# XML AGENT ON SMART CARDS

Sayyaparaju Sunil
*Kanwal Rekhi School of Information Technology,*
*IIT Bombay,*
*India.*
sunil@it.iitb.ac.in

Dr. Deepak B. Phatak
*Kanwal Rekhi School of Information Technology,*
*IIT Bombay,*
*India.*
dbp@it.iitb.ac.in

**Abstract** With the increase of resources and processing power on the Smart Cards, in recent days, their applicability has moved from a domain specific application to a more generic one. Multi-Application Frameworks, designed to host multiple applications on a card, also aim at increasing the interoperability between different vendors and their applications. This paper proposes a new model named *XML Agent,* which significantly enhances the inter-operability of the smart card applications. The XML Agent becomes an interface through which the external world can interact with the on-card applications. The XML Agent passes the commands and data to the appropriate application as per its requirements. This model will also facilitate the programmer in developing off-card applications, with minimal information about the on-card applications.

## 1. Introduction

Smart Cards have a hardware specific firmware operating system, called *Card Operating System (COS),* to manage all the resources on the card. Additionally, there may be a *Runtime Environment* which adds more functionality to the Card Operating System. The applications residing on the card are called *On-Card Applications.* The external applications, called the *Off-Card Applications,* communicate with the on-card applications by sending *Application Protocol Data Units (APDUs).* The Card OS interprets these commands and performs the corresponding predefined operations.

*Multi-Application Frameworks* provides a platform to securely host more than one application on a card, making them multi-purpose. They define how the on-card applications should communicate and share data between each other. A firewall is deployed to protect and isolate applications residing on the same card. All the communication between on-card applications should pass through this firewall. Widely used on-card Multi-Application Frameworks are: Java Card and MULTOS.

Interoperability is an important aspect in Smart Card application development. Ideally, the off-card application on any terminal should be able to interact with an on-card application on any type of card. Even though the standards and on-card frameworks enforce a uniform interface, the problem is only partially solved. We will discuss this problem in more detail in the next section. Our proposed model significantly enhances the interoperability of the applications, however it doesn't solve the problem completely. It also eliminates the need to rebuild the off-card application to support a new type of card.

Abstraction for the developers is one of the most important principles in software engineering. Abstraction hides all the unnecessary details from the developer. It is difficult to define what is necessary and what is not. A vague definition can be: Whatever the developer is not using is unnecessary to him. He should not be bothered with the details of what he is not going to use. Then he can concentrate more on the programming logic.

Our model discussed in this paper provides powerful abstraction to the developers. In our model, the developers need to have very limited information about the on-card application, to develop an off-card application. As a result of this, there can be more number of market players who provide their own customized services and solutions to the consumers.

## 2. Motivation

This section highlights some of the problems faced during the development of an application. Presently, all the problems may not appear to be very significant. But they will become more prominent once the smart card industry expands, and different players come up with their custom applications, terminals and cards.

Generally, the on-card and off-card applications are developed by the same developer. This is because the off-card application is tightly coupled with the on-card application. It will definitely be advantageous if this dependency is eliminated or at least reduced. Then all the application developers can easily

develop different off-card applications based on a single on-card application. For example, based on an on-card application which maintains personal information of the card holder, numerous off-card applications like vehicle license, insurance policy, bank account etc. can be developed.

Applications on different cards may use different data formats and communication sequences to communicate with the off-card application on the terminal. By data format, we mean the data type and the type of encoding used to represent data. For example, the balance in an E-purse application may be an integer or a real number. In addition, if it is floating-point representation then some base(radix) is decided upon and a fixed number of bits is used to represent mantissa and exponent. By communication sequence, we mean the order in which the control commands and data are sent. This is decided upon during the design and implementation of the on-card application. The off-card applications cannot interact with the on-card application unless they know the exact details of the their data formats and communication sequence.

The card has to interact with a terminal whose architecture may be completely different from that of the card. Then the data formats used by the on-card and off-card applications may be different, but still the communication has to take place. For example, one system may be using 2 bytes to represent an integer while another system may be using 4 bytes. Same is the case with big-endian and little-endian representations. Similarly, for strings, different types of encoding like ASCII, UTF-8, UTF-16 can be used. If both the terminal and the card are following the same data formats, then there will not be any problem. If not, the off-card and/or on-card applications have to do proper transformations to get the exact data. This will incur extra overhead in processing.

There should be some mechanism to hide all these problems from the developers. These problems should be handled at a lower level without exposing them to the application developers. Then the developers can concentrate more on the application process logic rather than on the nitty-gritties of the underlying system.

## 3. Related Work

This section introduces some of the earlier work describing efforts made in the direction of integrating the XML with the Smart Card technology. Their objective was to increase the interoperability of the Smart Card applications. They also attempted to eliminate the need to rebuild the whole off-card application to support a new type of card.

## 3.1 Dynamic Card Service Factory

*Open Card Framework*(*OCF*) provides standardized high-level APIs for card programming [Ocfgim98]. Its **CardService** layer implements the logic of the off-card application for a specific Card Operating System. So there can be many possible CardService implementations for the same application. All knowledge about a family of CardServices is encapsulated in a **CardService-Factory**. For an application to support multiple cards, corresponding Card-Service implementations have to be bundled with it. This increases the size and loading time of the application. A bigger disadvantage is that, an off-card application cannot support card types which are not present during its development.

*DynamicCardServiceFactory*(*DCSF*) [DCSF] is a CardServiceFactory implementation. Initially, it downloads the up to date information about all known card types along with the location of CardServices for those cards and stores it in an XML file in the terminal. When a card is inserted into the terminal, using XML messages, it downloads the appropriate CardService implementation for the requested application and serves the card. This model allows the off-card application to support new cards without changing the application logic.

## 3.2 SmartX

This technology was introduced by Gemplus. It provides a solution for writing host-to-card protocols in XML. This allows the APDU protocol to be portable to any language and platform that has a SmartX implementation. It provides a framework where the application process logic is dissociated from the application protocol [Xavier99]. So it is required to implement the application logic only once. This methodology can support cards that are not present during the development of the off-card application.

The application protocol details are represented using an XML-based language called *SmartX Markup Language*(*SML*). The SML documents, called *Dictionary* [SmartX00] contains card profiles which in turn will contain different card processes. Each card process is implemented for a given card profile using the card specific commands. The SmartX engine running on the terminal downloads the appropriate SML document from the dictionary and executes the application process logic. However, this technology is not in use. The development and support was withdrawn by its initiators.

## 4. XML Agent

In the previous section, we saw two models which enhance the interoperability of applications. Both the models run some sort of an engine on the terminals, which can download the up-to-date information about different types of cards. It will be beneficial if the need to dynamically download information can be avoided, without compromising the functionality provided by it. It is a costlier option to keep the terminals connected to internet round the clock. The above two models have serious limitations in situations where the terminal is not connected to internet.

With the increase of resources and processing power on the Smart Card, it is definitely not a bad idea to implement a small engine on the card itself. So our XML Agent is an engine which runs on the card. It doesn't demand any internet connectivity from the terminal. So the terminals can be deployed anywhere to perform all kinds of off-line transactions with the cards.

### 4.1 Basic Model

The XML Agent is a middleware between the off-card applications and the on-card applications. The off-card and the on-card applications communicate only through the XML Agent. In other words, XML Agent becomes an interface for the off-card applications to communicate with the on-card applications. The off-card applications communicate with the XML Agent in XML only. We chose XML because of the wide support available for it on different platforms.

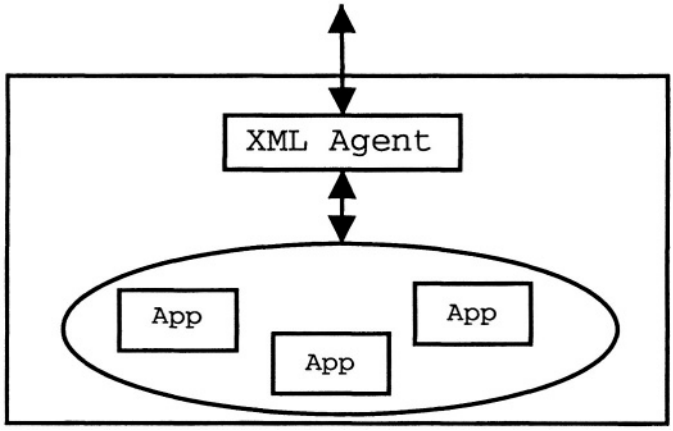


*Figure 1.* XML Agent

The XML Agent transforms the XML documents sent by the off-card applications into a byte stream which is understandable by the on-card applications. The on-card applications respond to the commands passed to them. This response reaches the XML Agent in a byte stream. It will construct an XML document out of the response and will send it to the calling off-card application. So the off-card applications communicate only in XML and the on-card applications communicate only in bytes. Eventually, the XML Agent is transforming the XML document to bytes and vice versa. It is important to note that the XML document can be sent as data in the standard APDU commands. So our model can fit in the existing standards and frameworks.

To transform the XML documents in this manner, the XML Agent should have a knowledge about the on-card applications. First, it should know which applications are installed on the card. Then, for each on-card application, the XML Agent should know the mapping between the fields in the XML document and the corresponding commands and its arguments which should be sent to the on-card application. It should also know the communication sequence for each on-card application. So the on-card application has to describe itself to the XML Agent, or the XML Agent has to be notified explicitly by the installing application. Similarly the information about the mapping rules should be specified by the on-card application or any external application. With this information, the XML Agent can parse the XML document, prepare a byte stream and pass it to the appropriate application.

## 4.2 Advantages

Ideally, all the data entering and leaving the card should be in XML only. In recent days, XML has become the defacto cross platform data representation standard. XML is a simple text-based language and is also extensible. There is wide support for XML parsing on all the platforms. As explained before, the off-card application on the terminal will send and receive only XML documents. It becomes easy for the terminal to communicate in simple characters and strings. The terminal need not worry about the details of the data formats and communication sequence of the on-card application. So the interoperability of the applications will increase significantly.

Since the XML Agent is a middleware between the on-card and off-card applications, the legacy on-card applications need not be changed at all. But the off-card applications should be modified to make them communicate in XML. But there is a workaround to this problem also. That part of the code which has to parse the XML document from the card and extract different fields can be written as wrapper functions to the original off-card application. So the mi-

gration from the existing model to our proposed model can be very smooth.

This model also allows the developers to develop an off-card application, with limited information about the on-card application. Consider a simple example of an on-card personal identification application. The XML document, which is sent out in response to a request may be as follows.

```
<Name: >Sunil< /Name: >
<Age>22< /Age>
<Homeplace>Vedureswaram< /Homeplace>
<State: Exp >Andhra Pradesh< /State: Exp >
<Country>India< /Country>
<Occupation>Student< /Occupation>
<SSN>123-45-6789< /SSN>
```

If suppose an external application wants to know just the name and social security number of a person, it can parse the whole XML document which will be transmitted by the XML Agent. Then it refers to the DTD specification of this data, understands the fields, and extracts the required information from it. In this example the application extracts the information between the tags <Name: >< /Name: > and <SSN>< /SSN>. The developer need not know about the data types of the values or the order in which they are arriving. He simply has to parse the XML document and extract the appropriate fields. This approach has an additional advantage. If the on-card application is changed by adding some more fields, or if some fields are removed (which no off-card application is using) then the already existing off-card applications need not be modified at all.

## 4.3 Disadvantages

It is obvious that the model demands more processing power on the card. It might be very costly to do the XML processing on currently available cards which have limited resources. So we decided not to validate the data against its DTD on the card. All the data which comes to the card is assumed to be valid data. The off-card application has to take care about this. Second disadvantage is that there is lot of movement of unnecessary data between OS, XML Agent and the applications. The verbose nature of XML makes this even more worse. So there should also be more memory on the card. But, we believe that these shortcomings can be overcomed with the increase in resources and processing power on the card.

## 5. Implementation Alternatives

The basic model of XML Agent can be implemented in different ways. In this section we discuss the different approaches and their pros and cons. In the following discussion we will be using the terms 'Runtime Environment' and 'Card OS' interchangeably. The same discussion holds true in both the cases unless explicitly mentioned.

### 5.1 Along with Runtime Environment

The easiest and probably the most secure way is to integrate the XML Agent with the Runtime Environment and/or the OS. All the APDUs first come to the OS to be processed. So the OS (along with XML Agent) can do the job of translating the XML document to byte stream and send it to the appropriate on-card application. Since the OS has complete privileges over all the applications, the XML Agent can access any functions of the on-card application directly. It can trap the responses from the applications, prepare an XML document out of it and send it to the calling application. Since the XML Agent is integrated with the OS, the semantics of the on-card applications will not change. They will not be aware of the presence of the XML Agent. So there is no need to rebuild the existing on-card applications. The problem with this approach is that the developers are at the mercy of the card manufacturer to have an XML Agent. It also steals the opportunity from the developers to build their own Agents. The schematic representation is shown in the figure 2.

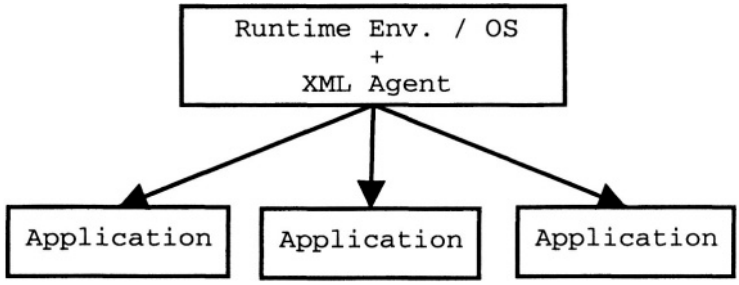


*Figure 2.* XML Agent along with Runtime Environment

### 5.2 Between OS and Applications

In this method the OS passes the XML document to the XML Agent which will generate the byte stream and pass it to the appropriate application. There can be two possible approaches in this scheme. One in which the off-card

applications are aware of the XML agents, but no extra support from OS is needed to achieve this. Other method in which the off-card applications are ignorant of the presence of the XML agents, but some extra support is required from OS. The schematic representation is shown in the figure 3.

The first approach is where the off-card application is aware of the presence of the XML agent. The XML agent can be assigned a fixed/reserved Application Identifier (AID). The off-card applications can directly pass the XML document to XML agent along with the AID of the application to which the final data should be dispatched. The XML agent arranges the data accordingly and dispatches using delegation.

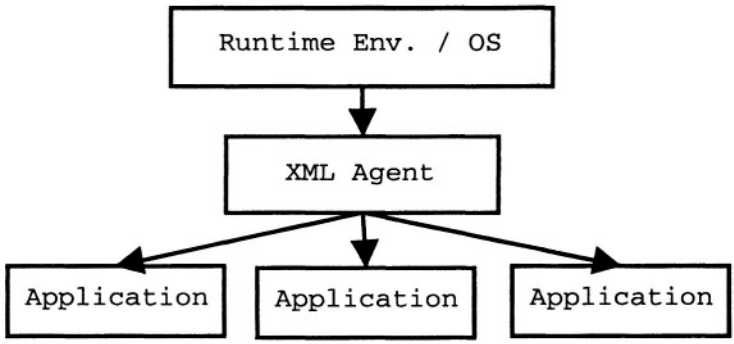


*Figure 3.* XML Agent Between Runtime Environment and Applications

The second approach, where the off-card application is unaware of the presence of XML, requires some support from the OS. The OS should handover the XML document, sent by the off-card application, first to the XML agent. This should be done without the knowledge of the on-card and off-card applications. This is diversion from the ordinary flow of the data where the data is directly sent to the concerned application. The XML agent will then process the data and finally dispatch it to the destination application.

In both the methods the XML Agent is not privileged to access the on-card applications directly because it is just a standard application. So all the on-card application should be made shareable to it. Another alternative is to force that the XML agent should be given special privilege just to access the on-card applications directly. This is more sensible because other on-card applications should not be given a chance to misuse the fact that all applications are accessible by the XML Agent.

## 5.3 As a Standard Application

In this approach, the XML Agent has no special privileges. Here, instead of XML Agent invoking other applications, they will invoke the XML Agent and get their job done. The XML Agent behaves more or less like a library of functions. But the opportunity lies completely in the hands of the developers to implement their own specific XML Agent.

In this model, the applications will directly get the XML document passed by the OS. Instead of processing it, the applications should pass the data to the XML agent. The agent will arrange the data in required format and pass it back to the application. Then the application can proceed in the normal way of processing the commands and/or data. In this approach, there is lot of overhead in movement of data to and fro between the XML agents and the applications. So this is not a preferred way of implementing the XML Agent. The schematic representation is as shown in the figure 4.

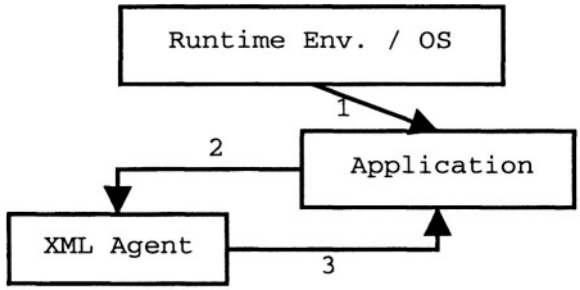


*Figure 4.* XML Agent as a Standard Application

## 6. Implementation Issues

In this section we will discuss some issues during implementation of XML Agent on two of the most popular Multi-Application Frameworks: Java Card [JavaCard] and MULTOS [MULTOS]. We will discuss the features which are supporting or hindering the implementation of XML Agent on them. We are not biased towards any of the frameworks. We view both of them just from a research perspective.

## 6.1 Java Card

Java Card technology adapts the Java platform for use on smart cards. It has a Java Card Runtime Environment(JCRE) which consists of Java Virtual

Machine(JVM), Java Card Framework APIs. As Java Card provides support for Java, we initially assumed that it will be most promising to implement our XML Agent on them. But later we found that the limited support (currently) provided by JVM on the card is not adequate to implement the XML Agent. The following features, which are important for our implementation are not supported [Chen00] on Java Card.

- Characters and Strings
- Dynamic Class Loading
- Object Serialization

XML is a text-based technology but Java Card currently has no support for characters and strings. It only support bytes, integer support is optional. So the characters should be encoded using some encoding scheme like Unicode or UTF. (UTF-8 has the advantage that it is fully compatible with the 7-bit ASCII encoding.) Processing the XML document will incur extra overhead of encoding and decoding. This overhead may be an overkill on the cards with limited processing power and resources.

Dynamic class loading is not supported in Java Cards. This feature allows the loading of classes, which share a common interface, even without any knowledge of their existence at the time of development of the main application. This is absolutely necessary in our model to support post issuance loading of card applets. There is no way in which the XML Agent can interact with the applets installed after it. So the XML Agent should be the last applet to be installed. And also it should hardcode the names of other classes which are already existing in the card. This is undesirable because there will be different applets on different cards. The size of the XML Agent will increase if we make it generic to handle the above situation.

Object Serialization will be helpful if present. In object serialization, rules can be defined to transform the object into a stream of bits. We can define our own rules to serialize an object. These serialization rules can be such that proper XML document is generated when the object is serialized. Then the on-card application has to send an object containing the response data to the XML Agent. The XML Agent can serialize the object, producing the XML document to be sent to the calling off-card application.

## 6.2 MULTOS

After encountering many limitations in Java Cards discussed in the previous section, we diverted our exploration towards MULTOS framework. We

noticed that MULTOS has a very generic architecture to run programs. After analyzing the architecture we found that it can add more functionality to the XML Agent, than we had initially thought about. We have no doubt in saying that, currently MUTLOS does appear more appropriate than Java Card for implementing our idea. Two of the most promising features of MULTOS that help in the implementation are:

- Shell Application
- Delegation Mechanism

The shell application[MShell99] is implicitly selected when a MULTOS card is powered up. The shell application receives all the commands sent to the card, except the commands sent to the MULTOS Security Manager(MSM). The rationale behind this is to support non-ISO interface devices and commands which need to communicate with the cards (which are ISO standard). We can safely exploit this facility provided by MULTOS cards. We can load the XML Agent as a shell application. Since the commands first go to the shell application, we can send actual commands enclosed in XML document to the shell application directly. The XML Agent which is running as a shell application will unwrap the data and invoke the appropriate application and pass it the commands and data using delegation. It can do the processing in reverse direction as well. It can encapsulate the data returned by the application in an XML document and send it back to the off-card application.

The memory architecture and the delegation mechanism [MAPRM97] in MULTOS will be helpful in increasing the efficiency of the XML Agent. In MULTOS, some part of the memory is reserved as "public" area. All the applications can access this data space. During delegation, the delegator application creates a command APDU in public area and activates the delegate application. The delegate application will process the APDU in public area and then return the response in the public area itself. The delegator application will then examine the response set by the delegate application. Since there is no physical copy or movement of data between the applications, the delegation will be faster. In case of XML Agent, if there is no movement of XML documents between XML Agent and the applications, the performance of the system will definitely be better.

## 7. Our Implementation

As we do not have the required resources to modify the Runtime Environment and OS of the card, we couldn't implement the first model where the XML Agent is integrated into the OS of the card. So, as a proof-of-concept we have implemented the second model where the XML Agent sits between

OS and the applications. The implemented model is one where the off-card application is aware of the presence of XML Agent on the card.

As the XML parser of the XML Agent is the heart of the problem, it should be as efficient as possible. We considered different parsing techniques[XML-Parsers] before choosing one. *Tree model* parsers, like DOM parser, are very expensive in this scenario because they will build a complete tree out of the whole XML Document. We did not find the *Push model* parsers, like SAX parsers, very useful because of the cumbersome callback mechanism. After exploring other alternatives, we found *Pull Model* parsers best suited for our requirements. We adopted the API[PullParser], which is actually developed for the Java language, for our implementation in C language. We implemented a simplified parser which can parse basic XML syntax. To decrease the parsing overhead, we are assuming that all strings are ascii encoded. In order to be able to load on currently available commonly used cards we didn't include the support for costly(in terms of processing) features like namespaces, references etc. Finally the size of the XML agent (ALU file for MULTOS) along with the parser came around 5KB. The XML parser alone came to a size of around 3KB.

We developed a very simple application which holds a value. It has only three functions *query( ), incr( )* and *decr( )* to query, increment and decrement the value respectively. The APDU prefixes of these functions are stored in an array datastructure with the XML Agent. As the XML Agent is aware of this application, it can send APDUs accordingly. The XML document is sent as data in the APDU to the XML Agent. The XML command to increment the stored value by some amount is as follows:

```
<xml><app1><incr val="10"/></app1></xml>
```

The latest value is sent in response to this command. The XML document returned by the XML Agent is as follows.

```
<xml><app1><resp success="1" val="110"/></app1></xml>
```

## 8. Conclusion

The traditional approach of developing on-card and off-card portion of an application as a single cohesive bundle does not permit them to be made interoperable easily. There is a great need to have an interoperable framework which would be independent, not only of the differences in cards and terminals, but also of the data structure details of the on-card and the off-card application components.

In this paper, we proposed a novel framework to achieve this interoperability with the use of and XML Agent on the card. The XML Agent which runs

as an engine on the card, transforms the incoming XML document to byte stream and send it to the on-card applications. Similarly, it will prepare an XML document encapsulating the response from the on-card application and send it back to the calling off-card application. This model also provides a powerful abstraction for the developers. It hides many details about the underlying architecture of the card and the terminal. Therefore the developers need to have only a limited information about the on-card application, to develop an off-card application.

The downside of the XML Agent is that it requires more processing power on the card. As the resources and processing power of the cards are increasing continuously, this may not be limitation for the cards in the future. XML Agent is a desirable thing because its advantages outweigh the disadvantages.

## 9. Future Work

In this paper we have not explored security related concerns in depth. There may be some security compromises as a result of our on-card data sharing model. This is because the XML Agent is assumed to have the privilege of accessing any on-card applications directly. We will be exploring these security concerns in our future work.

## References

[Chen00] Zhiqun Chen. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* The Java Series. Addison Wesley.

[DCSF] Wangjammers. Dynamic Card Service Factory Website : http://www.wangjammers.com/smartcards/dcsf.html

[JavaCard] Java Card Technology Homepage : http://java.sun.com/products/javacard/

[MAPRM97] (1997). *MULTOS Application Programmers Reference Manual v1.0.* MULTOS. MAOSCO Limited.

[MDev00] (1997). *MULTOS Developers Guide v1.3.* MULTOS. MAOSCO Limited.

[MShell99] (1999). *Shell Applications, MULTOS Technical Bulletin* MULTOS. MAOSCO Limited.

[MULTOS] MULTOS Homepage: http://www.multos.com

[Ocfgim98] (1998). *General Information Web Document.* OpenCard Framework. OpenCard Consortium.

[PullParser] Common API for XML Pull Parsing : http://www.xmlpull.org/v1/doc/api/org/xmlpull/v1/package-summary.html

[SmartX00] (2000). *SmartX User Guide v1.2.* Think Pulse.

[Xavier99] Xavier Lorphelin. (1999). *Internet and Smart Card Application Deployment.* JSource, USA.

[XMLParsers] A Survey of APIs and Techniques for Processing XML : http://www.xml.com/pub/a/2003/07/09/xmlapis.html