Exceptions in Object Modeling

Questions from an educational experience

Yolande AHRONOVITZ, Marianne HUCHARD
LIRMM UMR5506 161 rue Ada, 34392 Montpellier Cedex 5
email: yolande@lirmm.fr, huchard@lirmm.fr

1 Introduction

In a course on object-oriented modeling and programming, we teach UML formalism and Java and C++ languages. Through lessons and exercises, we systematically link analysis, design and programming.
In this framework, learning exceptions joins two sorts of difficulties: first of all, understanding what is an exception and what is not one, and reifying exceptions; second, understanding and using exception handling.

Most of the today's object-oriented languages have an exception handling system. The handling mechanism is easy enough to teach[1]. It is generally well-documented, its concepts are clear.
In the following, we suppose that handling exceptions is Java-like. To sum up: we throw an instance of a class *ExcA*; this instance can be caught if it is thrown inside a block `try{...}`, and if there is, either a block `catch(ExcA e){...}`, or a block `catch(ExcB e){...}` with *ExcB* ancestor class of *ExcA*, which can be reached at this time.

Some languages also supply a predefined class hierarchy, and offer the possibility of adding user defined classes to it.
But modeling and design methods are very silent about this topic: they provide few means, and only to represent the dynamic sides of exceptions.
For example, in the UML metamodel [Rat97], an exception is a *Signal*, dedicated to errors. Different dynamic diagrams allow us to represent how to throw it, how to catch it. It is linked by associations to *behavioral features*[2], which catch it and throw it; it is *generalizable* and *specializable*; it can have parameters. However, we can't describe it using features as attributes, methods, and/or associations. The only provided "feature" is a *body* the content of which must be "*a description of the exception in a format not defined in UML*". Nothing allows us to model an exception "class-like" (exception classes of Java can't be modeled with this formalism).

Thus UML does not help us much to think up, describe, organize,..., exceptions.
For our purpose and for teaching, we would like to have concepts, and not only recipes.
Literature gives us the (good) following advice: study methods' preconditions and postconditions, define for each object what a consistent state is. At closer look, we may not find these sufficient; how, for example, can one study the interaction between several classes?
From the exercises we used for the students a fundamental problem arises: how to design exceptions in connection to others parts of an object model (classes, attributes, methods, associations, specialization relations).
In Sections 2 to 5, we present all kinds of exceptions we have met, and possible solutions for modeling, in terms of classes and class hierarchies. In Section 6, we take stock of all raised questions. Especially, we propose some additions to the UML metamodel, in order to have the means to represent with this formalism the solutions we propose.

2 Exceptions and Types

We make the assumption that exceptions must be devised in connection with program types, i.e. we study all exceptions associated with every class when building the class. For instance, when we construct the *Stack* class, we must make an inventory of all the problems linked to the state of the stack, and of all the problems which may arise when we use the stack.

[1] This work does not cover research on handling mechanism, as it is not our field of research.
[2] UML term, which can refer to an operation or to a method.

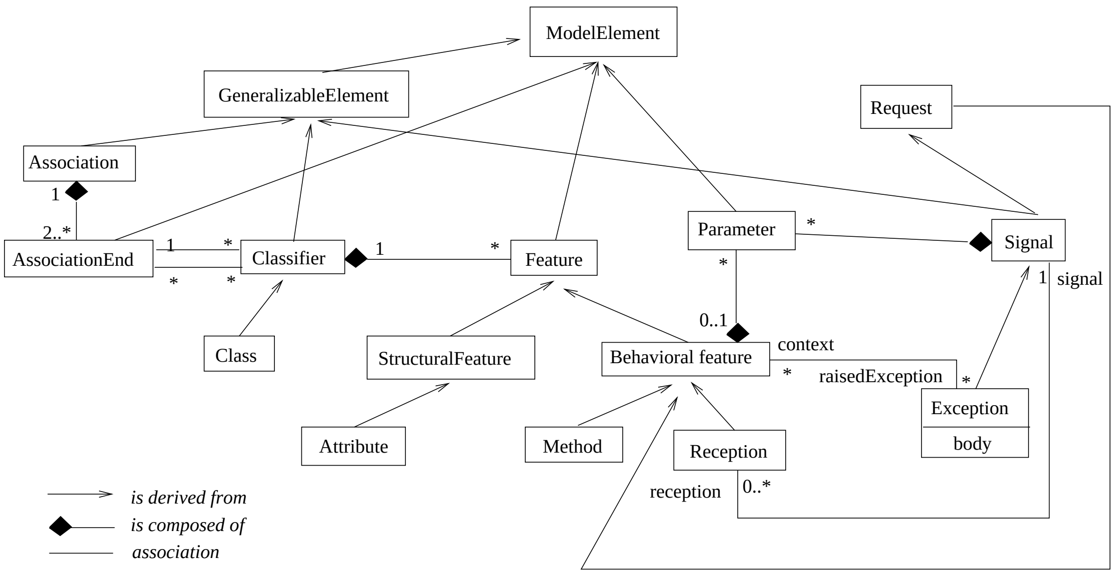

Figure 1: Extract from UML metamodel

We want the exceptions associated with a type to have the same abstraction level as the type itself; many people think the same thing [DPW91]. For a given type, this leads us to define exceptions which will be used to "encapsulate" low-level exceptions. For example, we create an "impossible push" exception associated with the *Stack* type; such an exception occurs every time a push is impossible, e.g. when a "memory overflow" occurs. That represents, at the conceptual level, what we do in practice when we catch an instance of "memory overflow" and treat it by throwing an instance of "impossible push".
To put to a single hierarchy all the exceptions associated with a given type helps to factorize their features [Don90].

The previous remarks leads us to the following principle: if X is a type, X 's own exceptions are all rooted under a general exception class *ExcTypeX*.

3 Exceptions and Features

According to UML formalism, features of a class can be split up into primitive attributes (this means that the attribute's type is not a class), methods and associations.
Methods and attributes
For each method *meth* of X which gives rise to exceptions, we can put, under the main root *ExcTypeX*, an exception class *ExcMeth*, which will be the mother of all the exceptions thrown by *meth*. We can do the same for each primitive attribute *attrP* of X, assuming that exceptions associated with an attribute mean exceptions its accessor methods can signal.
One can think this is very heavy. If two methods *meth1* and *meth2* can signal the same kind of exception E, there will be one class *E1* under the root *ExcMeth1*, which represents E in *meth1*'s context, and another class *E2* under the root *ExcMeth2*, which represents E in *meth2*'s context. Do we need that?
The answer is given when making progress in the design. Either the two contexts are the same, and we can simplify : we remove *E1*, *E2*, even *ExcMeth1* and *ExcMeth2*, and define only one class E. Or the two contexts are different, and we keep all the classes (and ever add E as a superclass for *E1* and *E2*).
For instance, we can associate an *EmptyStack* exception with the class *Stack*. The two methods *pop* and *peek* (look at the top object without pop) can signal this exception. Must we have an *EmptyStackPop* exception under *ExcPop*, and an *EmptyStackPeek* under *ExcPeek*? It depends on the framework in which the stack is used. We can wish to treat differently the *EmptyStack* problem when we try to pop, and when we only want to see; that being the case, it is better to have the means to catch the two exceptions separately.

Associations
We now look at associations: can they give rise to exceptions into the classes they link? If so, how and where can we represent them?

First, when a class X is linked to another class Y, an exception of one of them can generate an exception of the other through the association. Let us take for example a dispenser, containing several compartments. We model this using an "aggregation" between the class *DrinksDispenser* and the class *Compartment*. The existence of this aggregation relation, and the existence of an *Empty* exception defined on the *Compartment* class imply the existence of an *AnEmptyCompartment* exception defined on the *DrinksDispenser* class. In the code, we express this by saying that the dispenser catches the instance of the *Empty* exception thrown by a compartment, and treats it by throwing an instance of the *AnEmptyCompartment* exception.
On the other hand, a compartment will not catch an exception thrown by the dispenser.
This is typical of a composition relation. To respect the encapsulation, the composite takes responsibility for its components' exceptions, and the opposite would not happen.
For an arbitrary association, there is no such precise rules: some exceptions of X can generate exceptions of Y, some exceptions of Y can generate exceptions of X, and each of the two classes can have exceptions which have no influence at all on the other class.
Let us examine at two classes *Person* and *Train*, linked by the *driver* association. This association makes sense as *isDriverOf* from *Person* to *Train*, and *hasForDriver* in the other direction. An error on the person's address must not be transmitted to the train. A person's problem, which implies that the special button has not been pushed for a too long interval, must be transmitted to the train[3].

Second, exceptions can arise in a class as a result of the definition of the association itself. For instance, suppose we have a *birthDate* association between classes *Person* and *Date*. A well-formed date causes no error in the *Date* class. As a *birthDate* for a *Person*, it can cause error, e.g. if it is later than the birthdate of one of the person's children.

To sum things up, we find in a class two kinds of exceptions: ones which are generated by an exception defined on the other end of the association (one may say that they "encapsulate" an exception defined on the other end), others which are the result of the association's semantics (of the role the association plays in the class).

In practice, implementing an association means defining attributes and methods in its ends' classes. We can thus enrich the hierarchy we talked about in the Paragraph 2: under the main root *ExcTypeX*, we add roots corresponding to each attribute or method able to have exceptions. As for the primitive attributes, exceptions associated with an attribute means exceptions its accessor methods can signal.
For each attribute *attrY* of Y type (or of collection of Y type, according to the multiplicity of the association), we can add under its root *ExcAttrY* two subclasses *ExcCapTypeY*, root of the exceptions which are intended to "encapsulate" Y's exceptions, and *ExcRoleY*, root of the exceptions which result from the role of *attrY* in the X class (see Figure 2).

In some cases, it can be useful to insert, between *ExcTypeX* and the roots corresponding to attributes and methods implementing the association, a root *ExcAssoc* which represents the association itself (dotted in Figure 2).

4 Exceptions and specialization

Modeling
It seems natural that the hierarchy of the exception classes reflects the hierarchy of the classes of the application. For example, let us consider a *PostObject* class, which has two subclasses *Letter* and *Parcel*; every exception which can occur in every *PostObject*, can a priori occur in a *Letter*, and additional or more specific exceptions can occur in *Letter*.
Figure 3 shows the general diagram of such a hierarchy of exceptions.

[3]Dead man system. In very fast trains, the driver must push a special button at regular intervals. If he doesn't, he is presumed dead, or at least unable to drive, and the train's braking system is automatically started.

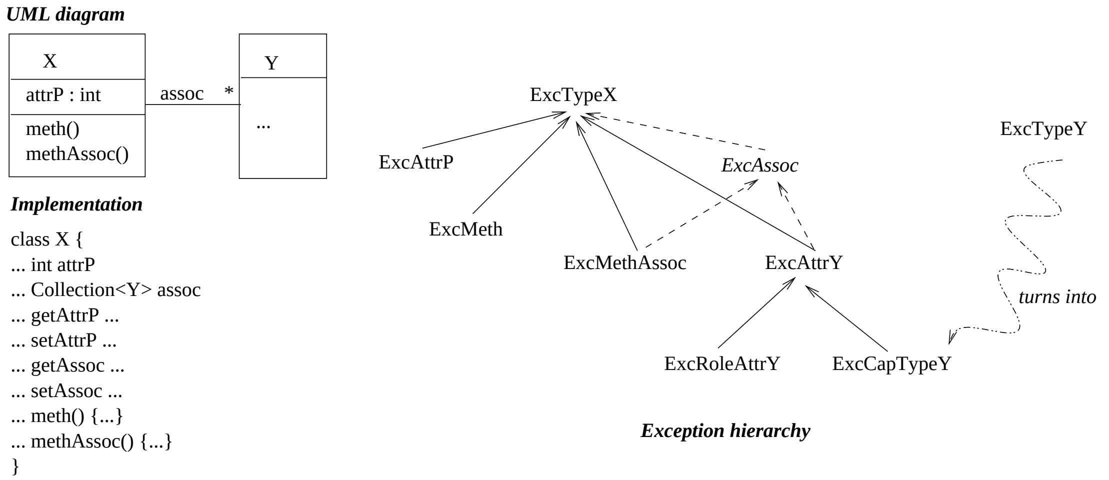

Figure 2: General diagram of a hierarchy of exceptions associated with a class' features

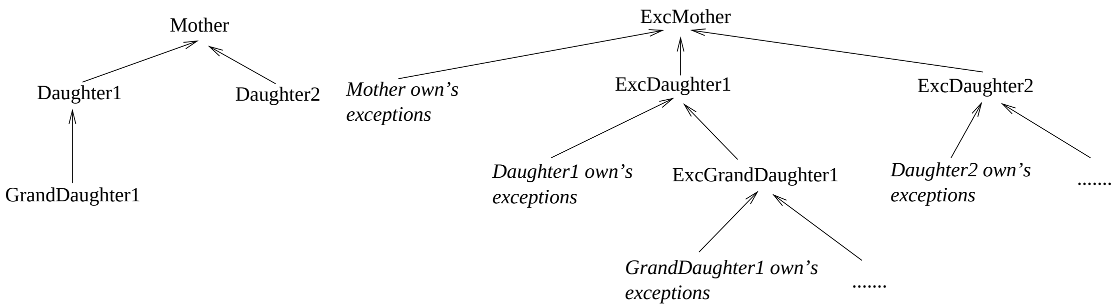

Figure 3: General diagram of a hierarchy of exceptions associated with a hierarchy of classes

In order to refine this first approach, we can split up the exceptions associated to a *Daughter*[4] class as follows:

- exceptions associated with *Daughter*'s own features;
- exceptions associated with features *Daughter* inherited from *Mother*, themselves split up in:
  - exceptions identical to the *Mother*'s ones, thus not redefined; these exceptions do not necessarily generate classes under the *ExcDaughter* root;
  - exceptions which are specializations of the *Mother*'s ones;
  - exceptions associated with new constraints on the inherited features.

The part of this decomposition showing inherited features is illustrated by the figure 4; a concrete case, where we need only a subset of the hierarchy, is shown by Figure 5.

Subtyping

One can define subtyping using substituability: an instance of a *Daughter* subclass must always be able to take the place of an instance of the *Mother* class. In the case of exceptions, this implies that a method redefined in *Daughter* does not signal exceptions which the original method has not told itself it can signal.
From the modeling point of view, this is not satisfactory: when specializing a method, we see that constraints can be added, and thus new causes of exceptions can arise.

Implementation problems

Java [AG98] and C++ [Str98] respect substituability: they allow the *Daughter*'s method to signal at most the same exceptions as the *Mother*'s method, or exceptions which are subclasses of

[4]The *Mother*, *Daughter* names refer to the place of the class in the hierarchy.

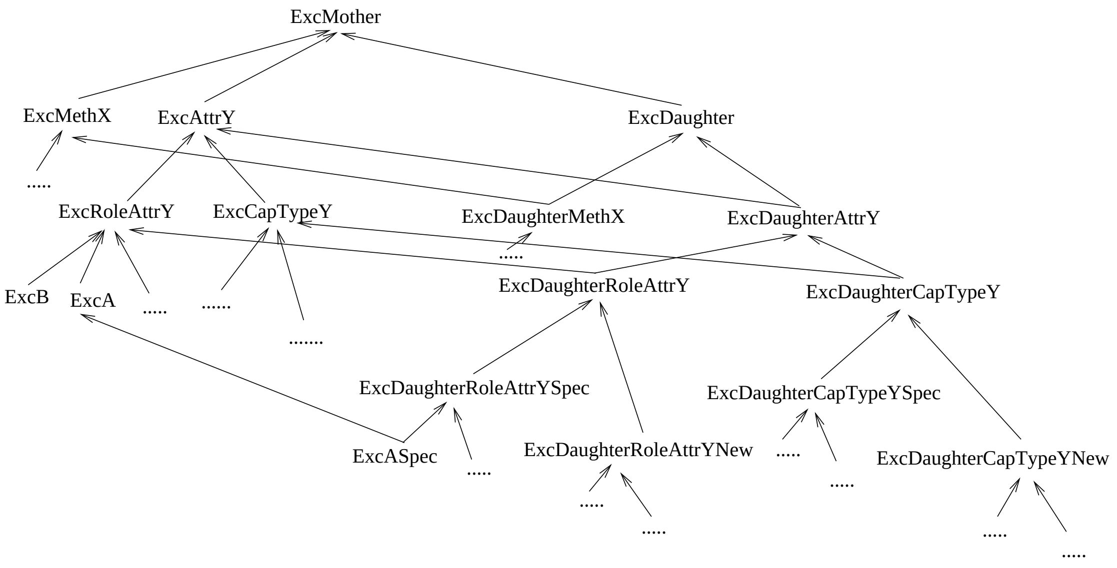

Figure 4: More detailed diagram for a hierarchy of exceptions associated with *Mother* and *Daughter*

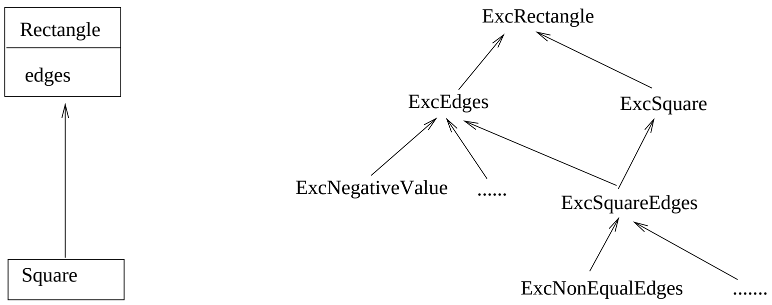

Figure 5: A concrete case

the exceptions the *Mother*'s method can signal.
All the "root" exception classes we described above are abstract classes, and we used them in order to classify the concrete exceptions found during the modeling phase. We will also use abstract classes as an implementation artifice, in order to respect substituability.
Let us look one more time at Figure 5[5]. Suppose that the *Rectangle* class has a *setEdge* method, which can signal an *ExcNegativeValue* exception; suppose that the *setEdge* method redefined in *Square* can signal, in addition to that, an *ExcNonEqualEdges* exception. The two abstract classes *ExcEdge* and *ExcSquareEdge* found during the modeling phase allow us to respect substituability: every *setEdge* method will announce in its signature that it can signal *ExcEdge*.

When we want to implement this inheritance hierarchy in C++, there is no problem, because C++ allows multiple inheritance. The situation in Java is not as good; here we must use all the usual means in order to simulate multiple inheritance [RBP+91]: duplicating properties and classes, using interfaces.

5 Composition of exceptions

Another problem we met is the case of errors which can occur separately or simultaneously. For instance, in a *Date* class, only the day may be wrong, or only the month, or both; in a *DrinksDispenser*, one can lack separately or simultaneously cups, sugar, spoons, drink doses.

[5]This example does not intend to answer the old question "must or must not *Square* be a subclass of *Rectangle*"; how could we give this answer context-free? It only shows how one can organize exceptions if one wishes *Square* to be a subclass of *Rectangle*.

When errors occur simultaneously, we can throw an exception corresponding to the first one we detect. However it would be helpful, e.g. for the dispenser manager, to know all the problems which have occurred.
This leads us to add, in some cases of modeling, the notions of single and composite exceptions. These notions are an application of the *Composite* pattern proposed in [GHJV94], as shown in Figure 6. This allows us to represent every single exception (*Leaf* in the *Composite* pattern), and also an exception composed of several exceptions (*Composite* in the *Composite* pattern).

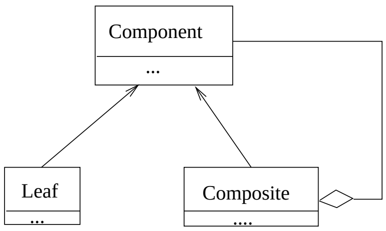

Figure 6: The UML *Composite* pattern

6 Proposals by way of conclusion

Discussion
The search for exceptions involves much modeling work; the result can be complicated, and difficult to implement in some languages.
However, when the final result is complicated, it often means that the problem we wanted to model was complex. This happens particularly when we want a software to be as reliable as possible.
For everyday case, the above proposals make up a kind of guide for modeling: there's no need to implement everything. Furthermore, even if the implemented hierarchy of exceptions is complex, it is well separated from the hierarchy of the classes of the application. Thus it does not obscure the main development. Moreover, this makes future developments of the software easier (e.g. see Paragraph 4 about problems linked to subtyping).
Large hierarchies of exceptions help us to represent accurately, and also to catch accurately, all error cases, since the exceptions are caught according to their types ([AG98]).

UML notations and methodology
We first make an inventory of elements we lacked in UML.
As we have said at the beginning of this paper, exceptions in UML are not recognized as *Classifiers*, so they can't have attributes, methods or associations. This is one of the first points to modify, at least to be compatible with programming languages (which are ahead of modeling language in that respect).
A simple solution would be that the *Exception* metaclass becomes a *Classifier* specialization in the metamodel.
Still in the metamodel, we would like to add:

- an *isExceptionFor* association between *Exception* and *Classifier*, meaning that an exception belongs to the exception hierarchy associated with a given type;
- an *isCompatibleWith* association on *Exception*, meaning that two exceptions can belong together to a composite exception;
- a *generatesException* ternary association between *Exception* and *Association*; an instance of this association would be defined in a model between two exceptions *E1* and *E2* and one association *A* in order to say that the exception *E1* and the association *A* generate the exception *E2*;
- a specialization of the *Exception* metaclass by *SingleException* and *CompositeException*, in order to apply the *Composite* pattern cited above.

Figure 7 shows these proposals in the UML model.

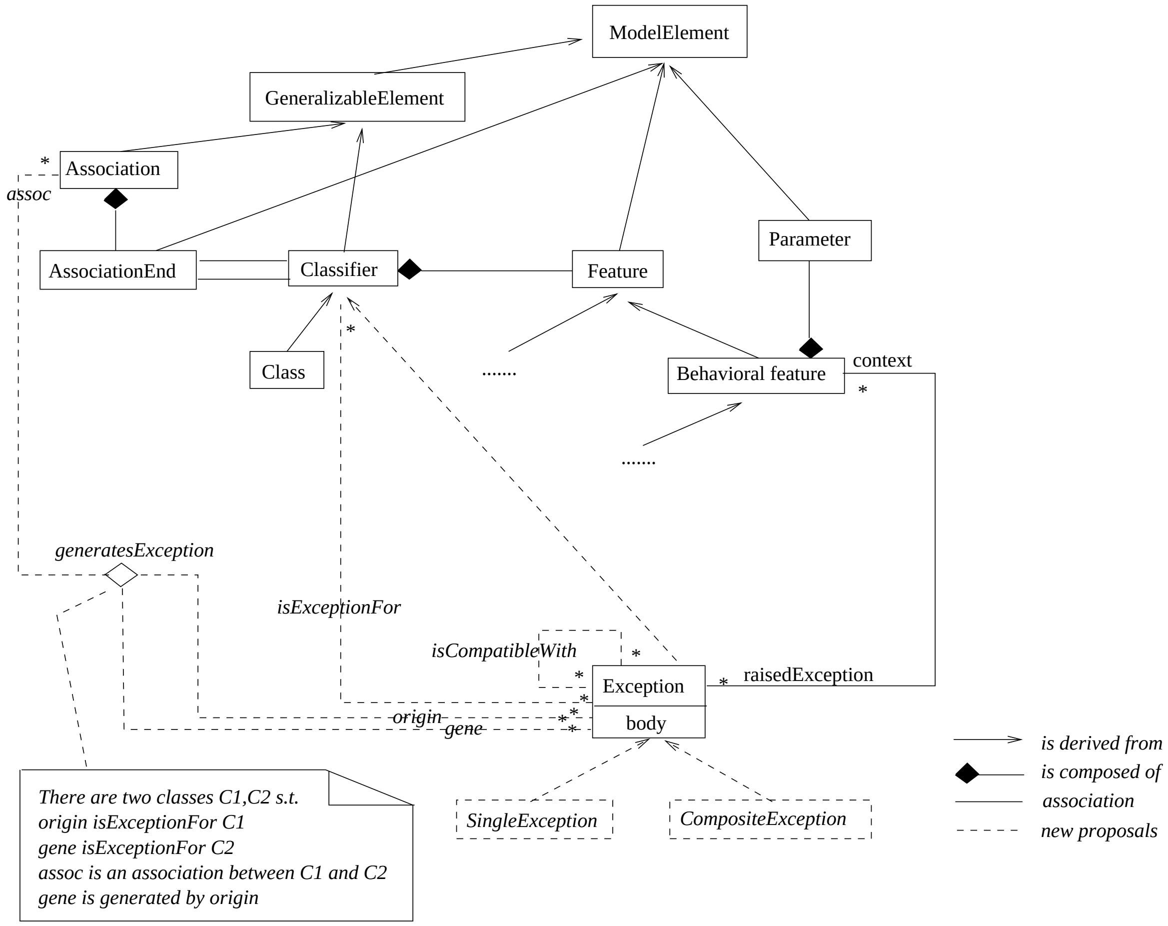

Figure 7: Some proposals about the UML metamodel

From the methodological point of view, we would like to have rules which help us to determine what kinds of exceptions we must search for to associate with a class, e.g. rules based on the nature of the class' features, especially that of its associations (aggregation, symmetrical or dissymmetrical relation).

Open questions
From a more general point of view, many questions arise, among other ones:

- how can we associate the classification we propose, based on model elements, and other classifications built along other criteria (e.g. the classic "fatal and recoverable exceptions", or `logic_error`, `runtime_error`, ... as in C++[Str98])?

- can we model exceptions without having in mind a particular exception handling mechanism?

As a conclusion, it seems to us that there is a fundamental work left to do, in order to define a methodology for the conception of exceptions, and for an appropriate associated notation.

7 Acknowledgements

We would like to thank Christophe Dony, Roland Ducournau, and Thérèse Libourel for their comments and suggestions; we also thank Anne Berry for her help with our English.

References

[AG98] K. Arnold and J. Gosling. *The Java Programming Language. Second Edition.* Addison-Wesley, 1998.

[Don90] C. Dony. Exception Handling and Object Oriented Programming: Towards a Synthesis. *ACM SIGPLAN Notices*, 25(10):322–330, october 1990. Proceedings of ECOOP-OOPSLA'90.

[DPW91] C. Dony, J. Purchase, and R. Winder. Exception Handling in Object Oriented systems: Report on ECOOP'91 Workshop W4, 1991.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[Rat97] Rational Software Corporation. *UML v 1.1, Semantics*, septembre 1997. version 1.1 ad/97-08-04.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice Hall Inc. Englewood Cliffs, 1991.

[Str98] B. Stroustrup. *The C++ Programming Language. Third Edition.* Addison-Wesley, 1998.