UML in Practice

The Art of Modeling Software Systems Demonstrated through Worked Examples and Solutions

Pascal Roques

JOHN WILEY & SONS, LTD

UML in Practice

UML in Practice

The Art of Modeling Software Systems Demonstrated through Worked Examples and Solutions

Pascal Roques

JOHN WILEY & SONS, LTD

Translation from the French language edition of: *UML par la pratique by Pascal Roques*
© 2001 Editions Eyrolles, Paris, France

Translation Copyright © 2004 John Wiley & Sons Ltd, The Atrium, Southern Gate,
Chichester, West Sussex PO19 8SQ, England
Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 33 Park Road, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0-470-84831-6

Translated and Typeset by Cybertechnics Ltd, Sheffield
Printed and bound in Great Britain by Biddles Ltd, Kings Lynn
This book is printed on acid-free paper responsibly manufactured from sustainable forestry
in which at least two trees are planted for each one used for paper production.

"A is a good model of B if satisfactory answers can be given by A to questions predefined on B."

Douglas T. Ross

"The difference between theory and practice is that in theory, there is no difference between theory and practice, but in practice, there is."

Jan van de Sneptscheut

"Since ancient times, man has searched for a language, which is both universal and synthetic. Their search led them to discover images, symbols that – by reducing them to the essential – express the richest and most complex realities. The images, the symbols speak – they have a language."

O.M. Aïvanhov

Contents

Foreword ix
Introduction xi
Acknowledgements xv

PART 1 FUNCTIONAL VIEW 1

1 Case study: automatic teller machine 3
- 1.1 Step 1 – Identifying the actors of the ATM 5
- 1.2 Step 2 – Identifying use cases 8
- 1.3 Step 3 – Creating use case diagrams 10
- 1.4 Step 4 – Textual description of use cases 14
- 1.5 Step 5 – Graphical description of use cases 20
- 1.6 Step 6 – Organising the use cases 26

2 Complementary exercises 37
- 2.1 Step 1 – Business modelling 53
- 2.2 Step 2 – Defining system requirements 57

Appendix A: Glossary & tips 65

PART 2 STATIC VIEW 71

3 Case study: flight booking system 73
- 3.1 Step 1 – Modelling sentences 1 and 2 75
- 3.2 Step 2 – Modelling sentences 6, 7 and 10 77
- 3.3 Step 3 – Modelling sentences 8 and 9 82
- 3.4 Step 4 – Modelling sentences 3, 4 and 5 86
- 3.5 Step 5 – Adding attributes, constraints and qualifiers 89
- 3.6 Step 6 – Using analysis patterns 94
- 3.7 Step 7 – Structuring into packages 98
- 3.8 Step 8 – Generalisation and re-use 105

4 Complementary exercises 113

Appendix B: Glossary & tips 149

PART 3 DYNAMIC VIEW 157

5 Case study: coin-operated pay phone 159
5.1 Step 1 – Identifying the actors and use cases 161
5.2 Step 2 – Realising the system sequence diagram 164
5.3 Step 3 – Representing the dynamic context 166
5.4 Step 4 – In-depth description using a state diagram 168

6 Complementary exercises 185

Apendix C: Glossary & tips 207

PART 4 DESIGN 213

7 Case study: training request 215
7.1 Step 1 – Defining iterations 217
7.2 Step 2 – Defining the system architecture 219
7.3 Step 3 – Defining system operations (iteration 1) 224
7.4 Step 4 – Operation contracts (iteration 1) 225
7.5 Step 5 – Interaction diagrams (iteration 1) 228
7.6 Step 6 – Design class diagrams (iteration 1) 237
7.7 Step 7 – Defining the system operations (iteration 2) 245
7.8 Step 8 – Operation contracts (iteration 2) 247
7.9 Step 9 – Interaction diagrams (iteration 2) 250
7.10 Step 10 – Design class diagrams (iteration 2) 252
7.11 Step 11 – Back to architecture 253
7.12 Step 12 – Transition to Java code 254
7.13 Step 13 – Putting the application into action 262

8 Complementary exercises 267

Appendix D: Glossary & tips 283

Index 293

Foreword

The heart of the challenge in building software-intensive systems is complexity. Computers are universal machines, and as David Eck examined in *The Most Complex Machine*, software "machines" are the most complex things humans build. Compounding this is the many degrees of freedom we as software developers "enjoy" in building systems; there are so many algorithms, components, and ways of connecting things. No wonder we both suffer and delight in the creative opportunities of software development!

The essential weapons against this complexity are abstraction and decomposition. And abstraction is a function of our languages. Our language deeply influences our view. Choosing a spreadsheet language, dance, Java, or the UML to describe a problem and solution shapes how we think about it.

Research indicates that approximately 50% of the cerebral cortex in primates (including us) is involved in vision processing. Communicating and exploring with visual languages plays to a major strength of our brains. Size, spatial relationships, color contrasts, and so on are subconsciously processed with breathtaking speed, conveying much–and fast.

These facts should not be lost sight of in the on-going debates of the value of visual vs. textual programming languages. Textual code (e.g., Java source) is a very low level of abstraction, and does not leverage the natural strength of the human brain as an optimized system for visual analysis. My interest is not just to focus on useful code manipulation-optimizing techniques, such as Extreme Programming or IDEs with refactoring tools, but to find ways to understand and build software using more human-oriented languages, iconic and visual. Make computers understand languages our brains favor, not vice versa.

This is part of the vision of the UML. It isn't just about drawing sketches; it is a vision of tackling complexity and increasing abstraction with better human-oriented languages. Not an easy goal, but worthy. We can't achieve order-of-magnitude improvements in productivity with the current levels of abstraction offered by today's textual computer languages that are not substantively different than FORTRAN-54.

I know that my friend Pascal Roques shares this vision. And Pascal is involved in day-to-day software development. As such, he cares about the practical use of the UML to add value–not simply as an academic toy. Pascal is an expert developer, modeler, and a thoughtful and sensitive teacher. You can see this in his detailed discussion of the trade-offs in different solutions to the problems–it is a great educational contribution to see how a skilled modeler and designer sees alternatives, and makes choices.

By using this excellent book of UML examples and practice, you will gain much in understanding and becoming fluid in the UML. Enjoy!

Craig Larman

Bracebridge, Ontario

Dec 2003

www.craiglarman.com

Introduction

Aims of the book

For several years now, there has been a constant increase in the number of works on UML and object modelling. However, my practical experience of training (more than a thousand or so people trained in OMT, then UML since 1993...) convinced me that there is still another need that is not tended to by the multitude of books available at the moment: a book of marked exercises. In fact, during the seminars that I lead, I am devoting more and more time to discussion sessions with trainees on the compared merits of such or such modelling solution. Furthermore, I am firmly convinced that these interactive discussions on concrete topics have a far more lasting impact for them than the theoretical presentation of the subtleties of UML formalism!

This led me to form an extensive database of exercises, the majority of which have been taken from current or past training courses offered by the company of Valtech. I also drew my inspiration from core books, which have helped me to further my own knowledge of this subject, in particular that of J. Rumbaugh on OMT[1] (one of the first to suggest giving exercises after each introductory chapter on a topic) and the best seller of C. Larman[2] on object-oriented analysis and design.

It is this educational material, based on hours of enriching discussions with trainees from all backgrounds and abilities, that I would like to share with you today. From their questions and suggestions, they compelled me to take into account the most diverse points of view on the shared problem of modelling, as well as improve my argumentation and sometimes to envisage new solutions, to which I had not given any thought at all!

Prerequisites

The reader is assumed to have mastered the core concepts of the object-oriented approach (class, instance, encapsulation, inheritance, polymorphism), having had, for example, practical experience of an object-oriented programming language, such as C++ or Java.

1. *Object-Oriented Modeling and Design,* J. Rumbaugh et al., Prentice Hall, 1991.

2. *Applying UML and Patterns,* C. Larman, Prentice Hall, 1997.

For a complete overview of UML formalism, the reader will be able to refer to comprehensive manuals, such as:

- *The Unified Modeling Language User Guide*, G. Booch, Addison-Wesley, 1999;
- *The Unified Modeling Language Reference Manual*, J. Rumbaugh, Addison-Wesley, 1999;
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd Edition), M. Fowler, K. Scott, Addison-Wesley, 2003.

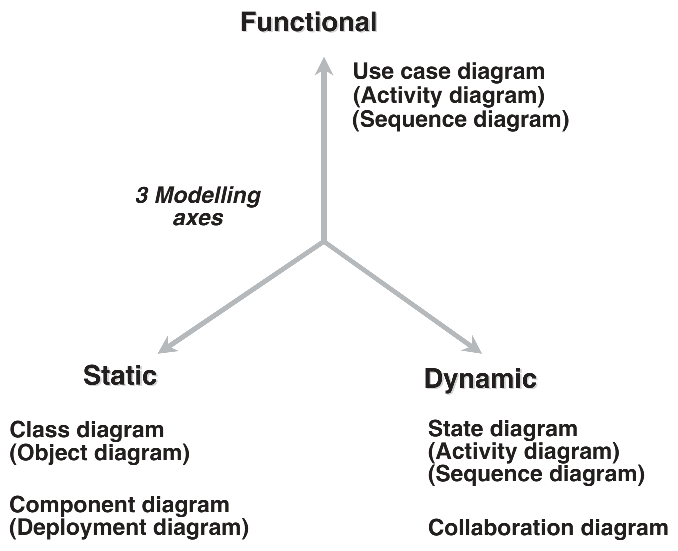
Note that the latest version of the UML Specifications can be found on the OMG web site (www.omg.org, or www.uml.org).

Layout of the book

To avoid confusing matters, the book is divided into parts in accordance with the three views of modelling: functional, static and dynamic, whilst emphasising for each the dominating UML diagram or diagrams (those which are not in parentheses on the next figure).

In order to make a second differentiation – this time between the levels of abstraction – a distinction has been made between:

- an "analysis" level comprising the functional view, as well as a subset of static and dynamic views, excluding the component, deployment and collaboration diagrams;
- a "design" view, which places emphasis on collaboration diagrams and the design detail of class diagrams, and which also introduces component and deployment diagrams.



The first three parts of the book, therefore, each correspond to an analytical view of modelling, and the fourth part to design.

For each part, one main, specific case study acts as the first chapter. Complementary exercises can be found in the subsequent chapter.

A condensed table of contents is given below.

Part 1 Functional view

Chapter 1: Case study: ATM

Chapter 2: Complementary exercises

Appendix A: Glossary & tips

Part 2 Static view

Chapter 3: Case study: flight booking system

Chapter 4: Complementary exercises

Appendix B: Glossary & tips

Part 3 Dynamic view

Chapter 5: Case study: pay phone

Chapter 6: Complementary exercises

Appendix C: Glossary & tips

Part 4 Design

Chapter 7: Case study: training request

Chapter 8: Complementary exercises

Appendix D: Glossary & tips

Typographical conventions

In order to clarify matters somewhat whilst reading this book, the exercises and solutions are given prominence through the use of different character fonts and graphical symbols. Examples of these are given below:

Case study 1 – Problem statement

This case study concerns a simplified system of the automatic teller machine (ATM). The ATM offers the following services:

...

** 1.1 Identify the main actors of the ATM.

Answer 1.1

What are the external entities that interact directly with the ATM?

...

In order to guide the reader a little more, the level of difficulty of the questions is evaluated by assigning it between one and four stars:

*	: easy question,
**	: question of medium difficulty,
***	: fairly difficult question that involves some advanced concepts of UML,
****	: difficult question that requires complex argumentation.

Occasionally, in order to break up the monotony of the text, I have also used the following symbol to set apart a comment concerning a question of advanced level:

Graphical representations of an actor

The standard graphical representation of the actor in UML is the icon called *stick man*, with the name of the actor below the drawing. It is also possible to show an actor as a class rectangle, with the <<`actor`>> keyword. A third representation (halfway between the first two) is also possible, as indicated below:

...

Acknowledgements

This book would not have been able to see the light of day without agreement from the management of Valtech, who allowed me to utilise the material accumulated in the various training courses on UML which I have presented.

I am therefore eager to give special thanks to all those who have participated over the years in developing UML Valtech course support, such as Pierre Chouvalidzé, Thibault Cuvillier, Michel Ezran, Patrick Le Go, Franck Vallée, Philippe Riaux, Philippe Dubosq, Yann Le Tanou, Françoise Caron, Christophe Addinquy, etc., without forgetting our American colleagues, in particular, Craig Larman, Ken Howard and Chris Jones.

I would also like to thank all those whose discussions, comments and suggestions led me to improve my argumentation. First and foremost, I think of my numerous trainees, as well as my correspondents during consultancy work on the introduction of UML in various projects.

Thanks also to Éric Sulpice of Éditions Eyrolles for expressing renewed confidence, and especially for knowing how to motivate me by suggesting that I write this book of marked exercises.

Finally, a big thank you to Sylvie, who supported me for this English edition by her loving encouragements.

Part 1

Functional view

1

Case study: automatic teller machine

Aims of the chapter

By means of the first case study, this chapter will allow us to illustrate the main difficulties step by step, which are connected to implementing the technique of use cases.

Once we have identified the actors that interact with the system, we will develop our first UML model at a system level, in order to be able to establish precisely the boundaries of the system.

We will then learn how to identify use cases, and how to construct use case diagrams linking actors and use cases. Then we will see how to specify the functional view by explaining in detail the different ways in which actors can use the system. For this goal, we will learn to write textual descriptions as well as to draw complementary UML diagrams (such as sequence or activity diagrams).

Elements involved

- Actor
- Static context diagram
- Use case
- Use case diagram
- Primary actor, secondary actor
- Textual description of a use case
- Scenario, sequence
- System sequence diagram
- Activity diagram

- Inclusion, extension and generalisation of use cases
- Packaging use cases.

Case study 1 – Problem statement

This case study concerns a simplified system of the automatic teller machine (ATM). The ATM offers the following services:

1. Distribution of money to every holder of a smartcard via a card reader and a cash dispenser.
2. Consultation of account balance, cash and cheque deposit facilities for bank customers who hold a smartcard from their bank.

Do not forget either that:

3. All transactions are made secure.
4. It is sometimes necessary to refill the dispenser, etc.

From these four sentences, we will work through the following activities:

- Identify the actors,
- Identify the use cases,
- Construct a use case diagram,
- Write a textual description of the use cases,
- Complete the descriptions with dynamic diagrams,
- Organise and structure the use cases.

Watch out: the preceding problem statement is deliberately incomplete and imprecise, just as it is in real projects!

Note also that the problem and its solution are based on French banking systems and the use of smartcards: the system you actually use in your country may be significantly different! It is not very important. What is important is the way of thinking to solve this functional problem as well as the UML concepts and diagrams that we use.

1.1 Step 1 – Identifying the actors of the ATM

First, we will identify the actors of the ATM system.

An actor is a construct employed in use cases that define a role that a user or any other system plays when interacting with the system under consideration. It is a type of entity that interacts, but which is itself external to the subject. Actors may represent human users, external hardware, or other subjects. An actor does not necessarily represent a specific physical entity. For instance, a single physical entity may play the role of several different actors and, conversely, a given actor may be played by multiple physical entities.[3]

** 1.1 Identify the main actors of the ATM.

Answer 1.1

What are the external entities that interact directly with the ATM?

Let's look at each of the sentences of the exposition in turn.

Sentence 1 allows us to identify an obvious initial actor straight away: every "holder of a smartcard". He or she will be able to use the ATM to withdraw money using his or her smartcard.

However, be careful: the card reader and cash dispenser constitute part of the ATM. They can therefore not be considered as actors! You can note down that the identification of actors requires the boundary between the system being studied and its environment to be set out exactly. If we restrict the study to the control/command system of physical elements of the ATM, the card reader and cash dispenser then become actors.

Another trap: is the smartcard itself an actor? The card is certainly external to the ATM, and it interacts with it... Yet, we do not recommend that you list it as an actor, as we are putting into practice the following principle: eliminate "physical" actors as much as possible to the advantage of "logical" actors. The actor is the who or what that benefits from using the system. It is the card holder who withdraws money to spend it, not the card itself!

Sentence 2 identifies additional services that are only offered to bank customers who hold a smartcard from this bank. This is therefore a different profile from the previous one, which we will realise by a second actor called *Bank customer*.

Sentence 3 encourages us to take into account the fact that all transactions are made secure. But who makes them secure? There are therefore other external entities, which play the role of authorisation system and with which the ATM

3. From the OMG document: "Unified Modeling Language: Superstructure (version 2.0)".

communicates directly. An interview with the domain expert[4] is necessary to allow us to identify two different actors:

- the Visa authorisation system (VISA AS) for withdrawal transactions carried out using a Visa smartcard (we restrict the ATM to Visa smartcards for reasons of simplification);
- the information system of the bank (Bank IS) to authorise all transactions carried out by a customer using his or her bank smartcard, but also to access the account balance.

Finally, sentence 4 reminds us that an ATM also requires maintenance work, such as refilling the dispenser with bank notes, retrieving cards that have been swallowed, etc. These maintenance tasks are carried out by a new actor, which – to simplify matters – we will call *Maintenance operator*.

Graphical representations of an actor

The standard graphical representation of the actor in UML is the icon called *stick man* with the name of the actor below the drawing. It is also possible to show an actor as a class rectangle with the <<`actor`>> keyword. A third representation (halfway between the first two) is also possible, as indicated below.

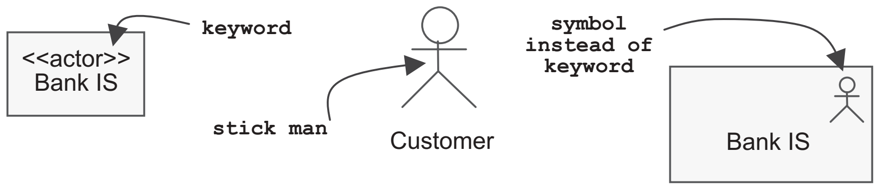


Figure 1.1 Possible graphical representations of an actor

A good piece of advice consists in using the graphical form of the *stick man* for human actors and that of the first rectangular representation for connected systems.

4. Remember that the domain refers to French banking systems, which may explain differences with your own knowledge and experience.

Rather than simply depicting the list of actors as in the previous figure, which does not provide any additional information with regard to a textual list, we can draw a diagram that we will call *static context diagram*. To do this, simply use a class diagram in which each actor is linked to a central class representing the system by an association, which enables the number of instances of actors connected to the system at a given time to be specified.

Even though this is not a traditional UML diagram, we have found this kind of "context diagram" very useful in our practical experience.

** 1.2 Map out the static context diagram of the ATM.

Answer 1.2

The ATM is fundamentally a single user system: at any moment, there is only one instance of each actor (at the most) connected to the system.

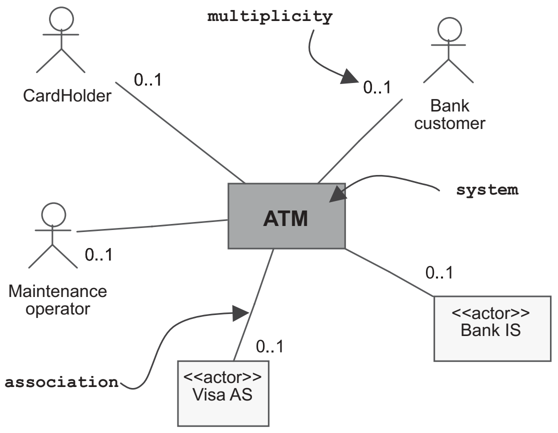


Figure 1.2 Static context diagram of the ATM

We should really add a graphical note to indicate that the human actors, *Bank customer* and *CardHolder* are, furthermore, mutually exclusive, which is not implicit according to the multiplicities of the associations.

Another solution, which is a little more developed, consists in considering *Bank customer* as a specialisation of *CardHolder*, as illustrated in the following figure. The aforementioned problem of exclusivity is therefore solved by adding an extra detail to the diagram.

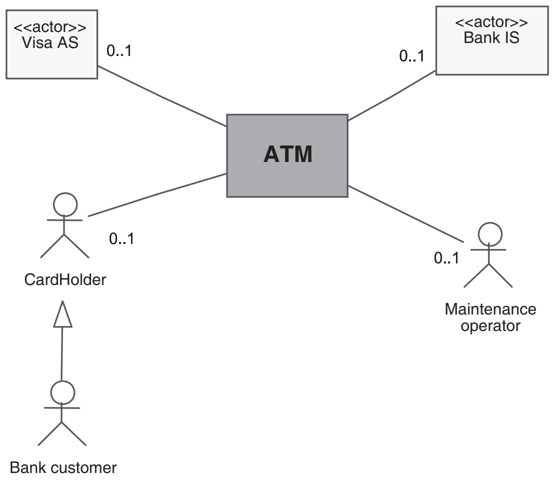


Figure 1.3 A more developed version of the static context diagram of the ATM

1.2 Step 2 – Identifying use cases

We are now going to identify the use cases.

A *use case* represents the specification of a sequence of actions, including variants, that a system can perform, interacting with actors of the system.[5]

A use case models a service offered by the system. It expresses the actor/system interactions and yields an observable result of value to an actor.

For each actor identified previously, it is advisable to search for the different business goals, according to which is using the system.

**

1.3 Prepare a preliminary list of use cases of the ATM, in order of actor.

Answer 1.3

Let's take the five actors one by one and list the different ways in which they can use the ATM:
CardHolder:

- Withdraw money.

5. From the OMG document: "Unified Modeling Language: Superstructure (version 2.0)".

Bank customer:

- Withdraw money (something not to forget!).
- Consult the balance of one or more accounts.
- Deposit cash.
- Deposit cheques.

Maintenance operator:

- Refill dispenser.
- Retrieve cards that have been swallowed.
- Retrieve cheques that have been deposited.

Visa authorisation system (AS):

- None.

Bank information system (IS):

- None.

Primary or secondary actor

Contrary to what we might believe, all actors do not necessarily use the system! We call the one for whom the use case produces an observable result the *primary* actor. In contrast, *secondary* actors constitute the other participants of the use case.[6] Secondary actors are requested for additional information; they can only consult or inform the system when the use case is being executed.

This is exactly the case of the two "non-human" actors in our example: the *Visa AS* and the *Bank IS* are only requested by the ATM within the context of realising certain use cases. However, they themselves do not have their own way of using the ATM.

6. In his excellent book, *Writing Effective Use Cases* (Addison-Wesley, 2001), A. Cockburn defines similarly *supporting actors*: "A supporting actor in a use case is an external actor that provides a service to the system under design."

1.3 Step 3 – Creating use case diagrams

We are now going to give concrete expression to our identification of use cases by realising UML diagrams, aptly called use case diagrams. A use case diagram shows the relationships among actors and the subject (system), and use cases.

We can easily obtain a preliminary diagram by copying out the previous answer on a diagram that shows the use cases (ellipses) inside the ATM system (box) and linked by associations (lines) to their primary actors (the "stick man" icon).

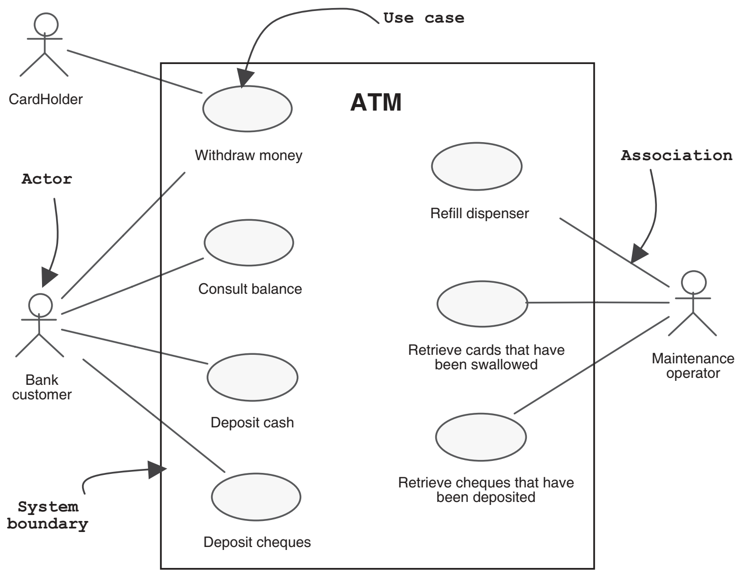


Figure 1.4 Preliminary use case diagram of the ATM

*** 1.4 Propose another, more sophisticated version of this preliminary use case diagram.

Answer 1.4

The *Withdraw money* use case has two possible primary actors (but they cannot be simultaneous). Another way to express this notion is to consider the *Bank customer* actor as a specialisation (in the sense of the inheritance relationship) of the more general *CardHolder* actor. A bank customer is actually a particular card holder who has all the privileges of the latter, as well as others that are specific to him or her as a customer.

UML enables the depiction of a generalisation/specialisation relationship between actors, as indicated on the diagram below.

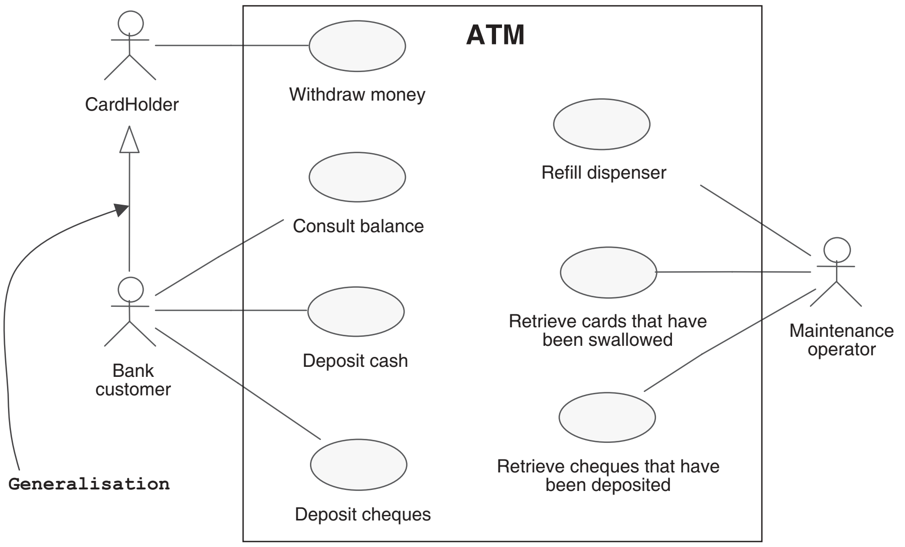


Figure 1.5 A more sophisticated version of the preliminary use case diagram

However, the significance of this generalisation relationship is not evident in our example. Certainly, it enables the association between the *Bank customer* actor and the *Withdraw money* use case to be removed, which is now inherited from the *CardHolder* actor, but on the other hand, it adds the symbol for generalisation between the two actors... Moreover, we will see in the following paragraph that the requested secondary actors are not the same in the case of the CardHolder and in that of the bank customer.

We will therefore not use this solution and, to reinforce this choice, we will rename the primary actor *Visa CardHolder*, to clarify matters a little more.

We now have to add the secondary actors in order to complete the use case diagram. To do this, we will simply make these actors appear with additional associations towards the existing use case.

Graphical precisions on the use case diagram

As far as we are concerned, we recommend that you adopt the following conventions in order to improve the informative content of these diagrams:

- by default, the role of an actor is "primary"; if this is not the case, indicate explicitly that the role is "secondary" on the association to the side of the actor;
- as far as possible, place the primary actors to the left of the use cases and the secondary actors to the right.

** 1.5 Complete the preliminary use case diagram by adding the secondary actors. To simplify matters, leave out the maintenance operator for the time being.

Answer 1.5

For all use cases appropriate for the bank customer, you must explicitly bring in *Bank IS* as a secondary actor.

But a problem arises for the shared use case, *Withdraw money*. Indeed, if the primary actor is a Visa card holder, the *Visa AS* must be called on (which will then be responsible for contacting the IS of the holder's bank); whereas the ATM will contact the *Bank IS* directly if it concerns a bank customer.[7]

One solution consists in adding an association with each of the two non-human actors. This simplistic modelling does not make it clear to the reader of the diagram that the actors are selectively participating two by two and not all together.

7. Remember that the domain refers to French banking systems, which may explain differences with your knowledge and experience.

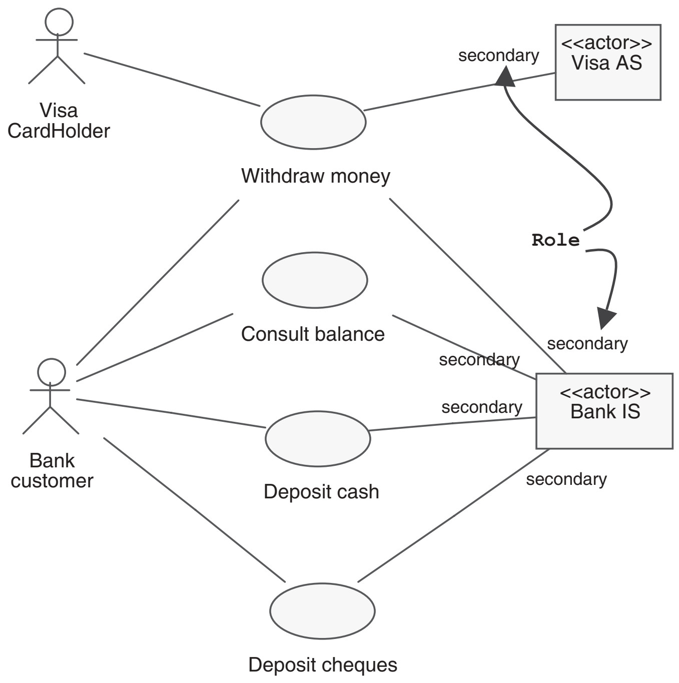


Figure 1.6 Simple version of the completed use case diagram

Another solution would be to distinguish two use cases for the withdrawal of money: *Withdraw money using a Visa card* and *Withdraw money using a bank card*. This more precise, yet more cumbersome, modelling is easier for the reader of the diagram to grasp. Furthermore, it clearly tells against the use of generalisation between actors, which was mentioned beforehand. Indeed, the distinction between the two use cases is contradictory with the attempt at inheritance of the unique *Withdraw money* case, which had been viewed more highly, while the secondary actors had not yet been added. We will keep this second solution for the follow-up to the exercise.

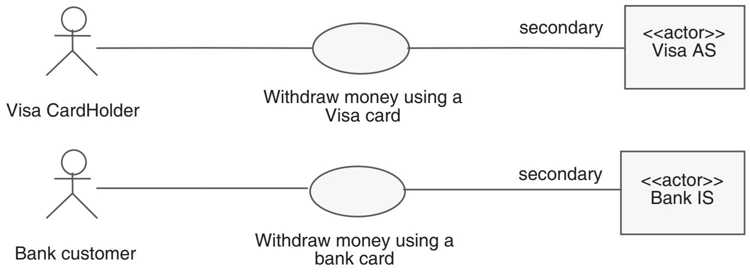


Figure 1.7 Fragment of the more precise version of the completed use case diagram