Paper 161-29

Modeling Object-Oriented SAS/AF® Applications Using UML®

Dominic Roy, DMR Conseil / Fujitsu Consulting, Sainte-Foy, Quebec, Canada
André Milliard, DMR Conseil / Fujitsu Consulting, Sainte-Foy, Quebec, Canada

ABSTRACT

UML is a widely used modeling standard for developing object-oriented systems; it stands for ***The OMG's Unified Modeling Language™*** (**UML®**). It is becoming more mature and integrated in many development methodologies.

Confronted with a complex SAS/AF® application, it was decided to turn to a real object-oriented approach, on a small scale and use a UML way to represent the model.

It revealed to be helpful in building a better application architecture, with a simpler user interface and a concept of business layer classes. It allowed to divide the work among many programmers and to support full-fledged development standards.

This paper establishes the link between the main UML diagrams and a SAS/AF application.

INTRODUCTION

An actuarial liability valuation system may need a very complex user interface application when it is built on a concept of parameterized calculations within a conciliation approach. Roy & Baillargeon (2004) explains the details of such an application in its context.

Previous work (like Booch (1994)) has reported that object-oriented approaches have been developed to deal with the growing complexity of the systems. During the last decade, UML was developed as an industry standard for modeling and communicating the intricacies of object-oriented systems.

Because of the size of the user interface application project and the involvement of only two programmers, Microsoft® Visio® 2000 Professional was selected to model the application. The professional version of the software supports a modeling rules layer and a database behind its graphical representation for UML modeling and database modeling; this means that if an element is added in a diagram it then becomes available in another type of diagram (for example a static diagram versus a dynamic diagram). The software worked very well for our needs and purposes but we did not even use it to its full potential: the models were not shared, and there were interesting features for generating or interpreting code but they were not used. For a larger project, it might be necessary for another tool to be used.

The objective of this paper is to witness that a UML development can be made on a small scale using SAS/AF. SAS® developers who may want to try to develop a SAS/AF application using UML modeling will benefit from this paper but it should not be considered as an introduction to UML. It is recommended that anyone who wants to understand UML should read introductory textbooks, followed by the OMG standard, or to attend a class on the subject.

Moreover, readers should consider that we do not pretend to do state-of-the-art UML. We have only scratched the surface of UML in order to solve our problem. We have tried to follow the principles as much as possible and to respect good design rules. The rules followed were those established by the UML version 1.4, published in September 2001.

COMPLEXITY

Complexity is a curious concept. It should be a neutral concrete measure that has nothing to do with the intellectual capacities of the analyst. Complexity in this context does not translate as “what is difficult to understand”, but rather as the number of interactions and their types, the dependence between interactions and the probability of occurrence of such interactions. There are standard measures for complexity, particularly in information technology, but for the sake of simplicity, it could be described in layman’s terms. The system we are dealing with is not extremely complex (as a meteorological or a stock exchange financial system could be), but it is somewhat complex.

Most often, SAS systems are batch jobs that are run in a rather sequential fashion. These systems can be described as “process centric systems”. In their simplest form, the data table is created and written once and for all, or a report is printed. The calculations may be very complex, but it is typical that those jobs can be managed through

SAS/Warehouse administrator®. When possible, a good design of SAS systems tends to divide the work into batch jobs of this kind, thus reducing the interaction between jobs.

Sometimes, SAS batch jobs can become very complex, for example, when a job produces too many data tables or different reports. Two main problems occur at this moment: the validation process and the evolution of the code. Validating the results becomes almost impossible because of this complexity. The job must leave intermediate results or reports that can be validated progressively in order to check the quality of the result. It can also become necessary to import data into worksheets to imitate the calculations. This validation can be done when the code of the program is modified, but whether that new data combination could generate a false result will remain uncertain. As the code becomes more complex, programmers require more time to make changes in the code and test them. This is the reason why a good design tends to split complex batch jobs into smaller and simpler ones, thus intermediate data tables become permanent data tables.

Actuarial simulation systems in shared environments represent many other levels of complexity over the preceding batch jobs level: 1-Jobs are often made of complex calculations. 2-Calculations have to be parameterized. 3-Many calculation parameters must be shared between many calculations in order to define a coherent scenario. 4-Many actuaries work simultaneously. 5-Many scenarios could be tested at the same time.

Many difficulties arise from this complexity. Results become almost impossible to verify. The integrity of the scenarios can raise doubts. As certain calculations may depend on others, they must be executed in a proper sequence. All the calculations of only one scenario could take more than an hour to be executed.

The way we chose to cope with so many requirements was to build an application to manage calculations, follow the approval process, control the integrity of the parameters and manage the environments. This application should allow many users to work simultaneously, manage user rights, and remain independent from the scenarios. It must also allow them to see the intermediate results of the calculations, lock the calculation when approved, and check the integrity of the approval process.

The following examples are based on such an application.

GENERAL VIEW OF UML

“The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.” (Unified Modeling Language Specification” (1998))

UML defines common artifacts appropriate for most projects:

- ❑ Use Case Diagrams, define the relationships between users and other systems with the current system. We use them in this context.
- ❑ Class Diagrams are the formal description of the classes and the static relationships between classes. We use them in this context.
- ❑ Behavior Diagrams are numerous forms of dynamic diagrams: Statechart diagram, Activity diagram, Interaction diagram, Sequence diagram and Collaboration diagram. In this context, we used only the Sequence diagrams.
- ❑ Implementation Diagrams represent the physical way the system has been designed. Only the Component Diagrams have been used in this context.

USE CASE DIAGRAMS

Use case diagrams are the opening doors to the system. They help to define who interacts with the system and what the major functions of the application are.

In figure 1 (Use Case Diagram), two types of actors are defined. The actors can be described as the role in which the people are interacting with the system. For example, the major actor of this application is the “Actuary”, and the secondary actor is the “Actuary in charge”. This actor inherits from the “Actuary” which means that he is an “Actuary” with a particular added characteristic. In other words, the “Actuary in charge” is an “Actuary” who has the right to approve the calculation group.

The ellipses represent the use cases; what is done is written in normal English. A direct line between an actor and a use case represents an association, which means that the actor is a part of the use case. For the “Actuary” actor, only four use cases are defined and for the “Actuary in charge” actor, only one is defined. This representation greatly simplifies what is actually done by the application. There are indirect use cases that are accessible to the

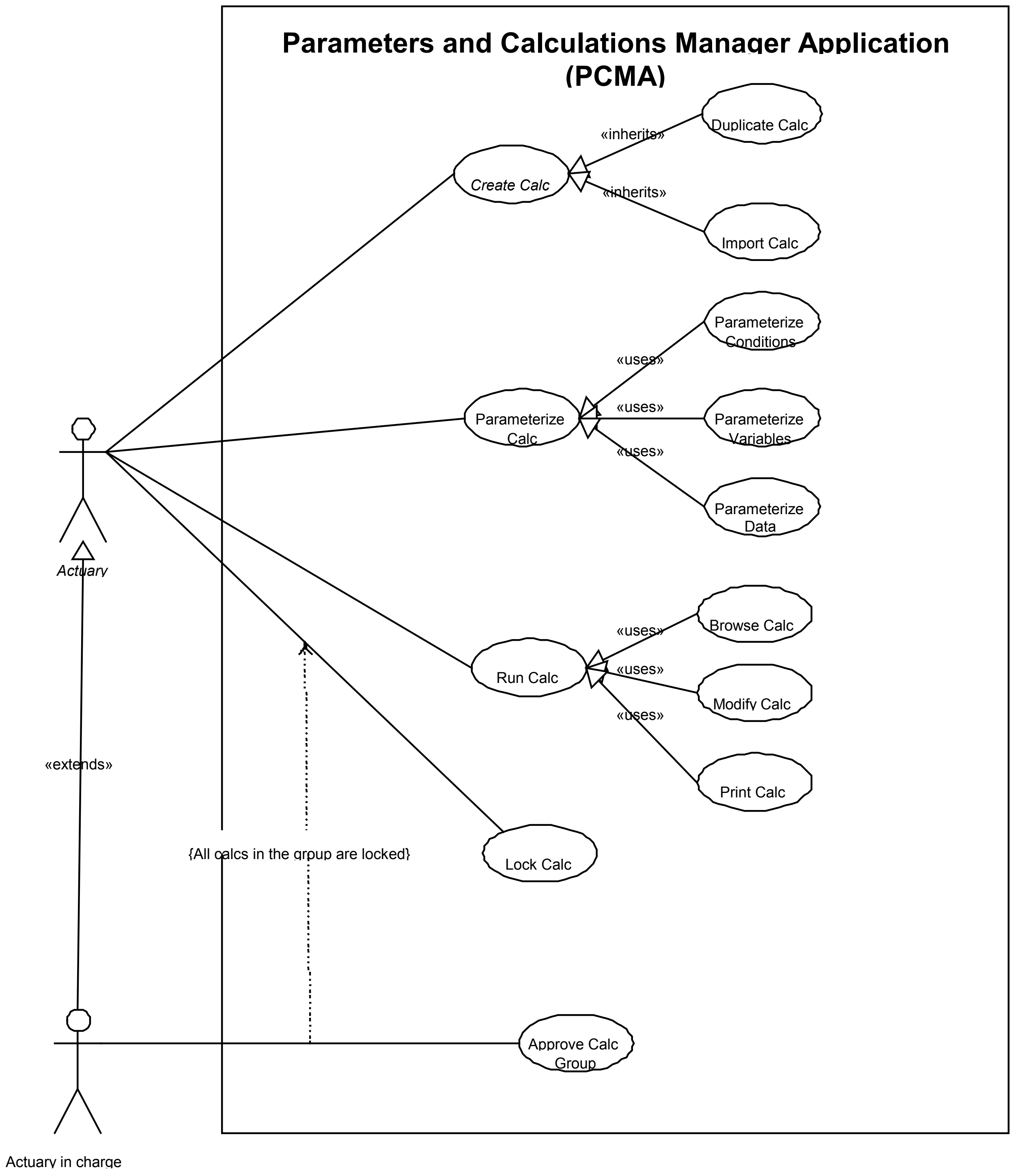

Figure 1 Use Case Diagram

actors. The diagram allows adorning certain conditions between the relationship such as the one used to mention that the calculations in a group must be locked before the calculation group is approved.

CLASS DIAGRAMS

PACKAGING

In system design, packaging is the most important concept in order to deal with the complexity. Even though the concept is simple, if the rules are not followed, the application definitely ends up a mess of code. Packages can contains other packages and packages apply to everything in UML: use cases, components, classes, etc.

In this project, class packages played an important role. In figure 2 Class Packages, the resulting packages were represented at the top level (remember this is a very small-scale project). Class packages are often related to the component packaging, which in SAS can correspond to a catalog. Technically, since a SAS catalog could contain many class packages, it would therefore be important for the analyst to maintain the packaging division. However, it is desirable to maintain a close relationship between the component packaging and the class packaging. Component packages give a decisive test to the class packaging.

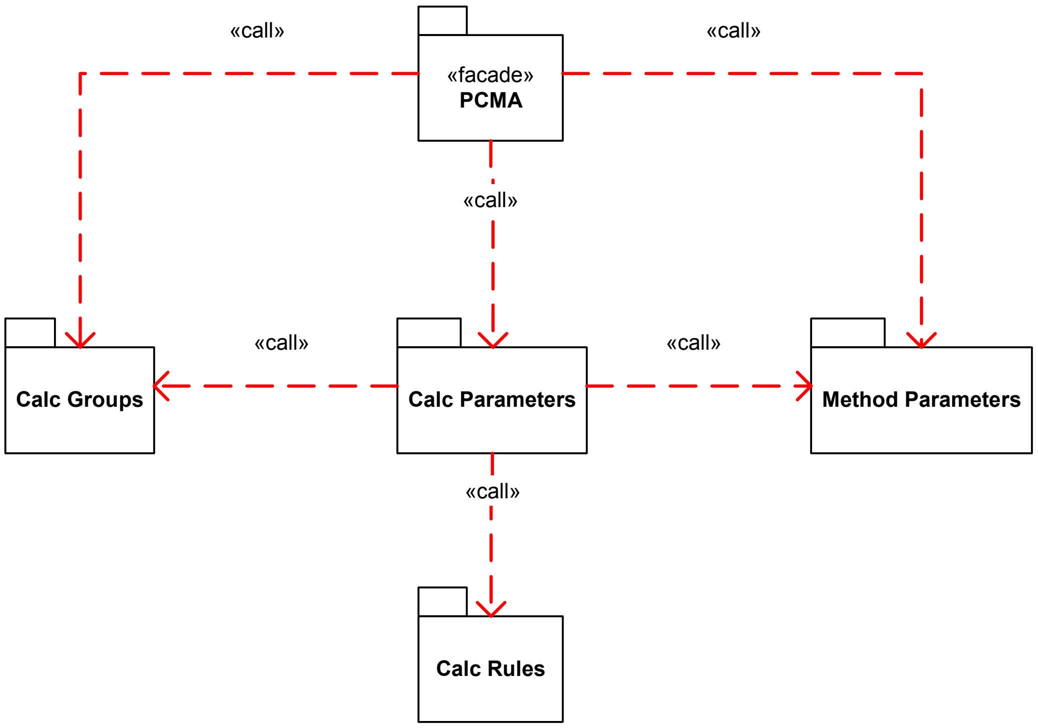

Figure 2 Class Packages Diagram

When many programmers are working on the same application, you may want as little interactions as possible between the packages (the catalogs). This is also a major rule in packaging. This can result in something that could be counter-intuitive at the first glance and could be disputable among analysts. The figure 2 results from an incremental process of packaging. Packaging can change as we add new features or change the scope of the project. It is preferable to be prepared to accept changes during the run.

Figure 2 represents a user interface application PCMA that relies directly on three business rule packages. This packaging may not be optimal because of the number of relationship. However, it can be observed that the dependency links head in only one direction, which is positive and almost mandatory; when two packages, call each other, development problems are inevitable.

CLASS STATIC DIAGRAMS

Classes are the basis of the object-oriented design and programming. They exist within SAS/AF through the SAS Component Object Model (SCOM) ®. Our main concern at the beginning of the project was to understand how close to UML this object model is. The answer is clear: as far (not very far) as this approach was used, we never reached a limitation.

The class diagram in the figure 3 shows two classes in a lower level class package: “variables collection class” and “variable class”. It is only a smaller part of the whole classes model but it is representative of what is done many times.

It also shows a SAS component class “OrderedCollection” that is part of the SASNonVisual package; this is why it is shaded. This package is an arbitrary package that was defined within Visio UML to support SCOM.

The “Variables Collection” class inherits from the SAS ordered collection class. It is responsible for managing all the variables related to a calculation, like adding and removing variables, keeping track of what was recently modified and saving all the variables to a SAS table. In order to do its job, an instance of “Variables collection” must

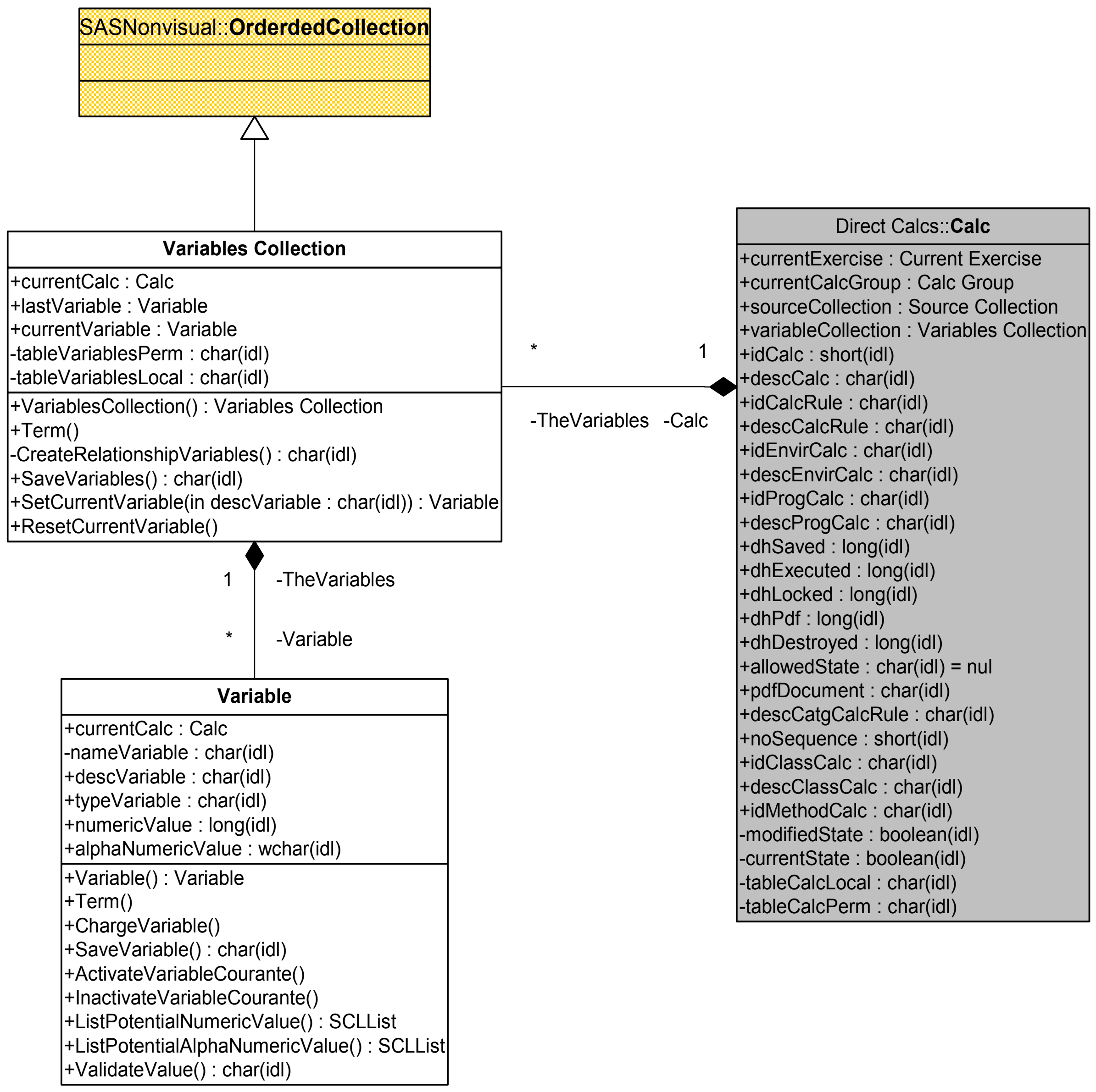
SASNonvisual::OrderdedCollection
Variables Collection
+currentCalc : Calc
+lastVariable : Variable
+currentVariable : Variable
-tableVariablesPerm : char(idl)
-tableVariablesLocal : char(idl)
+VariablesCollection() : Variables Collection
+Term()
-CreateRelationshipVariables() : char(idl)
+SaveVariables() : char(idl)
+SetCurrentVariable(in descVariable : char(idl)) : Variable
+ResetCurrentVariable()
*
1
-TheVariables
-Calc
Direct Calcs::Calc
+currentExercise : Current Exercise
+currentCalcGroup : Calc Group
+sourceCollection : Source Collection
+variableCollection : Variables Collection
+idCalc : short(idl)
+descCalc : char(idl)
+idCalcRule : char(idl)
+descCalcRule : char(idl)
+idEnvirCalc : char(idl)
+descEnvirCalc : char(idl)
+idProgCalc : char(idl)
+descProgCalc : char(idl)
+dhSaved : long(idl)
+dhExecuted : long(idl)
+dhLocked : long(idl)
+dhPdf : long(idl)
+dhDestroyed : long(idl)
+allowedState : char(idl) = nul
+pdfDocument : char(idl)
+descCatgCalcRule : char(idl)
+noSequence : short(idl)
+idClassCalc : char(idl)
+descClassCalc : char(idl)
+idMethodCalc : char(idl)
-modifiedState : boolean(idl)
-currentState : boolean(idl)
-tableCalcLocal : char(idl)
-tableCalcPerm : char(idl)
1
-TheVariables
*
-Variable
Variable
+currentCalc : Calc
-nameVariable : char(idl)
+descVariable : char(idl)
+typeVariable : char(idl)
+numericValue : long(idl)
+alphaNumericValue : wchar(idl)
+Variable() : Variable
+Term()
+ChargeVariable()
+SaveVariable() : char(idl)
+ActivateVariableCourante()
+InactivateVariableCourante()
+ListPotentialNumericValue() : SCLList
+ListPotentialAlphaNumericValue() : SCLList
+ValidateValue() : char(idl)

Figure 3 Static Class Diagram

know the calculation to which it is linked; this is why the object currentCalc, an instance of the class “Calc”, is inserted as an attribute into the “Variables collection” class.

Because of the **currentCalc** attribute, the methods of the «Variables collection» easily have access to all the information needed. For example in SAS/AF, the execution datetime of the calculation is accessible as **currentCalc.dhExecuted** and the current exercise year as **currentCalc.currentExercise.yearProduction** (not shown). This dot notation used in SAS/AF is very useful and it is coherent with a part of UML called OCL, the Object Constraint Language. The information’s accessibility (or the navigability) from an instance to the other is generally controlled by the visibility (or scope) attribute of the class attribute. For example, in the ‘”Calc” class, the **currentExercise** attribute is adorned by a plus sign which means that this attribute has a public visibility. The minus sign, as seen with **modifiedState,** is associated with a private visibility. While allowed by SCOM, we used it to a very limited extent but no problem was encountered. However, managing the visibility was not an easy task because more discipline was required. We are however confident that a more robust design will be generated.

Modeling detailed class diagrams with Microsoft Visio gives one the impression that the work is done more than once because every tiny change in the attributes and the operations must be made in Visio and SAS. This is a common comment for all programming languages; this is why Visio, like others, suggests a code generation feature and a reverse engineering feature. It is readily available for Java, but, as usual, not for SAS; it seems to be possible to develop such macros for SAS but it was not done within this project. We decided to maintain both the model and the code.

We separated and organized the SCL code for classes in three entries: 1-the class definition entry, 2- the method code entry, and 3-the test code entry. The class definition entry is the code that defines the class; rather than using the class editor to create and modify the classes, the formal code definition was used. It simplifies maintenance, compilation and impact analysis, provided you have a very good code editor. This entry cannot be directly compiled particularly with PROC BUILD; it is necessary to use the SAVECLASS command. Since this entry did not change very often, it was acceptable. The method code entry contains the code for the methods defined in the class definition entry.

BEHAVIOR DIAGRAMS

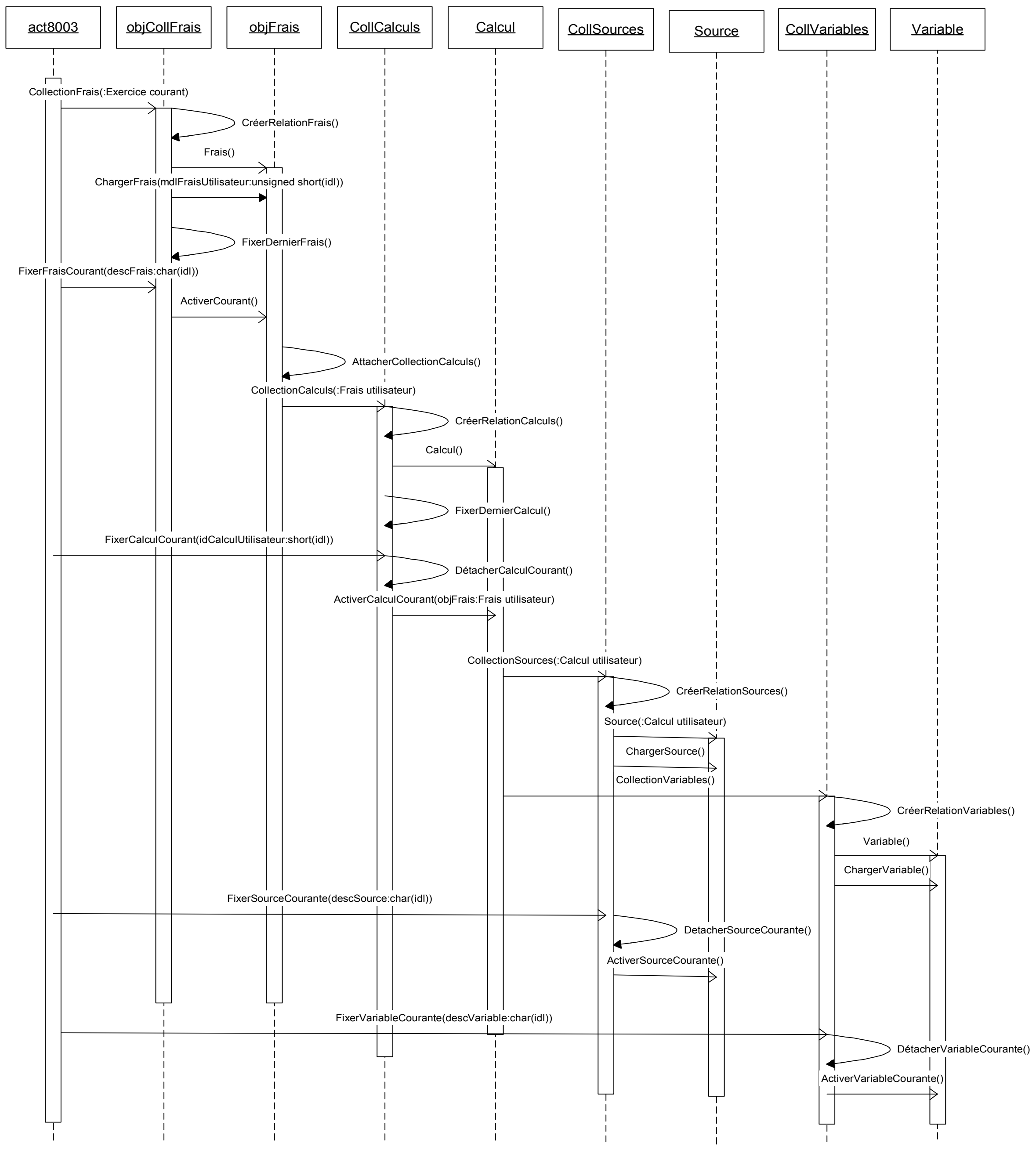
act8003
objCollFrais
objFrais
CollCalculs
Calcul
CollSources
Source
CollVariables
Variable
CollectionFrais(:Exercice courant)
CréerRelationFrais()
Frais()
ChargerFrais(mdIFraisUtilisateur:unsigned short(idl))
FixerDernierFrais()
FixerFraisCourant(descFrais:char(idl))
ActiverCourant()
AttacherCollectionCalculs()
CollectionCalculs(:Frais utilisateur)
CréerRelationCalculs()
Calcul()
FixerDernierCalcul()
FixerCalculCourant(idCalculUtilisateur:short(idl))
DétacherCalculCourant()
ActiverCalculCourant(objFrais:Frais utilisateur)
CollectionSources(:Calcul utilisateur)
CréerRelationSources()
Source(:Calcul utilisateur)
ChargerSource()
CollectionVariables()
CréerRelationVariables()
Variable()
ChargerVariable()
FixerSourceCourante(descSource:char(idl))
DetacherSourceCourante()
ActiverSourceCourante()
FixerVariableCourante(descVariable:char(idl))
DétacherVariableCourante()
ActiverVariableCourante()

Figure 4 Sequence Diagram

The behavior diagrams are used to present specific dynamic situations that need a more detailed explanation. We concentrated our work only on the sequence diagram, which we found useful. One of the most useful is the set up of all the business layer classes at the opening of the application. Readers should be aware that this representation does not follow the standard, mainly with the use of the activation symbol. It had been left as it is in order to communicate the complexity – and the organization - of the operation.

The leftmost object is the main Frame application. In its INIT phase, five messages are sent to the business layer objects: 1- Load the collection of all categories of calculation, 2- Assign the current calculation category, 3- Assign the current calculation, 4- Assign the current source, and 5- Assign the current variable. Those simple messages trigger the creation of temporary tables and the loading of the data into object collections for the four business layers (each layer being made up of a collection object and the objects making up the collection).

Sequence diagrams tend to become complex because we want to explain all the single interactions. We learned to group the messages in a concept close to the interface (the UML concept). It made simpler diagrams, even if it requires two or three diagrams to explain all the interactions with a business layer.

COMPONENT DIAGRAMS

Deployment diagrams and component diagrams are less explained and standardized in the current textbooks and in the OMG UML specification. Ambler (2003) within his Web site recognized it and gives much help on what to do. It can be explained as the sign of the complexity of the task, because all development software products are different in their component structure.

More than following a strict direction on how to design the perfect diagram, we devised a component diagram that can communicate much detail about the way we implemented the different modules. Its main value was to separate the responsibilities of the components. The representation of a component in the diagram is a rectangle with two small rectangles superimposed on it. Those small rectangles stand for the ports to which interfaces can be attached (the lollipops). We used the interface lollipops as descriptors of types of interaction between the components, rather than physical interfaces, such as the ones found in the classes.

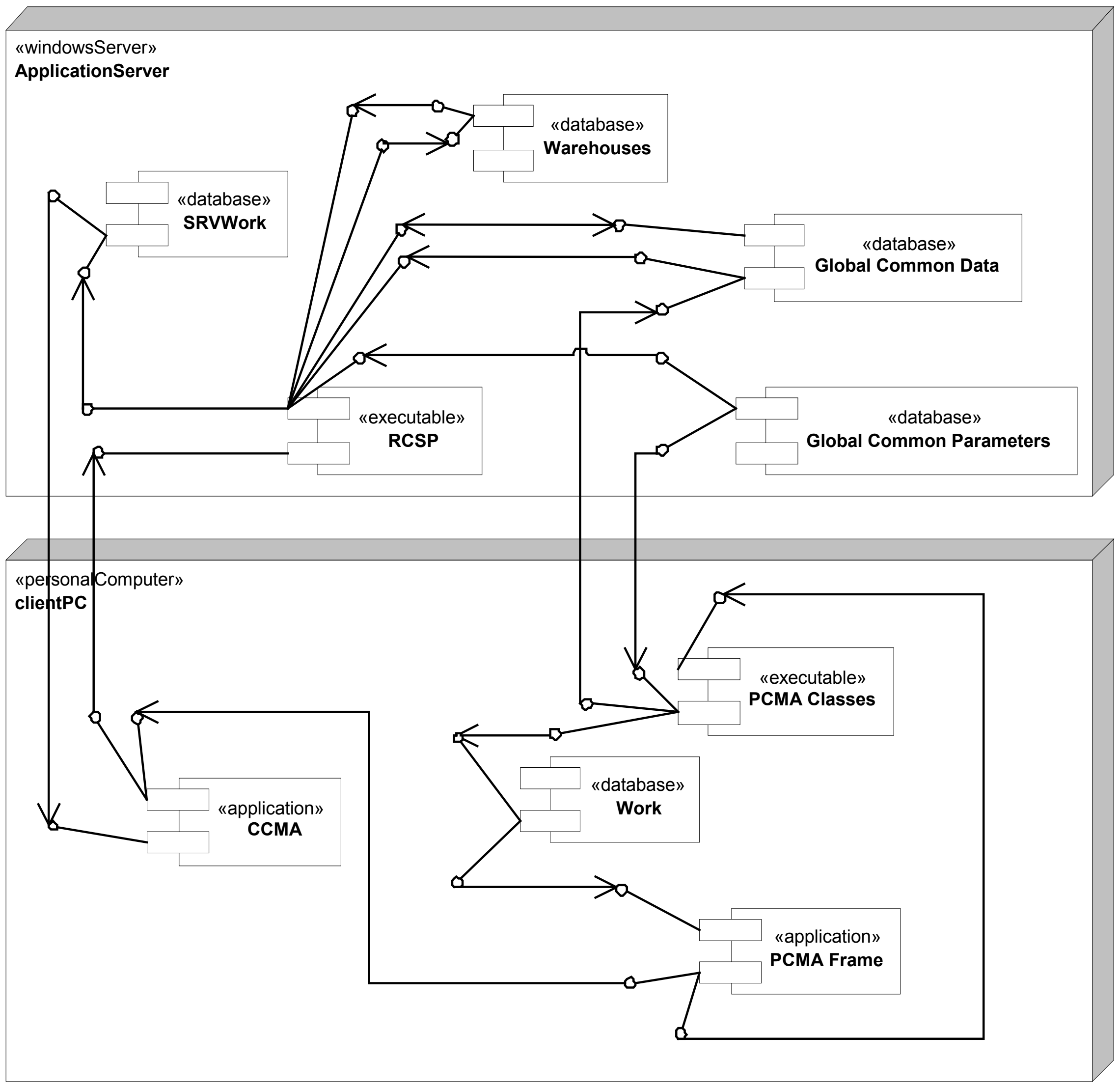

Figure 5 Component Diagram

In the figure, the PCMA Frame is an application that interact specifically with three other components: 1-it reads data tables only from a local work database (the common SAS work library); 2-it gives orders to the PCMA Classes that will fill the Work database; 3- it triggers the CCMA application. It becomes clear that the PCMA application does not have access to the permanent data otherwise than collaborating with the PCMA Classes; this is a delegation with the business layers classes.

The CCMA application (a SAS/AF frame application) does a very similar delegation with the RCSP executable (actually this is a collection of base SAS programs and macros with SAS/IML modules). All the exposable results are written by the RCSP executable into the server SRVWork database (the server's SAS work library) and the CCMA application shows them in its screen. RCSP executable also writes its different results in the Warehouses database and the Global Common Data database. In our case, a database stereotype was used to designate a real data warehouse in which results can be tables, metadata and documents.

Not all the links are currently represented in our diagram for sake of simplicity. Actually, the current diagram attempts to show the relationships between the three major phases of the project, which are the PCMA and CCMA applications, and the RCSP executable. There will also be specific detailed diagrams for each of them in which all the relationships will be exposed.

CONCLUSION

Making an object-oriented design for an application is interesting because it can be used for any true object-oriented programming language. In theory, a part of the current application could have been programmed in Visual Basic (.NET) or in JAVA. In practice, all the programming was executed using SAS/AF, but the weight of the Frame code was reduced in a dramatic proportion. Major changes were made in the Frame design without taking care of the business rules.

As Booch, G. (1994) wrote, there is no limit to the complexity of the systems. Whatever your IQ, you will face the complexity wall. If your IQ is very high and you program a 20 000 line-SAS/AF frame program, your colleagues and boss will not necessarily be appreciative and you will end up with a lifetime maintenance job on your hands. The only way to cope with the complexity is to take a structured and standard approach.

We think that having chosen SAS/AF as a programming language was a clever choice, because it supported any concept of a UML object-oriented programming language. We are conscious that we did not push things to the limit. As we have chosen to use only SAS in the system and particularly the graphical user interfaces, we ended up with a simplified relationship between the data, the workspace and the programming language. It would have been different if we had chosen a different database and a different programming language for the graphical user interface.

REFERENCES

CROSS-REFERENCES

Roy, D. (2003), "Building an Actuarial System using SAS/Warehouse Administrator®," *Proceedings of the Sixteenth Annual North-East SAS Users Group (NESUG) Conference (et001)*

Roy, D. and Baillargeon, A. (2004), "Architectural Views of Building an Actuarial System using SAS/Warehouse Administrator®," *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference*

Roy, D. and Genois, J. (2004), "Business Views of Building an Actuarial System using SAS/Warehouse Administrator®," *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference*

GENERAL REFERENCES

Booch, G. (1994), "Object-Oriented Analysis and Design with Applications (second edition)," The Benjamin/Cummings Publishing Company, Inc: Redwood City, California, 589 p. (a third edition is to be released in May 2004 with Addison-Wesley).

Booch, G., Rumbaugh, J. and Jacobson, I. (1999), "The Unified Modeling Language User Guide," Addison-Wesley, Reading, Mass., 512 p.

Booch, G., Rumbaugh, J. and Jacobson, I. (1999), "The Unified Modeling Language Reference Guide," Addison-Wesley, Reading, Mass., 576 p.

"Unified Modeling Language Specification" (1998), Object Management Group, Framingham, Mass. It is not recommended to begin with this reference because it is a formal paper. UML is a language that defines itself with a meta-model and a meta-meta-model. This paper is written using a meta-language that could be difficult to read for a beginner.

WEB SITES

Further readings and information on UML can be found on the following Web sites. Please note that OMG is currently preparing the delivery of the version 2 of UML.

Object Management Group: http://www.omg.org/
UML web Site: http://www.uml.org/
Ambler, Scott W, the official Agile Modeling site: http://www.agilemodeling.com
Ambler, Scott W, Modeling with style: http://www.modelingstyle.info/componentDiagram.html

ACKNOWLEDGMENTS

We want to thank our employer, the Quebec City office of DMR Conseil, a subsidiary of Fujitsu Consulting for their support, and particularly to Mr. Pierre Hamel who encouraged us to go ahead with this paper.

The authors would like to express their appreciation to the following people for their assistance in this paper:
Susan Assad, DMR Conseil
Dave Couture, DMR Conseil
Alain Baillargeon, DMR Conseil

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Dominic Roy
DMR Conseil / Fujitsu Consulting
2960 boulevard Laurier
Sainte-Foy, QUEBEC
CANADA, G1V 4S1
Work Phone: 418-653-6881
Fax: 418-653-4428
Email: dominic.roy@consulting.fujitsu.com
Web: http://globalservices.fujitsu.com/services/

André Milliard
DMR Conseil / Fujitsu Consulting
2960 boulevard Laurier
Sainte-Foy, QUEBEC
CANADA, G1V 4S1
Work Phone: 418-653-6881
Fax: 418-653-4428
Email: Andre.Milliard@consulting.fujitsu.com
Web: http://globalservices.fujitsu.com/services/