Implementation of the BLOOM data model on ObjectStore

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Alberto Abelló & Fèlix Saltor

April 23, 1999

Contents

Chapter 1

Introduction

These pages try to explain the implementation of the BarceLona Object Oriented Model (BLOOM for short) on an ObjectStore DBMS (DataBase Management System).

BLOOM is a semantic extension of object oriented (O-O) models, with a high degree of expressiveness. It was designed specifically to satisfy all desirable characteristics found in the analysis [SCGS91], both in expressiveness and in semantic relativism. That paper shows the needed characteristics of a data model to be used as Canonical Data Model (CDM) of a federation of databases.

The BLOOM model is similar to other O-O models. However, since it is an enrichment, it has things others do not have. Thus, it needs some extra features to be implemented. That is the aim of this work, improving an O-O DBMS with some code to get the desired effect.

To avoid beginning from zero, the implementation was done on a commercial object oriented DBMS. ObjectStore was chosen as the DBMS because it is one of the best products. All needed features were implemented on it to simulate a true BLOOM DBMS.

The main non-functional requirements for this work have been flexibility and modifiability. BLOOM is evolving, yet, and this is just the first version of the implementation. Therefore, it is desirable preparing it to be easily modified. Obviously, this implies having good documentation about the work done and full comments in the code, let alone its clarity. On the other hand, since this is not thought as a commercial system, performance was never a weighty argument to make a decision.

The paper is organized as follows. The second chapter contains the explanation of the improvements done on ObjectStore in order to enrich it. Each section explains a BLOOM feature and the decisions taken to implement it. The third chapter shows the BLOOM DBMS wrappers (i.e. *DBMS*, *Database*, *Transaction* and *Root*). They try to hide the existence of ObjectStore to the BLOOM user. The fourth chapter lists the user classes already defined, some needed to implement BLOOM features common to all user classes and others with the pseudo-primitive types (i.e. *Boolean*, *Int* and *String*). Finally, the fifth chapter contains the conclusion and future work. The last pages contain the bibliography and two appendices. The first appendix contains some programming standards adopted, while the second gives some clues about getting an executable file from the source code.

Chapter 2

Implementation issues

2.1 Multiple and dynamic classification

BLOOM allows you to work with multiple and dynamic classification. This means that once an object has been created it can be inserted in any other class, even if it is not a subclass of the bottom class it was inserted before. When an object is inserted in a new class, it does not lose any property it had for being an instance of the classes it was already in. For example, if a student, John, finds a job, he becomes an employee. However, it does not imply he is not going to study any more. John could work and study at the same time. Therefore, the object representing John should be an instance of both, *Student* and *Employee* classes. That object should be allowed to be inserted in or deleted from any of its classes in order to represent the real world.

This situation creates a really hard problem because programming languages do not support this kind of situation, where an object can dynamically be an instance of multiple classes that are not implied using a superclass hierarchy [MO92].

The first solution proposed in the bibliography is to have a subclass for every possible combination of classes. When an object, already in a given class, is inserted in a new one, the system creates an instance of the subclass of both, copies the data and destroys the old one. The main problem for this solution is the combinatorial explosion it generates in the number of classes needed, as shown in [MO96].

That problem is almost solved in BLOOM due to the definition of different types of Generalization and Specialization (see [CSGS91]) which limits that explosion thanks to the ability of the database designer. However, even when the combinatorial explosion is solved, some other problems remain. What happens with the references to the old object? They should point now to the new one. What happens to the object identifier? It should not change. How do I copy the information from an object to another if I do not know the attributes in advance? Maybe byte by byte as proposed in [GSR96].

Neither C++ nor ObjectStore help you to solve the problem in that way. Rather, they interfere with it. C++ has a special implementation of multiple inheritance which difficulties the copy of objects. Besides, ObjectStore does not provide a good, fast and clean way for schema evolution (probably, the best solution).

Another common technique that supports multiple and dynamic classification called *object slicing* is proposed in [MO96], [GSR96] and [Fow98]. In object slicing, an object with multiple classification can be thought of as being "sliced" into multiple pieces. Each piece is then distributed to one of the object's various classes.

In our case, these slices are implemented as subclasses of the *Slice* class and the "unsliced" version of the object is an instance of *BloomObject*, which is a subclass of *Container* (see figure 2.1). Each container keeps a set with all the slices that make up the object.

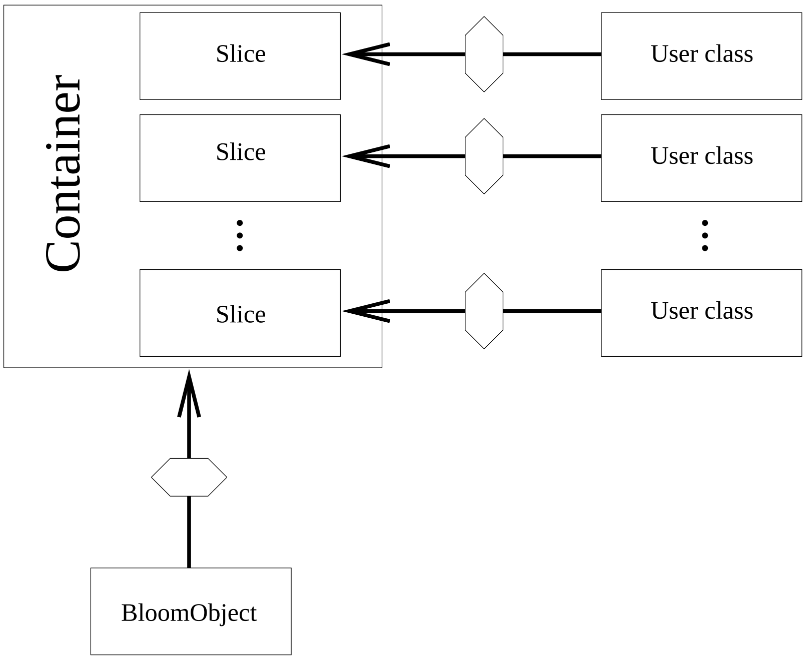

Figure 2.1: Container-Slices implementation schema

Following with our example about John, he would be represented by an instance of *BloomObject*. Since he is a student, it would contain an instance of the *Student* class, which is a subclass of *Slice*. When John finds a job, an instance of *Employee* (also subclass of *Slice*), should be inserted in the container. If later on, John is fired or quits his studies, it is enough removing the corresponding slice from the container.

Method calls

When a method is called, the instance of *BloomObject* delegates its execution to the corresponding slice. From here comes the problem of this beautiful solution, some extra code must be supplied to implement that execution delegation. This extra code is in the *BloomObject* class. Every existing method has an homonym in *BloomObject* which, following a inverse topological sort of the hierarchy graph, choose a slice and tries to call the method in that slice. Just in case somebody wants to directly call a method in a given slice, there is also defined a cast method from the *BloomObject* class to every slice class.

If the *Employee* class has a *riseSalary* method, the *BloomObject* class will have an homonym method. When the *BloomObject* method is called, it looks for the *Employee* slice, and calls its method.

There is a huge difference rising the salary of an employee or the salary of a boss. Therefore, the *Boss* class (subclass of *Employee*) would also have a *riseSalary* method ("overwriting" the previous one). Therefore, since *Boss* would be found first in an inverse topological sort, the *BloomObject::riseSalary* method would try to call *Boss::riseSalary* before calling the *Employee::riseSalary* method.

Inserts and deletes

Besides all those methods in the *BloomObject* class, every class in the model implements two methods (i.e. *insert* and *del*) which recursively create or destroy all the slices needed in order to put in or pull out a *BloomObject* from the given class. *insert* method will create slices of all the superclasses and *del* method will destroy all the slices of any subclass.

If John could pull wires at his new job, he could begin as a boss. However, since the *Employee* class is a superclass of *Boss* the *Boss::insert* method should create both, the *Employee* and the *Boss* slices, putting them into the container.

Slice identification

The slices of an object are kept together in a set. In order to identify the slices in a given set, the method *getClassName* is declared in the *Slice* class. That method uses the ObjectStore Meta Object Protocol

(MOP) to return the name of the class the slice belongs to. Since this method is used to query about the slices in the container, it is marked as a "query function" (see section 2.12).

An alternative implementation to using MOP would be a pure virtual method, implemented by every class, returning its name. It should be an static method, but C++ does not allow to define a virtual and static method.

The first way was chosen mainly because it gives less work to the precompiler and compiler in spite of it is, probably, much less efficient. Its efficiency depends on the ObjectStore implementation and the overhead of C++ virtual functions. The efficiency could be improved getting the name of the class only once in the constructor and keeping it as an attribute of the slice (it is a commitment between memory space and process time).

Attributes placement

As a side effect of this implementation, every class just contains its own data. Take into account a class does not contain the data in its superclasses. Therefore, you are not able to directly access those attributes. You have two different options to access them: explicitly cast the *BloomObject* to get the desired slice or declare two ad hoc methods (i.e. "setAttributeName" and "getAttributeName").

BloomObject has a cast method for every user class. This method returns the demanded slice. Once the slice is returned, the desired attributes can be accessed as usual.

The second possibility implies the user should declare a set and get methods for every attribute in the slice. The BloomObject will have the same methods, and the attribute will be accessible through them. A good choice for the names would be "setAttributeName" and "getAttributeName". If those methods are not defined, an explicit cast of the container will be mandatory to access an attribute.

Return values

Another important thing to take into account is that every method returning an object must return a reference to an object of class *BloomObject*. *this* points to a slice, hence, returning it means losing the object as a whole. A reference to the container is available in *Slice::container* to be used instead of *this*. Note it is important returning a reference or a pointer, never an object by value. If you return an object, it will be copied and the new copy will not be persistent.

2.2 Classification constraints

Depending on the kind of classification the user uses, the insertions and deletes of objects in the classes have different constraints as defined in [CSGS91] (see figure 2.2):

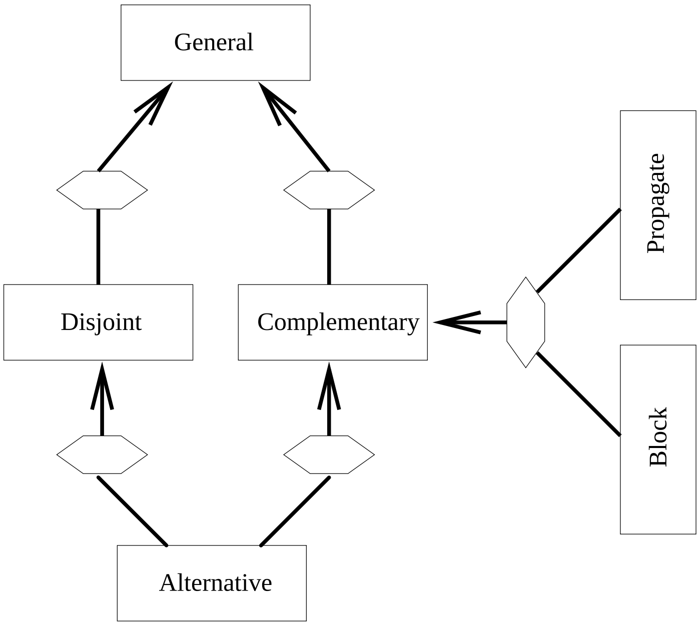

Figure 2.2: Kinds of Generalization/Specialization in the BLOOM model

Disjoint means each object of the superclass belongs at most to one subclass. A soccer supporter cannot support F.C.Barcelona and R.Madrid at the same time. Therefore, John instantiating *Barcelona* and *Madrid* violates an integrity constraint. John could hate soccer and not support any of them, but he is not allowed to support both at the same time.

Complementary means each object of the superclass belongs at least to one subclass. For example, a person should have a citizenship. The citizenship of John is Spanish because he was born in Spain. However, he could have more than one citizenship depending were his parents were born. So, if his parents were born in Chili, John would have Chilean and Spanish citizenship. Therefore, John should have at least one citizenship (where he was born), but could have more.
Depending on the behavior on delete, complementary classification has two different subtypes:

Block forbids the deletion of an object in a subclass if it is not an instance of a sibling subclass.

Propagate deletes the object from the superclass if it is not an instance of a sibling subclass.

Alternative is the conjunction of the previous two kinds (i.e. *disjoint* and *complementary*). Each object belongs to one and only one subclass. A person must be a male or a female. John is an instance of the *Male* class and he cannot be an instance of *Female* at the same time.
Note that, since an alternative specialization is complementary, it has the two different behaviors on delete, as well.

General does not have any constraints. An object being instance of a given class does not affects the same object being instance of another class. For example, John being an instance of the *Music* class in a "lovers" specialization, does not affect John being an instance of the *Tennis* class in the same specialization.

Those constrains imply doing something when a constraint is violated. Usually, an exception is thrown when it happens. The first choice implementing that is where are you going to throw the exception. This could be done in every single object. However, it also can be done in the relational way, in the transaction. You allow the user to do whatever he wants, but he cannot commit a transaction if the database is not consistent.

If you want to control every single object by itself, you get some problems. You must allow the object to temporarily violate some constraints, because it is mandatory to reach some consistent states (it cannot be inserted in or deleted from everywhere at a time). This is a problem because you have to define a special behavior for a "improper" object and check consistency at any time. Therefore, the best and simple way is controlling consistency just on commit and let the objects to temporarily violate the constraints in the transactions.

You also have different ways checking the constraints on commit. Firstly, you could check all the constraints for every object. Obviously, that is much more than you need in spite of it is really simple. Actually, an object just could violate a small set of constraints depending on which classes it belongs to ,was inserted in and was deleted from. Thus, just that small set should be checked.

When you insert or delete an object, depending on the kind of classification, if it violates a constraint, there are some other operations each one of them fixes it. This operations could be seen as pending operations. When any of these pending operations is done, the constraint is fixed. Note a constraint cannot be fixed if none of its pending operations is done.

In a *disjoint* specialization

Insert an object

In the subclass, violates the constraint if the object is in any other sibling subclass (pending delete from all but one of the sibling subclasses).

Delete an object

From the subclass, fixes the constraint if the object was in any other sibling subclass, i.e. the constraint was violated and now it is not (done the deletion from all but one of the sibling subclasses).

In a *complementary* specialization

Insert an object

In the superclass, always violates the constraint until the object is inserted in a subclass (pending the insertion in a subclass).

In the subclass, fixes the constraint if the object is not in any other subclass, i.e. the constraint was not already fixed (done the insertion in a subclass).

Delete an object

From the superclass, always fixes the constraint (done the insertion in a subclass).

From the subclass, violates the constraint if the object is not in any other subclass (pending insertion in a subclasses).

Since *Alternative* is the conjunction of the *Disjoint* and *Complementary*, its cases are the cases of both put together.

In order to check consistency on commit, the transaction needs to know how many pending operations it has. Every transaction has its own counter of pending operations and it does not commit if the counter is not zero. When an object is inserted in or deleted from a class, its number of pending operations is added to or subtracted from the counter of pending operations of the current transaction.

The problem of this implementation is the lack of information for the user. She[1] cannot know which are the pending operations of the transaction but how many. To offer that extra information, every transaction should know which objects where modified inside it. Every modification of an object would be entered in a list. This list could be in a database (persistent or transient) or in common memory. It should not be in a persistent database because it is slower and actually it is not persistent data and it cannot be in a transient database because you are not allowed to create neither objects nor roots. Therefore, it should be in common memory. Even in this case, it is a waste of time and memory just to keep user information.

[1] We use she for "he or she", her for "his or her"

Delete effect

As it was said above, the deletion of a class propagates it to its subclasses in any kind of specialization. However, if it is a *complementary* or *alternative* specialization, it could have to be propagated to its superclasses, as well. Obviously, this creates cycles in the deletion process. That cycles are easily broken marking a slice as being deleted. A deletion is not propagated to a slice being deleted. In the same way, it is not blocked if the superclass slice is being deleted.

The block exception must be thrown at the beginning of the method, before the slice is destroyed. However, the deletion is propagated to the superclass at the end of the method, after the destruction of the slice.

2.3 Classification operations abortion

In some circumstances a classification operation (i.e. insert and delete) must be aborted. It could be due to different reasons: *complementary* specialization constraint, existence dependency constraint or user exception. In any case, it is not desirable to have half object created or half object destroyed. Therefore, those operations are done inside a special internal transaction.

This internal transaction begins at the beginning of the first classification method (i.e. *insert*, *del*, *BloomObject::create* or *BloomObject::destroy*) called. If it already exists, it does not begin in the next call to any of those methods, only one of this transactions can exist at a time. This transaction is aborted when an *Exception* is thrown, or committed at the end of the method that began it.

2.4 Existence dependencies

As defined in [CSGS94] and [CSGS91], we consider nine different kinds of existence dependencies (see figure 2.3). It could also be seen as two orthogonal dimensions (see figure 2.4):

- Interest of dependent:
  - *Strict*: The existence of the dependent object strictly depends on the existence of the one on which it depends, called dependor here. This dependencies can reflect a situation where an object cannot exist in the real world independently of another one, or else an interest dependency where the dependent continues existing but it is not of interest anymore.

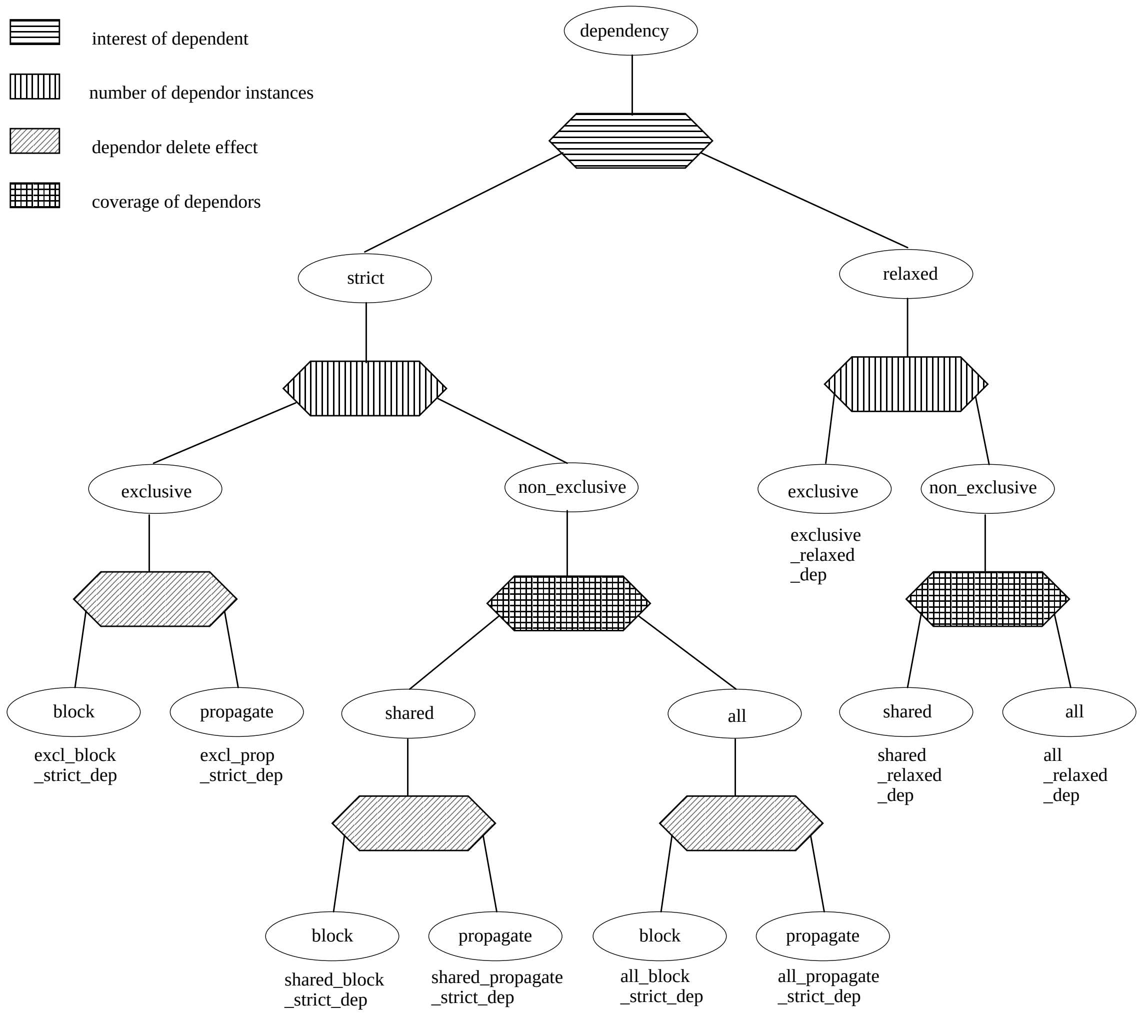

Figure 2.3: The existence dependencies in the BLOOM model

The insertion constraint linked to this concept assures that the dependent object cannot be created if the object upon which it depends has not been created before. With respect to the deletion constraint, this can have two different implications: in one case the deletion of the dependor object entails the deletion of the dependent object (*Propagate*) and in the other case the deletion of the dependor simply cannot be made if there are objects dependent on it (*Block*).

– *Relaxed*: It is a relaxed version of the strict dependency. The idea is that the dependent object cannot be created if the object on which it depends does not exist, but once it is created, its existence turns out to be independent of it. Therefore, its implications are only with respect to insertion, since the deletion of the dependor has no impact on the dependent.

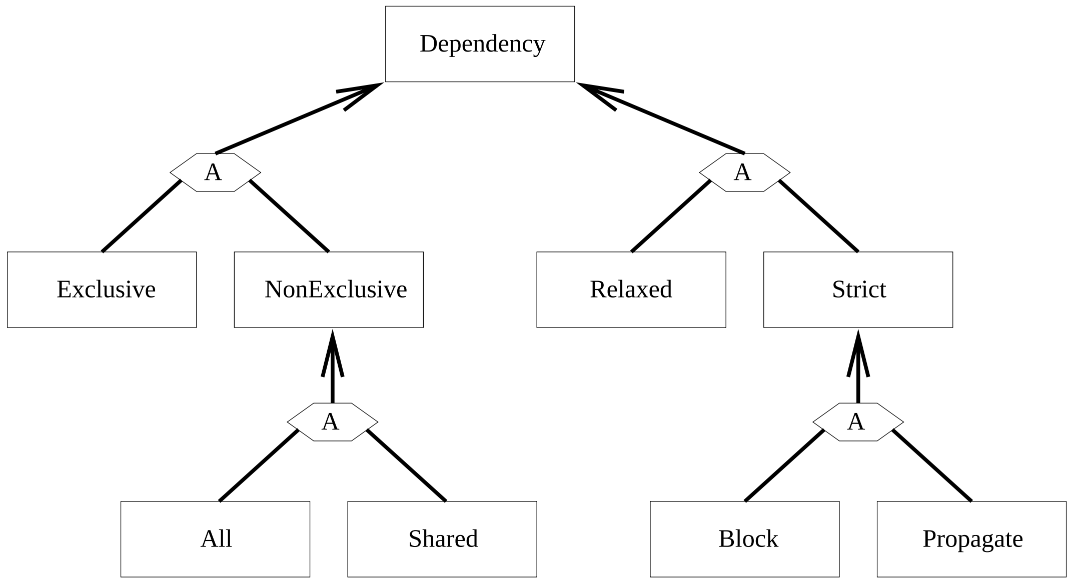

Figure 2.4: Existence dependency class hierarchy

- Number of dependor instances:
  - *Exclusive*: The dependent object depends exclusively on one instance of the dependor object class.
  - *Non exclusive*: This is the case that the dependent is shared by several instances of the dependor class leading to two kinds of dependency. In one of them, the dependent existence relies on that of at least one dependor instance (*Shared*). This implies that for its insertion it will be enough that one such instance exists, and the deletion of a dependor instance will be either blocked or propagated to the dependent only if it is the last dependor instance. In the other case, the existence of the dependent relies on that of all of the dependor instances and the deletion of any of them will either block or entail the deletion of the dependent (*All*).

A nice way to implement it is using the own model to represent the dependency. Since there are two orthogonal dimensions, they could be implemented as different branches in the class hierarchy (see figure 2.4). The methods implementing the behavior would be grouped in slices and these inserted in a container as needed. This approach is flexible and modifiable. However, it creates unnecessary overhead and all the slices are almost empty. It is much better implementing all together in a single class and change the behavior using flags.

The relation between objects could be implemented like a “m:n” relationship in the relational model. The dependency class should have the code of the kind of dependency it is, a set of dependors and a dependent. With that and some extra code in the *insert* methods of the user classes, the existence dependencies could be handled. However, everything is more natural if every class has its dependencies as data members.

Every dependor has to keep information about its dependents, at least in order to propagate the deletion. It could be enough keeping a set of dependents in the dependor side. However, if you define a symmetric class for the dependent side, which keeps the dependors, things are simpler and more coherent with the model. Moreover, doing it, dependencies are easily navigable. Actually, both sides have been implemented in the *BloomDependency* class.

Each dependent should have a *BloomDependency* as data member, and the dependors should have its symmetrical *BloomDependency*. Having them as data members, the objects are completely bound to their dependencies. You cannot create or destroy a slice without creating or destroying its *BloomDependency* data members.

Each *BloomDependency* in the dependent should match its symmetrical *BloomDependency* in the dependor side. It means you are not allowed to bind a dependency declared *All* and *Relaxed* in the dependent side if you declared it *Exclusive* and *Propagate* in the dependor side.

BloomDependency class has a parameter in the constructor saying the name of the slice containing the other side of the dependency. Another parameter indicates the distance in bytes from the beginning of the slice to the symmetrical *BloomDependency* instance. Using both parameters, a *BloomDependency* gets its symmetrical in a given *BloomObject*.

The dependors are passed as parameter to the constructor of the dependent side to force the existence of the dependors before the creation of the dependent. Just if the dependency is *Shared*, you are allowed to add or remove dependors from the set.

The delete effect is implemented in the destructor of *BloomDependency* class. Since the *BloomDependency* instances are part of the slices, when a slice is destroyed, its dependencies are destroyed with it. So, the constructor makes the links between both sides and the destructor breaks them.

The different cardinalities and delete effects are implemented using flags. These flags are passed in a bit-wise disjunction to the constructor and it converts it into flags.

Cycles detection

Since the existence of the dependor is mandatory for that of the dependent, it seems existence dependencies must not have cycles. However, the *Shared* dependencies break this rule. The user is allowed to add and remove dependents and dependors to the dependency. Thus, cycles can appear.

If cycles are not detected, the code loops forever when trying to propagate a deletion. To avoid that, there is a flag marking a container as in deletion process. When a container is being destroyed the *BloomObject* class does not try to destroy it again.

2.5 Aggregation

Aggregation, as in [CSGS91], is the abstraction by which several objects are aggregated into a new, complex object. In BLOOM, there are three different types of aggregation:

Simple: It is employed to express that the attributes involved in it are just properties of the object being described. The object is not generally perceived as being the result of the aggregation of these attributes, which constitute merely the object properties. This kind of aggregation provides a means for specifying the attributes of an object type. Going back to John, he could be the simple aggregation of his name, age and passport number.

Collection: The new objects are composed of collections of a given class. The aggregate object is perceived as being formed by aggregating the members of the collection. For example, a club could be seen as a collection of persons. As defined in [GSSC96], we distinguish four different kinds of collections, attending to the component class (see figure 2.5):

- **Disjoint:** Each instance of the component class is able to be at most in one collection. That is, John is allowed to be member of at most one club.
- **Covering:** Each instance of the component class must be at least in one collection. In this case, John must be member of at least one club.
- **Partitioning:** Each instance of the component class must be in exactly one collection. Now, John must be member of exactly one club.
- **General:** There is no restriction about the instances of the component class. John is allowed to join as many clubs as he wants or none of them if he does not feel like joining any.

Composition: In this, the last kind of aggregation, there is a fixed set of component classes, and an aggregate object is formed by aggregating one or more objects from each of these classes. *Simple* *aggregation* and *composition* differ in that the former simply express properties while the latter is the addition of the components. A type of existence dependency should be declared for every component. Simplifying, John's car could be understood as composed by engine and chassis. The car would strictly depend on the chassis. However, it would have a relaxed dependency with the engine.

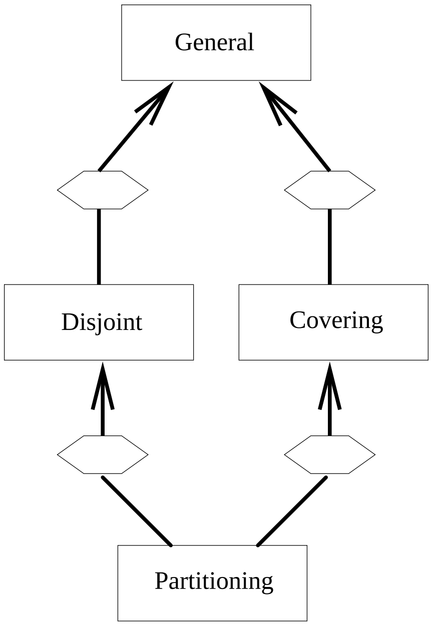

Figure 2.5: Kinds of collection in the BLOOM model

Each one of the aggregation types, can be identified as a kind of existence dependency. In the simple aggregation, the attributes strictly depend on the objects where they are aggregated . When a simple aggregated object is deleted, it should be propagated to its attributes if they are not aggregated in any other object. On the contrary, a composed object depends on all its components, and the deletion of those components could be blocked or propagated depending on each class. In the last kind of aggregation, the members share the existence of the collection. In this case, the deletion of the last element in the collection could have any effect: block its deletion, propagate it to the collection itself or even a relaxed deletion effect allowing the existence of an empty collection.

For the *composition*, the user should define the kind of dependency between the object and its components. It means the user is responsible for declaring the kind of existence dependency for each component class.

In collection aggregation, a translation is needed. Every collection should be translated as a shared existence dependency and every element should have a symmetrical existence dependency. The user will indicate the delete effect (i.e. relaxed, block or propagate) and the kind of collection (i.e. disjoint, covering, alternative or general) for each case.

The simple aggregation should have been translated as a shared dependency with propagate delete effect. However, it would mean that if an object aggregated in instances of two different classes stops being referenced in one of those classes, it would be destroyed. In spite of that, the simple aggregation has been implemented using the *BloomDependency*, too. Every attribute in a class must be declared as a dependency without type. This makes the attributes declaration more homogeneous. Methods have been defined in the *BloomDependency* class to make easy and almost transparent the assignation of values to the attributes.

Thus, every attribute must be of *BloomDependency* class. This class defines the behavior of the attributes. You declare its actual type in the constructor of the class, calling the constructor of *BloomDependency* with the name of the slice class as parameter. Note that if you would declare it directly in a concrete class, it would not inherit the attributes of its superclasses.

In the other side, every object used as attribute in a given slice, must have a *BloomDependency* created passing the name of that slice as parameter.

2.6 Type declaration

The idea behind the BLOOM model is the dynamic classification. Where is it if we fix the class or classes an object should be in? Having dynamic classification is useless if you must fix the class of the objects at declaration time. However, it is needed having typed attributes. This is a huge problem since you cannot declare a given object as instance of a class, but you insert it in that class dynamically. If everything is declared as *BloomObject* in order to have dynamic classification, you lose the actual types.

The first, and almost evident, solution would be declaring *BloomObject* as a template. However this does not work because, since depending on the parameter you get an absolutely different class, you lose everything you wanted for the dynamic classification. Therefore, the adopted solution was fixing it with a precompiler. You allow the user to declare the objects with type and then a compiler translates it to a sequence of creates and inserts in the desired classes.

That solution could be improved passing another argument to the *BloomObject::create* method. This argument would be the name of the class the object should be in. Using that information, every transaction would have a pending operation if a slice with that class name is not in the container. That could be checked in the *Container::setSlice* and *Container::unsetSlice* methods. Controlling types in this way, the user is not so tied up by them.

The user would be allowed to delete the object from the declared class, but he would have to insert it again before the commit of the transaction. This could be avoided by throwing an exception in the *Container::unsetSlice* method instead of marking it as a pending operation. This mechanism it is much less flexible for the user.

However, all that is not needed if the aggregation dimension is controlled through dependencies. Since the dependency binding is at the slice level, an object being aggregated in another one implies the former contains a given slice and thus, it is in the desired class.

2.7 Pseudo-primitive types

Some classes have been considered as pseudo-primitive types (i.e. *Boolean*, *Int* and *String*). These classes have been implemented as basic types and will be used as bricks to build the user classes. Those classes have been chosen as basic mainly due to its semantics. They have an absolutely different meaning and none of them could be easily implemented using the others.

From the existence of those types as classes we can get some advantages:

- They hide to the user the actual implementation of the types.
- Since everything is an object, all looks more homogeneous.
- Using the class keys, the duplicity of values is controlled.
- Declaring subclasses, the user could get semantic domains.
- They could have existence dependencies as any other class.

Maybe, some other classes could have been considered as primitives. However, it is enough with those three (by now). The reals could have been implemented as a pseudo-primitive type. However, their implementation strongly depends on what they are used for and it is quite easy to implement them using the *Int* class.

2.8 Extension

Another lack of ObjectStore is extension handling. It does not keep the extensions of the classes in the database. When you need them, you must go over the database looking for the desired objects.

The simplest solution would be to keep a set with all the objects and query it to select the objects depending on the result of their *instanceOf* method. This solution is really slow getting an extension because it must query the whole set of objects in the database.

Another approach to avoid that query would be having some special roots automatically created in the database. Those roots would point to sets of *BloomObject* objects. Each one of these sets would keep all the objects in a given class which were not inserted in any of their subclasses. You would get the extension of a class joining its set and the sets of its subclasses.

This is probably the best solution , because it keeps every object just once. It needs time to copy every object also once when you ask for an extension. The methods *insert* and *del* would keep the sets up to date. The *insert* method would remove the object from the superclass extension set and inserts it in its own extension set. About the *del* method, it would remove the object from its own set and insert it in the superclass if it was not in its extension (the object could be in a sibling class).

However, this nice solution had to be discarded in order to implement class key control (see explanation in section 2.9). Due to the necessity of having an index on a set with all the slices of a given class, these sets have been used to get the extension of the classes, too. Once a set with all the slices of every class exists, you just need to path the corresponding set to get the extension of the desired class.

The solution adopted uses much more space than others, but it is simpler. The extensions can be actualized in the constructor and destructor of each slice, without any verification at all.

2.9 Class key

A class key is equivalent to the primary key in the relational model. In the BLOOM model, it is a set (possibly empty) of attributes which identify the objects in a class. However, since it was implemented using a ObjectStore unique index and due to an ObjectStore restriction about its indexes, they can just be declared on one attribute.

That ObjectStore index must be bound to a collection containing all the objects in a class (the extension).

The elements of the collection cannot be *BloomObject* instances because they do not have any data to index and ObjectStore does not allow to query about the slices and cast them. Thus, the set contains slices of a given class and the containers are known because every slice knows who contains it.

Two different cases should be distinguished: the pseudo-primitive type classes and the others. This distinction is due to the kind of attributes that make up the key. In the former, they are ObjectStore primitive attributes. About the later, the attributes will actually be instances of *BloomDependency*, because all attributes are. Therefore, the pseudo-primitive types will be identified by their value and the other user classes are identified depending on the *BloomDependency* declared as key.

Every *BloomDependency* instance has a *key* attribute which will be used as key value. It is gotten concatenating the OIDs of the objects involved in the dependency.

2.10 Includes

The implementation of the BLOOM model has three important include files:

- bloom.hh
- data.hh
- displacements.hh

bloom.hh

This header file has the includes of the interfaces you need to access a database (i.e. *DBMS*, *Database*, *Transaction* and *Root*). Moreover it declares a method (i.e. *force_vfts*) to force the "vftable" inclusion for collections templates. This method cannot be declared in any class, it must be in the main program.

The first thing you must do in order to use the BLOOM model is to include this header file in the source file of any application where you want to use BLOOM.

data.hh

This file has the include of every persistent class in the schema. It should be included in the schema file and in every application accessing the database. It contains, at least, an include for each of the following classes:

- BLOOM and implementation classes: *AllObjects* and *BloomObject*
- Pseudo-primitive types: *Boolean*, *Int* and *String*

displacements.hh

This include file contains the definitions of memory distances in the user classes. They are needed to link both sides of a dependency. Therefore it must be included in every file containing code calling the constructor of a *BloomDependency* instance.

2.11 Schema file

The *schema file* should begin with:

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/manschem.hh>
#include "data.hh"
OS_MARK_QUERY_FUNCTION(Slice,getClassName);
```

In order to have persistence, you should declare in this file what classes in your database you want to be persistent. You just have to include the header of each persistent class and mark the class as persistent in the *schema file*. If a class is declared persistent, automatically, all its attributes and superclasses are treated as persistent. You mark a class as persistent using the ObjectStore macro:

```
OS_MARK_SCHEMA_TYPE(ClassName);
```

Where “ClassName” is the name of the class you want to declare persistent. Every class included in “data.hh” should be declared persistent.

2.12 Query functions

To make a method callable by a query you must do four things (all them due to ObjectStore requirements):

1. Declare an attribute of type *os_backptr* in the class.
2. In the header file, call the define *os_query_function(classname,methodname,returntype)*.
3. Call the define *os_query_function_body(classname,methodname,returntype,backptrattribname)*, in the implementation file.
4. Use the macro *OS_MARK_QUERY_FUNCTION(classname,methodname)* to mark the method in the schema file.

Chapter 3

DBMS classes

In order to handle a database, you need to use some classes and methods from a DBMS. In this case you have some classes, built on ObjectStore, which will help you to do it:

- DBMS
- Database
- Transaction
- Root

3.1 Interfaces

These interfaces had been defined to make everything more understandable and modular. Moreover, they will allow you to change the DBMS without almost any problem for your applications.

3.1.1 DBMS

This is supposed to be a class without instances. All ObjectStore general methods are in this interface, together with some useful methods that do not fit in any other interface.

Before using a BLOOM database, you must initialize the ObjectStore DBMS. This is done by the method *DBMS::initialize*. And, when you are not going to use ObjectStore anymore, you must call *DBMS::uninitialize*. This is mandatory due to ObjectStore needs some calls in order to use it. It is not because the BLOOM data model.

Methods

- initialize : Initializes ObjectStore and establishes the start of the fault handler block[1] in UNIX systems.
- uninitialize : Ends the fault handler block.
- getOID : This method is used to keep a counter in the database of the created objects. It uses a root to access it. If the root does not exist, the method creates it and initializes the counter to zero. Every time the method is called, the counter is increased. This method is just accessible from the *Container* class.
  - Parameters

    db *os_database** The ObjectStore database containing the counter.
  - Return

    unsigned long Number of objects created in the database plus one.
  - Exceptions

    TypeMismatch The database counter is corrupted.
- getKey : Returns a string of characters identifying an object. It is gotten converting its OID[2] to characters.
  - Parameters

    obj *BloomObject* Object to obtain the key from.
  - Return

    *char** Key identifying the object.
- getKey : Returns a string of characters identifying a set. It is gotten converting their OIDs to characters and concatenating them.
  - Parameters

    obj *os_Set<BloomObject* >* Set to obtain the key from.
  - Return

    *char** Key identifying the set.

[1] On some UNIX systems, the POSIX thread environment gives each thread its own set of UNIX signal handlers. On such systems, the ObjectStore handlers for SIGSEGV (and in some cases SIGBUS) must be established in each thread.
[2] Object Identifier

3.1.2 Database

The first thing you need to access or create a database is a *Database* object bound to it. You must open a database before doing anything in it and not close it until everything else is closed.

Methods

- Database : Constructor for the class. It keeps the name of the database.
  - Parameters

    dbName *char** Name for the database.
- ~ Database : Destructor for the class. It frees the memory allocated in the constructor.
- open : Creates/Opens a new the database.
  - Parameters

    readOnly *boolean* True implies the database is opened in read-only mode.

    creation *boolean* If it is true and the database is not found, it will be created. If it is false, an exception will be thrown when the database is not found.

    creationMode *int* UNIX permissions to be used if the database is created.
  - Exceptions

    NotFound Thrown if the creation mode is false and the database is not found.

    Repeated Thrown if the object is opened twice.
- close : Closes the database.
  - Exceptions

    OddOperation Thrown if the database was not opened before.
- destroy : Closes and destroys the database. You will not be able to use it again.
  - Exceptions

    OddOperation Thrown if the database was not opened before.
- () : Converts a *Database* into an *os_database*. It simplifies some calls.
  - Return

    os_database ObjectStore database wrapped by the object.

3.1.3 Transaction

In ObjectStore, all must be done inside a transaction except, of course, open and close the database. This means you must create a transaction and begin it before doing anything else. It is advisable not to open or close a database within a transaction. Another important thing is taking into account that if a transaction is not committed, it will be automatically aborted by the destructor of this class.

Some different kinds of transaction can be declared: update, read-only and abort-only. Thereby, you are not allowed neither to update a database inside a read-only transaction nor commit an abort-only transaction.

In the other hand, this class keeps a set of active transactions. It is used to check objects consistency before the commit. Every *Transaction* keeps a counter of pending classification operations (i.e. inserts and deletes). If the counter of a *Transaction* is not zero, it cannot be committed.

This class also contains some methods used to create an internal transaction. This transaction is begun every time the user tries to create new slices or containers. It is used to roll everything back if an exception is thrown while the slices or containers are being created.

ObjectStore offers much more methods about transactions than these offered by this class. Much of these methods just give more flexibility to transactions but do not do anything different from the others. Due to that, they were not added to this interface, by now.

Methods

- UPDATE : The database can be updated in the transaction.
- READ_ONLY : The database cannot be updated in the transaction.
- ABORT_ONLY : The transaction cannot be committed.

Methods

- getCurrent : This is a static private method used to get the current *Transaction*. It looks for it in the active transactions set.
  - Return

    Transaction The current transaction.

– Exceptions

NotFound There is not active transaction or it is not in the active transactions set.

- pending : Increases the pending operations counter of the current transaction. This static method should not be called by the user. It is used to check the integrity constraints of the data model before a transaction is committed.

  – Exceptions

  NotFound There is no active transaction.

- done : Decreases the pending operations counter of the current transaction. This static method should never be called by the user. It is used to check the integrity constraints of the data model before a transaction is committed.

  – Exceptions

  NotFound There is no active transaction.

- Transaction : Constructor for the class. It keeps the type of transaction.

  – Parameters

  trType *int* Type of transaction (i.e. update, read-only or abort-only).

  – Exceptions

  WrongType The type parameter value is not allowed.

- ˜ Transaction : Destructor for the class. It aborts the transaction if it was not committed before.

- begin : Begins a transaction and adds it to the active transactions set.

  – Exceptions

  Repeated A Transaction can only begin once.

- commit : Commits a transaction and removes it from the active transactions set. It does not do it if the transaction has any pending operation.

  – Exceptions

  OddOperation You cannot commit a Transaction which was not begun before.

  AbortOnly You are not allowed to commit an abort-only Transaction.

  ClassOperationsPending The database objects are not consistent. There are some classification operations (i.e. insert or delete) pending.

- abort : Aborts a transaction, removes it from the active transactions set and resets the pending operations counter.

- Exceptions

  OddOperation You cannot commit a Transaction which was not begun before.

- beginInternal : This method should never be called by the user. It is used to begin an internal transaction every time the user calls a classification operation (i.e. *BloomObject::create*, *BloomObject::destroy*, *insert* or *del*) in order to roll it back if any exception is thrown or commit it if everything was OK.

  - Exceptions

    OddOperation There is not any parent transaction begun.

    ReadOnly The current transaction is read-only.

- abortInternal : This method should never be called by the user. It is called in order to abort the internal transaction if an exception is thrown.

- commitInternal : This method should never be called by the user. It is called in order to commit the internal transaction when everything was OK.

3.1.4 Root

This class represents access points to a database. You must open a *Root* before using it in order to have a similar behavior with *Database* interface (the *Root* interface is designed to look like the *Database* one).

Once a *Root* has been opened, you can handle it. Be careful, it will not throw any exception if you set a value to a root which already has one. In the other hand, you can either delete a root leaving the object (value) or delete the object (value) leaving the root (and after that, set a new object to the same root).

Another problem is that ObjectStore allows you to access a destroyed object if you keep its reference. So, if you destroy an object but keep a root pointing to it, you will be able to keep working with that object. However, you are not allowed to destroy it again.

The values pointed by the roots are always going to be instances of *BloomObject*, since every object in the database will actually be an instance of that class.

Methods

- Root : Constructor for the class. It keeps the name of the root.

  - Parameters

db *Database* Database where the root is going to be created.

name *char** Root name.

- ˜ Root : Destructor for the class. It frees memory allocated in the constructor.
- open : It prepares the *Root* to be used. Firstly, it tries to find it in the database. If it is not found, depending on the creation mode, it creates it or throws an exception. It is done in this way to have the same behavior of a Database.
  - Parameters

    creation *boolean* Creation mode. True implies the root will be created if it is not found in the database.
  - Exceptions

    NotFound Thrown if the creation mode is false and the root is not found in the database.

    Repeated Thrown if the object is opened twice.

    Transient Thrown if the database is transient.
- setValue : Bind a root with an object. It does not check if the root already has a value because it is not possible to know if that previous value was already deleted from the database.
  - Parameters

    value *BloomObject* Value to be bound to the root.
  - Exceptions

    OddOperation The *Root* must be opened before the binding.

    WrongValue The value bound to a root must be an object in the same database.
- getValue : Returns the value bound to the *Root*. It does it even if the value was deleted from the database.
  - Return

    BloomObject Object bound to the root.
  - Exceptions

    OddOperation The *Root* must be opened before you try to get the value or the root does not have a value bound.

    TypeMismatch The object bound to the root is not a BloomObject.
- close : Closes the *Root*.
  - Exceptions

    OddOperation Thrown if the *Root* was not opened before.

- destroy : Closes and destroys the root. You cannot access it anymore.
  - Exceptions

    OddOperation Thrown if the *Root* was not opened before.

3.2 Interfaces' mistakes

Some errors are not handled by those interfaces. In spite of that, ObjectStore is still going to generate them. If that happens, it will abort your application.

err_write_during_query You are trying to write during a read-only transaction.

err_no_tran You forgot beginning a transaction.

err_opened_read_only You are trying to write on a read-only database.

3.3 Exception class

Besides, you also have another class to make programmers' live much more easy. It is the class of all possible exceptions. It has all the information needed to handle an exception. It is not part of the BLOOM model at all.

It should be the top of an exceptions class hierarchy. However, instead of a hierarchy, it has a constant defined for each kind of exception generated. That is done this way due to C++ deficiencies. If you catch an exception as one of its superclasses, you are not able to throw it again as an instance of the class it actually is in. Doing it with an attribute, you can always catch and throw it again without any loss at all. The problem is you need some extra code to check the kind of exception it is.

Kinds of exception

1. NotFound : Something was not found (i.e. the database, an active transaction, a slice in a given container, etc.).
2. ReadOnly : The current transaction is read-only.
3. Transient : The database is transient.

4. TypeMismatch : This is used in two different cases:
   (a) A root is pointing to an unexpected object.
   (b) The two parts of a dependency do not match.
5. ClassKey : Key constraint violation.
6. OddOperation : Wrong order in a sequence of method calls.
7. Repeated : Something was already done (i.e. open a database, begin a transaction, insert a slice, etc.).
8. WrongValue : This is used in two different cases:
   (a) A root value is not in the same database than the root itself.
   (b) A parameter is not in its actual domain (i.e. transaction type).
9. ClassOperationsPending : Due to classification constrains, a transaction is not committable.
10. AbortOnly : An abort-only transaction is not committable.
11. Cardinality : A given dependency should have a given cardinality depending on its kind.
12. ForbittenOperation : A method is not callable in a given kind of *BloomDependency*.
13. Block : An operations was blocked due to either classification or dependency constraints.
14. Null : A non-typed and mono-valued dependency (attribute) has no value.

Methods

- Exception : Constructor for the class. It keeps the kind, message and thrower.
  - Parameters

    paramKind *int* Kind of exception.

    paramThrower *char** Name of the thrower class.

    msg *char** Exception message.
- setKind : Sets the kind of the exception it is.
  - Parameters

    paramKind *int* Kind of exception.
- getKind : Gets the kind of the exception.

- Return

  int Kind of exception (hopefully one of the constants).

- setThrower : Sets the name of the thrower of the exception. It copies the parameter.
  - Parameters

    paramThrower *char** A class name.

- getThrower : Gets the name of the thrower class of the exception.
  - Return

    *char** Name of the thrower class.

- setMessage : Sets the message of the exception. It copies the parameter.
  - Parameters

    msg *char** An exception message.

- getMessage : Gets the message of the exception.
  - Return

    *char** Exception message.

Chapter 4

BLOOM data model classes

The classes listed in the previous chapter were directly related to the DBMS. They only depend on the DBMS under the implementation (ObjectStore). The classes listed here mainly depend on the BLOOM data model.

4.1 Implementation classes

4.1.1 Slice

Every user class is actually a subclass of this. It's mandatory in order to be inserted in a *Container*.

This class keeps a reference to its container. Thus, it needs the container as parameter as well as all its subclasses (all user classes) need it. This is done in this way to allow the constructors of the classes to access data in the superclasses as done in other methods. A slice cannot exists if it is not in a container. Due to that reason, the slice inserts and removes itself in and from the container.

This class also have a flag to indicate when it is in a deletion process. It is used to avoid deletion cycles.

Methods

- Slice : Constructor of the class. It inserts the slice in the container and keeps the reference to it.

- Parameters

  containerParam *BloomObject* Container where the slice is created.

- ˜ Slice : Destructor of the class. It removes the slice from the container.

- getClassName : This method uses the ObjectStore Meta Object Protocol to know the name of the class the slice belongs to and returns it. Another implementation would be a pure virtual method that would be implemented by every class returning its name. In the second case, it should be an static method, but C++ does not allow to define a virtual and static method. The first way was chosen mainly because it gives less work to the precompiler in spite of it is much less efficient. Its efficiency depends on the ObjectStore implementation. The efficiency could be improved getting the name of the class only once in the constructor and keeping it as an attribute of the slice (it is a commitment between memory space and process time).
  Anyway, it returns a string of characters with the name of the class the slice belongs to which is used to identify the slice in a container.

  - Return

    *char** Name of the class the slice belongs to.

4.1.2 Container

This class implements the application independent methods for the *BloomObject*. This is used to be able to compile all the classes without the *BloomObject* class. None of its methods should be called by the user.

This class also keeps the object identifier (OID) and a flag indicating when the object is being deleted to avoid cycles.

Methods

- Container : Constructor for the class. It gets the OID and marks the insertion in *AllObjects* as pending.

  - Exceptions

    NotFound There is no active transaction. This should never be thrown because it is always checked before creating a container.

    TypeMismatch The database OID is corrupted.

- ˜ Container : Destructor for the class. It unmarks the insertion in *AllObjects* as pending.

- Exceptions

  NotFound There is no active transaction. This should never be thrown because it is always checked before destroying a container.

- setSlice : Inserts a new slice in the container. This method should never be called by the user. It is only used in the *Slice* class (in the constructor).

  - Parameters

    slice *Slice** Slice to be inserted.

- unsetSlice : Removes a slice from the container. This method should never be called by the user. It is only used in the *Slice* class (in the destructor).

  - Parameters

    slice *Slice** Slice to be removed.

- getSlice : Looks for a slice of the given class in the container using the *os_collection::query_pick* method. It returns a pointer to it or throws an exception if not found. This method should never be called by the user.

  - Parameters

    className *char** The name of the desired class slice.

  - Return

    *Slice** A pointer to the demanded slice.

  - Throw

    NotFound The slice is not in the container.

- isSliceIn : Queries for a class slice in the container using the *os_collection::exists* method. It returns true if found. This method should never be called by the user.

  - Parameters

    className *char** The name of the desired class slice.

  - Return

    boolean True if a slice of the demanded class is in the container.

4.1.3 BloomObject

Every persistent object should be an instance of this class (even parameters and return values). All BLOOM model constraints and features has been implemented around this class, it is the core of the implementation. If an object is not in this class, it is not in the data model.

Two static methods have been implemented to easily create and destroy instances of *BloomObject*. This methods should be in *DBMS* class but they have a huge problem, they cannot know the class they are going to create because it will depend on those defined by the user. Therefore, they must be in this class.

For every class declared by the user, a method is defined in this class in order to return the corresponding slice. This allows the user to cast the objects and execute the desired method depending on the class. By default, the slice that is going to execute the method is chosen following an inverse topological sort of the hierarchical graph. Since everything is an instance of *BloomObject* and it can be cast to every class you defined, it is really easy to have type problems if you do not cast it explicitly to whatever type you want.

Methods

- create : Must be used (instead of *new*) in order to create a persistent object. It is static and should only be called to create instances in the BLOOM data model, that is, instances of *BloomObject*.
  - Parameters

    db *Database* Database object where you are going to create the new object.
  - Return

    BloomObject An instance of a basic BLOOM object which can be inserted in any other classes. Before you use it you must insert it in at least a class different from *AllObjects*. Declaring a *BloomObject* reference will allow you to work with the object.
  - Exceptions

    Those from the constructor of the class.
- destroy : Must be used (instead of *delete*) in order to destroy persistent objects. It is a static method and should only be called to destroy objects in BLOOM data model, that is, instances of *BloomObject*.
  - Parameters

    obj *BloomObject* Persistent object to be destroyed.
  - Exceptions

    Repeated The object has already been destroyed. Or it is not persistent.

    OddOperation There is not any transaction begun.

    Those from the destructor of the class.
- BloomObject : Constructor of the class. It inserts an *AllObjects* slice in the container.
  - Exceptions

Those from the *AllObjects::insert* method.

- ~ BloomObject : Destructor of the class. Removes all the slices from the container destroying them.
  - Exceptions

    Those from the *AllObjects::del* method.
- == : This method allows to check equality between two instances.
  - Parameters

    obj *BloomObject* The object to be compared with.
  - Return

    boolean True if the OIDs are equal.
- != : This method allows to check inequality between two instances.
  - Parameters

    obj *BloomObject* The object to be compared with.
  - Return

    boolean True if the OIDs are not equal.

4.1.4 BloomDependency

This class is used to link two bloom objects. An instance must be in both sides. Each side should know the slice containing its pair and the distance (in bytes) from the beginning of the slice to the *BloomDependency* instance.

This class implements different behaviors depending on the kind of dependency. Two basic kinds of dependencies are considered: with and without type. The dependencies without type can be mono-valued or multi-valued, and the typed dependencies can be exclusive or non-exclusive and have an interest of dependent relaxed or strict (see figure 2.4). A non-exclusive dependency can be shared or all, and a strict can be block or propagate.

All those different behaviors have been implemented using flags. The user can enable those flags suppling to the constructor the bit-wise disjunction (using |) of the constants signifying the desired behavior.

Constants

- DEPENDENT : The *BloomDependency* is in the dependent side.

- DEPENDOR : The *BloomDependency* is in the dependor side.
- RELAXED : It has a relaxed interest of dependent.
- STRICT : It has a block or propagate delete effect.
- BLOCK : It has a block delete effect.
- PROPAGATE : It has a propagate delete effect.
- EXCLUSIVE : It has exactly one element in the dependent side.
- NONEXCLUSIVE : It has at least one element in the dependent side.
- SHARED : Elements can be inserted in or removed from the dependent.
- ALL : Elements can neither be inserted in nor removed from the dependent.
- GENERAL : It is a general collection.
- DISJOINT : It is a disjoint collection.
- COVERING : It is a covering collection.
- ALTERNATIVE : It is an alternative collection.
- WITHOUTTYPE : The dependency does not have any of the above types (it is just an attribute).
- MONOVALUED : Equivalent to exclusive but without type.
- MULTIVALUED : Equivalent to non-exclusive but without type.

Methods

- updateKey : Private method that updates the key attribute of the dependency depending on the objects in its set of elements.
  - Exceptions

    ClassKey A key has been violated.
- getObject : Private method that returns the object containing the dependency.
  - Return

    BloomObject Container of the slice containing the dependency.
- getDependency : Private method that returns the *BloomDependency* instance at the other side of the dependency, for a given object.

- Parameters

  obj *BloomObject* Object to look for the slice in.

- Return

  BloomDependency The dependency potentially linked to this.

- Exceptions

  Notfound The object is not in the required class.

- setDependency : Private method that inserts an object in the set of elements. This method checks the kind of dependency it has (it should match with this) and actualizes the other side of the dependency.

  - Parameters

    dependency *BloomObject* Object to be linked by the dependency.

  - Exceptions

    TypeMismatch Thrown if the kind of dependency of the object to be linked does not match the kind of this.

    NotFound Thrown if the object is not in the required class or there is no active transaction (it should never happen).

    ClassKey A key has been violated.

- unsetDependency : Private method that removes an object from the set of elements. It actualizes the other side of the dependency as well.

  - Parameters

    dependency *BloomObject* Object to be removed from the set.

  - Exceptions

    NotFound There is no active transaction (it should never happen).

    ClassKey A key as been violated.

- preConstructor : Private method that keeps the parameters, decodes the kind of dependency and actualizes the booleans which actually describe the behavior of the object. This method is called at the beginning of every constructor.

  - Parameters

    className *char** Name of the slice class containing the symmetric *BloomDependency*.

    paramDisplacement *os_int32* Distance from the beginning of its slice to the memory position of the symmetric *BloomDependency*.

    paramKind *int* Bit pattern describing the behavior of the dependency (obtained forming the bit-wise disjunction of the corresponding constants).

paramWithKey *boolean* Indicates the necessity of actualizing the key attribute of the dependency.

– Exceptions

NotFound There is no active transaction (this should never happen).

- postConstructor : Private method that checks cardinalities and creates an index in the set if the key is enabled. This method is called at the end of every constructor.

  – Exceptions

  Cardinality Wrong cardinality for the kind of dependency.

- BloomDependency : Constructor for the class. Calls the *preConstructor* and *postConstructor* methods.

  – Parameters

  className *char** Name of the slice class containing the symmetric *BloomDependency*.

  paramDisplacement *os_int32* Distance from the beginning of its slice to the memory position of the symmetric *BloomDependency*.

  paramKind *int* Bit pattern describing the behavior of the dependency (obtained forming the bit-wise disjunction of the corresponding constants).

  paramWithKey *boolean* Indicates the necessity of actualizing the key attribute of the dependency.

  – Exceptions

  NotFound There is no active transaction (this should never happen).

  Cardinality Wrong cardinality for the kind of dependency.

- BloomDependency : Constructor for the class. Calls the *preConstructor* method, keeps the dependor and calls the *postConstructor* method.

  – Parameters

  className *char** Name of the slice class containing the symmetric *BloomDependency*.

  paramDisplacement *os_int32* Distance from the beginning of its slice to the memory position of the symmetric *BloomDependency*.

  paramKind *int* Bit pattern describing the behavior of the dependency (obtained forming the bit-wise disjunction of the corresponding constants).

  paramWithKey *boolean* Indicates the necessity of actualizing the key attribute of the dependency.

  dependor *BloomObject* Object at the other side of the dependency.

  – Exceptions

NotFound The dependor is not in the expected class or there is no active transaction (this should never happen).

TypeMismatch Thrown if the kind of dependency of the dependor does not match the kind of this.

Cardinality Wrong cardinality for the kind of dependency.

ClassKey A key as been violated.

- BloomDependency : Constructor for the class. Calls the *preConstructor* method, keeps the dependors set and calls the *postConstructor* method.

  – Parameters

  className *char** Name of the slice class containing the symmetric *BloomDependency*.

  paramDisplacement *os_int32* Distance from the beginning of its slice to the memory position of the symmetric *BloomDependency*.

  paramKind *int* Bit pattern describing the behavior of the dependency (obtained forming the bit-wise disjunction of the corresponding constants).

  paramWithKey *boolean* Indicates the necessity of actualizing the key attribute of the dependency.

  dependors *os_Set<BloomObject* >* Objects at the other side of the dependency.

  – Exceptions

  NotFound Some dependors are not in the expected class or there is no active transaction (this should never happen).

  TypeMismatch Thrown if the kind of dependency of any object in the set does not match the kind of this.

  Cardinality Wrong cardinality for the kind of dependency.

  ClassKey A key as been violated.

- ˜ BloomDependency : Destructor for the class. Unsets its dependencies, checking cardinality and kind of dependency in order to block or propagate it when needed. It frees memory as well.

  – Exceptions

  NotFound Some of the objects in the set are not in the expected class (it should never happen) or there is no active transaction (it should never happen neither).

  ClassKey A key has been violated.

  Block The object is bound to its dependents.

  Those coming from destroying the dependent objects (when applicable).

- () : Returns the set of elements of the *BloomDependency* if it is mono-valued.

– Return

os_Set<BloomObject >* Set of elements.

– Exceptions

ForbittenOperation Thrown if it is a mono-valued dependency.

- = : Allows to assign a set to a multi-valued *BloomDependency* without type.

  – Parameters

  newSet *os_Set<BloomObject* >* Set to be assigned.

  – Return

  BloomDependency The dependency itself.

  – Exceptions

  ForbittenOperation Thrown if it is a typed or mono-valued dependency.

  TypeMismatch Thrown if the kind of dependency of any object in the set does not match the kind of this.

  NotFound Thrown if any object is not in the required class or there is no active transaction (it should never happen).

  ClassKey A key has been violated.

- () : Returns the object in the elements set if it is a mono-valued or exclusive dependency.

  – Return

  BloomObject Object in the set.

  – Exceptions

  ForbittenOperation Thrown if it is not a mono-valued dependency.

  Null Thrown if the set does not have any element.

- = : Allows to assign an object to a mono-valued *BloomDependency*.

  – Parameters

  newObj *BloomObject* Object to be assigned.

  – Return

  BloomDependency The dependency itself.

  – Exceptions

  ForbittenOperation Thrown if it is a typed or multi-valued dependency.

  TypeMismatch Thrown if the kind of dependency of the object does not match the kind of this.

NotFound Thrown if the object is not in the required class or there is no active transaction (it should never happen).

ClassKey A key has been violated.

- add : Adds a dependent or dependor to a shared dependency.
  - Parameters

    dep *BloomObject* The object to be added to the dependency.
  - Exceptions

    TypeMismatch The dependency is not shared or both sides are of different kind.

    NotFound The object is not of the required class or there is no active transaction (it should never happen).

    ClassKey A key has been violated.
- remove : Removes a dependent or dependor from a shared dependency.
  - Parameters

    dep *BloomObject* The object to be removed from the dependency.
  - Exceptions

    TypeMismatch The dependency is not shared.

    Cardinality The object is bound (the dependent has only one dependor and it cannot be removed).

    NotFound The object is not of the required class or there is no active transaction (it should never happen).

    ClassKey A key has been violated.

4.2 User and pseudo-primitive types classes

In this section we can see the really worthy classes. That is, the classes which implement the BLOOM data model and the characteristics of the user classes. All these classes should be subclasses of *Slice* in order to be inserted in a container. None of them should be directly instantiated by the user.

All BLOOM user classes should look like this:

```
class ClassName: public Slice {
  components
  }
```

Moreover, every class you want to make persistent (all them) must define the following method signature:

```
static os_typespec *get_os_typespec()
```

The ObjectStore post–compiler will generate automatically the code for those methods.

All these classes will have some methods (automatically generated by a bloom–precompiler) which allow the applications to insert bloomObject in, remove a bloomObject from, know if a given bloomObject is in and get the extension of every class.

Special methods

- get_os_typespec : The ObjectStore schema generator will automatically supply a body for this method, which will return a pointer to a typespec for the class. This will allow you to create persistent slices of the class in the database.
  - Return

    *os_typespec** Pointer to the ObjectStore type specification of the class.
- insert : Inserts a bloomObject in the class and superclasses. It marks/unmarks the current transaction with pending operations depending on the data schema (the kind of generalizations and specializations the class has).

  It could have more parameters depending on the constructor of the class.
  - Parameters

    obj *BloomObject* Object to be inserted.

    Those for the constructor of the given class and superclasses.
  - Exceptions

    OddOperation There is not any transaction begun.

    ReadOnly You are not allowed to change the database in a read-only transaction.

    Repeated The object was already in the class.

    NotFound There is no active transaction (it should never happen).

    Those from the constructor of the class and superclasses.
- del : Deletes a bloomObject from the class and its subclasses. It marks/unmarks the current transaction with pending operations depending on the data schema. This implements checking for complementary specialization delete effects. It throws an exception or propagates the deletion to its superclasses if the class is a complementary specialization (with block or propagate delete effect, respectively) and the object is not in any other sibling class.

– Parameters

obj *BloomObject* Object to be deleted.

– Exceptions

OddOperation There is not any transaction begun.

ReadOnly You are not allowed to change the database in a read-only transaction.

NotFound The object was not in the class or there is no active transaction (internal error).

Block The object is not in any other sibling class (and the class is complementary or alternative specialization).

Those from the destructor of the given class and subclasses.

Those from the sub and superclasses deletion methods.

- instanceOf : Says if a bloomObject is an instance of the class. It queries the container looking for an slice of the class.

  – Parameters

  obj *BloomObject* Object to be checked.

  – Return

  boolean True if the bloomObject was inserted in the class.

- getExtension : Returns a set with the objects in the class. This method just copies the actual set in the database changing every slice by its container.

  – Parameters

  db *Database* Database to get the extension from.

  – Return

  os_Set<BloomObject>* A set containing all objects in the class.

  – Exceptions

  Transient The database is not persistent and an extension cannot be created.

  ReadOnly The current transaction is read-only and a new extension cannot be created.

  TypeMismatch The database extension is corrupted.

- internalGetExtension : This private method gets the extension set of the class kept in the database, using a root. If the set does not exist yet, a new empty one is created.

  – Parameters

  db *os_database** ObjectStore database to get the extension from.

  – Return

os_Set<ClassName>* A set containing all class slices.

– Exceptions

Transient The database is not persistent and an extension cannot be created.

ReadOnly The current transaction is read-only and a new extension cannot be created.

TypeMismatch The database extension is corrupted.

4.2.1 AllObjects

This is the superclass of all BLOOM objects. All the other classes in the BLOOM model are subclasses of this. All general characteristics of a BLOOM object are inherited from this class.

It is an alternative generalization of *Boolean*, *Int*, *String* and all the "top" classes defined by the user.

4.2.2 Boolean

This is a basic type. It wraps an ObjectStore boolean type in a BLOOM class and has exactly the same behavior. Mainly, it has just two methods to convert it into *boolean* and vice versa. So, you can use it instead of *boolean* with no difference.

Methods

- Boolean : Constructor for the boolean class. Adds the object to the extension and keeps the initial value (checking the key).

  – Parameters

    paramContainer *BloomObject** Container of the slice.

    initValue *boolean* Initial value for the boolean (by default, its value is false).

  – Exceptions

    Transient The database is not persistent and an extension cannot be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.

    ClassKey The class key constraint has been violated.

- ˜ Boolean : Destructor for the boolean class. Removes the object from the extension.

- Exceptions

  Transient The database is not persistent and an extension cannot be created.

  ReadOnly The current transaction is read-only and a new extension cannot be created.

  TypeMismatch The database extension is corrupted.

- = : Allows to assign a boolean to a *Boolean*.

  - Parameters

    newVal *boolean* New value for the object.

  - Return

    BloomObject Object container. It allows to chain assignments.

  - Exceptions

    ClassKey The class key constraint has been violated.

- () : Converts a *Boolean* into a boolean value. It allows you to operate a *Boolean* as you like.

  - Return

    boolean Boolean value.

- getByKey : Gets the object in the class with the given key.

  - Parameters

    db *Database* Database to get the extension from.

    keyValue *boolean* Value of the key of the desired object.

  - Return

    BloomObject The demanded object.

  - Exceptions

    Transient The database is not persistent and an extension cannot be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.

    NotFound There is not an object with the given key.

4.2.3 Int

This is a basic type. It wraps an ObjectStore integer type in a BLOOM class and has exactly the same behavior. Mainly, it has just two methods to convert it into *int* and vice versa. So, you can use it instead of *int* with no difference.

Constants

- MIN : Minimum integer value.
- MAX : Maximum integer value.

Methods

- Int : Constructor for the integer class. Adds the object to the extension and keeps the initial value (checking the key).
  - Parameters

    paramContainer *BloomObject** Container of the slice.

    initValue *int* Initial value for the integer (by default, its value is zero).
  - Exceptions

    Transient The database is not persistent and an extension cannot be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.

    ClassKey The class key constraint has been violated.
- ˜ Int : Destructor for the integer class. Removes the object from the extension.
  - Exceptions

    Transient The database is not persistent and an extension be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.
- = : Allows to assign an integer to an *Int*.
  - Parameters

    newVal *int* New value for the object.
  - Return

    BloomObject Object container. It allows to chain assignments.
  - Exceptions

    ClassKey The class key constraint has been violated.
- () : Converts an *Int* into an integer value. It allows you to operate an *Int* as you like.
  - Return

int Integer value.

- getByKey : Gets the object in the class with the given key.
  - Parameters

    db *Database* Database to get the extension from.

    keyValue *int* Value of the key of the desired object.
  - Return

    BloomObject The demanded object.
  - Exceptions

    Transient The database is not persistent and an extension cannot be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.

    NotFound There is not an object with the given key.

4.2.4 String

This is another basic type. It wraps an ObjectStore string type in a BLOOM class. Mainly, it also has two methods to convert it into *char** and vice versa. So, you can use it instead of *char** with no difference. It allocates and frees persistent memory for you. Therefore, you do not need to worry about that.

Methods

- String : Constructor for the string class. Adds the object to the extension and copies the initial value to persistent memory (checking the key).
  - Parameters

    paramContainer *BloomObject** Container of the slice.

    initValue *char** Initial value for the string (by default, its value is an empty string).
  - Exceptions

    Transient The database is not persistent and an extension cannot be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.

    ClassKey The class key constraint has been violated.

- ˜ String : Destructor for the string class. Removes the object from the extension and frees memory.
  - Exceptions

    Transient The database is not persistent and an extension cannot be created.

    TypeMismatch The database extension is corrupted.

    ReadOnly The current transaction is read-only and a new extension cannot be created.
- = : Allows to assign a string to a *String*.
  - Parameters

    newVal *char** New value for the object.
  - Return

    BloomObject Object container. It allows to chain assignments.
  - Exceptions

    ClassKey The class key constraint has been violated.
- () : Converts a *String* into a string value. It allows you to operate a *String* as you like.
  - Return

    *char** String value.
- getByKey : Gets the object in the class with the given key.
  - Parameters

    db *Database* Database to get the extension from.

    keyValue *char** Value of the key of the desired object.
  - Return

    BloomObject The demanded object.
  - Exceptions

    Transient The database is not persistent and an extension cannot be created.

    ReadOnly The current transaction is read-only and a new extension cannot be created.

    TypeMismatch The database extension is corrupted.

    NotFound There is not an object with the given key.

Chapter 5

Conclusions and future work

The first and more evident conclusion coming out from this work has been the necessity of an automatic code generator. Due to the deficiencies of ObjectStore and the enrichments of the BLOOM model, lots of code are needed in order to get the desired result. Fortunately, the needed code follows quite fixed patterns and it could be generated automatically.

Figure 5.1 shows the steps to get the code. Firstly, the user designs the database using a graphical interface, which validates the BLOOM schema. That interface generates the input for a BLOOM precompiler or code generator, which generates the needed C++ code to simulate the BLOOM model on a commercial DBMS. The grammar of the compiler will be published in a paper about the thesyntax of BLOOM schemas.

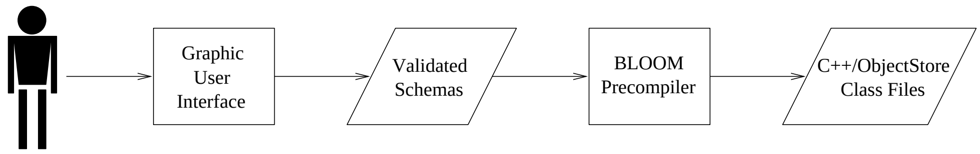

Figure 5.1: BLOOM code generation process

Another important issue about this work has been the implementation of the dynamic classification. It was the hardest point to solve and drove the rest of the work. Probably, a schema evolution would be enough in almost all the cases, depending on the frequency the objects move from a class to another. The adopted solution using slices generates a huge overhead calling methods and wastes lots of extra space for the slices. However, it was implemented to avoid the problems using the ObjectStore schema evolution, and because it is much more flexible and close to the desired effect.

About the existence dependencies, some gaps where found. They do not take into account every case and some new kinds should be studied. The nine different existence dependencies currently in the BLOOM model do not cover all the possibilities. It should be revised and the taxonomy extended.

Future work

The first and more urgent work to do is implementing a precompiler to generate all the extra code described in this paper. It is inconceivable thinking of a user generating all the code by hand. Some of the tasks of the compiler would be:

- Generating the *BloomObject* class and all its methods for casting and delegation.
- Modifying the attributes of the classes to implement them using the *BloomDependency* class.
- Modifying the constructor of user classes, calling the constructor of *Slice* and adding the code to link the dependencies to the desired attribute slices.
- Adding some extra methods to the user classes.
  - *os_get_typespec* to make the classes persistent.
  - *insert* and *del* to insert and delete the objects in and from the classes. These methods include creating superclasses and destroying subclasses slices when needed, as well as marking the pending operations depending on the different kinds of classification constraints.
  - *instanceOf* to know if an object is instance of a given class.
  - *getExtension* to get the extensions of the classes.
  - *getByKey* (in classes with class key) to retrieve an object given its key.
  - *add* and *remove* (in collection classes) to add and remove members in the collections.
- Adding code to create indexes on the extensions of classes with class key.

The code generated by the compiler could be modified to attach a relational database instead of an O-O one. It should not be difficult once you know the code needed to simulate BLOOM on a relational database. This is planned to be done once the precompiler starts working.

Since this is the first version, it has a couple of things left to next releases:

- The pseudo-primitive attributes should have a shared dependency with propagate delete effect. It means they should not exist when nobody references them. A garbage collector should be implemented or the dependencies should be improved to solve this problem.
- The performance was not a requirement. It should be taken into account in future versions.
  - Avoid using the ObjectStore Meta Object Protocol to get the name of a Slice.
  - Add indexes to the containers to gain time looking for slices.
  - Save space in the data structures (e.g. *BloomDependency* class).
- Test and improve the OID mechanism.
- Allow class keys with more than one attribute.
- Wrap all the ObjectStore exceptions.
- Add some extra features to the DBMS classes (e.g. to *Transaction*).
- Control about destroyed objects pointed by roots.

Together with all that things, it is also planned to implement a query language for BLOOM schemas as proposed in [Cam97], as well as a graphical user interface to make everything look nice.

Acknowledgments

This work has been supported by a grant of the Comissionat per a Universitats i Recerca de la Generalitat de Catalunya and the PRONTIC project TIC96-0903.

Bibliography

[Cam97] Benet Campderrich. A closed, data-model integrated, object-oriended query and view definition language. Escola Tècnica Superior d'Enginyeria (URV), Spain, November 1997.

[CSGS91] Malú Castellanos, Fèlix Saltor, and Manuel García-Solaco. The development of semantic concepts in the BLOOM model using an object metamodel. Technical Report 22, Universitat Politècnica de Catalunya, Dept. LSI, Barcelona, 1991.

[CSGS94] Malú Castellanos, Fèlix Saltor, and Manuel García-Solaco. A canonical model for the interoperability among object-oriented and relational databases. In Ozsu, Dayal, and Valduriez, editors, *Distributed Object Management*, pages 309–314. Morgan Kaufmann, 1994. Proceedings, Int. Workshop on Distributed Object Management, IWDOM, Edmonton, Canada, 1992.

[Fow98] Martin Fowler. Dealing with roles. http://www2.awl.com/cseng/titles/0-201-89542-0/roles2-1.html, 1998.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[GSSC96] Manuel García-Solaco, Fèlix Saltor, and Malú Castellanos. Semantic heterogeneity in multidatabase systems. In Omran Bukhres and Ahmed Elmagarmid, editors, *Object-Oriented Multidatabase Systems. A Solution for Advanced Applications*, chapter 5, pages 129–202. Prentice Hall, Barcelona, 1996.

[MO92] James Martin and James Odell. *Object-Oriented Analysis and Design.* Prentice–Hall, 1992.

[MO96] James Martin and James Odell. *Object-Oriented Methods.* Prentice–Hall, 1996.

[SCGS91] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of data models as canonical models for federated databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(4):44–48, December 1991.

[WdJS95] Roel Wieringa, Wiebren de Jonge, and Paul Spruit. Using dynamic classes and role classes to model object migration. *Teory and Practice of Object Systems*, 1(1):61–83, 1995.

Appendix A

Aspects of the code

Since the implementation has been done on ObjectStore, it drove almost all the implementation decisions. The main one was the programming language selection. ObjectStore offers three different interfaces: C++, ActiveX and Java. Since ActiveX is just a distributed components system, the choice was between C++ and Java.

C++ offers lots of functionalities and different ways to do things. Due to this, sometimes it's hard to understand the meaning of the code. In the other hand, Java does not have so many ways to do things, and, therefore, it is more understandable. However, C++ has been chosen as programming language, mainly because it allows multiple inheritance and lets you do whatever you want. It could be considered as a low level language.

It would be nice if C++ could, actually, look like Java. It is not possible; therefore, there will be some important differences to take into account:

- C++ does not have a garbage collector. Remember destroying every object you get, they will not be destroyed automatically.
- The methods will not return objects but references, because returning objects is very inefficient and hard to do on ObjectStore. In the other hand, pointers are dangerous and not clear (they "hide" the objects, and give access to memory without any control).

Indentation

Each indentation level will be two spaces to the right in relation to its container and the braces will be considered part of the inner block.

```
blockheader {
  contents
  }
```

Comments

A program never has enough comments. So, it is desirable to comment almost everything.

Classes: A general comment for each class giving an overview, author, date and version. Between "/*" and "*/". Each line beginning by "*".

```
//========================================================== ClassName
/*
 * Comments about the class.
 *
 * @author authorname
 * @version number.number date
 */
```

Methods: Short explanation about the method, list of parameters (name, type and explanation), return value (type and explanation) and exceptions thrown (which and when). Between "/*" and "*/". Each line beginning by "*".

```
//---------------------------------------------------------- methodName
/*
 * Explanation.
 *
 * @param name type explanation
 * @return type explanation
 * @throw type explanation
 */
```

Variables A short comment for every variable or group of variables. Using one line beginning by "//".

Code: Comment every action of the code using lines beginning by "//". The idea behind this is you can get the algorithm or pseudo-code removing the real code.

Identifiers

Classes: Lowercase except for the beginning of the words (Ex: ClassName).

Variables: Lowercase except for the beginning of words in the middle of the name (Ex: varName).

Methods: Lowercase except for the beginning of words in the middle of the name (Ex: methodName).

Macros: Uppercase with underscores between words for constants and lowercase (like methods) for functions.

Classes

Since an Object Oriented language is going to be used, everything, but the main function, is going to be in a class. The structure (header) of the class will be in a file and the implementation (code) will be in a different one.

```
class ClassName {
  private:
  protected:
  public:
    attributes
  private:
  protected:
  public:
    methods
  }
```

Methods

The methods will not have output parameters. If a method has to return an object, it will be specified as a return value. This has a little problem, the user should remember freeing the memory allocated by the method when the object returned is not needed.

While possible, the objects will be returned as references because it is much more simple and efficient. Besides, if you are working with persistent objects and a copy of them is returned this copy is not going to be persistent anymore.

Files

There will be two kinds of files: header files and code files. Each file will contain either the header or the code of exactly one class. The name of the file will be the same of the contained class plus ".hh" or ".cc" depending on the kind of file.

Error handling

The exceptions will be used to handle errors. It's much more clear than using return values. The compiler should help to know which exceptions you did not catch and which you are actually throwing. However, the C++ compiler does not support all those features, yet.

In order to use the exceptions, a hierarchy should have been defined, but C++ does not keep the actual class of an object when you get it as a superclass. So, if you catch an *Exception* instance you are not able to know which of its subclasses that instance was in. Therefore, a list of constants has been defined to indicate which kind of exception is generated.

Memory lacking has not been handled at all. If the system does not have enough memory to create a user object, it will not have to create an exception either. Therefore this kind of error will be directly handled by the operating system or ObjectStore (for persistent memory).

Appendix B

Compiling

Firstly, you must take into account ObjectStore lays under this BLOOM implementation. Therefore, you will use ObjectStore includes and libraries besides Bloom ones. You also must use the ObjectStore post-linker (os_postlink).

Compiling steps (see figure B.1):

1. Compile the schema source file using the ObjectStore schema compiler (ossg). It has these parameters:

   -assf filename.cc Specifies the name of the application schema source file to be produced by ossg.

   -asdb databasename.adb Specifies the name of the application schema database to be produced by ossg.

   -I$(objectstoreincludedir) Passes the ObjectStore include directory to the C++ preprocessor as directory for finding include files.

   -I$(bloomincludedir) Passes the BLOOM include directory to the C++ preprocessor as directory for finding include files.

   schemafilename Specifies the C++ source file that designates all the types you want to include in the schema.

2. Compile the application schema resulting from the previous step using a C++ compiler. It has the following parameters depending on the compiler you are using:

   -pta Instantiate whole template class.

   -vdelx Call "_vector_deletex_" instead of "_vector_delete_".

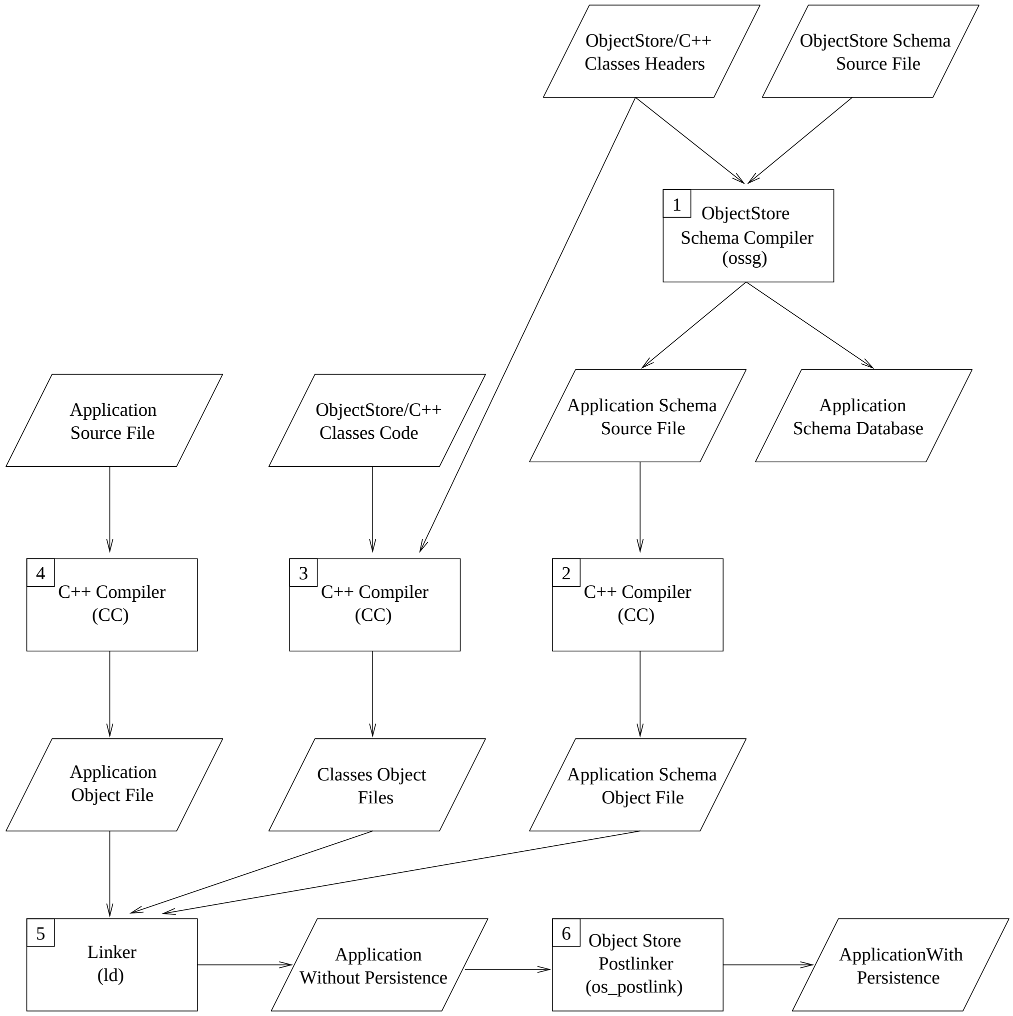

Figure B.1: BLOOM code compilation process

-I$(objectstoreincludedir) Specifies the ObjectStore include directory.

-I$(bloomincludedir) Specifies the BLOOM include directory.

-c Compiles only – produces object files, suppresses linking

3. Compile the data classes (including “AllObjects.cc”, “Boolean.cc”, “Int.cc” and “String.cc”) using the same parameters you used in the previous step.

4. Compile your C++ application using the same parameters again.

5. Link all together (including the object application schema resulting from step 2). Parameters (always depending on the linker you use):

-L$(bloomlibrarydir) -lbloom Adds the BLOOM library directory to the library search path and links with "libbloom.a".

-L$(objectstorelibrarydir) -los -losths -loscol -losqry -losmop Adds the ObjectStore library directory to the library search path and links with "libos.so", "libosths.so", "liboscol.so", "libosqry.so" and "libosmop.so".

6. Treat the application with the ObjectStore postlinker (os_postlink). This step makes persistent all classes you marked so in the schema, all their superclasses and their attributes.