Maintaining derived views on classes with dynamic data efficiently

Master Thesis

by

David Gràcia Llobet

Submitted to the Faculty IV, Electrical Engineering and Computer Science Database Systems and Information Management Group in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
as part of the ERASMUS MUNDUS programme IT4BI
at the
Technische Universität Berlin
July 31, 2015

Thesis Advisors:
Alexander Alexandrov
Anna Queralt

Thesis Supervisor:
Prof. Dr. Volker Markl

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Berlin, July 31, 2015

David Gràcia Llobet

Abstract

The goal of this thesis is to provide an approach for views on an object-oriented database. In order to achieve this, we reviewed the work done by others in the field of views on object-oriented databases. Approaches trying to solve this problem were not fully successful in the past: some of them didn't support fully derived information and some require an extra query language on top of the underlying system to support derived information.

In this thesis, we present an approach that supports derived information but that does not need a query language. It is designed to be used following the object-oriented paradigm. This approach is then adapted to dataClay, an object-oriented storage platform, with also the implementations details of part of the derivations.

Zusammenfassung

Das Ziel dieser These ist es, ein Ansatz zur Sichten in Objektdatenbank. Um dies zu erreichen, wir haben die Arbeit von anderen über Sichten in Objektdatenbank überprüft. Ansätze für dies Thema wurden nicht erfolgreich in die Vergangenheit: einige unterstützen nicht alle Ableitung Operationen und einige erfordern eine extra query Sprache über das zugrundeliegend System um abgeleiteten Informationen zu unterstützen.

In diese These, wir präsentieren einen Ansatz, der unterstützt abgeleiteten Informationen aber ohne eine extra query Sprache. Er ist entworfen für der Benutzer benutzt ihn folgen die Objektorientierte Programmierung. Dies Ansatz ist geeignet zu dataClay, eine Objektorientierte Lagerung Plattform, mit Implementierungsdetails über einen Teil von Ableitung Operationen.

Acknowledgements

I would like to dedicate a few words to the people who helped me through the process of this thesis, not only those that have contributed to it by means of feedback, advice and guidance but also to those that have supported me during this period.

I am very thankful for the opportunity I was given to continue my studies in the Information Technologies for Business Intelligence master program and also to the Barcelona Supercomputing Center, in particular to the Storage Systems group in the Computer Sciences Department, for allowing me to do my thesis with them.

I would also like to thank my advisors in this thesis, Alexander Alexandrov and Anna Queralt, for their advice and guidance through this thesis, both in the research and writing phase, and without it I could not have done this work.

Finally, I want to thank my family and friends for their trust in me and their unconditional support during my studies that have helped me keep working in the difficult moments.

Contents

Chapter 1

Introduction

1.1 Context

In a system, information can be divided between derived and base information. Base information is data entered to the system directly, while derived information is data that has been inferred from existing information in the system (base or derived) using derivation rules[8].

In Figure 1.1, we can see the conceptual model of the example that will be used in the thesis to apply the concepts and options explained. In this model, *Person* is a base type, i.e. non-derived; and the rest of types are all derived ones, but using different kinds of derivation.
For example, the *BlondePerson* type is derived from *Person* and contains only those *Person*s that have their *hairColour* blonde. The *Minor* type is a derived one too, in this case its instances are the *Person*s with an *age* lower than 18. The third derived type is a combination of these two, *BlondeMinor* has the same restriction as *Minor* but applied only to those *Person*s that are *BlondePerson*s. These three derived types don't expand the information already in the system but present it in different ways (more specific in the example given).

Another kind of derivation that does expand the information in the system is to create new information out of the one already in the system. This one is called capacity-augmenting derivation. For example, *Person* has information about *weight* and *height*, from these two, the *bodyMass* attribute is derived by dividing the *weight* by the *height*. This derivation expands the already existing elements with a new attribute inferred from their data.
In addition, the data in a system can also be expanded by deriving new elements from the existing information. An example of this is the type *Match*, representing all the possible matches between those *Person*s that have *hobbies* in common. In this case, the concept of a *Match* was not in the system before but it is inferred from the information in the system.

In the database field, derived information is usually created using views. A *view* is the subset of database elements that corresponds to the result of a stored query on the data.

A type of database are relational databases, in these databases the data is kept in the form of relations that consist of a table with columns and rows. A column is usually called an attribute and corresponds to a field of the data; and

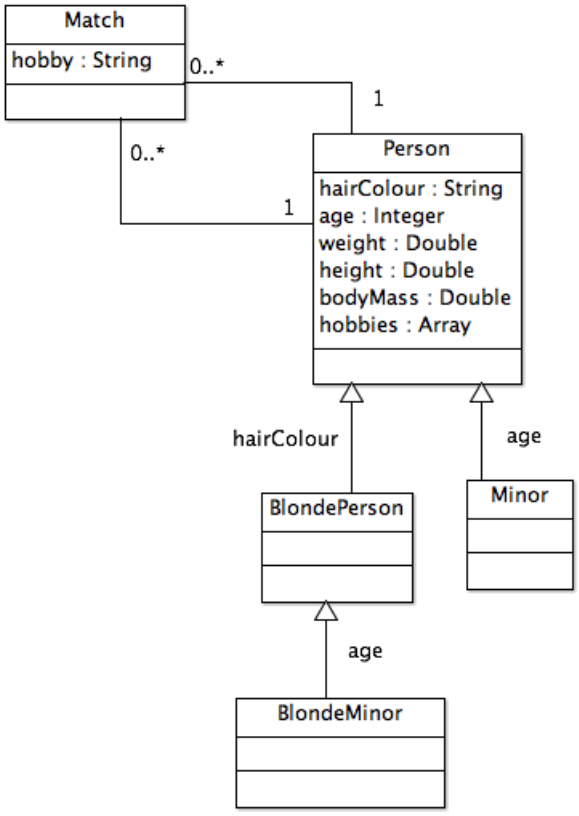

Figure 1.1: Conceptual representation of the *Person* example

a row is commonly called a tuple. The data elements correspond to tuples in a table that contain values for all the columns in the relation.

In relational databases, views are defined over tables or other views and only the definition of the view is stored in the system, the content of the view is calculated whenever the view is accessed.
Some databases offer the possibility to *materialize* views, i.e. to store not only the definition of the view but also its result, this allows for faster access to them since they are already calculated. With this comes the necessity for the system to keep them up-to-date with the changes made on the raw data.

The example of Figure 1.1 would correspond to a relational database with a *Person* table, then one would define the *BlondePeople* information as a view corresponding to a query that selects those tuples in *Person* with the value "blonde" in the *hairColour* column, same with *Minor* and in the case of *BlondeMinor*, it would be defined over the view *BlondePeople* instead of over *Person*. Following the example, the expanding of the elements in the system using their data would be done by creating a view with the same columns as *Person*, but adding another one called *bodyMass* calculated with the *height* and *weight* columns. The *Match*es information would be obtained by creating a view from the data returned by a query that joined the *Person*s that share *hobbies*.

Another type of database are object-oriented databases (OODBs), in contrast with relational ones, the data elements stored in an OODB are objects following the object-oriented model. The mapping to relational concepts is the following: a class contains the attributes that all its instances will have so it is the equivalent to a table, the relational rows correspond to the instances of a class and the columns are the attributes of the class.
Views on OODBs have not been studied as extensively as in relational databases. In the literature, *view* can define two different concepts:

The first meaning for *view* is a class that contains information that has been derived from the data in the system and is maintained by the system. It is similar to the relational definition of view. The information in a derived class can be obtained using different operations such as: selection (creating a subset of the instances of a class), projection (hiding an attribute of a class), union (creating a class that contains the instances of two other classes) and others.
This is the meaning used for a *view* from here on unless specified otherwise.

The second meaning for *view* is known as *view schema*. A view schema is a subschema of the global schema of a database, the global schema of a database is a schema with all the classes in the database, both base and derived. View schemas are customized for a particular application. This is useful so different applications can use different representations of the data model in the database. View schemas rely usually on derived classes as they are the tool to define how a class will be represented in the new schema.

Views in OODBs are also classified depending on what kind of semantics are used to create them:

- *Object-preserving semantics*: Objects that are members of an object-preserving view belong to another class. They do not represent a new concept of the real-world, but the same object seen in a different way. These views can also be capacity-augmenting, for example if they have extra fields with information from outside the system or with information derived from the system. In our example, *BlondePeople* is an object-

preserving view.

- *Object-generating semantics*: In this case, objects are concepts previously not in the system, they are generated by the view from the information already in the system. In our example, *Match*es is an object-generating view.

Some differences between object-oriented views and relational ones are:

- *Members*: The items in relational views are rows of values, whereas in object-oriented ones they are objects, this means that not only the data but the behaviour of the original object may be present in a view member.
- *Updatability*: This is linked to the previous one, in object-oriented views data can be modified and those changes propagated to the raw data, this is possible due to object identity. In many relational systems, views can't receive updates directly on the view data (they are updated when the raw data changes, accordingly to their derivation rules).

1.1.1 dataClay

dataClay is a distributed data storage platform that stores objects following the object-oriented paradigm. The dataClay platform focuses on data sharing between different applications that create, consume and combine shared objects, so it must be able to deal with large amounts of data in an efficient way. The platform supports both Java and Python objects and applications.

The system stores the objects together with their methods, the data itself is encapsulated with its code. Users register classes to the system and they are used as the data model, these classes do not need any kind of mapping which is the case for Hibernate. This allows the user to use the same language for his application and for his storage model.

Once a class is registered to the system, instances of it can be inserted to the system directly so the user uses the same data model for both persistent and non-persistent data. This also implies that accesses to attributes and methods of persistent objects are done as if they were in memory.

Since the data is encapsulated with its code and never separated from it, the data provider can share his data without losing control over it (since the code managing the data will be his). The system also allows classes to be enriched, a class can be extended with additional attributes or additional methods by both the provider and the data consumer. This gives more flexibility to the data consumer regarding how the data is manipulated.

The approach presented in this thesis is going to be implemented on the dataClay system.

1.2 Motivation

As said, views, or derived information, in OODBs haven't been researched as much as the relational ones, though some research has been done.

In the research field, some approaches supported object-preserving views and handled their updates, such is the case for the MultiView system[10]. This system supports views that contain objects from their base classes, which can

also include additional non-derived fields, i.e. object-preserving views, but does not support object-generating ones. These views are maintained up-to-date automatically by the system.
Other approaches focused more on supporting object-generating views, this is the case for the system presented by Samos[12], this approach supports object-generating and preserving views as well as update propagation between derived and base data, the approach was only presented as a theoretical one though.

In current OODB systems derived information is not fully supported, even though some systems provide similar features or features that provide similar behaviour. For example, in the GemStone[13] system, functionality similar to object-preserving views can be achieved through a functionality to store query results, these results are not maintained up-to-date so the query must be re-run to ensure current results.

On the other hand, since views contain query results without the need of processing the query again, they can be used to reduce the amount of queries done to the server during the execution of an application. This is specially important for a platform like dataClay, which is aimed at helping applications access large amounts of data as efficiently as possible, so it would greatly benefit from a view mechanism that provided direct and up-to-date access to objects satisfying a certain condition.
Moreover, thanks to the enrichment functionality provided by dataClay, new information (be it new fields or new classes) can be dynamically added at any time by any user. Thus, being able to expand existing data with derived information could potentially eliminate need for queries during application execution.

Hence, object-oriented systems and in particular dataClay, would greatly benefit from an approach that supports both object-generating and object-preserving views that always contains up-to-date information and allows manipulating it.

1.3 Objectives

The goal of this thesis is to present an approach for derived views in OODBs and bringing them to the dataClay system as a proof of concept.

This approach will support information derived from the data existing in the system, derivation is classified in three kinds:

- *Derived collection*: A class whose instances are the instances of another class that satisfy a condition given by the user. The user can't add nor remove objects from a derived collection directly, since they are maintained by the system, but can alter the membership of an object to a derived collection by changing the value of its attributes. In our example, *BlondePerson*, *Minor* and *BlondeMinor* are derived collections. As explained before, they are going to be updated by the system when an object receives a change in its *hairColour*, *age*, and to any of those, respectively. This has to be done avoiding data duplication, which differs from its relational counterpart; in a relational materialized view, the result is stored in the system, which leads to having data duplicated (the same data may be stored in the original table and in the materialized view).

- *Derived property*: A class attribute whose value is derived from other

information in the system, this value is stored and maintained up-to-date by the system. Changes done in a derived property should be propagated to the base data (if the user does not say otherwise) automatically, how these changes are propagated is specified by the user. The example of derived property in Figure 1.1 is the *bodyMass* attribute in *Person*, this derived property is based on *height* and *weight*. In case there is an update to *height* or *weight*, the value of *bodyMass* will be recalculated; and vice-versa, if the *bodyMass* received an update this could be translated, for example, into increasing that person's *weight* accordingly.

- *Derived class*: The instances of a derived class are created from information in other classes of the system. *Match*es is an example of derived class in Figure 1.1; it pairs *Person*s that have some value in common in their *hobbies* field. In order to keep the derived and base classes consistent, update policies are necessary, these policies are specified by the user so the changes are propagated as he desires. As with derived collections, instances of a derived class can not be created or removed directly by the user, though he can alter them by altering the base information. An example of update policy would be that when a *Person* is created, the system checks the persons in the system and creates the *Match*es with the new *Person*.

As said, the approach presented in this thesis will support the derivations above, and as a proof of concept, derived collections will be implemented in dataClay. This implementation will be following dataClay's spirit of making the persistence of data transparent to the user. The interaction with derived collections should be similar to accessing a non-derived collection, but with limitations. The limitation is that it won't be possible to directly add nor remove objects from a derived collection, instead this will be achieved by updating the value of the property that the derived collection is filtering on.
In this way, instead of running a query, a user will be able to access a derived collection that will always contain all the objects of a class that satisfy a certain condition (e.g. all the blonde people, all the blonde people under 18, ...).

The objective here is that, for the user, derived collections should require only some additional logic when registering the collection and then accessing it from the application should be done like accessing non-derived collections. This will be done only for Java but the approach has been adapted to be suitable for Python as well.

1.4 Structure of the document

The rest of this thesis is organized in four chapters. Chapter 2 presents the dataClay system, explaining its environment and those elements that are relevant to this thesis. The third chapter is dedicated to reviewing the work done by others related to views in object oriented databases and concepts important for this thesis. Chapter 4 presents the approach resulting of this thesis and also how this approach has been adapted to the dataClay system. The final chapter is for future work of this thesis and the conclusions.
This document contains an appendix with the use cases of the dataClay system after the adaptation of the approach presented in this thesis.

Chapter 2

dataClay

dataClay is a data sharing platform that aims to allow data providers to share their data with the consumer without losing control over it but also brings to the data consumer the flexibility to adapt the provided data to his necessities.

To make the provider keep the control over how his data is used, objects stored in dataClay are self-contained objects (SCO).
A SCO is an instance of a class registered to the system and behaves like an object in the object-oriented paradigm (OOP), but it also includes a data API that is used to manage them storage-wise. This includes storing them, which is done by executing a method on the SCO, and retrieving objects in the following ways:

- *Alias*: Users can tag SCOs with a given alias, this alias can be then used to query the system and retrieve that particular object.
- *Query by example*: dataClay offers the possibility to query its data with the query by example technique. This querying technique consists in providing the system with a prototype object and the system returns the instances of that class whose values match the values of the prototype object.

Of course, objects can also be accessed following references like in the object-oriented paradigm.

SCOs are encapsulated with the code of its provider which allows him to maintain control over his data.

In order to maintain the flexibility to adapt the data to the needs of both the consumer and the provider, dataClay allows the data models to be enriched by the data provider and the data consumer (if allowed by the provider).
The enrichment of the data model consists in expanding an already existing model with additional fields or behaviour[7].

The enrichments added to a class will be accessible from all the objects that can access the enrichment, both the already existing objects and objects added after it.
The behaviour of a class can be expanded in two ways by adding:

- *Method*: A class can be enriched with new methods, both the new methods and the original methods can be executed on the SCOs of that class, since the new methods are added to the class with the original methods.

- *Implementation*: In dataClay, methods can have multiple implementations, this concept differs from the one from the object-oriented paradigm though. In the latter, a method can have a different behaviour depending on which class is executed (for example, a subclass overriding a method of its superclass); in dataClay, methods can have multiple implementations in the same class. Different implementations will not change the effect of that method to the system (i.e. they have the same pre and post-conditions) but may differ in terms of their logic (for example, using different sorting algorithms). New implementations to an existing method can be added through the enrichment of classes.

2.1 Model

In dataClay, classes are organized in *namespace*s, which are domains that group the classes that are part of the same package. The class name is the unique identifier of classes within a namespace. The data model provider registers namespaces with their name to dataClay.

2.1.1 Class registration

In order for the user to be able to store objects in the system, he must first register a class to the system. The user registers a class in dataClay by providing the file with its code, the system then registers the given class in the user's namespace.

Once a class is registered, the user can then store instances of it, expand it with enrichments or share it with other users.

2.1.2 Model contract

When a user wants to share his model with another user, he (the data model provider) signs a model contract with the other user (the data model consumer). In order to create a model contract, the provider must define:

- *Set of interfaces*: An *interface* is a customization of a class for the contract. For each class the provider desires to share with the consumer, he must create an interface of it with the fields, methods and implementations that will be accessible through the contract.
- *Permissions*: The kind of enrichments, if any, that the consumer can do to the shared interfaces: fields, methods or implementations.
- *Expiration date*: The date in which the model contract expires.

The consumer can then use the classes included in the contract by retrieving them as stubs. A stub is a bytecode class file that represents a class, it is generated by dataClay from the contracts each user has. The stub of a particular class contains the union of the fields, methods and implementations in the interfaces of that class that the user can access through contracts, and a set of operations from the dataClay API:

- *makePersistent*: stores the SCO in the system, from this point on all the changes done to the object will be made persistent automatically. This method can also associate an alias to the object in order to retrieve it by alias.
- *deleteObject*: deletes the SCO from the system.
- *getByAlias*: if a tag is provided when the object is stored to the system, this method can be used to retrieve the SCO reference of the instance associated with the given tag.
- *getAlike*: retrieves all the references of the SCOs that have the same values as the given object (the prototype object).

2.1.3 Enrichment

The data model can also be changed through enrichments. As has been explained before, the provider and the consumer (if allowed by the contract) can expand a class with new fields, methods or implementations.

In order to do so, consumers download from dataClay the classes, each class contains the fields and the signatures of the methods visible to him through the data model contracts he is part of. The consumer adds to this class the new fields, new methods or new implementations (which is done by implementing the existing method headers of the class). The process finalizes with the user uploading the modified files to dataClay, the system then updates the original class to add the new fields and behaviour. These enrichments can be shared through new model contracts by their creator.

2.2 Data

2.2.1 Objects and datasets

As we have already mentioned, the data in dataClay is stored in the form of objects, in particular in the form of self-contained objects.

The objects stored in the system are grouped in datasets. A dataset is the enclosure of a set of SCOs which may be instances of different classes. Datasets are registered by the data owner to dataClay with a process analogous to registering a namespace.

2.2.2 Data contract

Datasets ease the task of defining how objects are shared between users. To share a set of objects, the data owner signs a data contract with the data consumer. In order to sign such a contract, the data provider defines:

- *Datasets*: The datasets whose SCOs the consumer will be able to access through the data contract.
- *Permissions*: The provider specifies whether the consumer is able to add new objects to the datasets or not.

- *Expiration date*: Like in a model contract, the data provider defines the expiration date of the contract.

Model and data contracts are the two different tools to share information between users in dataClay. A model contract defines what parts of the classes are shared, while the data contract defines which datasets (and which SCOs by extension) are shared.

Chapter 3

Related work

In this section we are going to review the state of the art regarding derived information in Object-Oriented Databases.
After that, the most relevant concepts from the related work for my thesis will be explained in more detail.

3.1 Views as classes

The MultiView[10], Tanaka et al. [14] and Peng et al. [9] approaches focus only on object-preserving views, which are those that only contain objects from previously existing classes.

In the MultiView system[10], views (called virtual classes in the paper) are defined with an object-oriented query using an object algebra given by the authors; views are defined over a set of base classes or other views.
The system supports object-preserving views (it does not support derived properties nor object-generating views) by using a technique called object-slicing (explained in Section 3.3.1). This mechanism allows the underlying system to support necessary features for the MultiView system (multiple classification, dynamic reclassification, etc).
These views are maintained up-to-date by the system, in order to do so, when a view is added to the system the user must also specify a set of properties that are relevant to the membership of objects to the class. This information is used in the *registration service*[5] to update the corresponding views whenever the properties to which they are registered receive an update. Views can also have additional non-derived attributes.

The approach by Tanaka et al. [14] is similar to MultiView but has no functionality to add an additional attribute to a view.

The approach presented by Peng et al. [9] supports object-preserving too but with an object deputy model. In this model, a real-world object has several deputy objects, which correspond to a custom representation of the original one (called the source object).
Deputy objects can inherit attributes and methods from the original objects but switching operations are needed. A switching operation translates an inherited attribute of a deputy object to the source attribute, in the case of inherited methods, the switching operations translate the parameters.

This approach also supports derived properties, for each derived property, an update propagation operation must be defined that transmits the changes done in the derived property to the original properties.

Other approaches, like those presented in Abiteboul et al. [1], Guerrini et al. [3], Samos et al. [12] and Alhajj et al. [2], support both object-preserving and object-generating views (views that contain objects that do not exist in other classes).

In Abiteboul et al. [1], a system is presented that supports object-preserving and object-generating views (views are called virtual classes and the objects created for an object-generating view are called imaginary objects).
Views are defined by its population, its position in the class hierarchy and the behaviour of its objects. The first one is specified by the user and the latter two are derived by the system. The user has three ways to specify the population of an object-preserving view: stating that the view is a superclass of existing classes (by generalization), declaring that its population is the result of a given query (by specialization) or stating that the view contains all the objects that have a particular behaviour (by behavioral generalization). For object-generating views, the user defines a query that returns a set of tuples and then the system attaches a new object identifier to each value. New objects can not be directly inserted to views, regardless of its type.

The approach presented in Guerrini et al. [3] keeps views in a hierarchy separated from the class one. Therefore a class can only inherit from a class and a view can only inherit from a view. Though, a base class and a view are related through a derivation relationship which relate the view to the base classes it is derived from. The derived classes are defined using a view definition language specified by the authors.
In addition to object-preserving and object-generating views, the system supports another kind of views called *set-tuple* views. A set-tuple view is a view whose objects do not need persistent identifiers hence they contain values instead of objects.

In Samos et al. [12], both object-preserving and generating views are based on classes or other views. The new objects in an object-generating view are created by processing the view definition. When defining a class, the user must specify the *core attributes* of the derived class, these are the attributes that are used to derive the object identifiers of the instances in the view. For object-preserving views the only core attribute is the object identifier of the base class. For update transmission, they proposed to have methods that propagate changes between the base and the derived classes, these are defined in the *derivation relationship*. This is a link between the base classes and the derived one that includes a propagation method for each modification method in the base and derived classes. This link is not present in the model but only in the data dictionary.
Derived classes can't have additional attributes that are not derived from the system.

In Alhajj et al. [2], the approach supports both types of views and it presents an additional type of view: *brother* classes. A brother class is an object-preserving view that has the same model as its base class and is obtained from its it using a selection predicate, in our approach this corresponds to the derived collections.

Hence there is no approach that supports object-preserving and object-

generating views but that does not limit the derived classes to have all their data derived from the system.

3.2 Views as schema

The MultiView system, presented by Rundensteiner [10], aims to give support to having multiple view schemata in an OODB, they do so by keeping a global schema that contains all base classes and virtual classes in the system. When a new virtual class is defined, it is placed by the system in the right position in the class hierarchy: this is done by analysing its definition and identifying how it is related to other classes regarding subsumption, the virtual class is then integrated in the hierarchy.
A view schema is closed when all classes that are used in a class of the schema are contained in the schema too, the closure property ensures that no class that belongs to a schema references a class that does not belong to it[15]. An algorithm for checking this closure property is used in the system to ensure that all the view schemas are closed (the system will automatically generate a closed schema from a non-closed one).

The system presented by Tanaka et al. [14] is a schema virtualization system. A schema is created from the base one and then the virtual classes are defined in it, after this the relationships between virtual classes are specified by the user. Schemas can be merged with other view schemas and deleted from a bigger schema.
In this approach, the integration of virtual classes is done during the creation of the view schema instead of being two different steps like in the MultiView system. Also the relationships among the classes in a view schema are specified by the user which could lead to inconsistencies in the schema. Finally, view schemas in this approach may not be closed.

3.3 Important concepts from related work

3.3.1 Object-slicing technique

This is an object representation technique[4] used in the MultiView system[10] to make the underlying OODB system support some features required by MultiView.
These requirements are:

- *Multiple classification*: An object can be member of more than one class at the same time.
- *Dynamic reclassification*: An object must be able to change its membership dynamically.
- *Dynamic restructuring*: A type must be modifiable dynamically, so the attributes or methods that a class has can be changed.
- *Capacity-augmenting virtual classes*: A virtual class that does not only contain information derived from its base classes but also non-derived new information.

Object-slicing underlying concept is that a real-world object corresponds to a hierarchy of implementation objects linked to a conceptual object. The conceptual object represents the object's identity in the real-world. An implementation object, on the other hand, corresponds to one of the classes the real-world object belongs to (hence, there will be as many implementation objects for a real-world object as classes it belongs to).

As said, a conceptual object represents the object in the real-world. It keeps

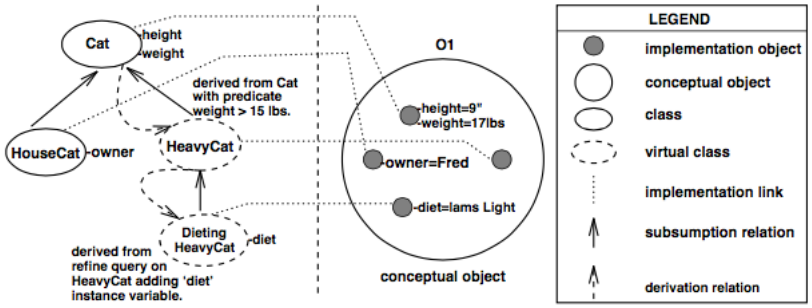

Figure 3.1: Object slicing example [11]

a hierarchy of implementation objects that corresponds to its type hierarchy. For each type, the identifier of the implementation object corresponding to it is kept.

An implementation object, on the other hand, has its own identifier and a reference to the conceptual object it represents. It also contains the methods that are specific to the class it is an instance of. An example of object-slicing can be seen in Figure 3.1, where the system contains two non-derived classes (*Cat* and *HouseCat*) and two derived classes (*HeavyCat* and *DietingHeavyCat*). An object that belongs to all four classes is depicted in the figure, this is represented by a conceptual object and four implementation objects, one for each class, each implementation object contains the functionalities particular to its class.

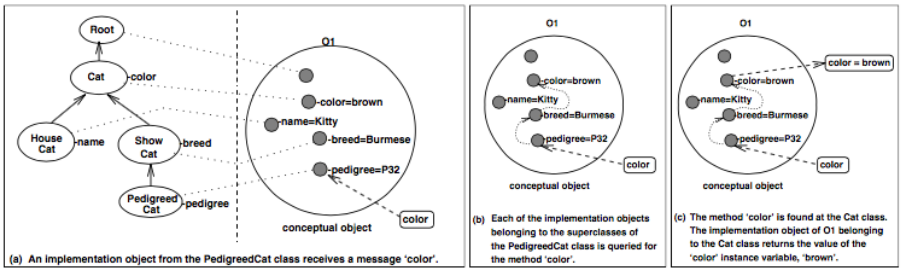

Figure 3.2: Object slicing inheritance [4]

Object slicing has its own inheritance mechanism: when a method is called on an implementation object, in case the method is defined in the class of the object, it can be executed. Otherwise, the implementation object delegates the execution to its conceptual object. This will do an upwards search through the

superclasses of the initial implementation object. Once the class that defines the given method is found, the method will be executed on the implementation object belonging to that class.

An example of this is shown in Figure 3.2. In the picture, the system consists of four classes (*Cat*, *HouseCat*, *ShowCat* and *PedigreedCat*) and an object with implementation objects for all the classes. A method defined in the *Cat* class is executed on the *PedigreedCat* implementation object, this execution is therefore delegated to the superclasses of *PedigreedCat* until the class that defines the method *color* is found, which returns the value of the *color* property.

3.3.2 Registration service

This concept is related to transmission of modifications between the base and virtual classes in the MultiView system[6]. The *registration service* is a service that notifies virtual classes when they must reassess the membership to them of an object.

In our example, we have a base class *Person* and a virtual class *BlondePerson* with the instances that have blonde hair. If the system reevaluates the membership to *BlondePerson* of a *Person* each time the instance is updated, this would lead to having to process a lot of virtual classes with every update.

In order to reduce the amount of virtual classes that need to be checked every time an update is made, the concept of registering the virtual classes to fields was introduced.
When a virtual class is created in the system, the user specifies which fields' updates could potentially affect the set membership of the new virtual class. With this, the system will only notify a virtual class when the value of a property it is registered to has been updated.

In the running example, the virtual class *BlondePerson* is registered to the field *hairColour* of *Person*, so only when a *Person* has its *hairColour* modified, the membership to *BlondePerson* of that instance is assessed.

The amount of processed virtual classes after an update is further reduced with the introduction of *early branch termination*.

In our example, *BlondeMinor* (virtual class registered to *age*) is based on *BlondePerson* (registered to *hairColour*) which is based on *Person*. In case a *Person* with black hair had its *age* modified, the system would reassess its membership to *BlondeMinor* since that virtual class is registered to that property. This would be done even though that instance will not be a member of *BlondeMinor* due to its hair not being blonde. Early branch termination covers this case.

Early branch termination consists in making virtual classes reassess an object's membership to them, only when it is an instance of its parent virtual class and it has received an update to the value of a property the virtual class is registered to. In Figure 3.3 you can see an example of these two concepts applied. In this case the *VeryLightCat* virtual class would only be reprocessed when there was an update to the *Cat.weight* field that made its membership to *LightCat* change.

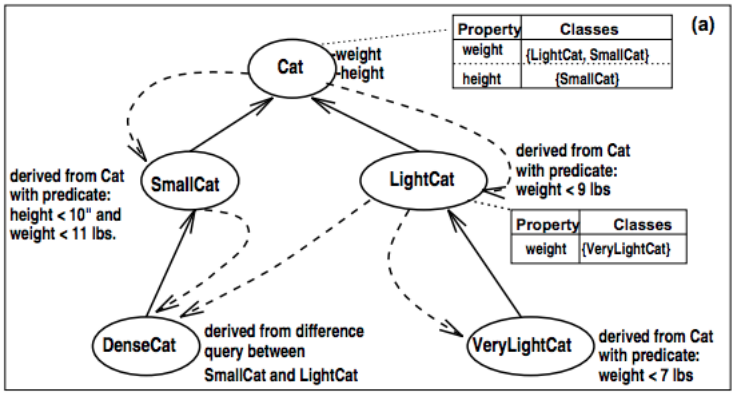

Figure 3.3: Registration service with early branch termination[11]

3.3.3 Derivation relationship

This concept is found in the work done by Samos[12]. A derivation relationship is the relationship between a derived class and its base classes, and it defines how the derived class is obtained. The derivation relationship is also going to be used to propagate the updates between the derived class and the base classes.

Like in other OODBs, the objects are identified by their identifiers, in the case of base objects these are given by the system, in the case of derived objects their identifiers depend on how the derived class has been defined. If it has been defined by object-preserving semantics, then it maintains the identifier of the base object. Otherwise their identifiers are created from a set of attributes called core attributes, this means that no two objects in a derived class will have the same values in the core attributes.

Kinds of derivation relationships

- *Derivation relationship of identity*: Relationship when a derived class uses the objects in the base classes to create its object identifiers.
- *Derivation relationship of value*: Relationship when a derived class does not use the objects in the base classes to create its object identifiers but only to create its values.

For us the interesting kind of derivation relationship is the one of value. Since in our approach the identifiers of derived objects are not based on the base objects.

Transmission of modifications

Derivation relationships can be further classified in static or dynamic:

- *Static derivation relationships*: They are defined between the set of base classes and the derived class and they show the correspondence between the base objects and the derived ones.

- *Dynamic derivation relationships*: They are the combination of a static derivation relationship and a translator (or update policy) which specifies how to transmit the changes done in the derived objects to the base objects.

In the work done by Samos, the derivation relationships contain the update policies, these are methods that, for each possible modification in both the derived and base classes, propagate said changes to the counterpart, i.e. a method of the base class that modifies it has an update policy method that propagates this change to the derived class and vice-versa.

Chapter 4

Solution

In this section, the approach resulting of this thesis is presented. We start by explaining the general approach, this consists in the theoretical system that addresses the issues tackled by this thesis. After that, we will discuss how this approach has been adapted to the dataClay system.

4.1 General approach

The general approach to our solution is divided in the three kinds of derivation already mentioned: derived collections, derived properties and derived classes.

For each derivation kind, we will briefly describe them and will explain how they are defined and maintained by the system.

For maintaining the derived information up-to-date, all three derivation kinds use a service similar to the registration service presented in Section 3.3.2, the *subscription service*. This service contains the information needed to propagate updates and maintain the derived information fresh. We will explain how each derivation interacts with this service.

4.1.1 Derived Collections

Derived collections are classes maintained by the system that contain all the elements in another class (its base class) that satisfy some conditions, this can be seen as keeping the result of a query materialized. Derived collections can be based on both non-derived and derived classes. Once a derived collection is created, the user interaction with it is purely for accessing its elements. The user can't add an element to a derived collection nor remove one from it directly, though he can modify the object's values (which could affect the membership of this element to the derived collection).

Our approach for derived collections has been based on the object-slicing technique (Section 3.3.1). These were the key points that made it suitable for supporting derived collections:

- *Multiple classification*: Multiple classification is necessary for a system to support an object being member of multiple classes at the same time (i.e. an object being member of several derived collections).

- *Dynamic reclassification*: With object-slicing, objects can dynamically change their membership state to a class. This is important for our approach so it is easy to update the derived collections that an object is member of.

As in object-slicing, in our approach an object will be represented by an original object (called conceptual object in the object-slicing technique) and a set of implementation objects.

The original object will be the object that was inserted to the system directly. An original object contains a structure for the implementation objects information, this structure contains the identifiers of all its implementation objects and the derived collections of each implementation object.
An implementation object represents that the original object is member of that derived collection. Implementation objects can have extra fields specified by the user which will be defined in the derived collection and only the objects members of the derived collection will have them. For example, the derived collection *Minor* could have an attribute called *tutor* that contained the legal tutor for that minor, this field would be stored in the *Minor* implementation objects.

Additionally, an implementation object will contain the following information:

- **OID**: The object identifier of the implementation object itself.
- **BOID**: The object identifier of its base object. This is the identifier of the implementation object in the derived collection's base class that represents the original object. This is going to be used for method execution as will be explained further below.
- **OOID**: The object identifier of the original object. This is going to be used also for method execution but also for equality assessment.
- **oDictionary**: A dictionary with the methods of the original object, this is going to be used for method execution too.

Derived collection definition

In order to create a derived collection, the user must provide the following information to the system additionally to the class that represents the derived collection:

- **Filter**: The filter of a derived collection is a boolean method that returns, for a given object, whether the object should be a member of the derived collection or not. In our example, the filter method of the *BlondePerson* derived collection would be:

```
BlondePerson.filterMethod(Person p){
  return p.hairColour == "blonde"
}
```

- **Base class**: The class on which the derived collection is based. The set of all instances of the base class (its extent) is a superset of the extent of the derived collection and it is the set of potential members for the derived

collection. In the *BlondePerson* case, the base class would be the *Person* class.

- **Subscription information**: The properties to which the derived collection needs to be subscribed to. These are the properties relevant for filtering the objects of the base class that need to be represented in the derived collection, i.e. the attributes important to determine whether an implementation object is created or not for a given base object. In the *BlondePerson* case, it would be registered to the *hairColour* property.

When the user creates a new derived collection, it will be subscribed to the properties given. The system will then go through the extent of the base class and execute the filter method on the objects. For each object that satisfies the condition, an implementation object of the derived collection will be created. When an implementation object is created, its *oDictionary* is set using the base class given by the user.

Derived collections and the subscription service

For derived collections, the subscription service keeps information regarding their base class and the properties they are subscribed to. The user specifies both of them when he creates the derived collection.

The subscription service keeps, for each property, which derived collections are subscribed to it. Once a property is modified, the subscription service notifies the subscribed derived collections so they can assess whether the updated object still belongs to the derived collection or not, this is done by executing the filter method on the updated object. If the updated object used to belong to the derived collection but it doesn't satisfy the filter any more, the implementation object is deleted; if it used to not belong to the derived collection but now satisfies its filter, an implementation object is created.

While for the base classes, it keeps for each class the set of derived collections that are based on it. This is used when an object is created, then the service notifies the collections that are based on the class of that object so they can process the new object.

The behaviour of the subscription service for derived collections is the following:

- **New object**: When an object (original or implementation) is created, the object is checked against the filter methods of the derived collections that have its class as base, if any. For each derived collection, in case the object satisfies the filter of the collection, the system creates an implementation object of that derived collection.

- **Property updated**: If the updated property has any derived collections subscribed to it, the service will update the object's membership to them. This means that their filters will be checked against the updated object and in case the membership to a derived collection has changed, it will be updated accordingly (if the object becomes a member of the derived collection, an implementation object for that derived collection is created; otherwise the corresponding implementation object is deleted).

- **Object deleted**: In case an object (original or implementation) is removed, the service will delete the implementation objects that were based on the removed object.

This behaviour applies in the following way to our *Minor* example (*Minor* (M) is a derived collection based on the *Person* (P) class and it's subscribed to the *age* (A) property; and *filterM*(o) being true means that the object o satisfies the filter for the *Minor* derived collection):

- **Insert**: An object o is inserted to P: if *filterM*(o), then $M = M \cup \{o\}$
- **Update**: An object o receives an update in the A property:
  - if *filterM*(o) = true: then if $\{o\} \notin M$, then $M = M \cup \{o\}$
  - if *filterM*(o) = false: then if $\{o\} \in M$, then $M = M - \{o\}$
- **Delete**: An object o is deleted from P: if $\{o\} \in M$, then $M = M - \{o\}$

The subscription service also uses the *early branch termination* concept explained before in order to optimize its derived information maintainance.
Our example also benefits from early branch termination, the behaviour of the subscription service in the case of *BlondeMinor* would be (*BlondeMinor* (BM) is a derived collection based on the *BlondePerson* (B) derived collection which is based on the *Person* (P) class; B is subscribed to the property *hairColour* (H) and BM is subscribed to the property *age* (A); and *filterX*(o) being true means that the object o satisfies the filter for the derived collection X):

- **Insert**: An object o is inserted in:
  - **P**: if *filterB*(o), then $B = B \cup \{o\}$
  - **B**: if *filterBM*(o), then $BM = BM \cup \{o\}$
- **Update**: An object o receives an update in the property:
  - **H**: if *filterB*(o) is:
    - **True**: then if $\{o\} \notin B$, then $B = B \cup \{o\}$
    - **False**: then if $\{o\} \in B$, then $B = B - \{o\}$
  - **A**: if $\{o\} \in B$, then if *filterBM*(o) is:
    - **True**: then if $\{o\} \notin BM$, then $BM = BM \cup \{o\}$
    - **False**: then if $\{o\} \in BM$, then $BM = BM - \{o\}$
- **Delete**: An object o is deleted from:
  - **P**: if $\{o\} \in B$, then $B = B - \{o\}$
  - **B**: if $\{o\} \in BM$, then $BM = BM - \{o\}$

Method execution

An implementation object must be able to execute not only those methods defined in its derived collection but also those defined in another class of its set of base classes (its base class, the base class of its base class and so on). In order to support this, two approaches were studied:

Option A

This alternative is closer to the original MultiView solution than the second one, in this option we need to keep a hierarchy of the derived collections ordered by parent-child relation. Also the implementation objects would not need to keep the base object identifier nor the dictionary of the original object methods (*BOID* and *oDictionary* respectively).
This is because the method invocation process when a method is invoked in an implementation object is the following:

1. If the method is in the definition of the derived collection to which the implementation object belongs, the method is executed on the implementation object.

2. Otherwise, it delegates the execution of the method to the original object.

3. The original object looks through the hierarchy of derived collections of which it has an implementation object to find the one that defines the given method.

4. In case it finds it, the method is invoked on the implementation object that corresponds to that derived collection.

5. If no derived collection that the object is member of defines the given method, an error is given to the user.

Option B

The second option, and the one we actually went for in our approach, does not need any hierarchy. The description of implementation and original objects for this alternative is the one given at the beginning of this section.
The method invocation process in this case when a method is invoked in an implementation object is as follows:

1. If the method is in the definition of the derived collection to which the implementation object belongs, the method is executed on the implementation object.

2. Otherwise, it first checks the dictionary of methods of the original object (the attribute *oDictionary* mentioned previously), and in case the invoked method is in the *oDictionary*, the method is delegated to the original object (using the *OOID*) on which will be invoked.

3. In case the invoked method is not in the *oDictionary*, the implementation object delegates the method to its base object, the object identified with the *BOID*.

4. If the base object defines the invoked method, it will execute it.

5. Otherwise, it will delegate again the method to its base object and so on until the derived collection of a base object defines the method.

6. In case that the original object is reached, it will give an error to the user.

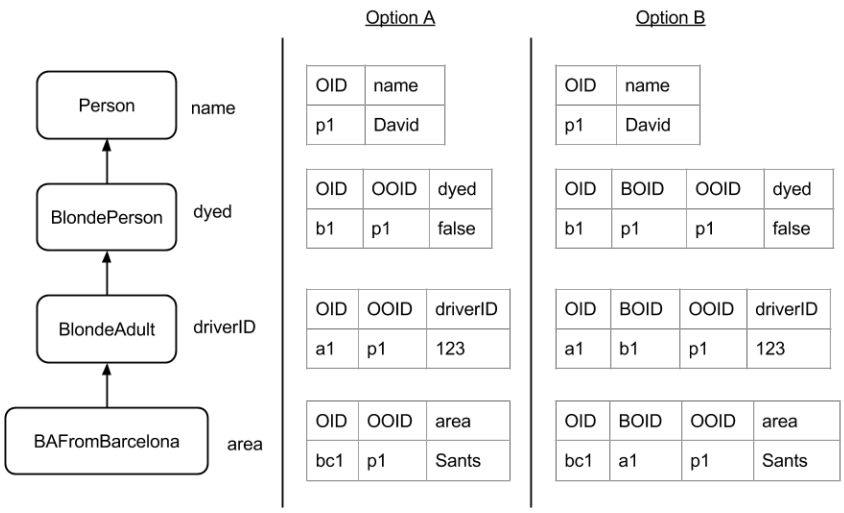

Figure 4.1: Example of options A and B of method execution

An example of the two options is shown in Figure 4.1. In this example, there is a class *Person* which has an attribute *name* (attributes not relevant to this method executing example, such as the properties being filtered on (*hairColour*, *age* and *city*) and the *oDictionaries* for Option B, have been omitted) and three derived collections with a property (and therefore a method, the method to retrieve that property) for the implementation objects of each derived collection:

- *BlondePerson*: Derived collection based on *Person* that has an attribute *dyed* that contains whether the colour is natural (false) or artificial (true).
- *BlondeAdult*: Derived collection based on *BlondePerson* that has a field with the driver license number (*driverID*).
- *BAFromBarcelona*: Derived collection based on *BlondeAdult* that has a property with the neighbourhood within Barcelona that person lives in (the *area*).

These classes can be seen in the left side of the figure. In the figure, the same example applied to options A and B is shown in the middle and right sections respectively.
The behaviours explained before for each option would apply to this example in the following way when the method to retrieve the *driverID* property is executed on the object *bc*1 (the implementation object of the derived collection *BAFromBarcelona*):

- **Option A**: *bc*1's class (*BAFromBarcelona*) does not define the method *driverID*, therefore the execution of this method is delegated to its original object, object *p*1. This object looks through the hierarchy and identifies that the derived collection *BlondeAdult* defines the method *driverID*, the method is then executed on the implementation object of that derived collection, object *a*1.

- **Option B**: $bc1$'s class (*BAFromBarcelona*) does not define the method *driverID*, therefore it checks whether the method is in the dictionary of methods of the original object (*oDictionary*) and, since that is not the case, it delegates the execution to its base object (the object *BOID*), object $a1$. $a1$'s derived collection (*BlondeAdult*) does define the method, therefore the method *driverID* is executed on $a1$.

The strong point for option A is that implementation objects do not need to keep as much information and therefore will be more lightweight than option B's implementation objects. And its weak point is the fact that there is the need to keep a derived collection hierarchy in the original object (*oDictionary*). With option B, there is no need for a hierarchy so the implementation objects information is simplified in the original object. The implementation objects, though, have additional attributes to keep one more object identifier (the *BOID*) and the dictionary of methods of the original object.
Since most of the methods that can be invoked in an object will be defined in the original object rather than in one of the rest of its base derived collections, with option B the system will be able to discard the original class without having to delegate the method execution to it (by simply checking the *oDictionary*). As opposed to having to delegate the execution to the original object and then check whether the method is defined in that class or not, which would be option A's behaviour.

4.1.2 Derived Properties

Derived properties are attributes whose value has been derived from some base objects. They can be added to an already existing class or be defined in a newly registered class.

In order to support derived properties, each object that has any derived properties will keep track of which base objects were used in order to obtain their value. In addition, each object (even if it has no derived properties) must maintain information about the derived properties it is a base object of. This information is necessary for update propagations.
So, objects will need the following additions:

- **Base objects map**: A map-based structure that links each derived property the object has, to the set of base objects that were used in deriving said field. This will be used when the value of a derived property is changed.

- **Derived properties set**: Set of the derived properties that the object is used as base object. This is going to be used when the object receives an update, so it can be propagated to these derived properties.

Derived property definition

In order to create a derived property, the user must provide the system with the following information:

- **Property creation method**: The method that gives value to the derived property of an object. This method will be also used to recalculate the value of the derived property when its base information changes. In this

method, the base objects must be identified and the connection between them and the derived property made. With the structure explained above, this means adding the base objects for this derived property to the map and also adding the derived property to the base objects' set of derived information. The creation method of the *bodyMass* derived property in our example would be the following:

```
Person.bodyMassCreationMethod(){
  this.bodyMass = this.weight/this.height
  this.setBaseObjects("bodyMass",this) //This
      ↪ method makes the connection between the
      ↪ derived property and the base objects
}
```

- **Update propagation method**: Update transmission method that propagates the changes done in the derived property to its base objects. The user can make the derived property read-only by not providing an update propagation method. For the *bodyMass* derived property, the update propagation method would be:

```
Person.bodyMassUpdatePropagationMethod(){
  this.baseObjects("bodyMass").weight = this.
      ↪ bodyMass*this.baseObjects("bodyMass").
      ↪ height
}
```

- **Subscription information**: The properties to which the new derived property needs to be subscribed to, for keeping it up-to-date. In the *bodyMass* example, the derived property is subscribed to the properties *height* and *weight*.

When a derived property is added to the system, the properties to which the derived property is subscribed are added to the subscription service. The system then iterates the extent of the derived property's class and gives value to the property for each object with the *property creation method*.

Derived properties and the subscription service

For derived properties, the subscription service keeps information regarding the properties they are subscribed to in order to keep them up-to-date. This is provided to the service when the derived property is defined in the system.

The subscribed properties serve the same purpose for derived properties as they do for derived collections.

Whenever the value of a property is changed, the subscription service notifies the objects, if any, that have derived properties subscribed to the updated property. Then, the derived property is recalculated with the property creation method.

When the derived property is updated, the subscription service will propagate the changes to the base objects. In order to do so, the system will execute the update propagation method given by the user. In this way, the changes done to the derived property are propagated to the base objects as the user desires.

If an instance of a class that contains a derived property is created, the system executes the creation methods of the derived properties in order to initialize them.

So in the *bodyMass* example, the behaviour of the system is as follows:

- **Insert**: A new *Person p* is created: the system executes the *bodyMassCreationMethod* on *p* to initialize the *bodyMass* derived property.
- **Update**: A *Person p* receives an update in the property:
  - **Weight**: The system executes the method *bodyMassCreationMethod* on *p* to update the value of *bodyMass*.
  - **Height**: The system executes the method *bodyMassCreationMethod* on *p* to update the value of *bodyMass*.
  - **bodyMass**: The system executes the method *bodyMassUpdatePropagationMethod* on *p* to propagate the changes to the base objects of the derived property.

4.1.3 Derived Classes

A derived class is a class whose objects are created from the derived class definition. An instance of a derived class is created from other objects in the system (its base objects). The difference between a derived class and a derived collection is that the instances of the first are created based on other information in the system whereas the instances of the second are just a different representation of the objects in the system.

The derived class members are maintained automatically by the system using the update information provided by the user. The derived class mechanism only creates or deletes the instances from the system, any extra property of a derived object can be derived, using the derived property mechanism explained in Section 4.1.2, or non-derived.

The approach to derived classes is very similar to the one used for derived properties, so objects need to keep extra information in order to keep derived classes up-to-date.
A derived object will contain a base objects set with the information regarding which objects have been used as base objects in order to create the instance. Also, objects (derived or non-derived ones) will have a derived objects set with the identifiers of the objects that have been derived from them (the derived objects that have them as base objects).

Derived class definition

In order to create a derived class, the user will provide the system with the following information:

- **Initial creation method**: This method creates all the objects of the derived class. It is meant to be used by the system only once, in the class creation moment. The connection between the base objects and the derived object must be made in this method too. The initial creation method for *Match*es would be:

```
Match.initialCreationMethod(){
  foreach Person p in Persons //Persons is a
     ↪ collection with all the instances of the
     ↪ class Person
    foreach Person p1 in Persons[p:] //Persons[p:]
       ↪ is a slice of the collection that starts
       ↪ in the Person right following p until the
       ↪  end of the collection
      foreach String h in p.hobbies
        foreach String h1 in p1.hobbies
          if h == h1 then
            m = new Match(p,p1)
            m.setBaseObjects([p,p1]) //[p,p1] is
                ↪ understood to be a collection
                ↪ with the objects p and p1
}
```

- **Update propagation methods**: The user must provide the system with the methods to maintain the derived classes up-to-date. Each method in the base classes or in the derived class that can affect the extent of the derived class, i.e. every method that may make an instance of the derived class to be removed or created, needs to have an update propagation method that updates the derived class. This also includes a method to propagate when an object of the base class is created and one for when an object is deleted. In our example, the update propagation methods for the *Match* class would be:

```
Match.personCreated(Person newP){ //newP is the
   ↪ newly created Person
  foreach Person p in Persons
    foreach String h in newP.hobbies
      foreach String h1 in p.hobbies
        if h == h1 then
          m = new Match(newP,p)
          m.setBaseObjects([newP,p])
}

Match.personDeleted(Person p){ //p is the deleted
   ↪ Person
  foreach Match m in p.getDerivedObjects("Match")
    m.delete() //This method deletes the object
       ↪ from the system and updates the derived
       ↪ objects set of its base objects
}

Match.hobbiesChanged(Person p){ //p is the updated
   ↪ Person
  foreach Match m in p.getDerivedObjects("Match")
    m.delete()
  foreach Person p1 in Persons
```

```
        if p != p1 then
          foreach String h in p.hobbies
            foreach String h1 in p1.hobbies
              if h == h1 then
                m = new Match(p,p1)
                m.setBaseObjects([p,p1])
}
```

- **Subscription information**: For derived classes, the subscription information consists in which update propagation method in the derived class is related to which method of the derived class or of the base classes, also for each base class the user must specify which update propagation method is used to propagate creations and which for deletions of objects of that base class. In the *Match*es example this would consist in specifying that the update propagation method *personCreated* is linked to the creation method of the class *Person*, *personDeleted* is linked to the deletion method of the class *Person* and *hobbiesChanged* to those methods that update the value of *hobbies*.

When a derived class is created, the system will link the update propagation methods with their associated methods and then will execute the initial creation method to generate the derived objects.

Derived classes and the subscription service

The subscription service is only responsible for keeping the derived classes instances up-to-date. Instances of the derived class will be added or removed by the system using the methods that the user provides at registration time.

Since every update propagation method is associated with a method of the base or derived class, whenever a method is executed, the subscription service will execute the update propagation methods associated with it and this update will be propagated as specified by the user.

In case an instance of one of the base classes is added or removed from the system, the subscription service will execute the update propagation method associated with the insertions or deletions of that base class.

An example of this behaviour for the derived class *Match* can be seen as follows (*Match* has as base class *Person*; in registration time the user provided three update propagation methods: *personCreated* associated with the insertions of *Person*s, *personDeleted* associated with the deletions of *Person*s and *hobbiesChanged* associated with the methods of *Person* that modify the *hobbies* property):

- **Insert**: A new *Person p* is created: the system executes the *personCreated* method passing *p* as argument, since it is the method associated with the creation of a *Person*.

- **Update**: A *Person p* receives an update in the property *hobbies*: the system executes the *hobbiesChanged* method passing *p* as argument.

- **Delete**: A *Person p* is removed from the system: the system executes the method *personDeleted* with *p* as argument.

4.2 Adaptation to dataClay

The approach presented in the previous section has been adapted to the dataClay system, in this section we will explain how this has been done.

We will start by reviewing the modified conceptual model of dataClay and the use cases relevant to derived information.

After this, the details in some of the implementation decisions for derived collections are given, this implementation has been designed and started for derived collections only. And finally, an example of how derived collections are used in dataClay will be presented.

For the adaptation to dataClay of the derived collections, there have been some important changes in the general approach. Even though these changes will be explained in the implementation subsection, they are reflected in the conceptual model and in the use cases, and therefore we must note them before going through the conceptual model and use cases.

In dataClay, derived collections will not use the object-slicing technique and therefore will not be classes with instances (the implementation objects in the general approach) but collections of objects, the reasoning behind this is found in the implementation subsection. This also means that instead of a base class, a derived collection will have a base collection (even though it can be the collection that contains all the instances of a class).

Additionally, filter methods will be part of the class of the elements of the derived collection instead of part of the derived collection itself.

4.2.1 Conceptual model

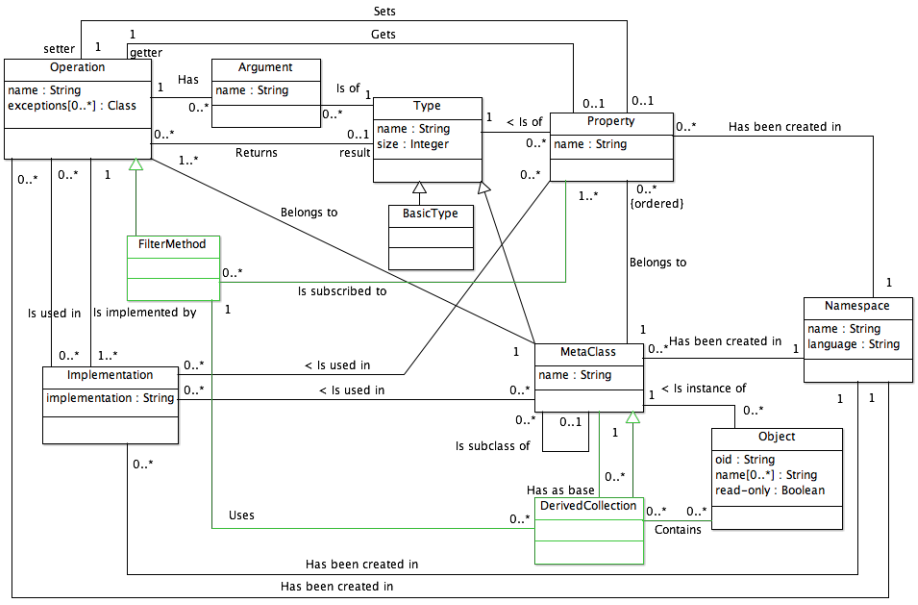

Figure 4.2: Part of the conceptual model of the dataClay system with derived collections

The conceptual model of the dataClay system has been modified in order to include the classes needed for supporting derived information. Figure 4.2 shows the parts of the dataClay's conceptual model that are relevant to the adaptation with the added elements for derived collections (the added elements are outlined in green).

In the dataClay system, a *metaClass* represents a class registered to the system. *Object*s are instances of a *metaClass*.

A *metaClass* has *properties* and *operation*s which correspond to the attributes and methods of said class.
A *property* has an associated *type* and its getter and setter *operation*s.
An *operation* has *argument*s with their associated *type*s as well as a return *type*.

In dataClay, *operation*s can have different *implementation*s though this concept is different than what is usually understood for implementation in the object-oriented paradigm (in which, that a method can have different implementations means that a signature of a method can have different behaviour depending on which class it is executed on).
In dataClay, different *implementation*s of a method have the same effect on the system (i.e. the same pre and post-conditions) but in a different way, for example, a sorting *operation* can have an *implementation* using the selection algorithm and another *implementation* using mergesort.

The system keeps the information regarding which *operation*s, *properties* and *metaClass*es are used in each *implementation*.

*MetaClass*es, *properties*, *operation*s and *implementation*s are created in a user's *namespace* which are entities to organize elements as if they were part of the same package.

These are the parts of the conceptual model in Figure 4.2 that were already present in the system. In order to support derived collections, the *derivedCollection* was necessary. A *derivedCollection* contains *object*s that satisfy the *filterMethod* it uses. A *derivedCollection* has a *metaClass* as its base collection (this can be another *derivedCollection* or the collection that contains all the instances of a *metaClass*).

A *filterMethod* is a special kind of *operation* that is subscribed to some *properties*.

The additions to the model made for supporting derived properties can be seen in Figure 4.3, outlined in green. The *derivedProperty* element is a special kind of *property* that is subscribed to some *properties*. It has two associated *operation*s, one for giving value to the derived property and another one for propagating its updates.

Each *derivedPropertyInObject* represents that an *object* has value in that *derivedProperty*, it is associated with the *object*s used to obtain the value of the *derivedProperty* (its base objects).

As for derived classes, the added elements can be seen in Figure 4.4 outlined in green again. A *derivedClass* has as many *derivationRelationship*s as base classes it has, and each *derivationRelationship* is related to the *operation*s that update the extent of the *derivedClass* when an instance of the base class is inserted or deleted.

A *derivedClass* has *updateMethod*s that track the *operation*s that affect the extent of the *derivedClass*.

An instance of a *derivedClass* is a *derivedObject*, and it is related to the *object*s that were used to create it (its base objects). The initial extent of the

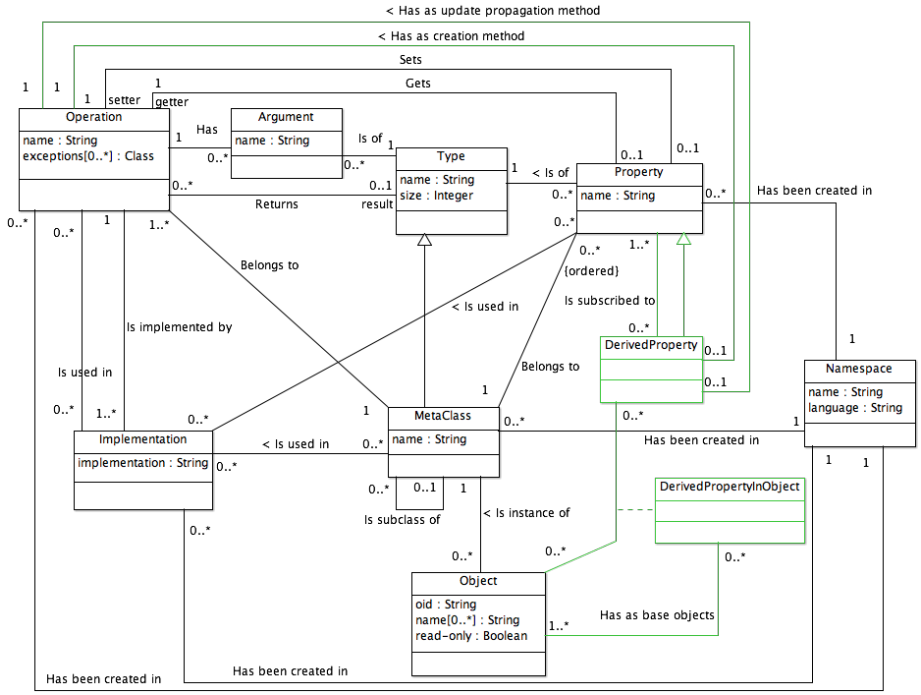

Figure 4.3: Part of the conceptual model of the dataClay system with derived properties

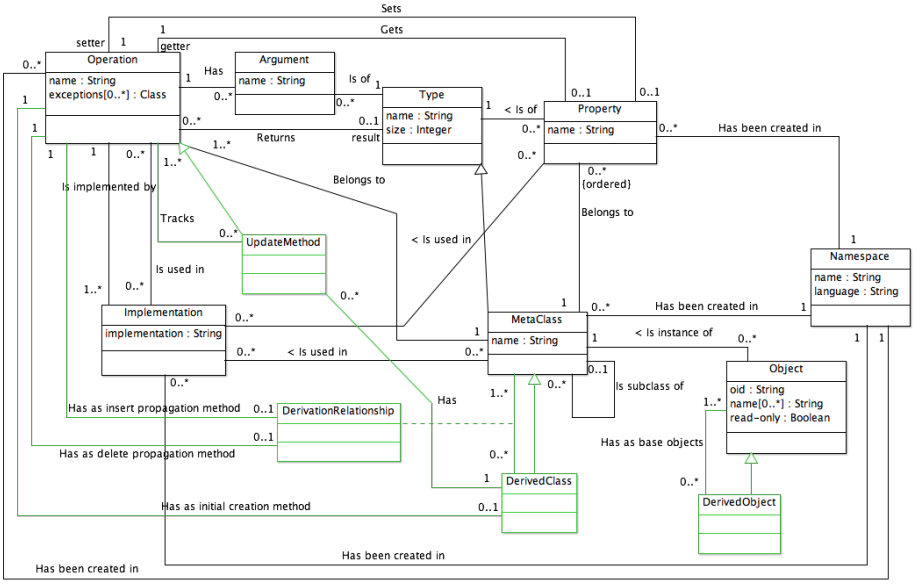

Figure 4.4: Part of the conceptual model of the dataClay system with derived classes

derived class is created using the initial creation method.

4.2.2 Use cases

DataClay use cases need to be changed in order to support derived information.

Currently, dataClay has use cases such as the *new metaClass* use case, which is used by the user to register a new class to the system so it can be instantiated or used in other classes.

The *new metaClass* use case executes, in its process, other use cases such as the *add property* and *add operation* which are executed for every attribute and method that the newly registered class has, respectively.

A use case related to these is the *new enrichment* one. This use case expands an already existing class with new functionality such as new methods or properties.

Other more different use cases are the *make persistent* and *execute operation* use cases. The first is used to store an object in the system, and the latter to execute a registered operation on a persistent object.

In order to support derived collections, we have added the following use cases to dataClay (this is a brief explanation of the use cases, for the whole use case see Appendix A):

- *Add derived collection*: This use case is used to register a new derived collection to the system. The user provides the system with the information of the derived collection, i.e. its name, its base collection and the filter method it must use. With these, the system creates the derived collection and then iterates the objects in the base collection and adds those that satisfy the filter method to the derived collection. The user may specify a metaClass instead of a derived collection for the base collection, in that case, the system uses the collection with all the instances of the given class as the base collection for the new derived collection.

- *Remove derived collection*: This is the counterpart of the *add derived collection* use case. This use case removes the derived collection specified by the user from the system. The derived collection can not be deleted if it's the base collection of another derived collection.

- *Add as filter method*: This use case turns an operation into a filter method that can be used by a derived collection. The user provides the system with the affected operation (along with the class it belongs to) and the set of properties to subscribe the new filter method to. With this, the operation becomes a filter method and the system subscribes the filter method to the given properties. For a method to be able to be added as filter method it must be boolean and have no parameters.

- *Remove filter method*: *Add as filter method* counterpart. The system removes the status of filter method from the operation provided by the user, this can only be done if the filter method is not used by any derived collection.

Some of the already existing use cases needed to be modified, this is the case for the following:

- *Remove metaClass from namespace*: This use case removes the given metaClass from a namespace. If the metaClass was created in the given namespace, the system removes the metaClass along with its interfaces, operations, implementations and properties. In case it was not created in the namespace, the system removes the interfaces that correspond to that metaClass as well as the enrichments and interfaces created in the namespace. This use case has been modified to prevent the user from removing a metaClass that has derived collections based on it.

- *Remove property*: In this use case, the user provides the system with the property to delete from the system, the metaClass that contains the property and the namespace in which it was created, then the system removes it (except if it is used in somewhere in the system) and removes its relationships with the metaClass and the namespace. It has been modified so the system prevents the user from removing a property that has filter methods subscribed to it.

- *Remove operation*: This use case removes the specified operation from the metaClass and namespace given by the user. The system removes its relationships with metaClass and namespace as well as its implementations. It has been modified so, in case the operation is a filter method and is not used by any derived collection, the system will execute the use case *remove filter method* before executing this use case. In case the operation is a filter method and is used by a derived collection, the operation is not removed.

- *Make object persistent*: This is used by the user to store an object in the system. The user provides the system with the object, the dataset to which the object must be added and the class of the object, and the system stores the given instance to the system. This use case has been modified so when an object is stored, the system executes the filter methods of the object's class. Then, the system adds the new object to the derived collections that use the satisfied filter methods if the object is member of their base collection.

- *Delete persistent object*: This use case is the counterpart of the previous one. The user specifies the identifier and the class of the object to delete. The system removes the given object and all its reachable objects (if the user has specified so). The reachable objects are all the objects that are linked to the given object and are accessible and deletable by the user. For derived collections, the system also removes it from the derived collections it is a member of.

- *Execute operation*: For this use case, the user provides an object, an operation to execute and its parameters. The system executes the given method on the object specified by the user. It has been modified so, in case the operation is the setter of a property, the system updates the derived collections that use some filter method subscribed to it accordingly. That is, if the execution of the setter has changed the result of a filter method, the system updates the derived collections that use it: if the updated object satisfies the filter method, it is added to the derived collections; otherwise, it is removed from them.

For adding filter methods, we considered modifying the *add operation* use case instead of creating a new use case, the *add as filter method*. This *add operation* would register the given operation to the system and, in case the operation was a filter method, it would subscribe the operation to the given properties.
Finally, we chose to create the *add as filter method* use case because it allows the user to use operations already in the system as filter methods without having to register a new method.

In order to support derived properties, the following use cases have been added to the system:

- *Add derived property*: This use case adds a derived property to a metaClass. The user provides the system with the following information: the namespace and metaClass in which the derived property is added, the name and type of the property, and the information for keeping it up-to-date, which includes the set of properties the derived property is subscribed to and the creation and update propagation methods. The system adds the derived property and the provided operations to the given metaClass and subscribes the derived property to the given properties. Then the extent of the metaClass is iterated and the new property is initialized by executing the creation method on each object.

- *Remove derived property*: This use case is the *add derived property* counterpart. The system removes the given derived property from the specified metaClass and namespace. The system also removes the operations associated with the derived property (i.e. the creation and update propagation methods). The derived property is not removed in case there is a filter method or derived property subscribed to it.

The following use cases have been modified for derived properties:

- *New metaClass*: As mentioned before, this use case is used to register a new class to the system. The user specifies the namespace to which the derived class must be added and provides the file implementing the metaClass. With this information, the system creates a new class and adds the properties and operations found in the file. This use case has been modified so it adds the derived properties in the file using the use case *add derived property*.

- *New enrichment*: This use case enriches a metaClass with new properties or methods. The user provides the metaClass to which the enrichment is added, its namespace and the file with the new properties and operations. The system then adds the new properties and operations to the given metaClass. It has been modified so it also adds the new derived properties in the enrichment using the use case *add derived property* and, if at least one derived property has been added, the system iterates the extent of the class and initializes the new derived properties using their creation methods.

- *Remove metaClass from namespace*: For derived properties, this use case has been modified so in case there is a derived property in another class that is subscribed to a property of the class being removed, the system does not remove the class.

- *Remove property*: This use case removes from the specified metaClass and namespace the given property. It has been modified so it executes the *remove derived property* in case the given property is derived and also to prevent the user from removing a property that has a derived property subscribed to it.

- *Remove operation*: For derived properties, this use case has been further modified in order to prevent the user from removing an operation that is the creation or update propagation method of a derived property.

- *Make object persistent*: To support derived properties, it has been modified so the system executes the creation methods of the derived properties contained in the object's class.

- *Execute operation*: For derived properties, this use case has been modified so, if the executed operation is the setter of a derived property, the system executes its update propagation method, and in case it is the setter of a property that has derived properties subscribed to it, the system executes the derived properties' creation method. It also prevents the user from executing the creation and update propagation methods of a derived property since these are methods to be executed only by the system.

The new use cases necessary to support derived classes are the following:

- *New derived class*: This use case is used to register a derived class to the system. The user provides the system with the namespace in which the derived class will be added and the file implementing the class. The system creates a new derived class and adds to it the properties (derived and non-derived) and the operations found in the file using the proper use cases (*add property*, *add operation* and *add derived property*). After this, the initial creation method is executed to create the extent of the derived class.

- *Remove derived class from namespace*: This use case is the counterpart of the *new derived class* use case. The system removes from the given namespace the derived class specified by the user. This removes all the operations and properties of the derived class. The class is not removed in case there it is a base class of another derived class or if there is some derived information subscribed to one of its properties.

And the ones modified for derived classes are:

- *Remove metaClass from namespace*: This use case has been modified to prevent the user from removing a class that is base class of a derived class.

- *Remove operation*: It has been modified so it prevents the user from removing operations related with a derived class, i.e. an update method, an operation being tracked by an update method, the insert propagation method or the delete propagation method of a derivation relationship.

- *Make object persistent*: For derived classes, in case the class of the object is the base class of a derived one, the system executes the insert propagation method of their derivation relationship.

- *Delete persistent object*: If the object's metaClass is a base class of a derived class, the system executes the respective delete propagation methods.
- *Execute operation*: In case the operation is tracked by an update method, the system executes the update method to propagate the changes to the derived classes. It also prevents the user from executing an update method, an insert or delete propagation method of a derivation relationship or a derived class' initial creation method, since these are methods to be executed only by the system.

The complete use cases modified or added as a result of the adaptation of the approach to dataClay can be found in Appendix A.

4.2.3 Implementation

We have explained how the general approach has been adapted to the dataClay system both in the conceptual model and in the use cases. Now we are going to explain how the derived collections are being implemented in the system.

Derived collections

In the dataClay system, derived collections are going to be a collection of objects instead of an instanciable class with implementation objects.

The user will access a derived collection by retrieving its content element by element, with this there could be issues in case the object membership to a derived collection changes while the user is still using that element. The user could be accessing the *BlondePeople* collection and while he is processing element *e*, its membership changes:

```
Iterator<BlondePerson> BPit = BlondePeople.getIterator();
    ↪ // Get the iterator of the derived collection
while (BPit.hasNext()) {
  BlondePerson e = BPit.next();
  ...
  // The membership of e to BlondePeople changes (e is no
     ↪ longer a member of BlondePeople)
  // From this point on the user has an instance of
     ↪ BlondePerson whose object is not a BlondePerson
}
```

This would be possible if derived collections were classes with implementation objects. In case of derived collections as collections, the object would still be instance of the same class, but in case the user was to read again the same derived collection, that object would not be returned. The same scenario with derived collections as collections:

```
Iterator<Person> BPit = BlondePeople.getIterator(); //
    ↪ Get the iterator of the derived collection
while (BPit.hasNext()) {
  Person e = BPit.next();
  ...
  // The membership of e to BlondePeople changes (e is no
     ↪ longer a member of BlondePeople)
```

```
    // The change in the membership of the object to
       ↪ BlondePeople doesn't affect the fact that the
       ↪ object still is an instance of Person
}
```

The object-slicing technique was selected as the way to support derived collections in the general approach because, among other reasons, it allowed

1. To have attributes specific to the objects of a derived collection, and
2. objects to be instances of more than one class.

These reasons are still necessary with derived collections being collections.

In dataClay, users will be able to define policies that restrict the behaviour of the objects in the system.

With these policies, one can achieve the same functionality as the reason 1 mentioned above. Creating an attribute X in class C and then defining a set of policies enforcing that the attribute X of an instance of C has value only when that object is a member of the derived collection D is equivalent to defining an attribute X for the derived collection D which has objects of the class C.

With policies, these restrictions will be made declaratively (the user will be able to state the attributes that only have value if the object is member of a derived collection). Currently, though, dataClay does not support policies. Until policies are supported in dataClay, it will be necessary to add them manually in the logic of the methods. This additional logic will check that the object the method is executed on belongs to the derived collection the method is defined in. For example, the getter for the attribute X would check whether the object belongs to the derived collection D or not, in case it does the value of X is returned, otherwise the method returns null.

With derived collections being actual collections, the reason 2 mentioned above is also achieved since the object can be part of two collections at the same time. For example, an object being member of the *Minor* and *BlondePerson* derived collections is reflected by having this object in both collections.

Another advantage of using collections instead of classes to support derived collections is that, with collections, the methods and attributes of an object are not split across implementation objects, but all in the same object. This means that, in our running example, a *Person* will have all the methods and attributes defined in the derived collections *BlondePerson*, *Minor* and *MinorBlonde*, instead of them being in the implementation objects.

This eases method execution since an object can execute all its methods without needing to delegate the execution to any implementation object as it did with object slicing. As mentioned before, an object will only execute the methods defined in its class or for a derived collection to which it belongs.

We have mentioned that derived collections can be based on other derived collections or on a class, in which case, a collection with all the instances of that class is used as base class, the class collection.

This collection contains all the instances of a class that have been stored in the system. It can be seen as a special kind of derived collection since:

- its filter method is always satisfied, and
- its offered by the system.

The system maintains class collections in the following way:

- **Insert**: When an object is stored in the system, this object is added to the class collection of its class. In case there are derived collections based on the class collection, the system would update them accordingly.
- **Delete**: When an object is deleted from the system, it is also removed from the class collection of its class. This may trigger other deletions in case there are derived collections based on the class collection.

Filter methods

In the general approach, a filter method was a method of the derived collection and the collection was subscribed to the properties that were relevant for filtering the objects (i.e. the properties that were important for its filter).
In the adaptation to dataClay, filter methods are operations of the class of the elements in a derived collection and the filter method is subscribed to the properties.

The filter method of a derived collection is a method of the class of its elements, instead of being a method of the derived collection. This avoids having methods duplicated in the system in two ways:

1. an already existing method of a class can be used as a filter method for a derived collection without adding it again when registering the derived collection, and
2. a filter method will only be defined once in the system regardless of how many derived collections use it.

Expanding on point 1, when a user wants to use a method M of a class as a filter method for a derived collection the process he has to go through depends on where filter methods are defined:

- *In the derived collection*: The user registers the derived collection to the system with the subscription information, this derived collection will have a method exactly like M, therefore the method is stored twice in the system.
- *In the class*: The user provides subscription information for the method in order to make it a filter method. Then he can register the derived collection specifying the method M to be used as filter. The method is only stored once in the system, the derived collection has only a reference to it.

So by having the filter methods in the class, the system does not need to have the same method stored again when an already existing method is used as filter method in a derived collection.

To expand on point 2, if the filter methods are in the derived collection, each filter method is exclusive to that derived collection and for another one to use it as filter method, the user would have to register a new derived collection with a method exactly like the previous.
By having the filter methods in the class, different derived collections can be created using the same filter method, therefore avoiding having the same method more than once in the system.

The second point is shown in the following example: a system that contains information of *Person*s and three derived collections with the people that live in different cities:

- *BCNPeople*: Persons who live in Barcelona.
- *GRNPeople*: Persons who live in Girona.
- *TGNPeople*: Persons who live in Tarragona.

In this scenario, if the user wanted to create derived collections with only the blonde *person*s for each city, he would create the following derived collections:

- *BCNBlonde*: Blonde people that are members of *BCNPeople*.
- *GRNBlonde*: Blonde people that are members of *GRNPeople*.
- *TGNBlonde*: Blonde people that are members of *TGNPeople*.

If the filter methods are in the derived collections, the user will need to register the filter method three times (one for each derived collection); whereas if the filter methods are in the class, all three derived collections would use the same filter method and therefore it would only be stored once in the system.

Having the filter methods in the class instead of in the derived collection is also useful for optimizations, this will be discussed in the next subsection.

A filter method can only be subscribed to properties of the class it belongs to. This avoids having to store extra information in the objects regarding what objects are relevant to its membership to the derived collections, since the only object involved in an object being part of a derived collection or not is the object itself.

An example of a filter method that should be subscribed to properties not in the class it belongs to would be the filter method *filterPeopleWithBlueCars* which is only satisfied for the instances of *Person* that have an associated car of colour blue. If the user wanted to create a derived collection using such a filter method, he would have to create derived properties for the properties in the other class (create a derived property called *carColour* that has the value of the colour of its associated car) and then to add a filter method subscribed to those properties.

Optimizations

Like in the general approach, when the value of a property is changed the system needs to update the derived collections that use filter methods subscribed to it.

In order to do so, the system executes the filter method on the updated object and, with that result, checks whether the object's membership to the derived collections using said filter method has changed or not.

To do this more efficiently, in the implementation of derived collections in dataClay, each object will store the current results of all the filter methods of its class.

The result of the filter methods is computed for each object when the filter method is added to dataClay. The system iterates through the instances of the method's class executing the filter method and storing its output.

This improves the performance of the system in two ways.

First: without storing the results of the filter methods, when a derived collection is registered to the system, this iterates through the base collection executing the filter method on each object and adding those that satisfy it to the derived collection. If the results of the filter methods are stored, the system does not need to execute the filter method on each object when a new derived collection is registered, instead it checks the stored result for that filter method in each object of the base collection and adds those that satisfy it to the derived collection.

Second: without storing the results of the filter methods, when the value of a property is updated, the system executes the filter methods subscribed to that property on the updated object. For each filter method, the system checks the derived collections that use it and updates them according to the output of the execution (i.e. if the filter method returned true and the object is not in the derived collection, the system adds it to the collection; and if the filter method returns false but the object is in the derived collection, the system removes it from the collection). Therefore, in case the update does not change the output of the filter method, the system checks the derived collections unnecessarily.
With the result of the filter methods stored, when the value of a property is changed, the system executes the filter method on the updated object and compares the new output with the previous one. Only in case it has changed, the system updates the derived collections that use the filter method accordingly.

Only the output of the filter methods is not enough to determine whether an object must be included in a derived collection or not. This is because an object may satisfy the filter method used by a derived collection and still not be part of it, this is the case when the object satisfies its filter method but the object is not member of its base collection.

For example, if the system has three derived collections:

- *BlondePeople*: Derived collection based on the *Person* class that uses the filter method *filterBlonde*.
- *BlondeMinor*: Derived collection based on the *BlondePeople* derived collection that uses the filter method *filterAge*.
- *BlondeMinorFromBarcelona*: Derived collection based on the *BlondeMinor* derived collection that uses the filter method *filterBCN*.

And an object o in the system that only satisfies *filterAge*. This object would not be in any of the derived collections since the only filter method it satisfies is used by a derived collection (*BlondeMinor*) that is based on a collection the object is not a member of (*BlondePeople*). Then object o received an update to its properties that made it satisfy the filter method *filterBCN*. In this scenario, only looking at the filter methods object o satisfies, the system would incorrectly add it to the *BlondeMinorFromBarcelona* derived collection (since the object satisfies the filter method of the derived collection and the filter method of its base collection). On the other hand, with the information of which derived collections the object is member of, the system would not add it since the object is not part of the derived collection *BlondeMinor*.

Therefore, the system needs to check whether the object is part of the base collection or not, as an additional optimization, each object will also store for

each derived collection of its class whether it belongs to the collection or not. This optimization allows the system to know whether an object belongs to a derived collection or not without accessing the collection and checking whether it contains the object or not.

Dynamic data

In dataClay, objects are grouped in datasets. A dataset is a set of objects that may not be instance of the same class that a user groups together.

Datasets can be shared between users using data contracts. Each user can access the objects of a class that belong to the datasets he has access to through data contracts. This means that different users may see different extents for the same class, and this needs to be the case for derived collections as well.

When a user creates a derived collection based on a class C, that derived collection contains the objects of his datasets that are instances of the class and satisfy the filter. If the user shared this derived collection with another user, the second user would see the derived collection based on the extent of the class for him. For example, two users A and B who have datasets 1 and 2, and 2 and 3 respectively. If user A creates a derived collection over the class, that derived collection will contain a subset of the datasets 1 and 2. In case he shared the derived collection with user B, this user would see the derived collection with a subset of datasets 2 and 3.

In order to support this efficiently and avoiding as much data duplication as possible, derived collections will have all the objects of the class, regardless of which datasets the user that created it can access. This will be done by keeping the derived collections at dataset level. Once a user requests the iterator to access a derived collection, the system will provide him with an iterator through the collections of the datasets that correspond to that user.

4.2.4 Usage

In this section an example of derived collections in dataClay is presented.

This example will present the *BlondePeople* derived collection. The system contains the class *Person* as follows:

```
public class Person {
  String hairColour;

  boolean filterBlonde() { //Filter method subscribed to
     ↪ Person.hairColour
    return this.hairColour == "blonde";
  }
}
```

In this class we can see that it has the attribute *hairColour* and a method called *filterBlonde* that has been added as filter method and is subscribed to the property *hairColour*.

When a filter method is added to the system, the system registers the operation as filter method if it is a boolean method with no parameters. In order to do so, the new filter method is added to the list of filter methods subscribed to the given properties. Then, the system executes and stores the result of the filter method on the objects of its class.

And when a derived collection is registered to the system, the given derived collection is added to the list of derived collections that use the specified filter method and to the collections based on its base collection. This base collection is then iterated and its elements that satisfy the filter method used by the new derived collection are added to it.

For the derived collection, the system provides the following class:

```
public class DerivedCollection<E> {
  Collection<E> elements;

  public String getFilterMethodSignature() {
    return null;
  }
  public String getName() {
    return null;
  }
  public String getBaseCollection() {
    return null;
  }
  public static Iterator<E> getIterator() {
    return elements.iterator();
  }
}
```

This class is the superclass of all the derived collections, it is necessary so the user implements the methods with the information that the system needs to register a derived collection. It contains:

- *elements*: The collection of objects, this is the actual content of the derived collection and it is what the user iterates through.
- *getFilterMethodSignature*: This method must be implemented by the user and must return the signature of the filter method that the derived collection uses.
- *getName*: This method must also be implemented by the user and returns the identifier of the derived collection for the user.
- *getBaseCollection*: The last method that the user needs to implement returns the identifier of the collection to base the new derived collection on. If this method returns null, the collection with all the instances of the class E is used.
- *getIterator*: This method is provided by the system and returns the iterator over the collection *elements*.

The *BlondePeople* derived collection would be implemented in the following way:

```
public class BlondePeople extends DerivedCollection<
    ↪ Person> {
  @Override
  public String getFilterMethodSignature() {
```

```
    return "filterBlonde";
  }
  @Override
  public String getName() {
    return "BlondePeople";
  }
  @Override
  public String getBaseCollection() {
    return null;
  }
}
```

The *BlondePeople* collection is created by implementing the methods *getBaseCollection*, *getFilterMethodSignature* and *getName* so they return the information for the derived collection. With this, the system creates a derived collection identified by the output of *getName* and that uses the filter method returned by *getFilterMethodSignature*. This derived collection will be based on the collection with all the instances of the class *Person* since the method *getBaseCollection* returns null.

The system then will iterate the collection with all the instances of *Person* and will check the output of the *filterBlonde* filter method, the objects that have *true* stored, will be added to the *BlondePeople collection*.

Finally, here is an example of the usage of the derived collection after it has been registered in the system:

```
Iterator<Person> blPeople = BlondePeople.getIterator();
while (blPeople.hasNext()) {
  Person p = blPeople.next();
  . . .
}
```

To use the derived collection, the user obtains the iterator of the collection and uses it like he would use a regular Java Iterator.

Chapter 5

Future work and conclusions

5.1 Future work

Even though the general approach has been adapted entirely to dataClay, the implementation of this has only been designed and started for derived collections, therefore the logic next step, after finishing the implementation of derived collections, is to design and implement derived properties and derived classes, following the adaptation to dataClay presented in Chapter 4.2.

Also, derived collections can be expanded in functionality.

5.1.1 Sorted derived collections

Derived collections contain a subset of the objects in their base collection, this allows the users to access them more easily than without derived collections. But a system like dataClay would also benefit from allowing the users to access derived collections sorted in a particular order.

In order to support sorted derived collections, a derived collection may contain additional methods, called *compareMethod*s. A compareMethod is a method that has two objects of the same type as parameters, these two objects are the objects to sort. The method returns an integer depending on how the objects should be sorted: if the returned integer is 0, both objects compare equally; if the integer has a negative value, the first object goes before the second one; and the other way around in case the integer is a positive number.

An example of a compareMethod to sort the *BlondePerson* derived collection by *age* ascendingly would be the following:

```
BlondePerson.compareByAge(Person p,Person p1){
  return p.age-p1.age
}
```

A derived collection has as many compareMethods as orders it needs to be accessed in. With orders, a derived collection consists of a set of collections of objects, each of them sorted using one of the compareMethods of the derived collection.

When a derived collection is registered to dataClay, the system will iterate through its base collection and execute its filter method on the objects. Those that satisfy it will be added to the derived collection and, in case the derived collection has orders, they will be placed in each collection using the corresponding compareMethod.
In case the derived collection already exists and the user expands it with a new order, the system will iterate through one of the sorted collections and add its objects to a new collection sorted with the new compareMethod.

When the user registers a compareMethod to the system, he must provide the properties to which this method must be subscribed to and an identification for that order. As with filter methods, a compare method is subscribed to the properties that are important for sorting two objects. The *compareByAge* method above would be subscribed to the property *age*.

As said, compareMethods can be registered with the derived collection or added to an already existing derived collection, an example of a derived collection registered with a compareMethod:

```
public class BlondePeople extends DerivedCollection<
    ↪ Person> {
  @Override
  public String getFilterMethodSignature() {
    return "filterBlonde";
  }
  @Override
  public String getName() {
    return "BlondePeople";
  }
  @Override
  public String getBaseCollection() {
    return null;
  }
  public int compareByAge(Person p,Person p1) { //
      ↪ CompareMethod identified by "ageSort" and
      ↪ subscribed to "Person.age"
    return p.age-p1.age;
  }
}
```

The behaviour of the system regarding compareMethods and filter methods is very similar. When the value of a property is updated, the system will update the derived collections that have a compareMethod subscribed to that property. The update will consist in placing the updated object in the position that corresponds to it inside the collection that uses the compareMethod subscribed to the updated property.

With sorted derived collections, when the user requests the iterator of a derived collection he could provide the order (out of the ones that the derived collection has) and the system would return the iterator for the derived collection sorted with that compareMethod. An example of this with the *BlondePeople* derived collection shown before would be:

```
Iterator<Person> blPeople = BlondePeople.getIterator("
    ↪ ageSort"); //The iterator returned by the system
```

```
    ↪ iterates the BlondePeople collection sorted with
    ↪ the compareMethod identified by "ageSort", i.e.
    ↪ compareByAge
while (blPeople.hasNext()) {
  Person p = blPeople.next(); //The elements will be
    ↪ obtained in order of their age
  . . .
}
```

The initial idea for sorted derived collections has been sketched but there's still the need to work on the details on how to implement and manage the orders and how to integrate them with dataClay regarding use cases and the conceptual model.

5.2 Conclusions

The goal of this thesis was to design an approach to views for object oriented databases and to adapt this approach to the dataClay system.

We started by reviewing the work related to my topic done by others in the past. With this, we were able to learn from their work and to understand the capabilities and lackings of other approaches. We identified two kinds of approaches in the literature: those that focused on only providing object-preserving views and those that tried to support object-generating views too. There was no approach with view definition without an object algebra and also none of the approaches supported object-preserving and object-generating views with both derived and non-derived data.

After that, we started the design of our approach by dividing the derivation in three kinds: derived collections, which present the data already in the system but in another way (for example, a subset of the instances of another class); derived properties, which are attributes of a class whose value is computed from the information in the system (for example, a property with the body mass based on the height and weight); and derived classes, which are classes whose instances are created from the data in the system and represent a concept previously not in the system (for example, possible pairings of a set of players). This approach was designed to be independent of the storage platform used, this is possible thanks to the object-slicing technique which allows systems to support the requirements of our approach.

The general approach was then adapted to dataClay, this was necessary so dataClay could benefit the most from our approach. Some of the adaptations were to not use the object-slicing technique since the same functionality can be achieved in dataClay's features. Adapting our general approach to a specific system was also useful in order to devise potential optimizations that depend on the platform the approach is implemented on.

Finally, the initial work of the implementation stage for derived collections has been completed, that is, the design of the implementation details. The rest of implementation stage is in development.

Appendix A

Full use cases

These are the use cases that have been modified or added to the system for supporting derived information:

Added use cases
These are all the use cases added to the system:
Title: Add derived collection
Actors: Namespace responsible
Main success scenario:
The user indicates:

- the namespace N of the derived collection
- the base collection for the derived collection
- the name of the derived collection
- the operation that the derived collection uses for filter method

The system creates a derived collection with the input. The created derived collection is based on the collection provided by the user, in case the user gives a class, the derived collection will be based on the collection that contains all the instances of that class. The derived collection uses the given operation as filter method. The system iterates through the base collection and adds the elements that satisfy the filter method to the new derived collection.
Extensions:

- The actor is not responsible for N → the derived collection is not created
- There is already a metaClass (derived collection or not) in N with the given name → the derived collection is not created
- The derived collection is not implemented in the language of N → the derived collection is not created
- The operation is not a filter method → the derived collection is not created
- The operation is not defined in the class of the members of the base collection → the derived collection is not created

Title: Remove derived collection
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N
- the name of the derived collection to be removed

The derived collection is removed from the system and its relations with metaClass and namespace too.
Extensions:

- The actor is not responsible for N → the derived collection is not removed
- There is no derived collection with the given name → the derived collection is not removed
- The derived collection is the base collection of another derived collection → the derived collection is not removed

Title: Add as filter method
Actors: Namespace responsible
Main success scenario:
The user indicates:

- the namespace N of the operation
- the metaClass of the operation
- the operation to be turn into a filter method
- a non-empty set of properties the filter method is subscribed to

The system makes the operation a filter method and subscribes it to the properties given. The system iterates through the instances of the class executing and storing the result of the filter method.
Extensions:

- The actor is not responsible for N → the filter method is not created
- The operation is not in the metaClass → the filter method is not created
- The properties are not part of the metaClass → the filter method is not created
- The operation is not boolean → the filter method is not created
- The operation has parameters → the filter method is not created

Title: Remove filter method
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N

- the metaClass of the filter method
- the filter method to remove

The system removes the operation as a filter method, removing its links with the subscribed properties. The operation is not removed from the system, only its state as filter method.
Extensions:

- The actor is not responsible for N → the filter method is not removed
- The operation is not a filter method in the metaClass → the filter method is not removed
- The filter method is being used by a derived collection → the filter method is not removed

Title: Add derived property
Actors: Namespace responsible
Main success scenario:
The user indicates:

- the namespace N in which the derived property is added
- the name of the metaClass to which the derived property will be added
- the name of the derived property
- the type of the derived property
- a non-empty set of properties to subscribe the derived property to
- the operation to use as creation method
- the operation to use as update propagation method

The system creates a new derived property with the indicated name, type, and default getter and setter operations, and associates it to the indicated metaClass and namespace. The system executes the use case add operation for the creation and update propagation methods and it associates the derived property to them. The derived property is subscribed to the properties given. If the type of the derived property is not a basic type and does not exist in N (created or imported by means of an active model contract), it is registered by executing the use case new metaClass (recursively).
Extensions:

- The actor is not responsible for N → the derived property is not created
- The metaClass does not belong to N and is not imported to N via an interface in contract associated to an active model contract → the derived property is not created
- The property already exists (a property or derived property with the same name+namespace N in the same metaClass) → the derived property is not created

- The type is an unsupported basic type → the derived property is not created
- The type corresponds to registered class that is not present in N (created or imported by means of an active model contract of the user) → the derived property is not created

Title: Remove derived property
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N in which the derived property was created
- the metaClass to which the derived property belongs
- the name of the derived property to be removed

The system removes the derived property (and its relationships with metaClass and namespace). The system executes the use case remove operation for the derived property's creation and update propagation methods.
Extensions:

- The actor is not responsible for N → the derived property is not removed
- The derived property was not created in N → the derived property is not removed
- The derived property belongs to an interface → the derived property is not removed
- The derived property is used in an implementation → the derived property is not removed
- The derived property has a filter method subscribed to it → the derived property is not removed
- The derived property has a derived property subscribed to it → the derived property is not removed

Title: New derived class
Actors: Namespace responsible
Main success scenario:
The user indicates:

- the namespace N of the derived class
- the file implementing the derived class to be created
- the classpath needed to compile the derived class

The system:

- executes the use case new metaClass for each class found in the file implementing the class (mainly the classes imported (Java meaning) or extended (Java meaning) by the new metaClass) until a class already present in N (created or imported by means of an active model contract) or a Java class is reached.

- creates a new derived class with the following data obtained from the input:
  - namespace N and derived class name (including the package, if any)
  - language in which the class has been implemented
  - for each property, the use case add property is executed to add the property to the new derived class. This use case also registers all the metaClasses needed by the property, if not already registered.
  - for each operation, the use case add operation is executed to add the operation to the new derived class. This use case also registers all the metaClasses needed by the operation and its implementations, if not already registered.
  - for each derived property, the use case add derived property is executed to add the derived property to the new derived class. This use case also registers all the metaClasses needed by the derived property, if not already registered.
  - for each update method, the use case add operation is executed to add the update method to the new derived class, after this the update method is associated with the method it tracks.
  - for each base class, the insert and delete propagation methods are added to the derived class and are associated with it and the base class.
- executes the initial creation method in order to create the instances of the derived class.

Extensions:

- The actor is not responsible for N → the derived class is not created
- There is another class (base or derived) with the given name in N → the derived class is not created
- The derived class is not implemented in the language of N → the derived class is not created
- There are two properties with the same name → the derived class is not created
- There are two operations with the same name and arguments → the derived class is not created
- The superclass/some type is neither registered in N, nor imported to N by means of some active model contract, and a file implementing it cannot be found through the classpath → the derived class is not created
- Any of the previous errors occur → all the metaClasses automatically registered from the "first" execution of the use case are eliminated

Title: Remove derived class from namespace
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N
- the derived class to be removed from N

If the derived class was created in N, the system removes the derived class and all its associated derived collections, interfaces, operations, implementations, and properties. Otherwise, the interfaces corresponding to the derived class are removed from the imported interfaces in N, and the enrichments (properties, operations and implementations) and interfaces created in N are removed.
Extensions:

- The actor is not responsible for N → the derived class is not removed
- The derived class does not exist in N (either created or imported) → the derived class is not removed
- The derived class has some interface created in N associated to a model contract → the derived class is not removed
- The derived class is used in an implementation → the derived class is not removed
- There is some other derived class with a property of this type in N → the derived class is not removed
- There is some other derived class that is subclass of this one in N → the derived class is not removed
- There is some operation with an argument or result of this type in N → the derived class is not removed
- There is some implementation created in N that uses an imported interface in contract corresponding to the derived class → the derived class is not removed
- There is a derived collection based on the derived class → the derived class is not removed
- There is some derived property subscribed to one of the properties of the derived class → the derived class is not removed
- There is some update method tracking one of the derived class methods → the derived class is not removed
- The derived class is the base class of another derived class → the derived class is not removed

Comments:

- All the interfaces, operations, implementations and properties in the class will be removed. This will only delete elements created in N, since in order to have these elements created in another namespace, a model contract is needed (and if there is a model contract associated to the derived class, it will not be deleted)

Modified use cases

These are all the use cases of the system that have been modified, the parts of the use cases that have been modified are highlighted with *italics*:

Title: New metaClass
Actors: Namespace responsible
Main success scenario:
The user indicates:

- the namespace N of the metaClass
- the file implementing the metaClass to be created
- the classpath needed to compile the metaClass

If the metaClass is a derived class, the system executes the use case new derived class instead.
The system:

- executes the use case new metaClass for each class found in the file implementing the class (mainly the classes imported (Java meaning) or extended (Java meaning) by the new metaClass) until a class already present in N (created or imported by means of an active model contract) or a Java class is reached.
- creates a new metaClass with the following data obtained from the input:
  - namespace N and metaClass name (including the package, if any)
  - language in which the class has been implemented
  - for each property, the use case add property is executed to add the property to the new metaClass. This use case also registers all the metaClasses needed by the property, if not already registered.
  - for each operation, the use case add operation is executed to add the operation to the new metaClass. This use case also registers all the metaClasses needed by the operation and its implementations, if not already registered.
  - *for each derived property, the use case add derived property is executed to add the derived property to the new metaClass. This use case also registers all the metaClasses needed by the derived property, if not already registered.*
  - if the metaClass C has a superclass S, C is stored as a subclass of S in case S has been created in N. Otherwise, C is stored as an extension of the interface in contract that imports S to N.

Extensions:

- The actor is not responsible for N → the metaClass is not created
- The metaClass name already exists in N → the metaClass is not created
- The metaClass is not implemented in the language of N → the metaClass is not created

- There are two properties with the same name → the metaClass is not created
- There are two operations with the same name and arguments → the metaClass is not created
- The superclass/some type is neither registered in N, nor imported to N by means of some active model contract, and a file implementing it cannot be found through the classpath → the metaClass is not created
- Any of the previous errors occur → all the metaClasses automatically registered from the "first" execution of the use case are eliminated

Title: New enrichment
Actors: Namespace responsible
Main success scenario:
The user indicates:

- the namespace N in which the enrichment is added
- the metaClass name to which the enrichment will be added
- the file containing the new properties and operations that enrich the metaClass
- the classpath needed to compile the enrichment

The system:

- executes the use case new metaClass for each class found in the enrichment file (classes imported (Java meaning)) until a class already present in N (created or imported by means of an active model contract) or a Java class is reached
- for each property, the use case add property is executed to add the property to the indicated metaClass. This use case also registers all the metaClasses needed by the property, if not already registered.
- for each operation, the use case add operation is executed to add the operation to the indicated metaClass. This use case also registers all the metaClasses needed by the operation and its implementations, if not already registered.
- *for each derived property, the use case add derived property is executed to add the derived property to the indicated metaClass. This use case also registers all the metaClasses needed by the derived property, if not already registered.*
- *if a derived property has been added, the system iterates the extent of the metaClass and executes the creation methods of the added derived properties.*

Extensions:

- The actor is not responsible for N → the enrichment is not created

- The metaClass is not imported to N via some interface in contract associated to an active model contract in which the user is the client → the enrichment is not created
- The enrichment file is not in the language of the namespace N → the enrichment is not created
- The enrichment includes two properties with the same name → the enrichment is not created
- The enrichment includes two operations with the same name and arguments → the enrichment is not created
- The enrichment is specified as a subclass of some class that the original class does not inherit from → the enrichment is not created
- Any of the previous errors occur → all the metaClasses automatically registered from the "first" execution of the use case are eliminated

Title: Remove metaClass from namespace
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N
- a metaClass to be removed from N

If the metaClass is a derived class, the system executes the use case remove derived class instead.
If the metaClass was created in N, the system removes the metaClass and all its associated interfaces, operations, implementations, and properties. Otherwise, the interfaces corresponding to the metaClass are removed from the imported interfaces in N, and the enrichments (properties, operations and implementations) and interfaces created in N are removed.
Extensions:

- The actor is not responsible for N → the metaClass is not removed
- The metaClass does not exist in N (either created or imported) → the metaClass is not removed
- The metaClass has some interface created in N associated to a model contract → the metaClass is not removed
- The metaClass is used in an implementation → the metaClass is not removed
- There is some other metaClass with a property of this type in N → the metaClass is not removed
- There is some other metaClass that is subclass of this one in N → the metaClass is not removed
- There is some operation with an argument or result of this type in N → the metaClass is not removed

- There is some implementation created in N that uses an imported interface in contract corresponding to the metaClass → the metaClass is not removed
- *There is a derived collection based on the metaClass → the metaClass is not removed*
- *There is some derived property in another class that is subscribed to a property of the class → the metaClass is not removed*
- *There is some update method that tracks one of the methods of the class → the metaClass is not removed*
- *The metaClass is the base class of a derived class → the metaClass is not removed*

Comments:

- All the interfaces, operations, implementations and properties in the class will be removed. This will only delete elements created in N, since in order to have these elements created in another namespace, a model contract is needed (and if there is a model contract associated to the metaClass, it will not be deleted)

Title: Remove property
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N in which the property was created
- the metaClass to which the property belongs
- the name of the property to be removed

If the property is a derived property, the system executes the use case remove derived property instead. The system removes the property (and its relationships with metaClass and namespace).
Extensions:

- The actor is not responsible for N → the property is not removed
- The property was not created in N → the property is not removed
- The property belongs to an interface → the property is not removed
- The property is used in an implementation → the property is not removed
- *The property has a filter method subscribed to it → the property is not removed*
- *The property has a derived property subscribed to it → the property is not removed*

Title: Remove operation
Actors: Namespace responsible
Main success scenario:
The user indicates:

- a namespace N in which the operation was created
- the metaClass to which the operation belongs
- the operation (signature) to be removed

If the operation is a filter method and it is not used by any derived collection, the system executes the use case remove filter method first. The system removes the operation, its associated implementations, and its relationships with metaClass and namespace.
Extensions:

- The actor is not responsible for N → the operation is not removed
- The operation was not created in N → the operation is not removed
- The operation belongs to an interface → the operation is not removed
- The operation is used in an implementation → the operation is not removed
- The operation is the getter of some property → the operation is not removed
- *The operation is used in a derived collection as a filter method → the operation is not removed*
- *The operation is the creation method of a derived property → the operation is not removed*
- *The operation is the update propagation method of a derived property → the operation is not removed*
- *The operation is an update method of a derived class → the operation is not removed*
- *There is an update method that tracks the operation → the operation is not removed*
- *The operation is the insert propagation method of a derivation relationship → the operation is not removed*
- *The operation is the delete propagation method of a derivation relationship → the operation is not removed*

Title: Make object persistent
Actors: User
Main success scenario:
The user indicates:

- the user class used to store the object
- the dataset in which the object will be stored
- the object data to be stored
- whether all the reachable objects must also be stored.

- the backend where the object(s) will be stored (optional)
- all the user classes obtained from the execution of get classes for development that provided the class of the object to be stored

The system stores the object data as an instance of the metaClass corresponding to the user class. In case that the reachable objects must also be stored, they are stored recursively in the same way until a persistent object is reached, using one of the provided user classes. The objects are stored in the indicated backend, if any. Otherwise, a random backend is selected by the system.
More precisely:

- a new unique OID is assigned to each object
- the objects are associated to the indicated user account (creator)
- the objects are associated to the backend in which they have been stored
- the objects are associated to the indicated dataset
- the objects are set to read/write

The system updates the derived information:

- *Derived collections: if the class of the object has derived collections, the system executes their filter methods and adds the object to them if the filter method is satisfied, this is only in case the object is member of the derived collection's base collection.*
- *Derived properties: the system executes the creation methods of the derived properties of the object's class.*
- *Derived classes: if there are derived classes based on the object's class, the system executes the insert propagation methods of their derivation relationships.*

The system returns:

- the OID for the initial object
- the destination backend

Extensions:

- The user is not a client of the contract(s) associated to the indicated user class(es), or some of the contracts is not active → the object is not stored
- The user does not have an active contract with the indicated dataset → the object is not stored
- The object is already persistent → the object is not stored
- An error is received from the backend → if the maximum number of retries is not reached another backend is selected. Otherwise, the object is not stored

- The user class used to store the object does not have the same language as the dataset → the object is not stored
- Any of the previous errors occurs at some point → all the objects created from the first execution of the use case are deleted

Title: Delete persistent object
Actors: User
Main success scenario:
The user indicates:

- the user class used to delete the object
- an OID
- a set of datasets
- whether all the reachable objects must also be deleted
- all the user classes obtained from the execution of get classes that provided the class of the object to be deleted

The system unregisters the requested object (from all the backends) and sets it as deleted in its instance of the user class (stub). In addition, if the reachable objects have to be deleted, the system executes delete persistent object for each object linked to OID (which may belong to a different dataset) that is accessible through the user classes provided, recursively until no persistent object that can be deleted by the user is reached. *The system updates the derived information:*

- *Derived collections: the system removes the object from the derived collections it is a member of.*
- *Derived classes: if there are derived classes based on the object's class, the system executes the delete propagation methods of their derivation relationships.*

Extensions:

- There is some contract associated to the user class that is not active, or does not include the user as a client → the object is not deleted
- The user is not the client of an active contract for some of the provided datasets → the object is not deleted
- The OID does not belong to any of the provided datasets → the object is not deleted
- The OID does not exist → the object is not deleted
- The OID does not correspond to the indicated user class → the object is not deleted
- An error is received from the backend → the object is not deleted

Title: Execute operation
Actors: User
Main success scenario:
The user indicates:

- an instance of a user class
- the operation to be executed
- the parameters of the operation
- all the user classes obtained from the execution of get development classes that generated the class used to call the operation
- a set of datasets

If the object is not persistent, the system executes the local implementation of the operation obtained from the user class.
If the object is persistent, the system executes the remote implementation in one of the backends where the object is stored. If the operation calls another operation during execution, the use case execute operation is executed, with the corresponding instance of the user class, operation and parameters of the call. If the user class has not been provided as input, the user of the recursive execution of the use case "is" the responsible of the namespace in which the implementation was created. More precisely, the execution of the so-called remote implementations (performed on persistent objects) is processed taking into account the execution context of the responsible of the implementation. That is, once the user calls a remote implementation it becomes impersonated so the Data Model Contracts of the responsible of the implementation are used from then on (because the implementation is the one that knows what it needs to be executed). However, only those objects belonging to datasets for which the original user has an active contract will be accessible. *The system updates the derived information:*

- *Derived collections: if the operation is the setter of a property that has filter methods subscribed to it, the system executes the filter method on the updated object and, in case its output has changed, the system updates the derived collections that use it accordingly.*
- *Derived properties: if the operation is the setter of a derived property, the system executes its update propagation method. If the operation is the setter of a property that has derived properties subscribed to it, the system executes the creation method of those derived property on the given object.*
- *Derived classes: if the operation is tracked by an update method, the system executes them to maintain the derived classes up-to-date.*

Extensions:

- There is some contract associated to the user class that is not active, or does not include the user as a client → the operation is not executed
- The operation does not belong to the user class → the operation is not executed
- The parameters of the call do not match with the operation → the operation is not executed
- The OID does not exist → the operation is not executed

- The OID does not correspond to the indicated user class → the operation is not executed
- The object with OID is read-only and the operation attempts to modify it → the changes will not persist
- The user is not the client of an active contract for some of the provided datasets → the operation is not executed
- The execution attempts to access an object that does not belong to any of the provided datasets → the operation is not executed
- *The operation is an update method, a derivation relationship's insert or delete propagation method, a derived property's creation or update propagation method or a derived class' initial creation method → the operation is not executed*

Bibliography

[1] Serge Abiteboul and Anthony J. Bonner. "Objects and Views". In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991.* 1991, pp. 238–247. DOI: 10.1145/115790.115830. URL: http://doi.acm.org/10.1145/115790.115830.

[2] Reda Alhajj, Faruk Polat, and Cem Yílmaz. "Views as first-class citizens in object-oriented databases". In: *VLDB J.* 14.2 (2005), pp. 155–169. DOI: 10.1007/s00778-004-0122-8. URL: http://dx.doi.org/10.1007/s00778-004-0122-8.

[3] Giovanna Guerrini et al. "A Formal View of Object-Oriented Database Systems". In: *TAPOS* 3.3 (1997), pp. 157–183.

[4] Harumi Kuno, Young-gook Ra, and Elke A. Rundensteiner. *The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation.* Tech. rep. University of Michigan, 1995.

[5] Harumi Kuno and Elke A. Rundensteiner. "Materialized Object-Oriented Views in MultiView". In: *Proceedings RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering - Distributed Object Management, Taipei, Taiwan, March 6-7, 1995.* 1995, pp. 78–85. DOI: 10.1109/RIDE.1995.378742. URL: http://doi.ieeecomputersociety.org/10.1109/RIDE.1995.378742.

[6] Harumi Kuno and Elke A. Rundensteiner. "Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views". In: *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana.* 1996, pp. 310–317. DOI: 10.1109/ICDE.1996.492178. URL: http://dx.doi.org/10.1109/ICDE.1996.492178.

[7] Jonathan Martí et al. "Towards DaaS 2.0: Enriching Data Models". In: *IEEE Ninth World Congress on Services, SERVICES 2013, Santa Clara, CA, USA, June 28 - July 3, 2013.* 2013, pp. 349–355. DOI: 10.1109/SERVICES.2013.59. URL: http://dx.doi.org/10.1109/SERVICES.2013.59.

[8] Antoni Olivé. *Conceptual modeling of information systems.* Springer, 2007. DOI: 10.1007/978-3-540-39390-0. URL: http://dx.doi.org/10.1007/978-3-540-39390-0.

[9] Zhiyong Peng et al. “Using Object Deputy Model to Prepare Data for Data Warehousing”. In: *IEEE Trans. Knowl. Data Eng.* 17.9 (2005), pp. 1274–1288. DOI: 10.1109/TKDE.2005.154. URL: http://doi.ieeecomputersociety.org/10.1109/TKDE.2005.154.

[10] Elke A. Rundensteiner. “Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases”. In: *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings.* 1992, pp. 187–198. URL: http://www.vldb.org/conf/1992/P187.PDF.

[11] Elke A. Rundensteiner. “Objected-Oriented View Technology: Challenges and Promises”. In: *CODAS*. 1996, pp. 299–308.

[12] José Samos, Fèlix Saltor, and Jaume Sistac. “Definition of Derived Classes in OODBs”. In: *IDEAS*. 1998, pp. 150–158.

[13] GemTalk Systems. *GemStone/S website*. 2015. URL: http://gemtalksystems.com/products/gs64/ (visited on 05/15/2015).

[14] K. Tanaka, M. Yoshikawa, and K. Ishihara. “Schema virtualization in object-oriented databases”. In: *Data Engineering, 1988. Proceedings. Fourth International Conference on*. Feb. 1988, pp. 23–30. DOI: 10.1109/ICDE.1988.105442.

[15] Manuel Torres and José Samos. “Closed External Schemas in Object-Oriented Databases”. In: *Database and Expert Systems Applications, 12th International Conference, DEXA 2001 Munich, Germany, September 3-5, 2001, Proceedings*. 2001, pp. 826–835. DOI: 10.1007/3-540-44759-8_80. URL: http://dx.doi.org/10.1007/3-540-44759-8_80.