID of the child DDSO
     */
    public void moveUp(DUID child) {
        int pos = children.indexOf(child);
        if(pos == -1 || pos == 0) return; // child not found or on top
        children.setElementAt(children.elementAt(pos-1),pos);
        children.setElementAt(child,pos-1);
    }

    /**
     * Moves the DDSO child one position down in the tree hierarchy.
     * @param child DUID of the child DDSO
     */
    public void moveDown(DUID child) {
        int pos = children.indexOf(child);
        // child not found or already at the bottom of the list
        if(pos == -1 || pos == children.size()-1) return;
        children.setElementAt(children.elementAt(pos+1),pos);
        children.setElementAt(child,pos+1);
    }
}
```

3.3.4. The Task Manager's DDSOs

As for the scenario manager, only one DDSO is absolutely mandatory in the *Task Manager*: the **Task** DDSO. Again, the innards of a task are implementation-dependant. As a result, the framework provides only one core DDSO for the task manager: **Task**. However, we also chose to add in the framework some utility DDSOs for this manager. Figure 21 shows the implemented DDSOs. It is a simple structure with only three subclasses of **DDSO** : **Switch**, **Task** and **Option**. A **Task** is used to capture the intention, or request of the user. A **Switch** is one aspect for which the **Task** has to be configured, and the configuration of a **Switch** means choosing an **Option**. Therefore, a **Switch** is composed of one or several **Option**s. If there are several **Option**s for a given **Switch**, one of them is defined as the DEFAULT_OPTION (this information is stored in the **type** field of the corresponding **DUID** object) and the others are defined as regular OPTIONs.

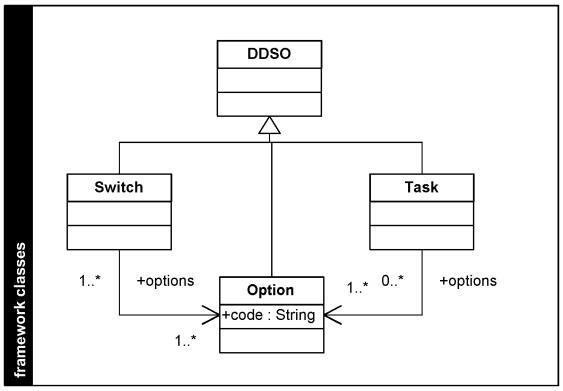

Figure 21. DDSOs of the Task manager

A **Task** is composed of exactly one **Option** from each **Switch** defined. When the user creates a new **Task**, the DSS should assign all the default options to it; of course, the user remains free to change the selected option of a given **Switch** at a later time (if she has the required privileges to do so). **Option** objects contain the actual information of the task; data is retrieved from these objects to derive a task submodel.

Example 12 shows the **Task** class. Note that this class has no hard-coded association with the utility classes of Figure 21. For example, the possible associations with instances of **Option** are managed through a generic **Vector** class member (`children`). Thus, this class remains generic enough to be used with other scenario models designed by DSS implementors.

Example 12. *The Task class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.Constants;
import ch.unifr.dicodess.core.DUID;
import java.net.UnknownHostException;
import java.util.Vector;

/**
```

```
 * This subclass of DDSO defines a JavaSpace entry representing a Task
 * DDSO
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 **/
public class Task extends DDSO {
    /** a Task knows its Options */
    public Vector children;

    /**
     * Required no-arg constructor. Should not be used directly.
     **/
    public Task() {
        /* super(); */ // implicit call
    }

    /**
     * Constructor. Creates a new Task. This constructor should
     * never be used to create Task templates.
     * @param name human-readable name of the task
     * @param description human-readable description of the task
     * @param author the Badge of the author of this task
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     **/
    public Task(String name, String description, Badge author)
                                        throws UnknownHostException {
        super(name, description, author, Constants.TASK);
        this.children = new Vector();
    }

    /**
     * Add an Option to this task
     * @param option DUID of the option
     **/
    public void addOption(DUID option) {
        children.add(option);
    }

    /**
     * Remove an Option from this task
     * @param option DUID of the option
     **/
    public void removeOption(DUID option) {
        children.remove(option);
    }
}
```

3.3.5. The Evaluation Manager's DDSOs

The *Evaluation Manager* is the only object manager using DDSOs created in other managers. Figure 22 shows the logical relationships between an **Evaluation** and the **FactBase**, **Scenario**, **Task** and **ExogenousDecision** classes. An **Evaluation** is composed of exactly one **FactBase** chosen from the facts manager, one **Scenario** chosen from the scenario manager and one **Task** chosen from the task manager. As DDSOs promote reusability, a given **FactBase**, **Scenario** or **Task** can be used in several **Evaluation**s. An **Evaluation** can also include zero, one, or several **ExogenousDecision** objects. Exogenous decisions are either user's input made in order to interactively make up a better alternative, or real *ex-ante* decisions to be respected in a group

decision process when decisions made by others have an effect on the users' own decisions. Example 13 shows the **Evaluation** class.

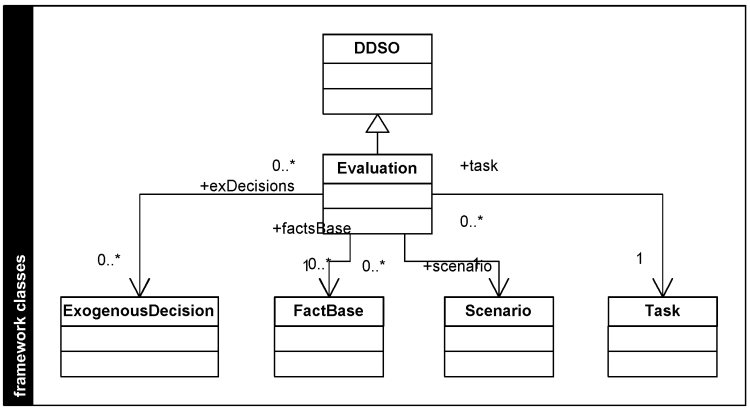

Figure 22. DDSOs of the Evaluation Manager

Example 13. *The Evaluation class*

```
package ch.unifr.dicodess.entry;

import ch.unifr.dicodess.core.Constants;
import ch.unifr.dicodess.core.DUID;

import java.net.UnknownHostException;
import java.util.Vector;

/**
 * This subclass of DDSO defines a JavaSpace entry representing an
 * Evaluation DDSO.
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 **/
public class Evaluation extends DDSO {
    /** an Evaluation knows the IDs of its factBase, Scenario and Task
     */
    public DUID factsBase, scenario, task;
    /** an Evaluation knows its exogenous decisions */
    public Vector exogenousDecisions;

    /**
     * Required no-arg constructor. Should not be used directly.
     **/
    public Evaluation() {
        /* super(); */ // implicit call
    }

    /**
     * Constructor. Creates a new Evaluation. This constructor should
     * never be
     * used to create Evaluation templates.
     * @param name human-readable name of the evaluation
     * @param description human-readable description of the evaluation
     * @param author the Badge of the author of this evaluation
     * @exception UnknownHostException if no IP address for the host
     *                                 could be found.
     **/
```

```
    public Evaluation(String name, String description, Badge author)
                                        throws UnknownHostException {
        super(name, description, author, Constants.EVALUATION);
    }

    /**
     * Add an exogenous decision to this evaluation
     * @param child DUID of the child ExogenousDecision
     **/
    public void addExogenousDecision(DUID child) {
        exogenousDecisions.add(child);
    }

    /**
     * Remove an exogenous decision from this evaluation
     * @param child DUID of the child ExogenousDecision
     **/
    public void removeExogenousDecision(DUID child) {
        exogenousDecisions.remove(child);
    }
}
```

3.3.6. The Report Manager's DDSOs

As with the facts manager, the *Report Manager* requires only one DDSO called **Report**. The framework assumes that, in its simplest form, a **Report** is nothing more than a template used to format a collection of data resulting from an **Evaluation**. According to the actual data structure used by the system, **Report** can remain the only class needed by the report manager, or can become the super-class of a large set of subclasses developed by the DSS implementor. However, this decision is implementation-dependant and cannot be influenced by the framework. The *reporting* is a complex area of decision support systems; it is very difficult to provide trivial template DDSOs for this domain, as we did for the scenario and task's manager DDSOs. A parallel project focusing on reporting has recently been started by the DS group of the University of Fribourg and will be integrated to the framework in a forthcoming phase. See section 4 for more information about this process.

3.4. Object Managers

The DDSOs presented above are partitioned, and for each partition an object manager is defined. An *object manager* supervises its DDSOs and keeps them consistent. Our vision of distributed *object managers* is quite different from the *object managers* originally described by Schroff. Figure 23 presents the classes needed by a typical, distributed *object manager*.

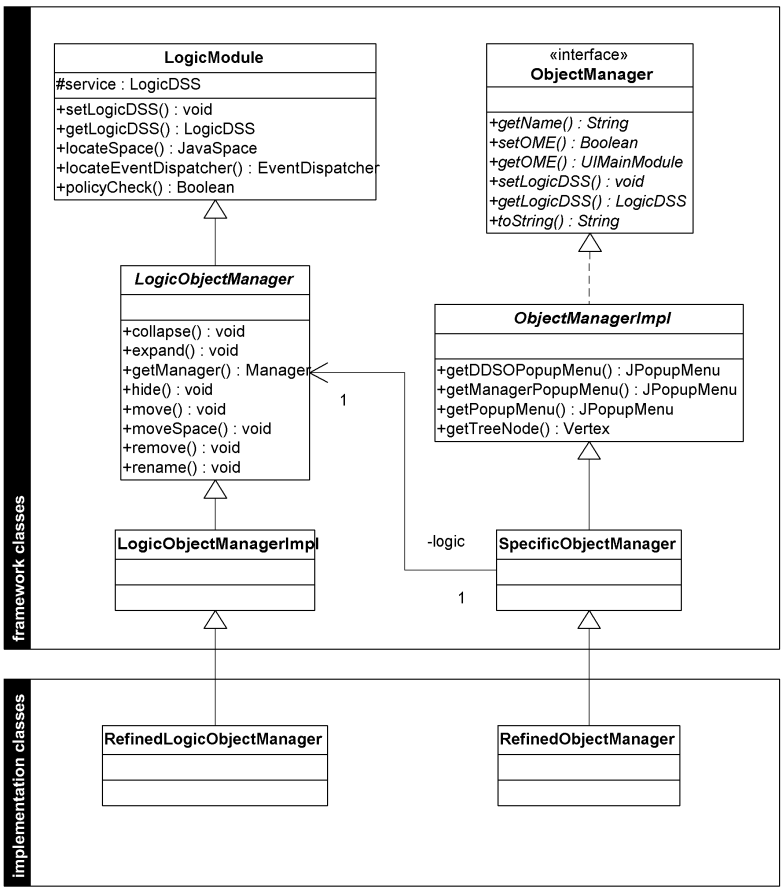

Figure 23. Internal structure of object managers

First and foremost, object managers are designed according to the presentation/logic separation presented in section 2.1.1. The **SpecificObjectManager** class (respectively the **RefinedObjectManager** class) is responsible for the *presentation* tasks of the manager, while the **LogicObjectManager** class (respectively the **LogicObjectManagerImpl** and **RefinedObjectManager** classes) is responsible for the *logic* operations of the manager. The **ObjectManager** interface, and its abstract implementation class (**ObjectManagerImpl**) are representative of the "adaptation by extension/by implementation" principle introduced in section 2.1.2. The **ObjectManager** interface defines generic methods, such as methods to retrieve the name of the manager (a **String** field) and to define with which central classes the manager is associated (`setOME()`/`setLogicDSS()`). These central classes have been introduced in section 2.3. The **ObjectManagerImpl** class provides a partial, abstract implementation of this interface (see Example 14). It augments it with functionality related to the user interface. According to the "adaptation by extension/by implementation" principle, the DSS implementor can either extend the **ObjectManagerImpl** class (as illustrated by Figure 23), or directly implement the **ObjectManager** interface.

Example 14. *The ObjectManagerImpl*

```
package ch.unifr.dicodess.manager;

import ch.unifr.dicodess.core.Constants;
import ch.unifr.dicodess.core.DUID;
import ch.unifr.dicodess.core.LogicDSS;
import ch.unifr.dicodess.core.NetworkAdmin;
import ch.unifr.dicodess.core.OME;
import ch.unifr.dicodess.core.Resources;
import ch.unifr.dicodess.entry.Manager;
import ch.unifr.dicodess.manager.LogicObjectManager;
import ch.unifr.dicodess.manager.ObjectManager;
import ch.unifr.gachet.graph.Vertex;
import ch.unifr.gachet.graph.TreeMutableVertex;
import java.rmi.RemoteException;
import java.util.Hashtable;
import javax.swing.JPopupMenu;
import javax.swing.tree.DefaultMutableTreeNode;

/**
 * This abstract class defines a basic framework to implement object
 * managers. Object managers belong to the "Cooperative DSS Framework"
 * layer of the overal framework.
 * This class defines the basic behavior shared by all object
 * managers. If this basic bahavior does not meet all the requirements
 * of the specific DSS, it can be subclassed and refined.
 * @author  <a href="mailto:alexandre.gachet@unifr.ch">Alexandre
 *          Gachet</a>
 * @version 1.0
 */
public abstract class ObjectManagerImpl implements ObjectManager {
    /**
     * an object manager knows the main module of the system (for
     * example, to get the list of active managers, to open modal
     * dialog boxes, etc.)
     */
    protected OME ui;
    /** the name associated with this object manager */
    protected String name;
    /** this fields are imported from UIManagerStrategy */
    /** popup menu associated with the manager */
    protected JPopupMenu jpmManager;
    /** popup menu associated with the children DDSOs */
    protected JPopupMenu jpmDDSO;
    /** instance of the class managing the external resources (icons,
     * etc.) */
    protected Resources res;
    /**
     * logic counterpart of this graphical manager (presentation/logic
     * separation)
     */
    protected LogicObjectManager logic;
    /**
     * fields used to store the currently selected entry (DDSO or
     * other) and its parent in the tree structure, when calling
     * this.getDDSOPopupMenu(). This information has to be sent back
     * to LogicExplorer
     */
    protected DefaultMutableTreeNode source, parent;

    /**
     * This simple constructor initializes a new manager with the
     * specified name
     * @param name name of the new manager
     */
```

```
public ObjectManagerImpl(String name) {
    this.name = name;
    /* this code is imported from UIManagerStrategy */
    res = Resources.getInstance();
}

/**
 * Getter method. Retrieves the manager popup menu
 * @param attributes contextual attributes giving useful
 *                   information for the dynamic generation of the
 *                   popup menu
 * @return the JPopupMenu object representing the manager popup
 *                        menu
 */
protected abstract JPopupMenu
                       getManagerPopupMenu(Hashtable attributes);

/**
 * Getter method. Retrieves a popup menu associated with the DDSO
 * @param source the currently selected DDSO
 * @param attributes contextual attributes giving useful
 * information for the
 *                   dynamic generation of the popup menu
 * @return the JPopupMenu object representing the DDSO popup menu
 */
protected abstract JPopupMenu
                  getDDSOPopupMenu(DefaultMutableTreeNode source,
                                   Hashtable attributes);

/**
 * Dispatch method. If the selected tree node represents a
 * manager, then return the popup menu of the manager, otherwise
 * return the popup menu of the DDSO. This method assumes that all
 * the DDSOs of this manager need the same popup menu. If this is
 * not the case, this method should be redefined by the subclass.
 * @param dmtn the selected tree node
 * @param attributes contextual attributes giving useful
 *                   information for the dynamic generation of the
 *                   popup menu
 * @return the corresponding JPopupMenu object
 */
public JPopupMenu getPopupMenu(DefaultMutableTreeNode dmtn,
                               Hashtable attributes) {
    TreeMutableVertex vertex =
        (TreeMutableVertex)dmtn.getUserObject();
    DUID id = (DUID)vertex.getUserObject();
    this.source = dmtn;
    this.parent = (DefaultMutableTreeNode)dmtn.getParent();
    if(id.getType().endsWith(Constants.MGR)) // "manager"
        return this.getManagerPopupMenu(attributes);
    else                                     // ddso
        return this.getDDSOPopupMenu(dmtn, attributes);
}

/**
 * This method returns a Vertex object representing the root of
 * this manager subtree. It is used by SwingTreeExplorer to build
 * the complete tree structure.
 * @return the Vertex object representing the root of this manager
 *         subtree
 */
public Vertex getTreeNode() {
    Manager mgr = null;
    try {
```

```
            mgr = logic.getManager(getOME().getID());
        } catch(RemoteException re) {
            ; /* logic is a remote class because it can be accessed by
               * remote Jini services. However, this class
               * <i>always</i> accesses it locally. Thus, this
               * exception <i>cannot</i> happen in this situation */
        }
        if(mgr == null) return null;
        return mgr.origin;
    }

    /**
     * Getter method. Retrieves the name of this manager
     * @return the name of this manager
     */
    public String getName() {
        return name;
    }

    /**
     * Setter method. Specify the main module (presentation class) of
     * the system.
     * @param ome the main module of the system
     */
    public void setOME(OME ome) {
        ui = ome;
    }

    /**
     * Getter method. Retrieves the main module (presentation class)
     * of the system
     * @return the main module of the system
     */
    public OME getOME() {
        return ui;
    }

    /**
     * This setter method specifies the main module (logic) of the
     * system.
     * Defined by UIManager
     * @param service the main module of the system
     */
    public void setLogicDSS(LogicDSS service) {
        logic.setLogicDSS(service);
    }

    /**
     * Getter method. Retrieves the main module (logic class) of the
     * system
     * @return the main module of the system
     */
    public LogicDSS getLogicDSS() {
        return logic.getLogicDSS();
    }

    /**
     * Trivial implementation of Object.toString()
     * @return the name of this UIManager
     **/
    public String toString() {
        return name;
    }
}
```

Basically, the abstract **LogicObjectManager** class (respectively the **LogicObjectManagerImpl** class) manages the operations associated with events triggered by the **ObjectManagerImpl** class (for example, menu selections). As the **LogicObjectManager** is a subclass of **LogicModule**, a central class that has been introduced in Figure 9 and in Figure 12, it has access to many classes of the Jini-based distributed framework and, in particular, to the various JavaSpaces managing the DDSOs. The exact relationships between JavaSpaces and object managers should be defined at design-time; the DSS implementor is free to put all the DDSOs of all the object managers in the same JavaSpace, or to create one or several JavaSpaces for each manager.

The framework provides six versions of the **SpecificObjectManager**, and of the **SpecificLogicObjectManager** classes: one combination for each object manager (*system*, *facts*, *scenario*, *task*, *evaluation* and *report* managers). Figure 24 shows a possible interaction between a user and one of the object managers (**aFactsManager**).

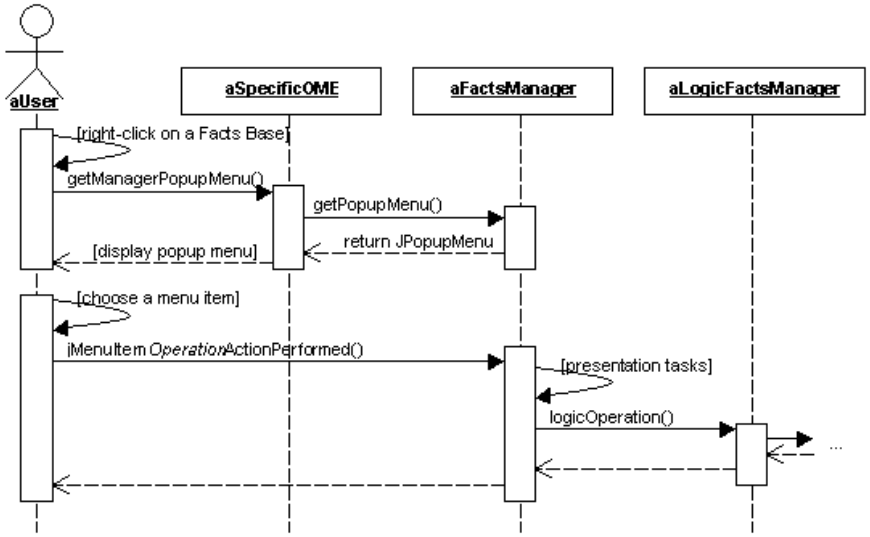

Figure 24. A possible interaction between an user and the Facts Manager

aSpecificOME is an instance of the **SpecificOMEImpl** class. This class will be presented in detail in the next section (3.5). In this figure, when the user right-clicks on a DDSO to display the corresponding popup menu, the request is handled by **aSpecificOME**, which delegates the request to the appropriate object manager (thanks to polymorphism). The object manager returns the corresponding **JPopupMenu**, and **aSpecificOME** displays it to the user. Later, when the user chooses a menu item from the popup menu, the request is directly handed to the appropriate object manager. The object manager takes care of the presentation tasks associated with the request, and delegates the logical tasks to its **LogicObjectManager** (according to the presentation/logic separation). As said before, **LogicObjectManager** is a subclass a **LogicModule** and can therefore use various classes of the framework to achieve its goals. Note that the class **SecurityManager** (not displayed in the figure) provides static methods that check the actions of the DSS user and prevent her from performing illegal operations.

3.5. The Object Manager Environment (OME)

We mentioned several times the *object manager environment* and its corresponding classes (**OME**, **OMEImpl**, **SpecificOME**). In section 2.3, we said that these classes were used to put together the various pieces of the *Jini-based distributed framework*. As showed by Figure 25, these classes are also the glue between the various pieces of the *cooperative DSS framework*. Figure 25 should be combined with Figure 16 and Figure 17 to get the big picture of the overall framework.

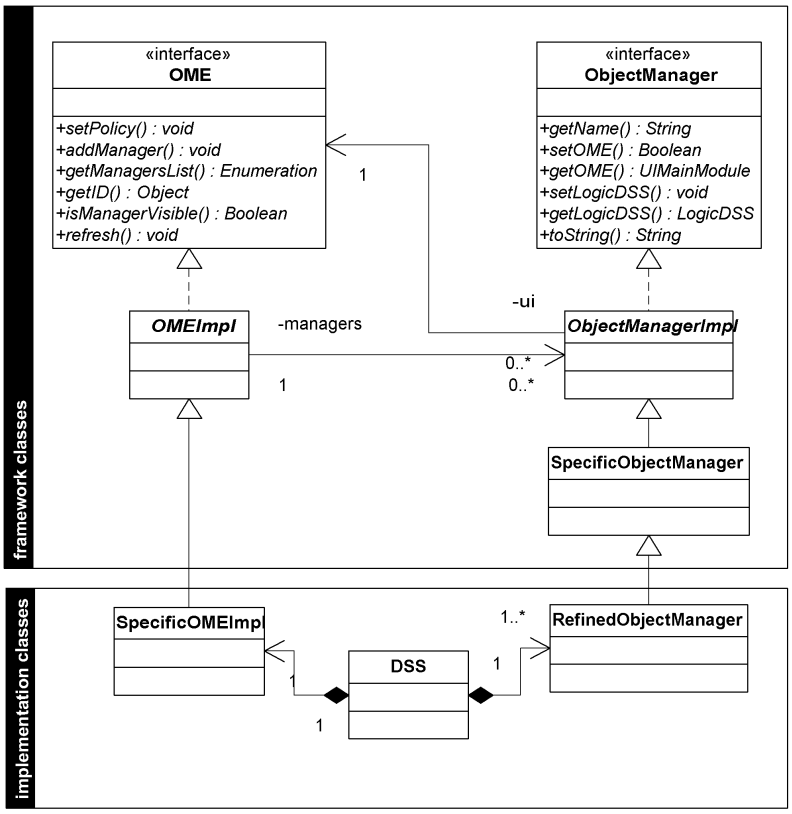

Figure 25. The relationships between OME and object managers

In this figure, **DSS** represents the main class of a specific DSS written by a DSS implementor. On the one hand, this class creates an instance of an object manager environment (**SpecificOMEImpl**). On the other hand, it creates one instance of each object manager (**RefinedObjectManager**) needed by the DSS (the framework provides six different managers, but it is easy to imagine a DSS which only needs a subset of these managers, or which needs some new, specific kind of managers). It is then the responsibility of **OMEImpl** to keep track of all the managers used by the specific DSS (`managers`), and to manage their presentation. The abstract class **OMEImpl** comes precustomized with a tree-like presentation (see Figure 26). Note that the graphical details of this tree-like presentation (for example, the icons used) can be customized for each individual DSS user. The **Resources** class (not displayed in the figure, but briefly introduced in appendice A.2.2) manages the graphical elements of the **JTree** component.

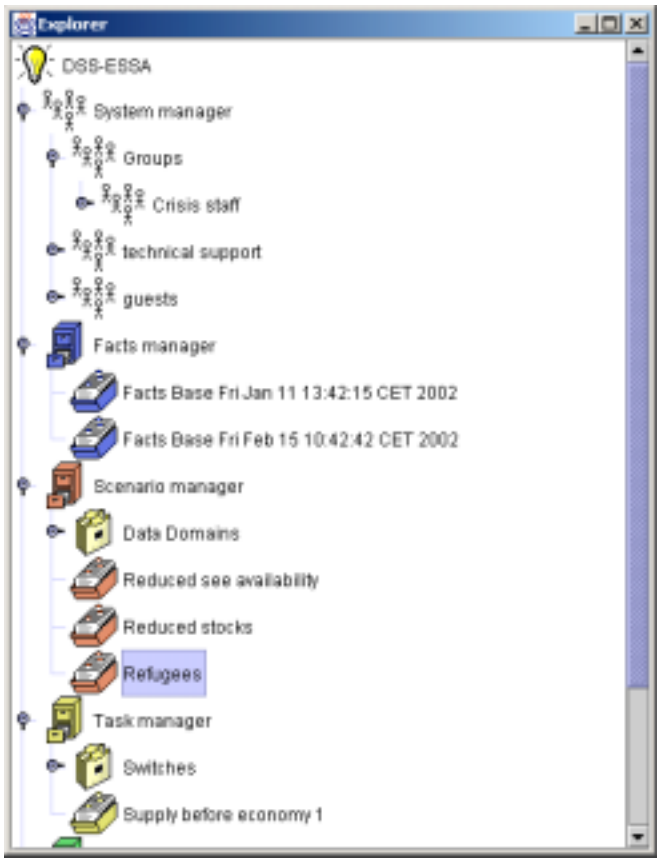

Figure 26. Managers displayed in a tree structure

Representing objects stored in JavaSpaces in a tree structure is not a trivial task, as a JavaSpace is mostly a simple *set* of objects. Structuring these objects is the task of the client of the JavaSpaces. The structure used by the framework is two part: *(1)* the **DDSO**s are stored in the JavaSpaces in an independant way. In other words, there is no direct association between **DDSO**s; only **DUID**s are used to emulate inter-DDSOs association (for example, an **Evaluation** DDSO only knows the **DUID** of its associated **FactBase**, **Scenario** and **Task** DDSOs). *(2)* At the same time, the **Manager** DDSO (introduced in section 2.2.2) manages the tree-like structure of the various DDSOs. There is one such **Manager** DDSO per object manager.

Another issue is due to the fact that the **DDSO**s themselves are not organized in a pure *tree* structure, but partly in a *network* structure. In other words, each **DDSO** can be the child of several fathers (predecessors), and have several children (successors). For example, a **Scenario** is always the child of the Scenario Manager, but it can also be the child of another **Scenario** (recursive relationship), or the child of an **Evaluation** DDSO. To make the matching between the **DDSO**s and a tree structure easier, we developed a specific library called *graph*, which defines data types typical of network structures (*vertices*), but that can be automatically displayed as **DefaultMutableTreeNode** in **JTree** components. A detailed presentation of this library is beyond the scope of this paper. However, the interested reader can read the Javadoc of the graph library.

Each **Manager** DDSO is responsible for its own DDSOs. Consequently, the contents of each **Manager** can be displayed as a subtree in the system. It is the task of the **OMEImpl** class to group these subtrees in the same **JTree** component. Figure 27 shows how a complete DSS tree structure is built when the system is launched.

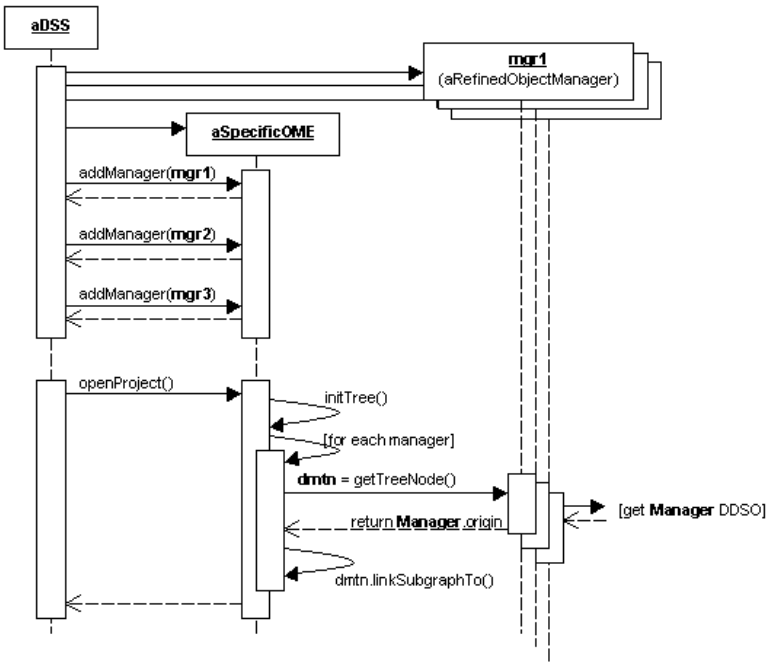

Figure 27. SpecificOME generating a new JTree

aDSS first instanciates the needed object managers. Then, it instanciates an instance of **SpecificOME**, and registers the managers with this new instance (`addManager()`). Later, when the user asks **aDSS** to open a project (`openProject()`), **aSpecificOME** initializes the **JTree** component (`initTree()`) and, for each registered manager, it asks **aRefinedObjectManager** to retrieve the root node (`origin`) from the **Manager** DDSO. Finally, it links the corresponding subgraph to the main **JTree** component (`linkSubgraphTo()`).

4. Conclusion/Future

This paper presented the *construction phase* of the development process of a Software Framework for Developing Distributed Cooperative Decision Support Systems. Starting from a basic architecture composed of a *Jini-based distributed Framework*, a *Cooperative DSS Framework*, and a *Specific Distributed DSS* (see Figure 1), it presented a reference implementation of the framework with comprehensive explanations, code extracts, and UML class and sequence diagrams. It is based on the Jini and JavaSpaces technologies, and it extends Schroff's model of *Decision Support Objects.*

The heart of the framework can be considered as mature; it provides all the abilities to build real *networks of services.* It needs now to be augmented with specialized

services before its transition into real-world DSS projects. As examples of possible specialized services, we can mention:

- a *communication tool*, supporting the exchange of structured and non-structured messages between the various actors of a system, in a synchronous or asynchronous way;
- a powerful, production-ready *reporting tool* able to display static, print-ready reports as well as interactive OLAP interfaces;
- a strong and complete *security model* suited to the specifics of a Jini distributed environment;
- a distributed *solver*, for DSSs based on optimization models.

The open, distributed Jini technology can also be the starting point for other extensions building up the framework, such as better support for mobile devices or multi-agent distributed sub-systems for specific tasks where the agent technology proved to be useful. Figure 28 shows a modified version of Figure 1, integrating three possible levels of extension rounding out the overall framework:

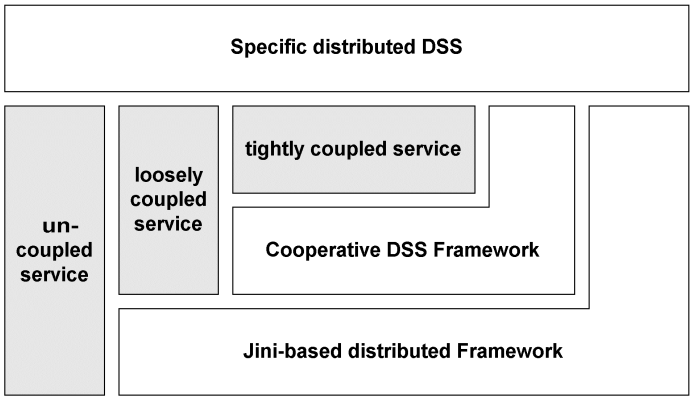

Figure 28. An extended view of the framework

In this figure, *tighly coupled services* are services taking advantage of the cooperative DSS framework and of the Jini-based distributed framework to offer a value-added functionality to the specific distributed DSS. For example, a communication service providing structured means of communication about specific DDSOs between DSS users would belong to this category. *Loosely coupled services* are services that do not take advantage of the cooperative DSS framework, but only of the Jini-based distributed framework. For example, a security service securing the Jini-based networking operations would belong to this category. Finally, *uncoupled services* are services providing value-added functionality to a specific DSS, but without relying on the layers of the framework. A videoconferencing service would belong to this category. The sole requirement that an uncoupled service should meet to be used in parallel to the framework is the support of IP networking.

We reached the end of this description of the *construction phase* of a Software Framework for Developing Distributed Cooperative DSSs. Note that not all the

classes of the framework have been presented in this paper. Specific exception classes, for example, have not been formally introduced. Please refer to the appendice for a complete overview of the existing packages and classes.

Appendice A. Class structure of the framework

A.1. Overview (Dicodess Framework API Documentation)

A.1.1. Packages

ch.unifr.dicodess.core	Central classes and interfaces of the *Jini-based distributed framework* (see Figure 1).
ch.unifr.dicodess.entry	Distributed Decision Support Objects (DDSOs) and other entries stored in JavaSpaces, mostly belonging to the *cooperative DSS framework* (see Figure 1).
ch.unifr.dicodess.event	Classes and interfaces dedicated to the distributed event mechanism provided by the framework (see section 2.1.3).
ch.unifr.dicodess.junit	Unit tests exercising units of production code. These tests verify that the code executes without a hitch, but they do not formally belong to the framework; nevertheless, this package is mentioned in this list.
ch.unifr.dicodess.manager	Classes and interfaces dedicated to the implementation of object managers belonging to the *cooperative DSS framework* (see Figure 1).
ch.unifr.dicodess.module	Classes of general-purpose modules defined by the framework.
ch.unifr.dicodess.templates	Sample implementation classes that can be used as templates by DSS implementors. The substructure of this package matches the structure of the **ch.unifr.dicodess** package. Used together, these classes build an example of a specific DDSS using LPL as knowledge engine.
ch.unifr.dicodess.util	Miscellaneous utility classes.

A.2. The ch.unifr.dicodess.core package

A.2.1. Interface Summary

Constants	This interface defines all the constants used throughout the framework.
LogicDSS	This is the central interface of any DSS developed with this framework. It is a remote object, as it

	could be passed as a reference to custom Jini services (loosely or tighly coupled services). It defines generic operations related to a DSS project and manages lease renewal for **LoginTicket** objects

A.2.2. Class Summary

DUID	Distributed Unique ID. Pair an instance of **InetAddress** with an instance of **UID**. Extends **ApplicationObject** to be compatible with the cloning process of **TreeMutableVertex**
LogicDSSImpl	This is the central interface of any DSS developed with this framework. It is a remote object, as it could be passed as a reference to custom Jini services (loosely or tighly coupled services). It defines generic operations related to a DSS project and manages lease renewal for **LoginTicket** objects
LogicModule	The **LogicModule** class is the super-class of all framework or user-defined classes used as logic modules in a specific DSS (presentation/logic separation). It provides access to JavaSpaces and manages the event dispatcher service used by different instances of the DSS to communicate. Even if it is intended to be subclassed, this class cannot be defined abstract because it is used directly by the **LogicService** class.
LogicService	The **LogicService** abstract class is the super-class of all framework or user-defined classes used as logic Jini services. It provides access to JavaSpaces and manages the event dispatching service used by different instances of the DSS to communicate.
NetworkAdmin	This class allows to programmatically set up a Jini federation. It provides static methods to start HTTP class servers, shared VM, lookup services, transaction manager services, JavaSpaces and other framework-specific services. It also provides methods allowing the system to wait until a specific service is found, as well as methods able to copy and move whole JavaSpaces.
OME	This interface defines a few basic methods that the central "presentation" class of a project must implement.
OMEImpl	This **JFrame** is the central graphical container of the framework. It creates and manages all events and operations related to the tree view of the managers and of the underlying **DDSO**s. It implements the "cooperative DSS framework" layer of the framework. It can be directly included in any Java application (as a **JFrame**) or it can be subclassed and further refined.

Resources	This class manages the generic resources (icons, etc.) stored somewhere on the client disk
SecurityManager	This class checks that the DSS user does not try to commit a forbidden action (e.g. paste a specific DDSO in a wrong container)
ServicesManager	This class is responsible for all the management tasks of Jini services. It uses **ServiceDiscoveryManager**s and **LookupCache**s to efficiently manage the services of the Jini federation. Its implementation follows the singleton design pattern, ensuring that only one instance of this class exists for each Jini federation used.

A.2.3. Exception Summary

DicodessException	General-purpose subclass of **java.lang.Exception** used to wrap generic exceptions that can occur while working with the framework.
NoLookupServiceException	This **Exception** is thrown when the lookup service is required but cannot be found
SpaceNotFoundException	This **Exception** is thrown when a JavaSpace service is required but cannot be found

A.3. The ch.unifr.dicodess.entry package

A.3.1. Class Summary

Badge	This abstract subclass of **DDSO** defines a general-purpose entry aimed at identifying users of the DSS and at storing their preferences
Clipboard	**Clipboard** enables distributed cut/paste operations using JavaSpaces as an intermediary storage place.
DetailLevel	This subclass of **DDSO** defines a JavaSpace entry representing a Detail Level DDSO.
DDSO	This abstract class defines a general-purpose Distributed Decision Support Object. It is subclassed by all classes defining decision objects stored in a JavaSpace. Its super-class is **net.jini.entry.AbstractEntry**, which implements the **Entry** interface, as well as the `equals()`, `toString()`, and `hashCode()` methods.
Entry	This subclass of **DDSO** defines a JavaSpace entry representing an Entry DDSO
Evaluation	This subclass of **DDSO** defines a JavaSpace entry representing an Evaluation DDSO.
FactBase	This subclass of **DDSO** defines a JavaSpace entry representing a Fact Base DDSO

Group	This subclass of **DDSO** defines a JavaSpace entry representing a Group DDSO
InfluenceFactor	This subclass of **DDSO** defines a JavaSpace entry representing an Influence Factor DDSO
LoginTicket	**LoginTicket** is a subclass of **AbstractEntry**. It represents a user currently connected to the system. If a user leaves the system for an unexpected reason, her **LoginTicket** is not renewed and is deleted from the JavaSpaces
Manager	This class defines a special **DDSO** that represents an object manager in a JavaSpace. A manager needs an unique ID and also needs to know its children.
Model	The **Model** class encapsulates the code of the model used in a project, and makes this entry storable in a JavaSpace
Option	This subclass of **DDSO** defines a JavaSpace entry representing an Option DDSO
Project	**Project** is a subclass of **AbstractEntry**. It represents a DSS project in a JavaSpace.
Report	This subclass of **DDSO** defines a JavaSpace entry representing a Report DDSO.
Scenario	This subclass of **DDSO** defines a JavaSpace entry representing a Scenario DDSO
Switch	This subclass of **DDSO** defines a JavaSpace entry representing a Switch DDSO
Task	This subclass of **DDSO** defines a JavaSpace entry representing a Task DDSO

A.4. The ch.unifr.dicodess.event package

A.4.1. Interface Summary

EventDispatcher	The **EventDispatcher** interface defines the methods needed to register, unregister and notify event listeners used to communicate between different instances of the specific DSS.

A.4.2. Class Summary

DistributedEvent	The **DistributedEvent** class defines remote events sent to the event dispatching service by the logic classes of the system. These events are then forwarded to the interested event listeners.
EventDispatcherImpl	This class implements the **EventDispatcher** interface, which defines the methods needed to register, deregister and notify event listeners used to communicate between different instances of the system.

A.4.3. Exception Summary

NoEventDispatcherException	This **Exception** is thrown when the event dispatcher service is required but cannot be found

A.5. The ch.unifr.dicodess.manager package

A.5.1. Interface Summary

ObjectManager	This interface defines basic methods that object managers need to implement. Object Managers belong to the "Cooperative DSS Framework" layer of the overall framework.

A.5.2. Class Summary

DSSManager	The DSS manager manages the overall system (i.e. the others object managers)
SystemManager	The System manager manages the hierarchies of users and user groups.
FactsManager	The Facts manager manages the hierarchies of fact bases.
ScenarioManager	The Scenario manager manages the hierarchies of scenarios.
TaskManager	The Tasks manager manages the hierarchies of tasks.
EvaluationManager	The Evaluation manager manages the hierarchies of evaluations.
ReportManager	The Report manager manages the hierarchies of reports.
LogicDSSManager	The **LogicDSSManager** class is the logic counterpart of the **ObjectManagerImpl** class
LogicSystemManager	The **LogicSystemManager** class is the logic counterpart of the **SystemManager** class
LogicFactsManager	The **LogicFactsManager** class is the logic counterpart of the **FactsManager** class
LogicScenarioManager	The **LogicScenarioManager** class is the logic counterpart of the **ScenarioManager** class
LogicTaskManager	The **LogicTaskManager** class is the logic counterpart of the **TaskManager** class
LogicEvaluationManager	The **LogicEvaluationManager** class is the logic counterpart of the **EvaluationManager** class
LogicReportManager	The **LogicReportManager** class is the logic counterpart of the **ReportManager** class
LogicObjectManager	The **LogicObjectManager** abstract class is the main class of all logic counterparts of the classes representing object managers
ObjectManagerImpl	This abstract class defines a basic framework to

	implement object managers. Object Managers belong to the "Cooperative DSS Framework" layer of the overall framework. This class defines the basic behavior shared by all object managers. If this basic bahavior does not meet all the requirements of the specific DSS, it can be subclassed and refined.

A.6. The ch.unifr.dicodess.module package

A.6.1. Interface Summary

PropertiesPanel	This simple interface must be implemented by all the **JPanel** subclasses that are added to the main **Properties** dialog box.

A.6.2. Class Summary

LogicProperties	The **LogicProperties** class is the logic counterpart of the **Properties** class
Properties	This class generates the main "Properties" dialog box. It can add external **JPanel**s as tabbed panels.

A.7. The ch.unifr.dicodess.template package

A.7.1. Interface Summary

core.SpecificLogicDSS	This interface complements the **LogicDSS** interface with operations specific to a specific DSS. As a subclass of **LogicDSS**, it is a **Remote** interface. This class can be used as a template. It offers generic operations for user management and knowledge engine management. It can be replaced by another interface but, as it is a template class and not a formal class of the framework, it is not intended to be subclassed.

A.7.2. Class Summary

core.DSS	This is a template of a specific DSS main class. It is the main entry point of a specific DSS and it is responsible to set up the the whole DSS environment.
core.SpecificLogicDSSImpl	This class represents the main logic class of a project. It is created by the **SpecificOMEImpl** class and it manages logical operations delegated by **OMEImpl** (related to JavaSpaces and modeling

	concepts)
core.SpecificOMEImpl	This subclass of **OMEImpl** is the central graphical container of the specific DDSS. It creates and manages all events and operations related to the tree view of the managers and of the underlying DDSOs. It implements the "cooperative DSS framework" layer of the framework.
core.ModelDictionary	This class encapsulates all the functionalities necessary to parse and extract information from a LPL model.
entry.BadgeNT	This subclass of **Badge** emulates a Windows NT login system, that is to say with username, password and domain (in this case, a domain is a **Group**)
event.RemoteEventListenerImpl	This class manages all the remote events sent by the "logic" classes of the system, and forwards them to the registered listeners.
manager. RefinedEvaluationManager	The **RefinedEvaluationManager** manages the hierarchies of evaluations in a specific DSS environment.
manager.RefinedFactsManager	The **RefinedFactsManager** manages the hierarchies of fact bases in a specific DSS environment.
manager. RefinedLogicEvaluationManager	The **RefinedLogicEvaluationManager** class is the specific, logic counterpart of the **RefinedEvaluationManager** class.
manager. RefinedLogicFactsManager	The **RefinedLogicFactsManager** class is the specific, logic counterpart of the **RefinedFactsManager** class.
manager. RefinedLogicReportManager	The **RefinedLogicReportManager** class is the specific, logic counterpart of the **RefinedReportManager** class.
manager. RefinedLogicScenarioManager	The **RefinedLogicScenarioManager** class is the specific, logic counterpart of the **RefinedScenarioManager** class.
manager. RefinedLogicSystemManager	The **RefinedLogicSystemManager** is the specific, logic counterpart of the **RefinedSystemManager** class
manager. RefinedLogicTaskManager	The **RefinedLogicTaskManager** class is the specific, logic counterpart of the **RefinedTaskManager** class.
manager. RefinedReportManager	The **RefinedReportManager** manages the hierarchies of reports in a specific DSS environment.
manager. RefinedScenarioManager	The **RefinedScenarioManager** manages the hierarchies of scenarios in a specific DSS environment.
manager. RefinedSystemManager	This class extends the **SystemManager** class and manages the hierarchies of users and user groups in a refined way.
manager.RefinedTaskManager	The **RefinedTaskManager** manages the hierarchies

	of tasks in a specific DSS environment
module.BadgeNTEditor	The **BadgeNTEditor** class creates a GUI panel allowing the user to configure a selected **BadgeNT** DDSO. This **JPanel** is intended to be added as a tabbed panel in the main **Properties** dialog box
module.EntryEditor	The **EntryEditor** class creates a GUI panel allowing the user to configure a selected **Entry** DDSO. This **JPanel** is intended to be added as a tabbed panel in the main **Properties** dialog box
module.LogicBadgeNTEditor	The **LogicBadgeNTEditor** class is the logic counterpart of the **BadgeNTEditor** class
module.LogicEntryEditor	The **LogicEntryEditor** class is the logic counterpart of the **EntryEditor** class
module.LogicNewDataDomain	The **LogicNewDataDomain** class is the logic counterpart of the **NewDataDomain** class
module.LogicNewDetailLevel	The **LogicNewDetailLevel** class is the logic counterpart of the **NewDetailLevel** class
module.LogicLogin	This class implements the functionality of a login service. It provides identification and authentication functionalities. **LogicLogin** has to implement the **LeaseListener** interface because of the lease renewal of the **LoginTicket**s (see last method call of the `login()` method)
module.LogicOptionEditor	The **LogicOptionEditor** class is the logic counterpart of the **OptionEditor** class
module.Login	The **Login** class creates a GUI dialog box allowing the user to log into the system of a specific DSS
module.NewDataDomain	The **NewDataDomain** class creates a GUI dialog box allowing the user to add a new **DataDomain** DDSO in the Scenario Manager
module.NewDetailLevel	The **NewDetailLevel** class creates a GUI dialog box allowing the user to add a new **DetailLevel** DDSO in the Scenario Manager
module.OptionEditor	The **OptionEditor** class creates a GUI dialog box allowing the user to edit a specific **Option** DDSO in the Task Manager

A.8. The ch.unifr.dicodess.util package

A.8.1. Class Summary

WizardJFrame	The **WizardJFrame** class is the graphical front-end of any wizard designed for the framework. It contains a left **JPanel** containing a **JLabel** to display a static image (or a logo), a **JLabel** to display the current state of the wizard and a **JProgressBar**. It also contains a right **JPanel** containing a **JTextArea**

	to display information to the end-user and an optional **JComboBox** to let the end-user select the appropriate action, when needed. The end-user validates its choice by clicking on the button "Continue".

References

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-oriented software architecture: A system of patterns*. John Wiley & Sons Ltd

Fowler, M. 2000. *UML Distilled: Second Edition.* Reading: Addison-Wesley

Gachet, A. 2001*a*. *A framework for developing distributed cooperative decision support systems: inception phase*. Krakow, Poland: Informing Science International Conference

Gachet, A. 2001*b*. *A software framework for developing distributed cooperative decision support systems: elaboration phase*. Fribourg, Switzerland: working paper 01-25

Griffiths, A. 2001. *Introducing JUnit*. Accessed on February 2002 at URL http://www.octopull.demon.co.uk/java/Introducing_JUnit.html

Hättenschwiler, P. 1999. *Neue Konzepte der Entscheidungsunterstützung*. Fribourg, Switzerland: working paper 99-04

Hättenschwiler, P., Moresino, M., and Schroff A. 1998. *Rapid prototyping of decision support system*. Tenerife, Spain: ICSC Symposium

Kumaran, S. I. 2002. *Jini technology: An overview*. Upper Saddle River: Prentice Hall

Li S. 2000. *Professional Jini.* Birmingham: Wrox Press

Schroff, A. 1998. *An approach to user oriented decision support systems*, Druckerei Horn, Bruchsal: Inaugural-Dissertation Nr. 1208