A problem in querying recursive patterns with OQL

Cédric Raguenaud, Jessie Kennedy, Peter J. Barclay
School of Computing
Napier University
10 Colinton Road
Edinburgh EH10 5DT
United Kingdom
{c.raguenaud, j.kennedy, p.barclay}@napier.ac.uk

Abstract

This paper analyses the problem generated by recursive patterns in typed query languages such as OQL [Cattell '97]. Recursive patterns describe hierarchical structures such as those defining classifications or part-explosion problems. A classification is composed of hierarchies of classes that eventually classify non-class objects. A part-explosion problem is composed of parts that contain other parts, thereby describing a complex object. They are commonplace in database schemas and software models but make it impossible to answer some queries that require the use of either or both attributes from classes and classified objects or attributes from different kinds of parts. The possible approaches to solve this problem are presented and discussed, and a solution proposed. This solution involves the creation of a simple well-defined operator that allows the expression of a type selection coupled with casting and error handling facilities. This way, it becomes possible without type error to query sub-classes of a particular type without writing programs.

Keywords: digital libraries, object-oriented databases, querying, OQL, patterns

1. Introduction

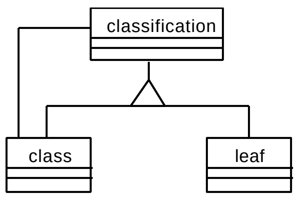

Figure 1: The classification pattern

Recursive patterns occur often in database schemas. Such patterns represent for example part-explosion problems, where parts may contain other parts to describe a complex entity (e.g. car parts); classification problems, where classes may contain other classes and classify other objects (e.g. library classifications); or composite patterns [Gamma '94] where a generic interface is created for all the classes recursively involved in the pattern.

Our interest is with classification problems, as found in plant taxonomy. The problem arose in the Prometheus project [Pullan '00] because the main component of a taxonomic database is a nested structure based on a recursive pattern. It became clear during the project that some of the queries taxonomists would be interested in could not be resolved using a typed query language such as OQL. Recognising that this a general problem, we have therefore developed the concept described in this paper.

Recursive structures such as those used to represent classifications are often modelled as shown in figure 1 (e.g. Prometheus [Raguenaud '00]). Figure 1 shows the minimal pattern necessary to handle such structures (which we will refer to as a classification pattern), where three classes are required: a *class* class, that will represent the classes that appear in the classification; a *leaf* class that is the generic class for classified objects; and a *classification* class, that subsumes the other two. A relationship between the *class* class and the *classification* class captures the recursive aspect of the problem. This relationship can be either a general association or an aggregation, depending on the semantics of the classification scheme (Odell [Odell '94] points out that some classifications are not member-bunch-based models whereas we model our classifications via aggregation). *Classes* can contain other *classes* or *leaves*, whereas *leaves* cannot contain other objects.

If such a pattern is used in a schema, one expects the query language offered by the database system to be able to query the pattern fully. For example, if such a pattern is used to create a library classification of books, it seems natural to expect the query language to support queries such as “extract the books published by Kennedy in the category fiction”. However, this kind of structure generates a type problem in typed query languages like OQL [Cattell '97] and it is impossible to query the different classes appearing in the pattern. As OQL is intended to be a standard query language for OODBs, this is can be considered a syntactical weakness.

We first define the problem in more detail in section 2 and then we examine the solutions that can be considered and expose their weaknesses in section 3. We then propose our solution in section 4, we show how the problem is a generalised problem in OQL and how our solution can be applied in section 5, and we conclude in section 6.

2. The problem

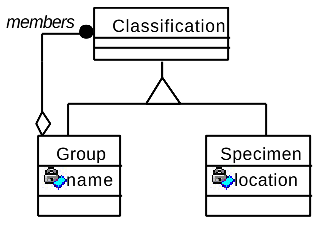

Figure 2: Example

Figure 2 shows a classification pattern applied to biological classifications.[1] Classifications are composed of *Groups* that contain other *Groups*, and eventually *Specimens*. *Groups*, that represent biological *taxa*, have a name and other attributes such as a type and a publication, but here we use only name for simplicity. *Specimens*, that represent physical biological specimens, have a location of collection and other attributes such as a group of collectors, a barcode, a herbarium code, but only location is represented in our example. As the analysis of the domain justifies, the relationship between groups is represented by an aggregation relationship. *Groups* and *Specimens* are semantically two entirely different concepts, however, when the classification pattern (or any other recursive pattern such as a part-explosion pattern) is used, logically they may serve the same function (as things that can be classified) and therefore can be seen as the same kind of object (a *Classification*) in some circumstances. When queries are used in this kind of schema, the type consistency of the database forbids the utilisation of attributes only defined in the sub-classes when

[1] The reader is referred to [Pullan '00] Martin R. Pullan, Mark F. Watson, Jessie B. Kennedy, Cedric Raguenaud and Roger Hyam, *The Prometheus Taxonomic Model: a practical approach to representing multiple taxonomies*, Taxon, 49 (1), pp 55-75, 2000 for more information on biological classifications.

the super-class (*Classification* in our example) is targeted by relationships. For example, when querying sub-*Groups* from a *Group* object point of view, only attributes defined in the *Classification* class can be used. Therefore, it is impossible to query the name attribute of a sub-*Group*. For example in our schema, the query

```
select g from Group g where g.members.name = “X”
```

would generate a type error because name is not an attribute of *Classification*, only an attribute of one of its sub-classes.

Although this is a good use of object-oriented principles (only inherited attributes can be seen when polymorphism is used), it is a great limitation that renders impossible the utilisation of particular schemas in practical applications. For example, in our schema, it is sometimes important to retrieve the name of a *Group* for constraint checking purposes (e.g. *Group* names should follow certain rules), for extraction of meaningful information (e.g. retrieving all *Groups* a specific *Specimen* identified by its location belongs to), or for simply searching the database. Using standard query languages such as OQL, these queries are impossible, and therefore some information is impossible to retrieve using the query language. The only solution is to write a program in a high-level language such as Java [Sun '95] or C++ [Stroustrup '91] that simulates the execution of a query.

In object-oriented programming languages, it is possible to “cast down” objects when the type of the objects in hand is known, and it is possible to test the type of an object when it is not known. For example, if we know in advance that the objects we will consider are only *Group* objects, we can cast the retrieved objects to G*roup* and thereafter use the properties defined only for these objects. If we do not know in advance the type of the objects we find (for example, if *Classification* is used), it is possible to test the type of each object, and discard *Specimen* objects, or apply the correct operations on each of the objects retrieved. The drawbacks to this method are that writing programs is more complicated that simply writing queries, and using the programming language instead of the query language avoids any benefit from the optimisations a query language might offer.

Since this problem occurs frequently in querying object-oriented schemas, it would seem that an extension to the object query language is required. Indeed, we could view it as a deficiency of the ODMG model that it can be used to define schemas that cannot be queried. Clearly, an equivalent of the type test in programming languages (e.g. *instanceof* in Java [Sun '95]) is required. However, type test operators are usually used in conjunction with an “if” statement in programming languages, but there is no equivalent of “if” in OQL and many other query languages.

3. Towards a solution

The pattern presented in section 2 appears to be a good representation of the application domain and captures well the available information. However as discusses, when querying such a pattern, problems arise because object-oriented environments ascribe considerable importance to type correctness. Therefore in order to use existing object-oriented databases with OQL, two kinds of solutions can be considered: the decomposition of the path expression using joins and changing the schema so that queries are type correct.

3.1. Joins

A way around the problem can be found by decomposing the path expression into many smaller path expressions and a join. It is possible to express queries such as the one presented in the example above as follows:

```
select g2 from Group g1, Group g2 where g1.name = "X" and g2.members = g1
```

which extracts *Groups* that contain the sub-*Group* called "X" at the next level in the classification.

However, this notation breaks path expressions and is unnatural, as it becomes impossible to write path expressions to traverse a graph of objects. Moreover, when the schema uses inheritance heavily queries become hard to write and understand. As can be seen, our extremely simple example requires the definition of two variables and a join.

More classification levels might be required in the query, for example to extract the groups that contain another group that contains the group called X (e.g. when the number or position of levels in classifications is important, as it is in biological classifications). For each new level added, more variables and joins are necessary, making the query even harder to write and understand.

For more complex queries such as recursive queries defined using extensional object-oriented languages (e.g. [Kifer '92], [Raguenaud '00]), i.e. queries that recurse down a classification until it finds specific objects, the notation above is not suitable as it would be impossible to specify recursive statements involving the join.

Finally, inefficiency could be an issue, as joins may be more expensive than following a path.

3.2. Changing the schema

Another solution is to change the schema so that querying becomes possible from a type point of view. Four approaches can be considered: the move of specific attributes to super-classes, the avoidance of inheritance and the creation of additional relationships, the use of the composite pattern, and the use of non object-oriented mechanisms.

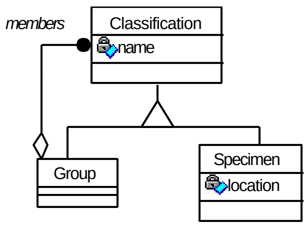

Figure 3: First solution

a) Moving attributes to super-classes: one partial solution to the problem caused by the query presented in section 2 would be to move the attributes queried to super-classes. In figure 3, the name attribute is defined on the *Classification* class, thereby implying that *Specimens* might have a name. As this is not true, the management of the attribute has to forbid this association dynamically in the *Specimen* class. Now, the query

```
select g from Group g where g.members.name = "X"
```

is sensible from a type point of view. But this solution requires that all the queries that will be run are known at design time, which is unlikely. It also implies the addition of properties to all the sub-classes of the *Classification* class (e.g. *name*) that do not necessarily make sense for some of the sub-classes. For example *Specimens* do not have a *name*.

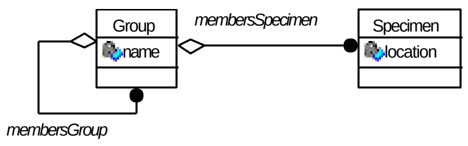

Figure 4: Second solution

b) Avoiding inheritance: we could also change the schema in order to avoid the use of the super-class when attributes need to refer to sub-classes of the classification (figure 4). In this solution, the *members* relationship is abandoned and two new relationships are created: one between *Group* and itself for members relationships that involve only a *Group* object, and one between *Group* and *Specimen* when the members are *Specimens*. Now, it is possible to query the database and choose the relationship or the type we want. For example:

```
select g from Group g where g.groupMembers.name = "X"
```

is valid because the *groupMembers* relationship only involves *Group* objects. The type of the query is correct. However, as can be seen, a new relationship has been created between Group and Specimen to replace the genericity provided by inheritance. If many classes are involved in the pattern (e.g. not only *Specimens* are classified), new relationships between *Group* and these other classes are necessary. It quickly becomes hard to manage all these relationships and to choose how to query them.

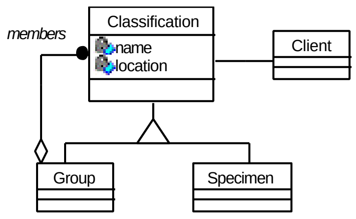

Figure 5: The composite pattern

c) Composite pattern: the schema could be changed to a composite pattern [Gamma '94] (figure 5). The composite pattern allows the definition of recursive structures with a single interface for the client. That way, the *Client* that uses the recursive structure does not need to know whether the object currently manipulated is a *Group* or a *Specimen*. A super-class for *Groups* and *Specimens* is created and needs to contain the accessible interface of both *Group* and *Specimen* objects. The following query then becomes possible:

```
select g from Group g where g.members.name = "X"
```

as all possible objects encountered would have a name property.

This solution is seductive, as the composite pattern is well known, but it has problems. It makes dubious use of inheritance as the super-class created contains all the properties of its sub-classes, which consequently means that all sub-classes inherit the whole interface. This may lead to nonsensical situations where objects need to respond to messages that are meaningless for them. For example in figure 5, the composite requires that *Classification* contains both properties name and location, therefore both *Specimen* and *Group* inherit these properties. However, name does not make sense for a *Specimen* and the method has to return an artificial value or no value at all, and location does not make sense for a *Group*.

d) Non object-oriented solution: another approach could be avoiding the use of inheritance in the pattern. In many object-oriented systems (e.g. Java), genericity is only supported by inheritance. In other object-oriented languages (e.g. C++, Eiffel [Meyer '92]), genericity is provided by generic classes or templates which can be parameterised by a particular class. Therefore, we could think of another approach to genericity that would not involve inheritance. Union types as in [Raguenaud '99] are a good candidate. Union types are types that are created from the union of a series of existing types. For example, in our sample schema, a union type *Classification* could be created by unioning *Specimen* and *Group*, thereby generating a new type that would create a logical relationship between these types.

However, union types have the disadvantage of not integrating well in an object-oriented environment because of their weak typing. Furthermore, querying union types involves ignoring the type of the classes that are part of the union, and querying blindly only on the name of the attributes used. Although it does not change the meaning of an object-oriented system or its querying, it requires making an exception in the treatment of types when a union type is encountered in a query. Indeed, we do not know what type we are using until we have found a representative object. It also hinders any kind of optimisation, which might compromise the evaluation of queries.

Given that our modelling of the problem is most accurately the one presented in section 2, these four solutions have the disadvantage of requiring the modification of the schema. They also either introduce incorrect concepts (*Groups* do not have a location of collection), lose of a form of genericity (if many classes are involved in the classification pattern, a new relationship between Group and each of these classes needs to be created to replace the generic relationship to the super-class), or the introduction of non type safe non object-oriented mechanisms. Moreover, in many existing applications, we do not have the liberty of altering the schema or schema evolution [Banerjee '87] may not be possible or too expensive.

4. Proposed solution

The solutions presented in section 3 are not entirely satisfying: each of them has drawbacks, and we believe the pattern presented in section 2 is the most suitable for classification patterns from a modelling point of view. The only solution is therefore to avoid changing the schema and extend the query language so that it can deal with our pattern.

It is possible to extend OQL with a simple well-defined operator that will allow the examination of sub-classes in normal queries. The meaning of this operator needs to be "I want to use one and only one of the sub-classes of the *classification* class because I want to refer to one of its attributes". The purpose of this operator is therefore triple: first it is a type selection operator (the object must be an instance of the chosen class/type), secondly it is a cast operator (so that queries are type safe and attributes can be accessed), and thirdly it handles errors so that evaluation of queries is not impaired (regular cast operators would stop query execution when an object of an undesirable type is encountered). We have extended OQL in our implementation of POOL [Raguenaud '00] and represent this operator as square brackets after the attribute it targets. It tests the membership of an object to a particular class, but in addition, it only keeps objects of that class in the query.

For example:

```
select g from Group where g.members[Group].name = "X"
```

can be explained as follows: select all *Group* objects that have a *members* relationship only to other *Group* objects, then check that the attribute name of these *Group* objects is X; (silently) discard *Specimen* objects.

The square brackets operator is not simply a cast operator (which already exists in OQL). A cast operator would mean that the objects found are cast to a particular type, then used in the query. But casting objects would create a type error if *Specimen* objects were used, therefore invalidating this query.

The square brackets operator acts in two parts of the query: when objects are examined (selection) and when they are extracted (projection). It is then possible to extract only an object belonging to a particular type when they are involved in a relationship. For example, extracting all the *Specimen* objects that are used in classifications (referred to by *Group* objects) can be performed as follows:

```
select g.members[Specimen] from Group g
```

Or finding the location of collection of all *Specimens* placed in a *Group* of name “X”:

```
select g.members[Specimen].location from Group g where g.name = “X”
```

5. Generalisation

Although we describe this problem in the context of the classification pattern, it is a general problem when sub-classes are chosen in path expressions (e.g. part-explosion problems).

The problem we have presented in this paper also occurs in ODMG bindings where parameterised polymorphism does not exist (e.g. the ODMG Java binding). In ODMG, collections are typed objects, i.e. they contain objects that belong to a particular type/class or to its sub-types/sub-classes. However, in the Java binding, because collections are not parameterised objects, all objects in a collection are of type *any* (from the ODMG point of view) or *Object* (from the Java point of view). This leads to the inability to query collections accurately. As it is impossible to select sub-classes of the class Object, it is not possible to reach their attributes. For example, the following query is not possible, if as above, members is a collection:

```
select g from Group g, g.members a where a.name = “X”
```

The variable *a* is of class Object in Java because g.members is of type collection, therefore the attribute is not defined in its type. In our system, the following statement is valid:

```
select g from Group g, g.members[Group] a where a.name = “X”
```

6. Conclusion

This paper has presented the problems that recursive patterns such as the classification pattern, along with other common patterns (part-explosion, composite pattern), generates in query languages. Because of its nested structure and because of the type system of OQL, this pattern cannot be fully queried, and sometimes generates situations where it is impossible to extract some information without writing a program to simulate the query. This problem can be avoided using the solutions we have proposed in section 3, but their use implies change to and redundancy in the schema, and loss of optimisation in the evaluation of the query. The problem also appears for example in ODMG bindings where collections are queried and parameterised polymorphism is not available (e.g. the Java binding).

Our solution involves the addition of a simple operator in OQL that allows the explicit declaration of the type used when a composite pattern is queried, thereby providing a means of querying sub-classes of the composite whilst keeping a query type safe. Since the use of our new operator is consistent with the type system, it does not affect the ability of the query engine to optimise queries. As the square bracket operator enforces type, type-based optimisations are still possible.

This extension has been incorporated in the POOL query language in the Prometheus extended object-oriented database [Raguenaud '00] used in the domain of plant taxonomy. The new operator has been vital to answering questions posed by taxonomists regarding their classifications.

7. References

[Banerjee '87] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk and N. Ballou, *Data model issues for object-oriented applications*, ACM Transactions on Office Information Systems, 5 (1), pp pp 3-26, 1987

[Cattell '97] R. G. G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland and Drew Wade, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, Inc., I.S.B.N. 1-55860-463-4, 1997

[Gamma '94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, I.S.B.N. 0-201-63361-2, 1994

[Kifer '92] Michael Kifer, Won Kim and Yehoshua Sagiv, *Querying Object-Oriented Databases*, Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, pp 393-402, 1992

[Meyer '92] Bertrand Meyer, *Eiffel: the language*, Prentice Hall International (UK) Ltd, I.S.B.N. 0-13-247925-7, 1992

[Odell '94] James Odell, *Six different kinds of composition*, Journal of Object-Oriented Programming, 6 (8), pp 10-15, 1994

[Pullan '00] Martin R. Pullan, Mark F. Watson, Jessie B. Kennedy, Cedric Raguenaud and Roger Hyam, *The Prometheus Taxonomic Model: a practical approach to representing multiple taxonomies*, Taxon, 49 (1), pp 55-75, 2000

[Raguenaud '99] Cedric Raguenaud, Jessie Kennedy and Peter J. Barclay, *The Prometheus Database Model*, Napier University, Edinburgh, UK, Prometheus report #2, 1999

[Raguenaud '00] Cedric Raguenaud, Jessie Kennedy and Peter J. Barclay, *The Prometheus Taxonomic Database*, IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE 2000), Arlington Virginia, USA, pp 63-70, 2000

[Stroustrup '91] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, I.S.B.N. 0-201-53992-6, 1991

[Sun '95] Sun, *"The Java Programming Language (white paper)*, Sun MicroSystems Inc, 1995