14 DEFINING VIEWS IN AN IMAGE DATABASE SYSTEM [*]

Vincent Oria, M. Tamer Özsu, Duane Szafron and Paul J. Iglinski

Department of Computing Science - University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{oria, ozsu, duane, iglinski}@cs.ualberta.ca

Abstract: A view mechanism can help handle the complex semantics in emerging application areas such as image databases. This paper presents the view mechanism we defined for the DISIMA image database system. Since DISIMA is being developed on top of an object-oriented database system, we first propose a powerful object-oriented view mechanism based on the separation between types (interface functions) and classes that manage objects of the same type. The image view mechanism uses our object-oriented view mechanism to allow us to give different semantics to the same image. The solution is based on the distinction between physical salient objects which are interesting objects in an image and logical salient objects which are the meanings of these objects.

14.1 INTRODUCTION

Views have been widely used in relational database management systems to extend modeling capabilities and to provide data independence. Basically, views in a relational database can be seen as formulae defining virtual relations that are not produced until the formulae are applied to real relations (view materialization is an implementation/optimization technique). View mechanisms are useful in other newly emerging application areas of database technology. In this paper, we discuss a view mechanism for one of those areas, image databases. This work is conducted within the context of the DISIMA (DIStributed Image database MAnagement system) prototype which is under development at the

[*]This research is supported by a strategic grant from the Natural Science and Engineering Research Council (NSERC) of Canada.

University of Alberta. Since DISIMA uses object-oriented technology, we deal with object-oriented views.

Despite several research efforts in the object-oriented community [4, 1, 14, 16], the objective of a view mechanism, as defined for the relational model, has not yet been achieved. The problem is more complex and may be too general in the object-oriented environment. Assume that a virtual class is defined from an existing schema. Will each virtual object in this virtual class get a new OID each time the view is activated? This violates object-oriented principles. Can this virtual class be considered as a normal class? In this case, what is its place in the class hierarchy?

Due to the volume and the complexity of image data, image databases are commonly built on top of object or object-relational database systems. Image databases, in particular, can benefit from a view mechanism. Specifically, an image can have several interpretations that a view mechanism can help to model. The DISIMA system [11] defines a model that is capable of handling an image and all the meta-data associated with it, including syntactic characterization (shape, color and texture) of *salient objects* contained in the image. The level at which the syntactic features are organized and stored is called the physical salient object level. Each physical salient object can then be given a meaning at the logical salient object level. How do we get this information? In general, salient object detection and annotation is a semi-automatic or a manual process.

Given the fact that we can manually or automatically extract meta-data information from images, how do we organize this information so that an image can be interpreted with regard to a context? That is, if the context of an image changes, the understanding of the image may change as well. Consider an electronic commerce system with a catalog containing photographs of people modeling clothes and shoes. From the customer's point of view, interesting objects in this catalog are shirts, shorts, dresses, etc. But the company may want to keep track of the models as well as clothes and shoes. Assume the models come from different modeling agencies. Each of the agencies may be interested in finding only pictures in which their models appear. All these users of the same database (i.e. the catalog) have different interpretations of the content of the same set of images.

Defining an image content with regard to a context helps capture more semantics, enhances image modeling capabilities, and allows the sharing of images among several user groups. Our mechanism of image views, currently being implemented in the DISIMA system, allows users to virtually create an image interpretation context that includes salient object semantics and representations.

Our class derivation mechanism is general enough to be applied to any object-oriented application and is presented in Section 14.2. Section 14.3 describes the DISIMA model and extends it to support views on images, Section 14.4 presents the image view definition language and describes the current implementation of the image views, Section 14.5 discusses the related work and Section 14.6 concludes.

14.2 DERIVED CLASSES

We separate the definition of object characteristics (a *type*) from the mechanism for maintaining instances of a particular type (a *class*) for several well known reasons [9]. A *type* defines behaviors (or properties) and encapsulates hidden behavioral implementations (including state) for objects created using the type as template. We use the term behaviors (or properties) to include both public interface functions (methods) and public state (public instance variables). The behaviors defined by a type describe the *interface* for the objects of that class. A *class* ties together the notion of *type* and *object instances*. The entire group of objects of a particular type, including its subtypes is known as the *extent* of the type and is managed by its class. We refer to this as *deep extent* and introduce *shallow extent* to refer only to those objects created directly from the given type without considerating its subtypes. For consistency reasons all the type names used in this paper start with $T\_$.

Let C be the set of class names. If C is a class name, $T(C)$ gives the type of C and $\Gamma(C)$ denotes the extent of the class C. We denote by $\mathcal{T}$, the graph representing the type hierarchy. We consider two types of derived classes: simple derived classes (derived from a single class called *the parent class*) and composed derived classes (derived from two or more parent classes). We will use the term root class to refer to a non-derived class. In the same way, a root object refers to an object of a root class. The derivation relationship is different from the specialization/generalization one in the sence that the objects and properties introduced are obtained from data previously stored in the database.

14.2.1 Simple Derived Class

A simple derived class is a virtual class derived from a single parent class.

Definition 1 *A* **derived class** C_d *is defined by* (C, Φ, Ψ) *where:*

- *C is the parent class*
- *Φ, the filter, is a formula that selects the valid objects from C for the extent of C_d*
- *Ψ, the interface function, defines the type of C_d by combining the functions* A: Augment *and* H: Hide *such that $\Psi = A \circ H$, where A maps a set of objects of a particular type to a set of corresponding objects in a type with some addtional properties. Similarily H hides some properties.*
- $\Gamma(C_d) = \Psi(\Phi(\Gamma(C)))$

As defined, Ψ, A and H have to be applied to sets of objects of a certain type to return sets of objects of another type. To avoid introducing news terms, we will extend their applications to types.

If $\alpha, \beta, \gamma, \delta$ are properties defined in T_C, $H(T\_C, \{\alpha, \beta\})$ will create a new type (let us call it $T\_restricted\_C$) in which only the properties γ, δ are defined. Hence $T\_restricted\_C$ is a supertype of T_C.

$A(T\_C, \{(\mu : f_1), (\nu : f_2)\})$ will create a type ($T\_augmented\_C$) with the additional properties μ and ν, where f_1 and f_2 are functions that implement them. $T\_augmented\_C$ is a subtype of T_C.

$A(H(T\_C, \{\alpha, \beta\}), \{(\mu : f_1), (\nu : f_2)\})$ defines the type T_C_d for a class C_d derived from a class C with the properties α, β of T_C hidden and μ, ν as new properties.

In general, the type $T(C_d)$ of a class C_d derived from the class C, is a sibling of $T(C)$. However, if no properties are hidden, $T(C_d) \leq T(C)$, where $\leq$ stands for a subtyping relationship and $\geq$ for a supertyping relationship. Alternatively, if no properties are added, $T(C_d) \geq T(C)$. The notion of sibling generalizes the notion of subtyping and supertyping. The most general case where some properties are removed and new ones are added is illustrated by Figure 14.1. In this example, we assume that the following properties are defined for the different types:

- T_Person(SIN: int, LastName: string, FirstName: string, Sex: char, DateOfBirth: date)
- T_Restricted_Person(SIN: int, LastName: string, FirstName: string, Sex: char)
- T_Augmented_Restricted_Person(SIN: int, LastName: string, FirstName: string, Sex: char, Age: int)

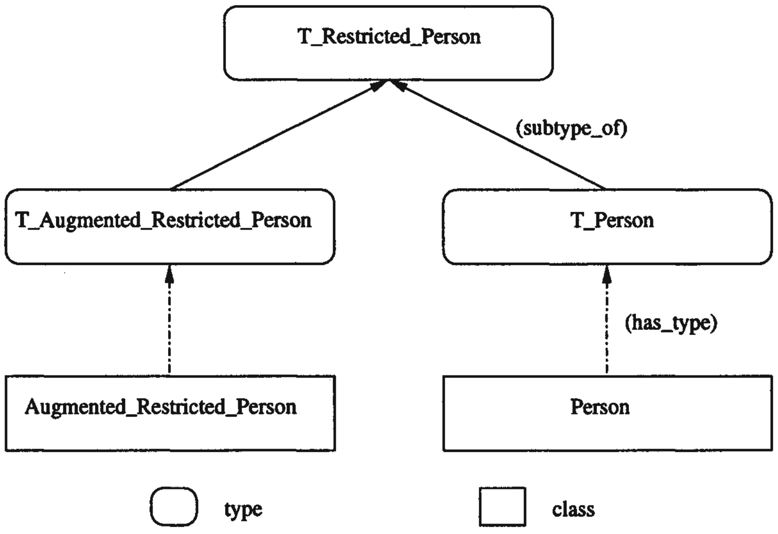

Figure 14.1: An Example of a Derived Class and its Type

In Figure 14.1, the extent of *Augmented_Restricted_Person* is a subset of the extent of *Person* with a different interface defined by the type *T_Augmented_Restricted_Person.*

14.2.2 Composed Derived Class

Assume that the type *T_Person* has two subtypes *T_Student* and *T_Faculty.* Some of the students teach and some faculty do only research. The Type *T_Student* has the properties *(Year: int)* and *(Teach: boolean)* while the properties *(HiringDate: date)* and *(Teach: boolean)* are defined for *Faculty.* We would like to derive a class *Teacher* of all the persons who teach with the property *(TimeServed: int)* obtained either from *HiringDate* or from *Year* depending on the type of the root object. The class *Teacher* cannot be directly derived from the class *Person* since the useful properties are not defined in *T_Person.* In the following, we propose a way (composed derived class) to solve this problem.

Definition 2 *Let $(C_1, C_2) \in \mathcal{C}^2$ be a pair of classes. Then:*

- *$C_d = C_1 * C_2$ with a filter Φ and an interface Ψ is a composed derived class with $\Gamma(C_d) = \Psi(\Phi(\Gamma(C_1) \cap \Gamma(C_2))$*
- *$C_d = C_1 + C_2$ with a filter Φ and an interface Ψ is a composed derived class with $\Gamma(C_d) = \Psi(\Phi(\Gamma(C_1) \cup \Gamma(C_2))$*
- *$C_d = C_1 - C_2$ with a filter Φ and an interface Ψ is a composed derived class with $\Gamma(C_d) = \Psi(\Phi(\Gamma(C_1) - \Gamma(C_2))$*

with $T(C_d)$ a sibling of $Anc(T(C_1), T(C_2))$ where $Anc(T(C_1), T(C_2))$ is a function that returns the first common ancestor of $(T(C_1), T(C_2))$ in the type hierarchy $\mathcal{T}$.

The semantics of the constructive operations $\{*, +, -\}$ are respectively based on the basic set operations $\cap, \cup$ and $-$. As defined, $\{*, +, -\}$ are binary operations but the formulae obtained can be seen as terms and be combined for more complex ones. Note that C_1 and C_2 can be derived classes as previously defined. The ancestor function *Anc* works fine when $\mathcal{T}$ is rooted. When this is not the case, a common supertype T_C is created for $T(C_1), T(C_2)$. In the worst case, T_C will not have any properties in it.

The problem of deriving a class *Teacher* can be solved by defining a simple derived class *Student_Teacher* whose extent is a subset of all the students. In the same way, we derive the class *Faculty_Teacher* from *Faculty. Teacher* is then defined as $Teacher = Student\_Teacher + Faculty\_Teacher$. The type *T_Teacher* is a subtype of *T_Person* which is the common ancestor (Figure 14.2).

14.2.3 Identifying and Updating a derived object

A derived object is always derived from one and only one root object although its properties can be totally different from the properties of the root object.

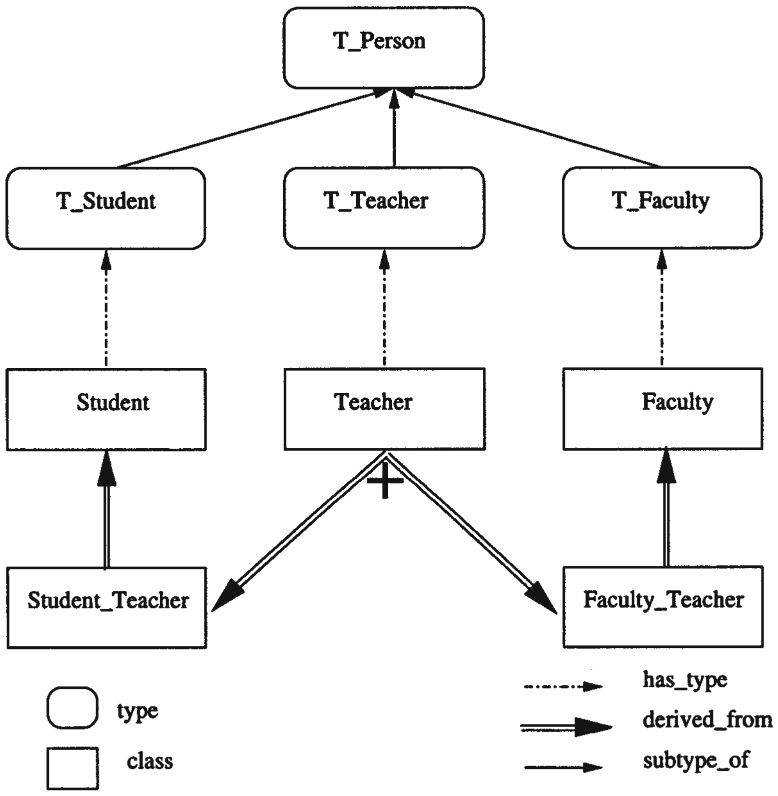

Figure 14.2: An Example of a Composed Derived Class and its Type.

This happens when all the properties of the root class are hidden and new ones are defined for the derived class. Hence, a derived object can be seen as a root object viewed from another angle (the interface function of the derived class). Both the derived object and its corresponding root object can be identified by the OID of the root object (ROOT_OID). If we redefine the notion of OID as follows: OID= $< class\_name, ROOT\_OID >$ then the root object can be differentiated from the derived one. This OID defines a logical idenfier for any object including the derived ones independently from any view implementation technique. In the case of view materialization with incremental maintenance, an active research area [3, 6, 12, 2], the derived object OID is a key candidate and can be directly used as identifier.

A derived object knows its root object. Therefore, updating a property inherited from the root type can easily be propagated to the root object. Creating new objects for a derived class should first create the objects in the root class with some possible unknown property values.

14.3 DEFINING IMAGE VIEWS IN DISIMA

The mechanism of image views presented in this paper is based on the DISIMA image DBMS, which is a research project for developing a distributed interoperable DBMS for image and spatial applications. The DISIMA model aims at organizing the image and associated meta-data to allow content-based queries.

14.3.1 The DISIMA Model: Overview

The model provides efficient representation of images and related data to support a wide range of queries. The DISIMA model, as depicted in Figure 14.3, is composed of two main blocks: the image block and the salient object block. We define a *block* as a group of semantically related entities.

14.3.1.1 The Image Block. The image block is made up of two layers: the *image* layer and the *image representation* layer. We distinguish an image from its representations to maintain an independence between them, referred to as *representation independence.*

At the *image* layer, the user defines an image type classification. Figure 14.4 depicts a type hierarchy for an electronic commerce application that represents the catalogs as classes. The general *T_Catalog* type is derived from the root type *T_Image*, the root image type provided by DISIMA. The type *T_Catalog* is specialized by two types: *T_ClothingCatalog*, and *T_ShoesCatalog.*

14.3.1.2 The Salient Object Block. The *salient object* block is designed to handle salient object organization. A simple example of a salient object hierarchy, corresponding to the image hierarchy defined in Figure 14.4, is given in Figure 14.5.

DISIMA distinguishes two kinds of salient objects: physical and logical salient objects. A *logical salient object* is an abstraction of a salient object

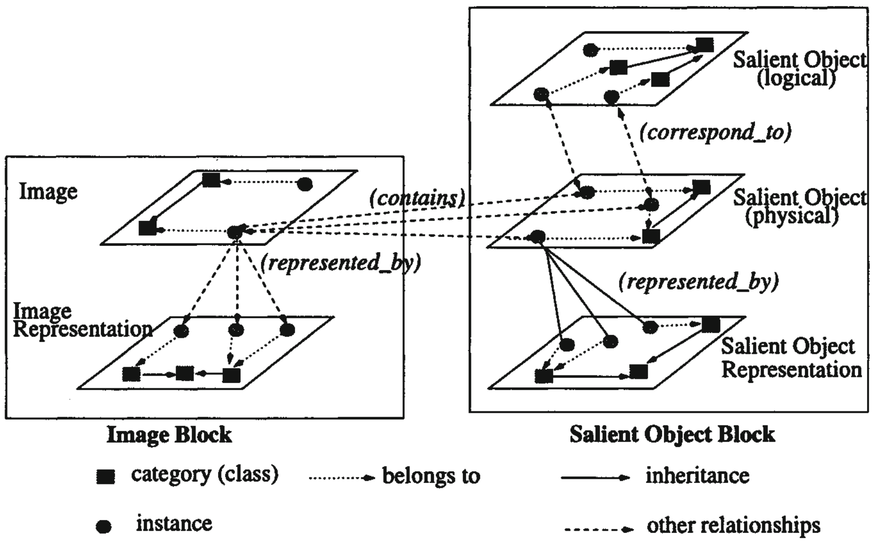

Figure 14.3: The DISIMA Model Overview.

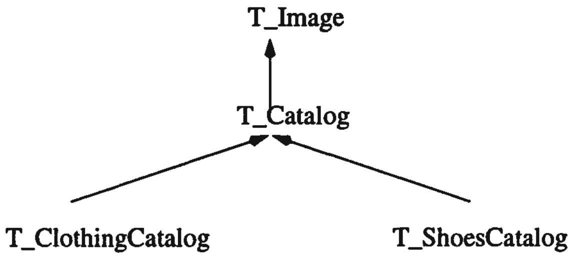

Figure 14.4: An Example of an Image Hierarchy.

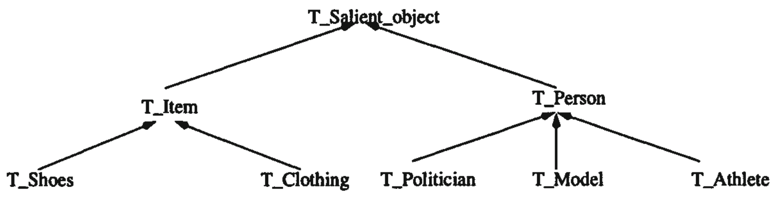

Figure 14.5: An Example of Logical Salient Object Hierarchy.

that is relevant to some application. For example, an object may be created as an instance of type *Politician* to represent President Clinton. The object "Clinton" is created and exists even if there is yet no image in the database in which President Clinton appears. This is called a *logical salient object*; it maintains the image independent generic information that might be stored about this object of interest (e.g., name, position, spouse). Particular instances of this object (called *physical salient objects*) may appear in specific images. There is a set of information (data and relationships) linked to the fact that "Clinton appears in an image". The data can be the colors of his clothes, his localization, or his shape in this image.

We now give a formal definition of the content of an image, using physical and logical salient objects.

Definition 3 *A* **physical salient object (PSO)** *is a region of an image, that is, a geometric object (without any semantics) in a space (defined by an image) with the following properties: shape, color, and texture.*
A **logical salient object (LSO)** *is the interpretation of a region. It is a meaningful object that is used to give semantics to a physical salient object.*

Definition 4 *Let $\mathcal{L}$ be the set of all logical salient objects and $\mathcal{P}$ be the set of all physical salient objects. The* **content of an image** *i is defined by a pair $Cont(i) = <\mathcal{P}^i, s>$ where:*

- $\mathcal{P}^i \subseteq \mathcal{P}$ is a set of physical salient objects,
- $s : \mathcal{P}^i \longrightarrow \mathcal{L}$ maps each physical salient object to a logical salient object.

An image is a basic unit in the DISIMA model and is defined as follow:

Definition 5 *An* **image** *i is defined by a triple <Rep(i), Cont(i), Desc(i)> where:*
- Rep(i) is a set of representations of the raw image in a format such as GIF, JPEG, etc;
- Cont(i) is the content of the image i;
- Desc(i) is a set of descriptive alpha-numeric data associated with i.

Color and texture characterizing the whole image are part of the *Desc*(i).

14.3.1.3 How to Recognize the Salient Objects of an Image. Despite progress in the computer vision field, automatic detection of objects is "hard" and application-dependent. The state of the art in computer vision does not permit automatic recognition of an arbitrary scene [15].

Assume an object is detected by the image analysis software. In the general case, this object is a syntactic object without any semantics. That is, it is a region of an image with properties such as color, shape and texture. Another challenge is to provide syntactic objects with semantics. Assume the object detected is a person. How can a computer assign a name to this person? This example explains why, in most cases, the image analysis is semi-automatic or manual.

One component of the DISIMA project is in charge of image processing and object detection. Our first concern was images with people. The image processing software detects the faces contained in the image with a minimum bounding rectangle (useful for spatial relationships) and a human-annotator assigns a logical salient object to the face. In addition, an image has some descriptive properties such as date and photographer that have to be provided. In the remainder of the paper, we assume that the information at the two levels of salient objects is provided.

The two levels of salient objects ensure the semantic independence and multi-representation of salient objects. The idea of image views is based on this semantic independence and the class derivation mechanism presented in Section 14.2.

14.3.2 Extending the DISIMA Model to Support Image Views

A DISIMA schema is composed of two sub-schemas: the image type hierarchy and the salient object type hierarchy. An image view can be defined by a derived image class or by giving different semantics to the salient objects an image contains using derived logical salient object classes.

Derived classes can be defined for both image and salient-object classes. Derived salient object classes are illustrated by examples shown in Section 2. The aim of a derived image class is to filter salient objects or to redefine their semantics through derived logical salient object classes.

14.3.2.1 Defining Image Views Using Derived Image Classes. A derived image class, in addition to defining a new type, converts some salient objects of a parent image class into non-salient in a derived one.

Definition 6 *A* **derived image class** *is a class derived from an image class that specifies the valid logical salient objects for images in its extent. If i_d is an image derived from an image i, then the set of physical salient objects contained in i_d is a subset of the set contained in i. The physical salient objects in i_d are those for which the corresponding logical salient objects belong to one of the valid logical salient objects.*

In addition to redefining the type, a derived image class redefines the content of the images it contains. For example, from the *ClothingCatalog* class defined in Figure 14.4, we can derive two different catalogs giving different interpretations of the images in the *ClothingCatalog* image class: the customer catalog class (*CustomerCatalog*) and the clothing company catalog (*CompanyCatalog*). The customers are interested in finding clothing from the catalog. Therefore, the valid logical salient object class is *Clothing*. In addition to the clothing, the company may be interested in keeping some information about the models.

A composed derived image class can also be created. For example, from *ClothingCatalog* we can derive the class *FemaleClothingCatalog*. We can also derive *FemaleShoesCatalog* from *ShoesCatalog*. *FemaleClothingCatalog* and *FemaleShoesCatalog* can be combined using the + operator to derive a class *FemaleApparelCatalog*. The common ancestor of *FemaleClothingCatalog*, and *FemaleShoesCatalog* is *Catalog*. Therefore the type of *FemaleApparelCatalog* has to be a sibling of the type of *Catalog* (Figure 14.6).

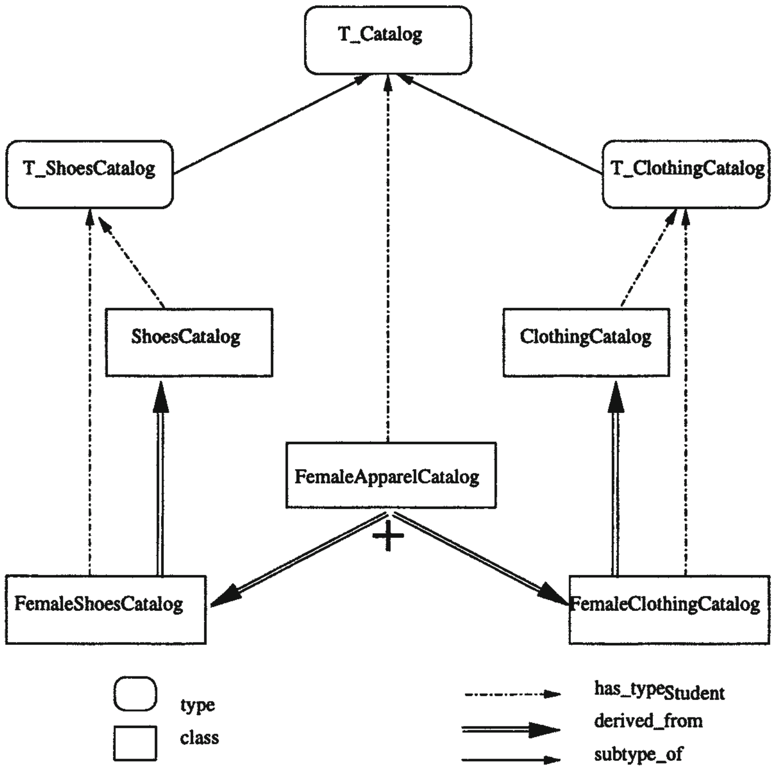

Figure 14.6: An Example of a Composed Derived Image Class

14.3.2.2 Defining Image Views Using Derived Logical Salient Object Classes. Definition 5 defines the content of an image i as a pair $Cont(i) =< \mathcal{P}^i, s >$ where $\mathcal{P}^i$ represents the physical salient objects and the function s maps each physical salient object to a logical salient object. An image i_d can be derived from i and $Cont(i_d) =< \mathcal{P}^i, s_d >$. Assume we derived a logical salient object class L_1 from the logical salient object class L and that all the physical salient objects in $\mathcal{P}^i$ are mapped to objects of L. If we note by f the interface function that transforms an object of L to an Object of L_1, and we define $s_d = s \circ f$, then i_d is a derived image that contains L_1 objects.

For example, the classes *FemaleClothing* and *FemaleShoes* can be respectively derived from *Clothing* and *Shoes* (Figure 14.5). A composed derived class *FemaleApparel* can be derived from the two previously derived classes and the derived image class *FemaleApparelCatalog* can be defined as images containing female apparels. Of course, *T_FemaleClothing* and *T_FemaleShoes* can respectively be different from *T_Clothing* and *T_Shoes*. *T_FemaleApparel* is then, a sibling of *T_Apparel*.

Definition 7 *An* **image view** *is defined by:*

- *a derived image class*
- *redefining the semantics of the physical salient objects an image contains through derived logical salient object classes.*

14.4 THE IMAGE VIEW DEFINITION LANGUAGE

The view definition language allows us to define derived classes. Queries in the view definition are expressed in MOQL (Multimedia Object Query Language) [10], the query language defined for DISIMA. MOQL extends the standard object query language, OQL [5] with predicates and functions to capture temporal and spatial relationships. Most of the extensions have been introduced in the *where* clause in the form of new predicates including the *contain* predicate to check if a salient object belongs to an image. The convention used in the language definition is: [ ] for optional, { } (different from **{}** which are part of the language) for 0 to n times, and | for one and only one among different possibilities. The view language allows us to create and delete derived classes.

- Create a derived class

```
derive { <derived class name> from <class definition >
    [ augment {<virtual property name> as <query> |
              <function name> ;}]
    [ hide    < property list>]
    {cast     < property> into <derived type> }
    [ content < valid salient object class list>]
    extent    <extent name > [as <query>]
};

    <class definition> := <class name> |
    (<class definition > union | intersect | minus <class definition>)
```

- Delete a derived class

```
    delete <derived class name>
```

The *derive* clause is used to define a derived logical salient object class, as well as derived image classes. The classes that the *derived classes* are derived from can be ordinary or derived classes. The query in the *extent* clause defines the derived class extent and must return a unique subset of the combination of the parent class extents. The *augment* clause is used to define new properties. A query can invoke an existing property. In this case, the keyword *this* is used to refer to the current object. If $(\alpha : T(C))$ is a property and C_d is a class derived from C, then the clause *cast* can be used to cast the type of α into $T(C_d)$.

The *content* clause allows us to define the valid logical salient objects. This clause is used only for derived image classes and does more than hide the previous image content and redefine a new one. It implements the image views using derived logical salient object classes. If the logical salient object class mentioned is a derived class, then it changes the semantics of the physical salient objects from parent to derived objects. Assume a salient object class S_d is derived from class S and an image i (element of the image class I) contains a salient object of type $T(S)$. If we derive an image class I_d from I with the clause *content* S_d, image i_d derived from i will contain a salient object of type $T(S_d)$ instead of $T(S)$. For example, in the image view *CustomerClothing* that follows, an image of *CustomerCatalog* contains elements of *CustomerClothing,* rather than *Clothing* as salient objects.

14.4.1 Examples of Image Views

In the following, we give some examples of image views derived from the catalog database. The corresponding schema expressed in the ODMG object model [5] is given in the Appendix. The schema given in the Appendix can be seen as the view of the company: each image contains models and clothes. The examples correspond to the Customer View, the Female Clothing Catalog View, and the Female Aparel Catalog view.

Image View 1 The CustomerCatalog view

```
    derive   {CustomerClothing from Clothing
    augment  inStock as this.inStock();
             avgPriceForType as
             avg(Select c.price
             From Clothes c
             Where c.type = this.type);
    hide     stock, lastOrderDate, lastArrivalDate, nextArrivalDate
    extent   CustomerClothes};

    derive   {CustomerCatalog from ClothingCatalog
    hide     photographer, date, time, place
    cast     accessories into Set<Ref<CustomerCatalogs>>
    extent   CustomerCatalogs
    content  CustomerClothing};
```

The derived class *CustomerClothes* redefined *Clothes* for the customers' use. Attributes *stock, lastOrderDate, lastArrivalDate, nextArrivalDate* are hidden and the virtual attribute *avgPriceForType* returns the average price for this type of clothing.

The image view *CustomerCatalog* uses the image class *Catalog* renamed as *CustomerCatalog* with *CustomerCatalogs* as its extent name. All the images are available but their content will be limited to objects of type *CustomerClothes* which redefines *Clothing*. Attributes ***photographer, date, time, place*** are hidden. The attribute ***accessories*** was defined as a set of images from *Catalog*. Its type has to be changed to set of *CustomerCatalogs* to ensure consistency.

Image View 2 The FemaleClothingCatalog view
```
    derive    {FemaleClothing from CustomerClothing
    extent    FemaleClothes as
              Select c
              From CustomerClothes c
              Where c.sex = 'female' or c.sex = 'unisex'};

    derive    {FemaleClothingCatalog from ClothingCatalog
    hide      photographer, date, time, place
    cast      accessories into Set<Ref<CustomerCatalogs>>
    extent    FemaleClothingCatalogs
    content   FemaleClothing};
```

Only images containing female items are selected from the clothing catalog. The salient objects are restricted to female clothing.

Image View 3 The FemaleApparelCatalog view
```
    derive    {FemaleShoes from Shoes;
    augment   inStock as this.inStock();
              avgPriceForType as
              avg(Select s.price
              From Shoes s
              Where s.type = this.type);
    hide      stock, lastOrderDate, lastArrivalDate, nextArrivalDate
    extent    FemaleShoesExtent as
              Select s
              From ShoesExtent c
              Where c.sex = 'female' };

    derive    {FemaleShoesCatalog from ShoesCatalog
    hide      photographer, date, time, place
    extent    FemaleShoesCatalogs
    content   FemaleShoes};

    derive    {FemaleApparelCatalog from FemaleClothingCatalog union
              FemaleShoesCatalog
    extent    FemaleApparelCatalogs};
```

The *FemaleApparelCatalog* combines the *FemaleClothingCatalog* and the *FemaleShoesCatalog* into a new derived catalog.

14.4.2 Implementing Derived Classes in DISIMA

The distinction between types and classes is not supported by most object-oriented languages in current use. DISIMA is being implemented on top of ObjectStore [8] using C++. DISIMA provides types for image and logical salient objects that can be subtyped by the user. The implementation we describe in this section simulates the idea using C++. We implement all our types as C++ classes. We call these C++ classes *type classes* and their names start with *T_*. For example *T_Person* will be a type class for the class *Person*. Our classes are objects of the C++ class *C_Class*. *C_Class* has a subclass *D_Class* for derived classes. The properties defined for *C_Class* are:

- Name: name of the class
- Type: type class name
- SuperclassList: list of the superclasses
- SubClassList: list of the subclasses
- ShallowExtent (virtual function): The shallow extent of the class
- DependentList: list of classes derived depending on this one

The properties defined for *D_Class* are:

- RootClassList: list of the classes it is derived from
- Filter: filter function
- ShallowExtent: redefined
- MaterializationFlag: set when the ShallowExtent is up-to-date
- Change: function used to unset the MaterializationFlag

The *DependentList* in the class *C_Class* contains all the classes derived from that class and also all the derived classes for which an augmented property is computed using objects of that class. Since the type of a derived class can be different from the type of its root class we choose to materialize the derived class extent. An object of *C_Class* represents a user's class and the extent (*ShallowExtent*) property returns objects of the type class (*Type*). The *SubClassList* can be used to recursively compute the deep extent. To simplify the materialization process, we only store one level of root class. That is, the *RootClassList* of a derived class contains only non-derived classes. A derived class extent is materialized the first time the class is referred to and the materialization flag is set. Each time new objects are created, modified or deleted in a root class, a change message is sent to each of the classes in the *DependentList* to unset the materialization flag. If the materialization flag is unset when a derived class is accessed, the derived class extent is recomputed and the materialization flag is reset.

When the augmented properties of a derived object are computed from the single root object without any aggregate, the management algorithm for incremental view maintenance can easily be implemented as follows. An object of a derived class contains the OID of the root object it is derived from. The *Change* method passes the OID of the changed root object (new, deleted or updated) to the derived class object where it is kept in the *ChangeList* of the derived class object. The *ChangeList* can then be visited to update or create the derived objects for modified or new root objects and to delete derived objects corresponding to deleted root objects.

14.5 RELATED WORK

To the best of our knowledge, there is no other view mechanism defined for image which can be compared to our solution. DISIMA, as well as most multimedia products and prototypes, is being developed on top of an object-oriented database system. We defined an object-oriented view mechanism used in the image view solution. This section will focus on two of the most representative object-oriented view solutions: O_2View and Multiview.

14.5.1 O_2View

O_2View is the view mechanism defined for the O_2 system. O_2View distinguishes two kinds of derived classes: virtual and imaginary classes. The main ideas are the following:

- A virtual class (1) selects through a query, objects existing in the root database; (2) is connected to the root hierarchy; and (3) provides a name for the extension of the virtual class. Its interface can be modified for hiding an attribute or adding a virtual one.
- An imaginary class (1) selects and restructures through a query data from the root database or the view, (2) turns them into objects, (3) is not connected to the root hierarchy, and (4) provides an extension.
- A virtual attribute attaches (possibly restructured) data to an object in the view, through a query on the root database or the view. It augments the original interface of virtual objects.
- An attribute hiding restricts the original interface of root objects. It hides the attributes of a virtual object not to be visible to the end-user

14.5.2 Multiview

Multiview [7] is a research prototype developed at the University of Michigan on top of the GemStone system. Multiview provides updatable materialized object-oriented database views. The main features of the system are:

- Integration of both virtual and base classes into a unified global schema. This is done through a classification algorithm [13] that restructures the whole class hierarchy. Hence, virtual classes participate in the inheritance hierarchy and can be used in the same way base classes are used.

- Generation of schemata composed of user-selected bases and virtual classes.
- Includes incremental view maintenance algorithm for view materialization.

O_2Views [14], makes the distinction between ***virtual classes*** that select through queries existing objects in the database and ***imaginary classes*** for which the selected objects are restructured and turned into objects. Virtual classes are connected to the generalization hierarchy by a *maybe* relationship whereas imaginary classes are not. Multiview [7] integrates the derived classes into the global class hierarchy using a complex classification algorithm [13]. Our solution is simpler and yet more powerful. A virtual class can be derived from one or several classes with its type integrated into the type hierarchy without any modification of the user-defined root classes. In addition to having the object-oriented views features, an image view should provide a semantic independence. That is, the content of the same image can be different from one view to another.

14.6 CONCLUSION

Several object view mechanisms have been proposed since the early 90s [4, 1, 14, 16, 7]. In general, the main problems with these views are [16] (i) expressive power (restrictions on queries defining views), (ii) reusability and modeling accuracy (insertion of the views into the generalization hierarchy), and (iii) consistency (stability in OID generation).

Problems (i) and (ii) are somewhat related. For example, using the view mechanism in [14], if the user wants the view class to be linked to the generalization hierarchy, the query that generates the view class has to be restricted. In addition, the problem (ii) raises a typing problem (how is the type of the virtual class related to the type hierarchy?) and a classification problem (how is the extent of a virtual class related to the existing ones?). Finding an answer to these two questions in an environment where the only relationship is the *is-a* relationship can lead to contradictions. The distinction between the derivation hierarchy and the generalization hierarchy in our proposal, based on the distinction between type and class, provides an elegant solution to problems (i) and (ii). In addition, the object-oriented view mechanism presented in this paper allows us to derive classes from several existing ones. Problem (iii) is also solved by the fact that a derived object is seen as a root object with a different interface function. A derived object and its root class share the same OID but are uniquely identified by the pair $< class\_name, OID >$ which is invariant even if the derived object is recomputed.

The DISIMA model separates the objects contained in an image (***physical salient objects***) from their semantics (***logical salient objects***). Using our object view mechanism, we proposed an image view mechanism that allows us to give different semantics to the same image. For example, a derived image class can be defined by deriving new logical salient object classes that give new semantics to the objects contained in an image or by hiding some of the objects by directly defining a derived image class.

The main contributions of this paper are the proposal of a powerful object-oriented view mechanism based on the distinction between class and type, a proposal of an image view mechanism based on image semantics and the image view implementation using a language that does not intrinsically support the distinction between class and type.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 238—247, Denver, Colorado, May 1991.

[2] B. Adelberg, H. Garcia-Molina, and J. Widom. The STRIP rule system for efficiently maintaining derived data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 147—158, Tucson, Arizona, May 1997.

[3] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouse. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 417—427, Tucson, Arizona, May 1997.

[4] E. Bertino. A view mechanism for object-oriented databases. In *Proceedings of International Conference on Extending Data Base Technology*, pages 136—151, Viena, Austria, March 1991.

[5] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, CA, 1997.

[6] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Munick, and K. A. Ross. Supporting multiple view maintenance policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 405—416, Tucson, Arizona, May 1997.

[7] H. A. Kuno and E. A. Rundensteiner. The MultiView OODB view system: Design and implementation. *Journal of Theory and Practice of Object Systems (TAPOS)*, 2(3):202—225, 1996.

[8] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of ACM*, 34(10):19—20, 1991.

[9] Y. Leontiev, M. T. Özsu, and D. Szafron. On separation between interface, implementation and representation in object DBMSs. In *Proceedings of Technology of Object-Oriented Languages and Systems 26th International Conference (TOOLS USA98)*, pages 155—167, Santa Barbara, August 1998.

[10] J. Z. Li, M. T. Özsu, D. Szafron, and V. Oria. MOQL: A multimedia object query language. In *Proceedings of the 3rd International Workshop on Multimedia Information Systems*, pages 19—28, Como, Italy, September 1997.

[11] V. Oria, M. T. Özsu, X. Li, L. Liu, J. Li, Y. Niu, and P. J. Iglinski. Modeling images for content-based queries: The DISIMA approach. In *Proceedings of 2nd International Conference of Visual Information Systems*, pages 339—346, San Diego, California, December 1997.

[12] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 507—518, Montreal, Canada, June 1996.

[13] E. A. Rundensteiner. A classification algorithm for supporting object-oriented views. In *Proceedings of International Conference on Information and Knowledge Management*, pages 18—25, June 1994.

[14] Casio Santos, Claude Delobel, and Serge Abiteboul. Virtual schema and bases. In *Proceedings of International Conference of extending Data Base Technology*, pages 81—94, Cambridge,UK, March 1994.

[15] A. W. M. Smeulders, M. L. Kersten, and T. Gevers. Crossing the divide between computer vision and data bases in search of image databases. In *Proceedings of 4th IFIP 2.6 Working Conference on Visual Database Systems - VDB 4*, pages 223—239, L'Aquila, Italy, May 1998.

[16] X. Ye, C. Parent, and S. Spaccapietra. Derived objects and classes in DOOD system. In *4th International Conference on Deductive and Object-Oriented Databases, DOOD 95*, pages 539—556, Singapore, December 1995.

Appendix: Sample Schema

```
class Image{
    Set<Ref<Representation>> representations;
    Set<Ref<PhysicalSalientObject>> physicalSalientObjects
            inverse image;
    // Methods
    display(); }
class Catalog: Image{
    Person photographer; Date date; Time time; String place; }
class LogicalSalientObject{
    Set<Ref<PhysicalSalientObject>> physicalSalientObjects
            inverse logicalSalientObject;
    //Methods
    Region region(Image m); // salient object's region in image m
    Color color(Image m); // salient object's color in image m
    Texture texture(Image m); // salient object's texture in image m }
class Person: LogicalSalientObject{
    String name; String occupation; Address address; }
class Model: Person{
    String : agency; }
class Apparel: LogicalSalientObject{
    String name; String type; Real price; Set<Real> size;
    Manufacturer manufacturer; Integer stock; String colors;
    Date lastOrderDate; Date lastArrivalDate; Date nextArrivalDate;
    //Methods
    Boolean inStock(); // true if the the clothing is in stock }
class Clothing: Apparel {
    Set<Ref<Catalog>> accessories;// images of items that match with the cloth }
class Shoes: Apparel {
    String sole;String upper; }
class PhysicalSalientObject{
    Ref<LogicalSalientObject> logicalSalientObject
            inverse physicalSalientObjects;
    Ref<Image> image
            inverse physicalSalientObjects;
    Region region; Color color; Texture texture }
Set<Ref<SalientObject>> SalientObjects; //all salient objects
Set<Ref<Person>> Persons; //salient objects of type Person
Set<Ref<Model>> Models; //salient objects of type Model
Set<Ref<Clothing>> Clothes; //salient objects of type Clothing
Set<Ref<Shoes>> ShoesExtent; //salient objects of type shoes
Set<Ref<Image>> Images; //all images
```