Journal of Mobile Multimedia, Vol. 9, No.3&4 (2014) 273-285

OBJECT SERIALIZATION WHITE FRAMEWORK IN J2ME AND ITS REFACTORING IN BLACK FRAMEWORK

MOHAMMED MAHIEDDINE[1u], MEHDIA AJANA EL-KHADDAR[2v], SALYHA OUKID[1w]

[1]*LRDSI Laboratory, University of Blida, Algeria*

[2]*SIM Laboratory, ENSIAS Rabat, Morocco*

[u]*mo_mahieddine@hotmail.com*

[v]*mehdia.ajana@gmail.com*

[w]*osalyha@yahoo.com*

J2ME mobile agents developers are very soon confronted with the problem of objects transfer over the communication flows (streams) provided by J2ME which, by themselves, take into account only the primitive types or simple String objects of JAVA. Serialization is the process of saving the state of an object on a flow of communication, transferring it in the net, and restoring its equivalent from this flow. Unfortunately J2ME software development cannot be used as standard JAVA because it has many strong restrictions, which greatly restricts its use relatively to JAVA. For example, in the case of mobile agent software development, J2ME does not provide tools for the serialization of objects. This research addresses the lack of a standard development environment for mobile agents under J2ME. In this work we propose a pattern-based white-box application framework, in order to achieve the serialization of complex JAVA Objects and its re-factoring into a black-box application framework.

Key words: Design Pattern, White-Box Framework, Black-Box Framework, Re-Factoring, Re-Engineering, Serialization, J2ME, Ubiquitous, Mobile Agent.

1 Introduction

J2ME [1] is a Java platform designed specifically for limited applications running on small devices such as mobile phones, PDAs, and so on.

The Design technologies of Java applications for mobile devices under J2ME is still in its infancy, but will play a significant role in the development of various applications in the nearest future. These technologies will mature and integrate object-oriented programming, design patterns, and application frameworks. This maturity will induce a considerable rise in the development of increasingly consistent applications in the ubiquitous world [2].

Reusability is one of the means, available to the software developer, to optimize and rationalize the design work. Indeed, reusability reduces the steps of design and testing of reused components, it also simplifies the maintenance and the evolution of applications [3].

The process of Serialization [4] allows storing object-related data in a stream, and recreating the state of the object after its transfer across a network.

A design pattern [5] is a solution to a common problem that repeatedly appears in the software design. This is not a complete design that could be directly converted into code, but a description or a model describing the relationships and interactions between classes and objects, while indicating how to solve a problem in different situations. The design patterns provide a variety of design techniques facilitating the analysis, the design, the re-use, and the maintainability of the framework. Design patterns allow the creation of reusable components and could make frameworks less rigid and less sensitive, by eliminating unexpected dependencies between software components. Therefore design patterns make maintenance tasks easier than in practice, and more stringent than the initial development [3].

A framework is a set of cooperative classes [6] that captures the expertise of a certain application domain in a generic way. Significant applications in this area can be specialized by inheritance (white box framework) or composition (black box framework) from this skeleton by implementing their abstract methods and by adding new application features to them.

The tendency towards development by means of pattern-based frameworks is becoming more and more essential since many years ago [6]. As for the expected benefits from the use of pattern-based frameworks, they are important to reduce production costs as well as maintenance costs, and to encourage reuse by users and developers.

Re-factoring [7] is the process of software change, so as not to affect its external behavior with respect to its users, while improving its design and internal structure. The objective of pattern-based refactoring is to improve the quality of software such as understandability, adaptability, and reusability.

The success of a consistent and important application with J2ME as an agent system, in our view point, depends much on the software architecture construction and structuring as well as the quality of the chosen software design methods.

The standard JAVA is applied in the design of software mobile agents because by itself supports the process of Serialization.

Unfortunately software development in J2ME has many strong restrictions, which cannot be easily used as the standard JAVA or Android [8] which is basically a rewrite and much more powerful alternative of J2ME.

For example, in the case of developing mobile agent software, J2ME does not provide tools for the serialization of objects.

Object serialization proposed in [4] consists of transforming objects into a string or an array of bytes or primitive type objects, which in practice is used to transfer them on a flow of communication across a network connection.

Tom Höfte [6] has affirmed that for the case of complex objects, the serialization process is not simple to achieve.

This research addresses the lack of a standard development environment for mobile agents under J2ME. In this paper, we propose a reusable technique [3] for serializing complex objects in J2ME, carried out by means of a pattern-based framework, while following optimized software design

techniques (maintainability, re-design and re-factoring) owing to the fact that good frameworks are often prone to improvement. Our work is mainly based on the observation that the behavior of serialization is not a natural behavior of objects, as that was the case in the solutions proposed by [2], but rather, that of the streams in which related objects will be transferred for their serialization. This observation has led us to attach the serialization behavior to streams and not to the objects themselves.

The paper is organized into five main sections as follows: in section 2 we present the basic idea we have followed to reach the design of a framework for JAVA object serialization. Section 3 concerns the description of the approach that led us to design of a white-box application framework. In section 4, we describe in detail the re-factoring of the white-box framework, leading to a black-box framework. Finally we give a conclusion and some perspectives in section 5.

2 Basic Design

In this section we supply an overall framework terminology needed to handle serialization as a comprehensive concept, and the essential basic design ideas needed in serialization framework understanding are provided.

2.1 Current Practice and Research

We noticed that it is not a natural behavior for objects to serialize themselves, but they rather serialize the streams in which they will be transferred [4].

The objective of the re-factoring operation is to enhance the *OutputStream* provided by J2ME, which allows a transfer of only primitive data types such as bytes or String objects, and then a more general stream: an *ObjectOutputStream*, which will be used for transferring *Objects*, as shown in Figure 1.

A framework is a reusable design of a family of applications in a given application domain. This design consists of a set of components (objects) that collaborate to carry out a set of responsibilities that form an application. The objects and their collaborations are described by a number of classes (usually abstract) of an object-oriented programming language. A variable aspect of a family of applications in a framework is called a *hot-spot*. An *abstract* method (method without implementation) represents the hot-spot of a framework. Therefore, the process of filling hot-spots consists of using inheritance to define the required functionality according to the interface specified by the abstract methods. A developer customizes the framework to a particular application by sub-classing (white-box framework) and composing (black-box framework) instances of framework classes [6].

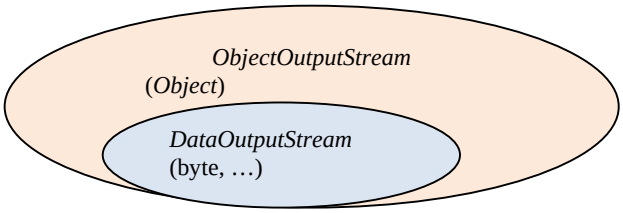

Figure 1 Upgrading of *DataOutputStream* of J2ME in *ObjectOutputStream*.

3 White-Box Application Framework of Serialization

In this section we describe our proposed process for the design of a white-box framework for object serialization in J2ME, and we provide a closer look at the software components of the architecture.

The framework for implementing serialization that we developed in J2ME, will allow an almost identical use of standard JAVA serialization, while providing a basic skeleton that developers can specialize, subsequently, to serialize their objects in J2ME.

The solution proposed in [2], needs to be re-factorized for two distinct purposes. The first one is for applying it to more complex objects, and the second one is for the purpose of maintainability, by means of software reusability through a design pattern-based framework.

3.1 Application Framework Architecture

The solution proposed in [2] consists of attaching to each object class, a special method whose role is to decompose the object into primitive types which can be sent in the J2ME *OutputStream* over the communication network.

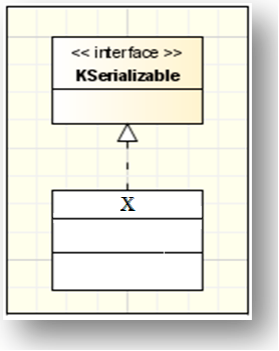

Figure 2 *X* class implementation from *KSerialisable*.

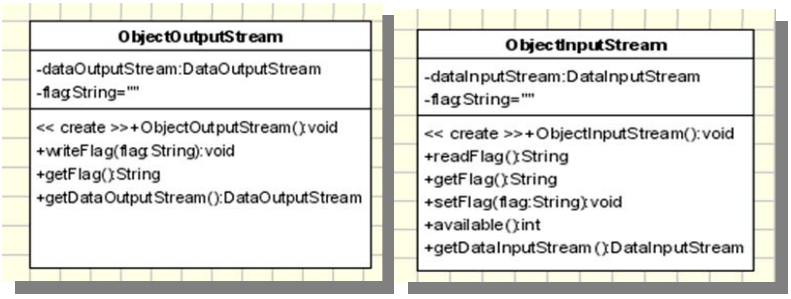

Figure 3 Class diagrams of *ObjectOutputStream* and *ObjectInputStream.*

These methods consist mainly of decomposing an *Object* into an array of *bytes*.

The weakness of this method is that it is not applicable to complex JAVA objects, in other words, objects whose structure includes references to other objects. The solution we propose, in this paper, is

to design the class to be serialized (e.g. that we would like to send objects over the network) as a class that implements the *KSerializable* interface. The latter, which comprises no method, is in fact a simple container of each general object that we would like to serialize (see Figure 2).

The *ObjectOutputStream* and *ObjectInputStream* classes, which we designed as the basic input stream and output stream for object serialization, should comprise the following methods (see Figure 3):

- *WriteObject(KSerializable),* and
- *ReadObject()*

Then, we simply create the *ObjectOutputStream* and *ObjectInputStream c*lasses specific to each class of JAVA Object to serialize, which will be inherited.

The *writeFlag ()*, and *readFlag ()* methods allow you to write or read an indicator of an object in the *DataOutputStream* and *DataInputStream*., meaning that we should create for each class of objects to be serialized its own indicator (as a solution the class name itself was chosen).

For each class of objects to be serialized, we must create its own input stream and output stream, for example in the case of *X* Class, we should create its *XObjectOutputStream* and *XObjectInputStream* respectively as the output and the input stream, which will inherit respectively the *writeObject ()* and *readObject ()* methods.

The streams created, in our case the *XObjectOutputStream* and *XObjectInputStream*, will inherit the *writeFlag ()*, and *readFlag ()* methods of *ObjectOutputStream* and *ObjectInputStream* respectively (see Figure 4).

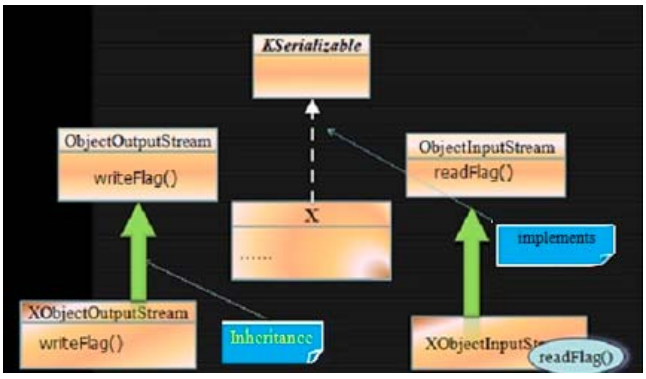

Figure 4 Streams of writing and reading Objects of general *X* class.

The first resulting problem was how to recognize specific types of *KSerializable* objects we receive through the output stream. We solved this problem using refactoring by adding the two methods, *writeFlag ()* and *readFlag ()*, to the classes of streams of each particular serializable object, as *XObjectOutputStream* and *XObjectInputStream* in our example of the class *X*. The role of these

methods is to set a flag, which is used as an indicator of the particular class of the object to be serialized.

The second problem we encountered was the problem of *framework* reusability. The problem is how to incorporate in the *framework*, the code for reading each particular kind of object class without using the case structure which needs to be refitted for each introduction of new class of objects to be serialized. The refactoring we propose is to use a design pattern which uses the weakest coupling possible, which led us to choose, for reasons of past expertise in the manipulation of this pattern, its achievability, and that of the elegance of the produced design, the *Observer* design pattern [2].

The idea is to associate a *Reader* as an observer of the *Receiver* for each class of object to serialize. The role of the *Reader* is, as an *observer*, once it is alerted by a good indicator to recover its object.

The *Receiver* is in fact only an abstraction built around the output stream of the network connection using the *Connector* class of J2ME.

Just now, for each instantiation of the framework for the case of serializing a particular object, such as of the *X* class, we attach a *XReader* and *XWriter* specialized for the recognition of the indicator that it provides, that is to say that it is provided by the *X* class in our example.

4 Black-Box Application Framework for Serialization

Stable frameworks usually result from several design iterations and a lot of work involving structural changes. These changes will involve a series of re-factoring [7]. Good frameworks should always "turn", e.g. evolve from white-box frameworks to black-box frameworks with time. Concerning our framework, is was designed as a white-box framework and we re-built into a black-box framework based on the *Observable*/ *Observer* design pattern.

We noticed that there were some parts that need to be re-factored or re-engineered in our white-box application framework. Therefore we thought introducing greater reusability to the framework by incorporating other possible JAVA interfaces, instead of some classes.

4.1. Framework Architecture

The first thing we could do on our white framework is to transform the abstraction that was not very natural, for example the class *ObjectOutputStream* has the characteristic structure of an indicator. This allows writing objects, while this is not a very natural characteristic for a stream, but it was done by pure goal of genericity.

In fact a general stream, in our case, is an abstraction for writing an object, not for writing an *indicator*, as was the case in our white framework. This led us to make some changes in this first framework. The same comment could have been for the case of the stream of reading.

Our process of re-factoring has led us, for reasons of reusability, to replace the *ObjectOutputStream* class, by the *ObjectOutputStreamable* interface, that will contain the abstract methods *writeObject(KSerializable object)*, and *readObject()*. The role of special streams for each object to serialize is to implement this interface, so they can define its methods according to their needs.

For the same reasons we have, for example, moved the *writeFlag (String flag)*, and *String readFlag ()* methods, to the *FlagOutputStreamable* and *FlagInputStreamable* interfaces, and therefore they have become abstract.

By what follows, we provide the transformation of the inheritance of the white *framework* by the composition required in a black *framework* (see Figure 6).

But in the process of re-factoring, after moving the method that handles the indicator toward the interface, we are faced with the problem of how to encrypt the indicator. To resolve this problem, we have provided a *FlagOutputStream* class, which implements this interface, and includes a reference to the *EncryptedInputOutputStream*. The encryption side will be treated individually with more detail in another paper.

4.2. Testing the Instanciation of the Framework

to is a pattern-based framework for mobile and intelligent agents in J2ME [2]. Being Given that we are constrained to operate only by pseudo mobility in J2ME, we preferred to migrate, in the *to* framework, the messages sent by the agents and not the agents themselves. In *to*, there are three kinds of messages. Since all these messages inherit from the *GMessage* class, this class should be the one which implements the *KSerializable* interface (see Figure 5). So, as explained in the case of any class *X*, we shall construct, for each of these messages, a special stream derived from J2ME *OutputStream* for its serialization, and a second derivative of J2ME *InputStream* for its deserialization, as shown in Figure 7. The problem was to integrate all these classes, with the server, originally built, and that we no more would re-factorize it as it always gives satisfaction. This is due to its architecture based on delegated classes, and the proxy pattern that is the basis for the design of the proxy server.

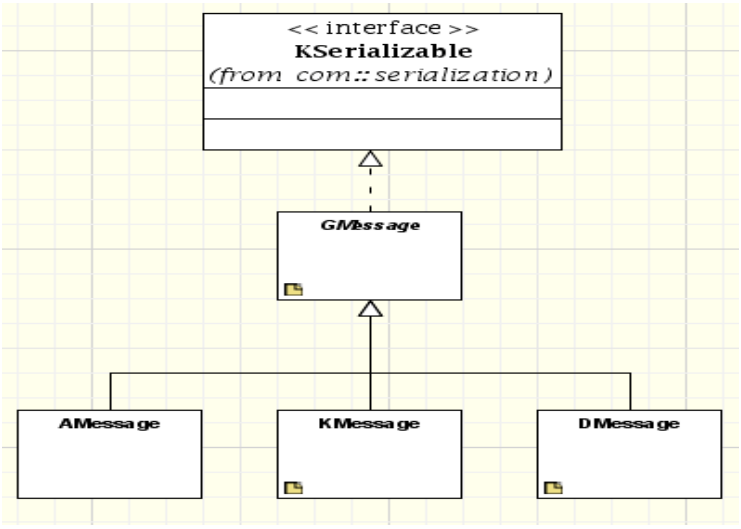

Figure 5 *DMessage* implements *KSerializable* through *GMessage*.

The integration of this part in the *Framework to*, led us to maintain the old *Server* class of *to*, but the problem that arose because of the new integration was how to preserve the architecture of the

framework, without falling into the gear of the *case* structures that should be incorporated for processing each message through its specific streams, and possibly other elements to migrate than other developers could develop later. This problem, as was explained above, was solved using the design pattern *Observer/Observable*. We split the *Client* abstraction, in two distinct abstractions, one modeled by a special delegate class called *Sender* and the other by a class called *Receiver* that will be the *Observer* of the *Client* and the *Server*, and which will be responsible for dealing each with a single direction of the double bond of the network transfer with the *Client*. We then delegate, for each class of messages, a particular object of the *Writer* class (as *AMessageObjectWriter*, *DMessageObjectWriter* and *KMessagaObjectWriter*), which has the role to be notified by the *Sender*, to perform the serialization of the message. (see Figures 6, 7, and 8).

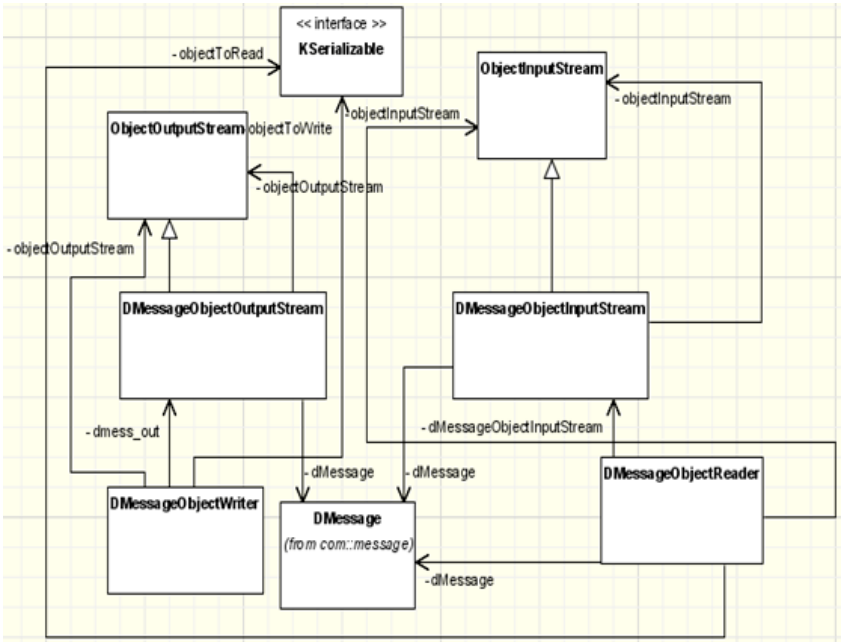

Figure 6 Class diagram showing the serialization of *DMessage*.

Figure 7 Flow of Serialization and De-serialization of the *DMessage* of *to*.

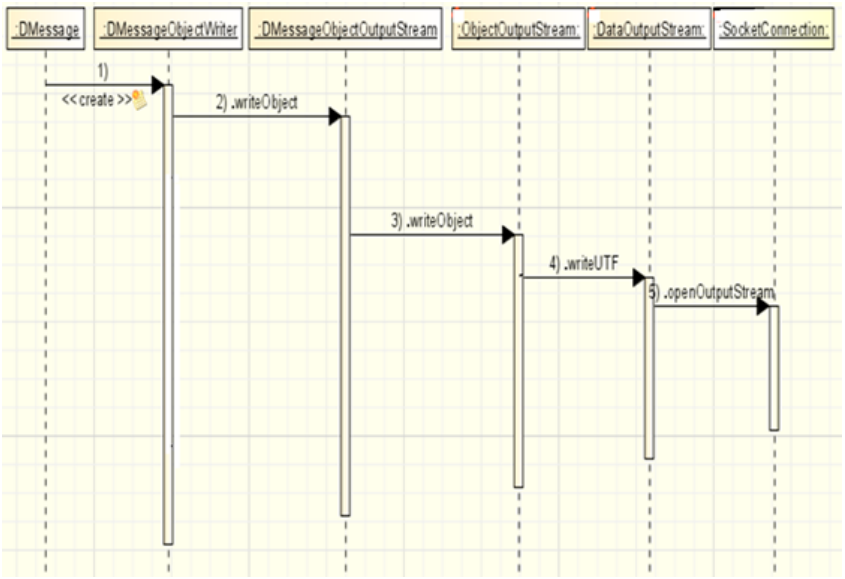

Figure 8 Diagram of sequence showing the dynamic of the serialization of *DMessage*.

When the information transferred to the stream arrives at the *ServerConnexion* (a delegate of the *Server*, having the task to deal with the connection to the *Server*), it will notify the *Receiver*, which by its turn notifies all of its *Observers* (as *AMessageObjectReader*, *DMessageObjectReader* and *KMessagaObjectReader*) (see Figures 9 and 10).

The role of a *Reader* is to perform the de-serialization of an object, and that after it is notified; along with the other *Readers* by the *Sender*, using the indicator (*flag*) attached to each particular class of objects to be transferred over the network.

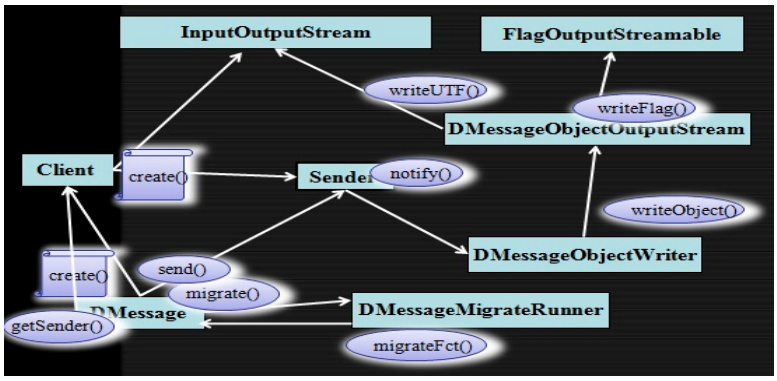

Figure 9 *DMessage Sender* side.

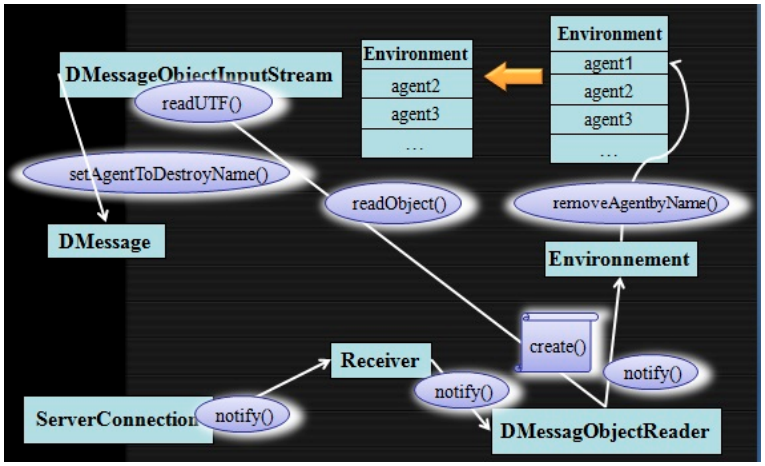

Figure 10 *DMessage Receiver* side.

5 Conclusion

The field of mobile phones is rising in crescendo nowadays. In our viewpoint it is time to start adapting high level design techniques for the field of designing and programming of small-scale models [2].

Our work is part of a pioneering work in the design through "heavy" software design tools in the field of mobile phones, particularly in the design of software agents [2]. It should be noticed that in the literature, the design of software using design patterns, frameworks, and re-factoring for mobile devices in J2ME, is always in an embryonic state. In this work we have initially proposed a pattern-based white framework for object serialization in J2ME.

We then showed how to solve the problem of genericity resulted from the architectural solution that we proposed, using the *Observer* design pattern.

At the end, we detailed the process of refactoring used to upgrade this white framework towards a black framework. The proposed black-box framework of serialization, was used to migrate the *to* agents on mobile devices (see Figures 11, 12, and 13). This later starts to become stable and general, it has all qualities of a pattern-based framework, however, it will always be subject to improvement every time it shows any weakness. The subsequent refactoring and reengineering of the proposed black-box framework could be achieved easily since it was designed using the *Observer* design pattern, and the natural relationship between design patterns and refactoring make the design more reusable and extensible.

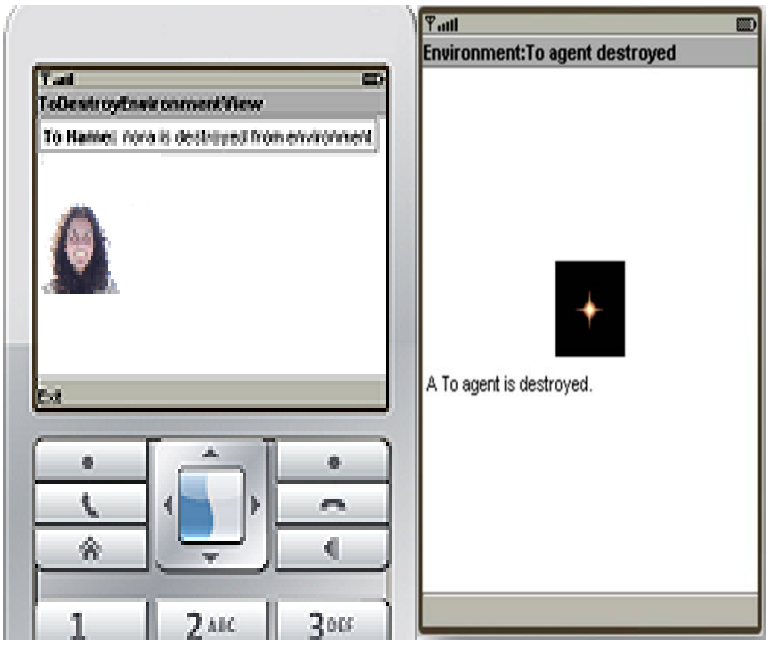

Figure 11 Application of a remote *DMessage* on Mobile Phone Result.

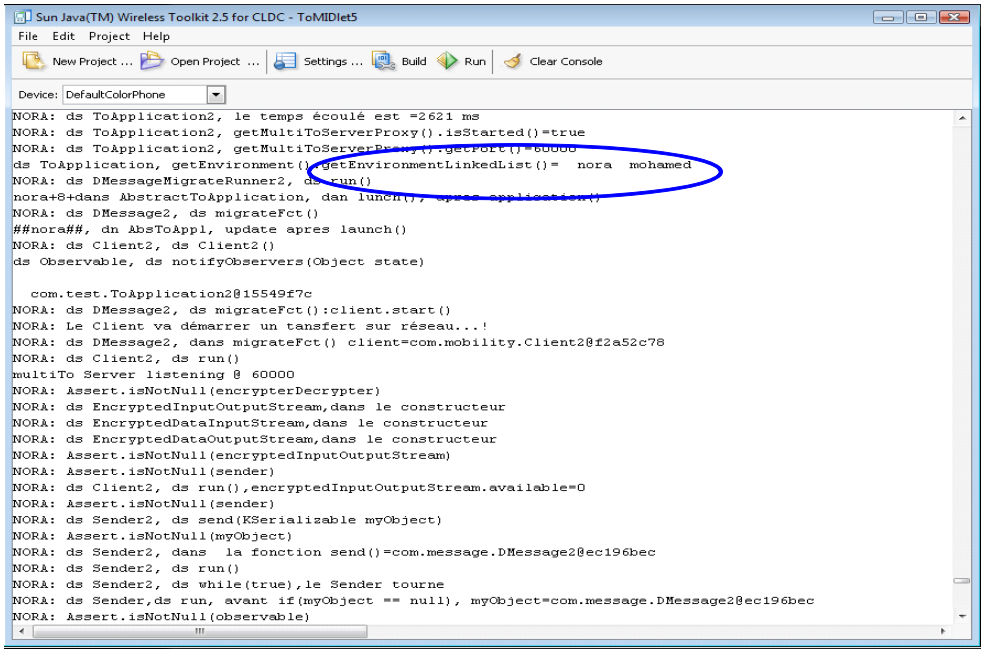

Figure 12 Application of a remote *DMessage* on an agent of the Environment.

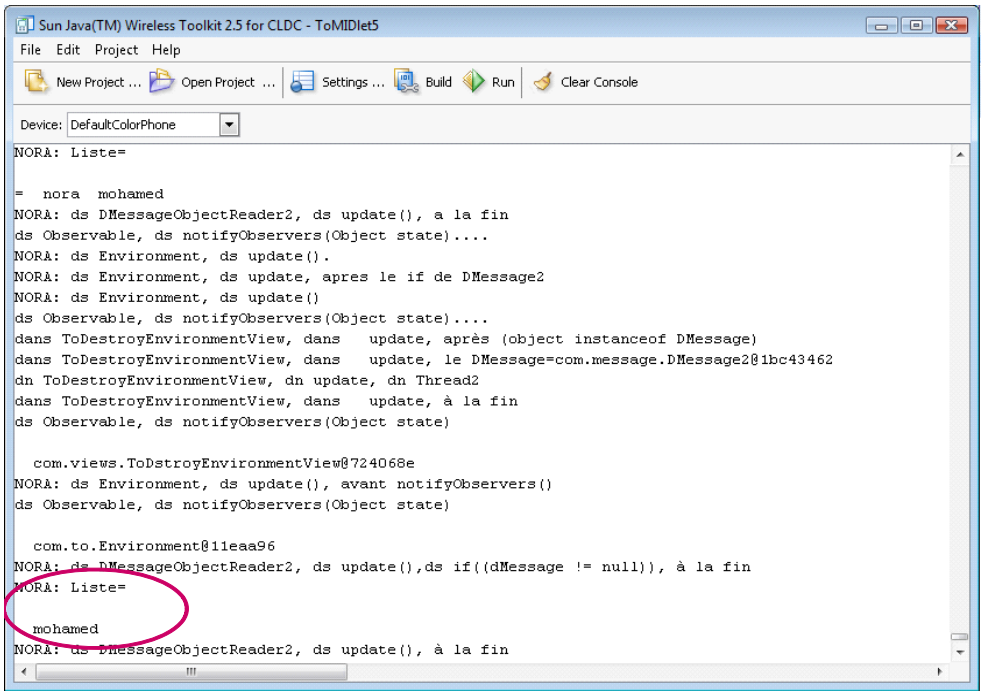

Figure 13 Resulting Environment after the application of a remote *DMessage* on an agent.

References

1. Sun Microsoystems Inc. Java 2 platform, micro edition (j2me), http://java.sun.com/j2me/, 2008.
2. Mazari, R. Study and Design of a pattern-based framework for intelligent and Mobile Agents under J2ME, Magister in System and Knowledge Engineering, Computer Science Department, Faculty of Sciences, Blida University, Algeria, July 2010.
3. Khomh, F. and Gueheneuc Y.-G. Do Design Patterns Impact Software Quality Positively?, CSMR '08, Proceedings of the 12th European Conference on Software Maintenance and Re-engineering, 2008.
4. Höfte, T. Vector and Object Serialization for J2ME-MIDP, http://www.it-eye.nl/weblog/2005/, 2005.
5. Srikanth, J., R Savithri, R. A New Approach for Improving Quality of Web Applications Using Design Patterns, International Journal of Electronics Communication and Computer Engineering, Volume 3, Issue 1, 2012.
6. Kirk Douglas, S. Understanding Object-Oriented Frameworks, Phd thesis, University of Strathclyde, Glasgow. August 2005.
7. Fowler, M. Refactoring: Improving the Design of Existing Code, http://www.refactoring.com, 2008.
8. Meir, R. Professional Android 4 Application Development (Wrox Professional Guides), Amazon, 2012.