# Java Card Applet validation: methodology and tools

L. du Bousquet[*], J.-L. Lanet[†], H. Martin[†]

## Introduction

Open-cards have introduced a new life cycle for smart card embedded applications. In the case of Java Card, they have raised the problem of embedded object-oriented applet validation.

We have developed a methodology to handle this problem. This methodology takes into account the fact that the only permanent elements of the card are the Java Card Virtual Machine (JCVM) and the Operating System (OS), and that most of security and robustness requirements are based on it. In order to optimize the applet validation process, the JCVM and OS validity are integrated as test hypotheses [1]. Thus, the validation phase is focused on the applet behavior.

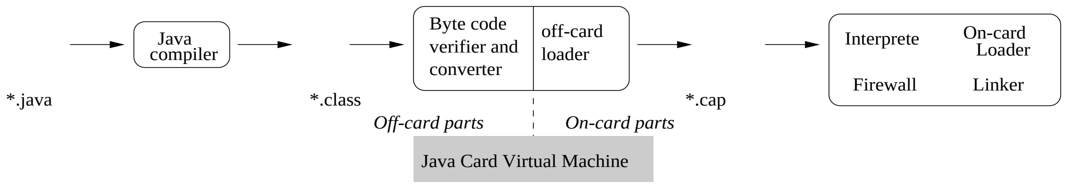

Figure 1: Java Card environment

## Approach and validation environment

Our main goal is then to verify the application conformance to its specification, and to verify the applet integration with other loaded applets within the card. Conformance testing is a black-box functional testing, recommended by ISO for protocol validation [2]. The key points are that there is a behavioral specification and a system implementation exhibiting the specified behaviors. The specification is a prescription of what the system should do. The goal of the testing is to check whether the implemented system satisfies this prescription, using the customer requirements as main purposes for the operational test process.

To apply such a process, we model the applet and its communication with other embedded elements in UML. The resulting model is used as a specification to automatically generate executable test suites, using COTE environment[1]. The COTE environment provides entry for customer main requirements that are used as main test purposes by the tools. The automation of the test generation phase aims at reducing the test development time and improving the quality of the tests (the produced tests conform the specification). Time saving is then both on test development and on test debugging.

[*]LSR-IMAG, BP72, 38402 St Martin d'Hères Cedex, France; lydie.du-bousquet@imag.fr

[†]Gemplus Research Labs, BP 100, 13881 Gémenos,France; {Jean-Louis.Lanet,Hugues.Martin}@gemplus.com

[1]This environment was partially elaborated during the COTE RNTL National project (2000 - 2002). http://www.irisa.fr/cote/

The central part of this environment (see fig. 2) is the UMLAUT/TGV test generator. From the UML specification (composed of a class diagram, state diagrams and an object diagram) and a test purpose (test goal), an abstract test case is generated [4, 3]. This abstract test case is expressed in UML sequence diagrams. Thanks to its precise level of description, it can then be automatically translated into an executable test case for Corba, Java/RMI, or .Net platforms. The upper part of the environment (TOBIAS tool [5]) allows generating massively test purposes from an abstract description we call test schemas. This facility was introduced to test combination of function calls for every normal use and every possible misuse (these misuses include wrong parameters in operation call, unexpected execution context, wrong order in the operation call...). In our methodology, test schemas correspond to the main customer requirements, in accordance with the applet specification.

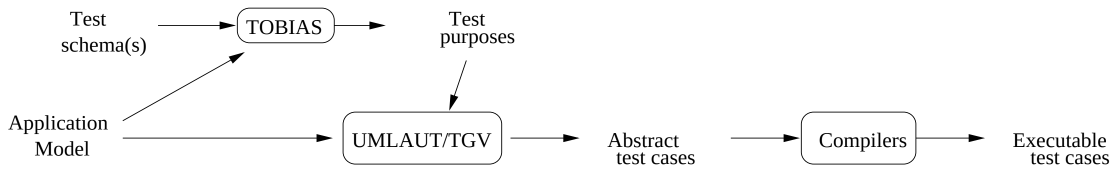

Figure 2: Synopsis of the prototype

## Application

This environment was used to valid a real Java card applet, based on an simpler application already used in the industry, in which we have integrated features that will potentially be used in future Java card applications. The applet consists in a banking application.

Our banking application specification was composed of one class diagram including seven classes: an account manager that creates and deletes accounts, a transfer class that defines spending and saving rules to transfer money from an account to another according to different thresholds and a balance class that allow the customer to have access to its accounts. The class diagram is given in Fig. 3.

To produce tests, UMLAUT/TGV needs the behavioral specification of the applet. It should be expressed with state-machines (associated with each class of the class diagram). UMLAUT/TGV also needs a description of the initial test configuration (expressed with object diagram). Figure 4 provides a sample of a state machine for the account class. Figure 5 provides one initial test configuration, in which: three accounts have been created for one customer in three different banks, no rule exists in the system and all the currency rate are known.

The informal test plan based on the initial use cases requires at least 40 scenarios. For example, one scenario says that “the customer has at least two accounts, execute a transfer and check the correct balances” while another scenario says that “if the customer id is wrong than an error is raised”.

After an abstraction phase, we were able to write only 5 schemas in TOBIAS, to express the 40 scenarios.

This can be express using TOBIAS with first defining the groups (sequences of methods). For this validation plan, we have defined six groups M1...M6. Once the groups are well defined, we apply them to objects defining five schemas.

S1 = transfers.M1 ; balance.M2
S2 = accman.M3 ; accman.M4; balance.M1
S3 = currency.M5 ; currency.M6
S4 = transfers.RegisterSavingRule
S5 = transfers.RegisterSpendingRule

$$with \begin{cases} M1 = \{getAccountsVector(a); getBalances(a) | a \in \{0,1,2,3\}\} \\ M2 = \{transfer(b,b,c) | b \in \{1,2,3,100\}, c \in \{0, 100.0, 1000\}\} \\ M3 = \{BOcreate(d); getBalance(1) | d \in \{1, e, 1000.0\}, e \in \{'\mathrm{CE}', '\mathrm{BNP}', '\mathrm{lsr}'\}\} \\ M4 = \{BOdelete(3)\} \\ M5 = \{setCurrency(f) | f \in \{'\mathrm{USD}', '\mathrm{EUR}', '\mathrm{CHF}'\}\} \\ M6 = \{amountToDisplay(1.0)\} \end{cases}$$

TOBIAS expands theses schemes into more than 1900 test objectives, and UMLAUT/TGV generates the corresponding test cases.

## Evaluation of the method

The most costly part of the validation process is the design of the test cases not their execution. With TOBIAS we have the opportunity to control the generation of interesting test cases from a UML model. One can notice that only five schemas allow to generate all the concrete test cases we needed.

The main difficulty with TOBIAS is that it is the responsibility of the test engineer to find the *good* data. Another drawback is the impossibility to call a method with the result of a previous call. Those points are currently studied.

During the COTE project, we have demonstrated the feasibility of a testing tool chain, even if yet some links or functionalities are missing. The result is promising, especially the TOBIAS [5], a tool to produce massively test purposes (and now test cases). This tool is well adapted to robustness validation of critical applications.

## References

[1] H.Martin. *Une méthodologie de génération automatique de suites de tests pour applets Java-Card*. PhD thesis, Université de Lille 1, 2001.

[2] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[3] C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology (IDPT'02)*, Pasadena, California, USA, June 2002.

[4] Y. Le Traon, T. Jéron, J.-M. Jézéquel, S. Pickin, C. Jard, and A. Le Guennec. System test synthesis from uml models of distributed software. In D. Peled and M. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, Houston,Texas, November 2002. LNCS.

[5] Y. Ledru. The tobias test generator and its adaptation to some ase challenges. In *Workshop on the State of Art in Automated Software Engineering*, June 2002. http://www.isr.uci.edu/events/ASE-Workshop-2002/program.html.

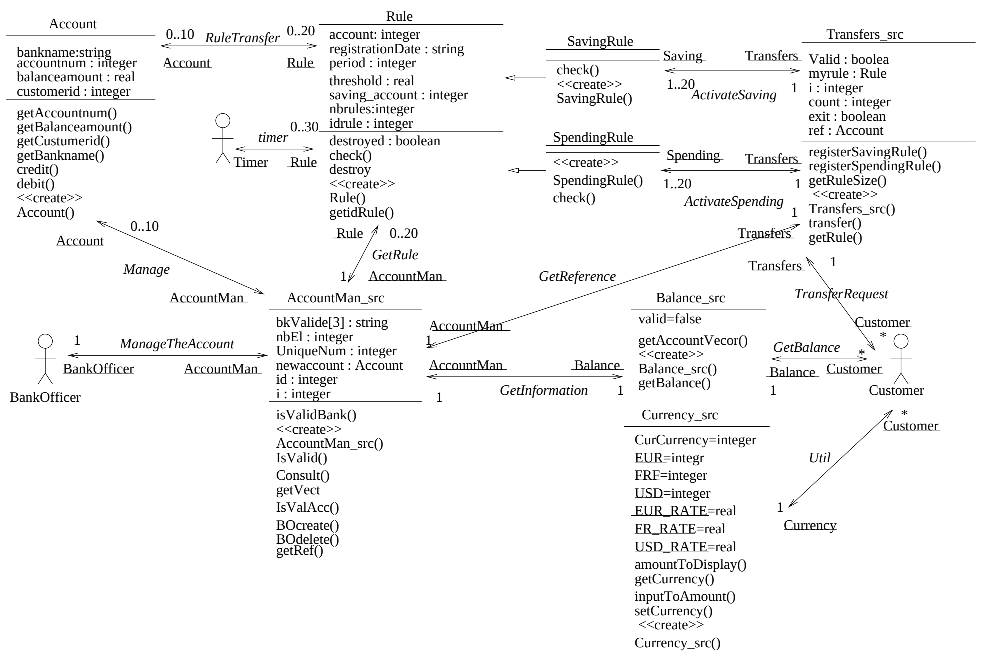

Figure 3: Class diagram of the banking application

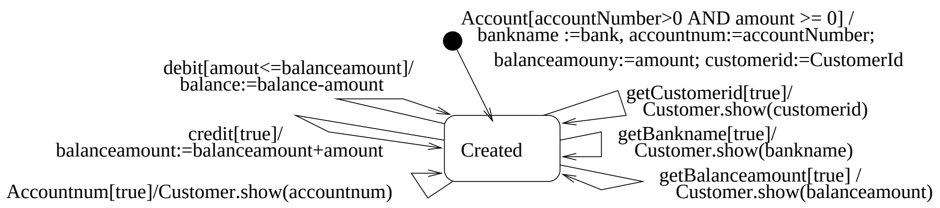
Account[accountNumber>0 AND amount >= 0] /
bankname :=bank, accountnum:=accountNumber;
balanceamouny:=amount; customerid:=CustomerId
debit[amout<=balanceamount]/
balance:=balance-amount
credit[true]/
balanceamount:=balanceamount+amount
Accountnum[true]/Customer.show(accountnum)
Created
getCustomerid[true]/
Customer.show(customerid)
getBankname[true]/
Customer.show(bankname)
getBalanceamount[true] /
Customer.show(balanceamount)

Figure 4: State machine of the Account class

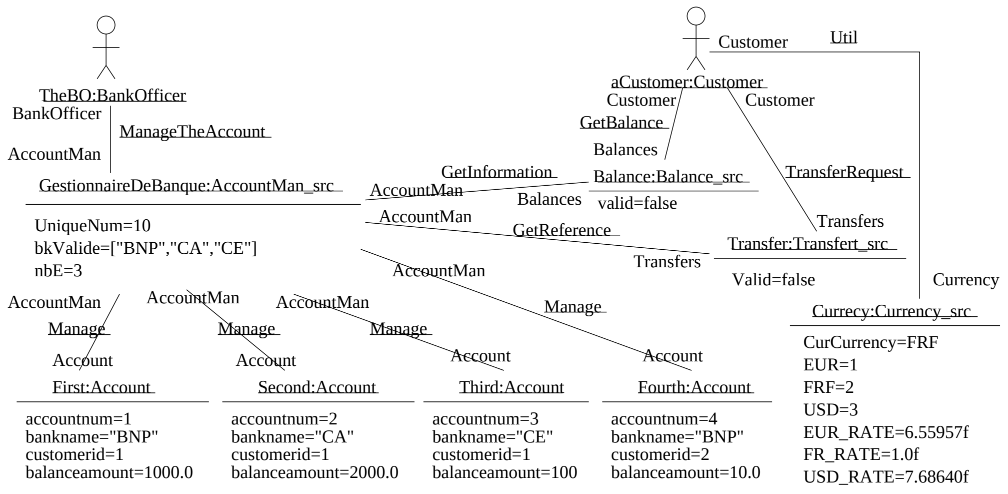

Figure 5: Object diagram of the banking application