

Modeling Data Integration with Updateable Object Views

Piotr Habela¹, Krzysztof Kaczmarski², Hanna Kozankiewicz³, Kazimierz Subieta^{1,3}

¹ Polish-Japanese Institute of Information Technology, Warsaw, Poland
habela@pjwstk.edu.pl

² Warsaw University of Technology, Warsaw, Poland
kaczmars@mini.pw.edu.pl

³ Institute of Computer Science PAS, Warsaw, Poland
{hanka, subieta}@ipipan.waw.pl

Abstract. Recently, a range of applications of views increases. Views are not anymore tightly related to classical databases – there are proposals to use them as means of data transformation and integration in distributed environment. Despite many aspects of view applications, there is still lack of suitable graphical notation that would help designers by providing clear notions from the very early stage of system development. Therefore, our objective is to propose a suitable extension of UML supporting the design process. We focus on modeling object-oriented updateable views in the context of data integration. We believe it is one of the most prominent appliances of views. The paper describes assumed general features of updateable object views and fits them into an object-oriented metamodel. Based on this, we suggest the necessary view-specific notation elements, and present some examples of view modeling.

1 Introduction

Views constitute one of the fundamental database mechanisms, which provide virtual images of data stored in a database. Views are an important component of many applications as they provide abstraction and generalization over data, transformation of data, access and merging data from multiple sources, etc. Views can be used in Web applications as means of integration of heterogeneous resources stored at remote sites into a unified ontology. Traditionally such applications were implemented in lower level languages like C or Java. An advantage of using query language to describe integration is a higher level of abstraction what in turns reduces the time required for development and the cost of maintenance. Although the idea to use views as a mean of data integration is not new (e.g. [2]) and views are an important component of many applications, there are still no satisfactory means to model them. The problem of view's modeling has already appeared in literature [1, 10]. Several useful notions were discussed in the context of relational databases, views, and mapping between object diagrams and database tables. The mentioned guides propose constructs for virtual objects (certain stereotype), view dependencies (dependency) and read-only attributes. However, they are not advanced due to limited capabilities of views in relational databases. They allow to represent flat, read-only views to stored data whereas

modern databases (relational, object-relational, object-oriented, XML-oriented) require much more advanced modeling features.

Earlier approaches to view modeling have presented structural view definition in contrast to operational definition used in classical systems, where a view is defined by a single query [5]. The idea was to provide a representation for resulting objects and their properties instead of defining how they are created. In this sense, our solution is rather structural. However, it is necessary to note that our object and view models are much more sophisticated than the ones assumed by the ODMG standard [3, 11].

We propose a new extension of UML that allows to model integration of data through updateable views [6, 8]. The view features covered by this notation are inspired by the implementation we have developed, based on the Stack-Based Approach. However, we believe that the notation can be also used for modeling views defined within other approaches e.g. in SQL using *instead-of triggers* like in Oracle and MS SQL Server. Our objective was to analyze the ways of extending existing modeling standards to support more advanced capabilities of database systems. We believe that a better view modeling would:

- create a reliable communication tool between business and DBMS designers;
- simplify the process of view design and implementation;
- clarify and standardize views documentation;
- allow to better comprehend the dependencies between objects and views;
- provide uniform expressiveness for all system components.

The proposed notions are dedicated for a design phase of a software lifecycle. We are skeptical about any automatic translation between view models and view implementation because graphical notations are so far less descriptive than programming/query languages. Therefore, we do not assume the elimination of the implementation phase.

The rest of the paper is structured as follows. Section 2 describes our approach to updateable views that is a basis for this paper. In Section 3, we summarize the existing UML notions related to view modeling and indicate their limitations. Section 4 presents the main contribution of this paper i.e., the extensions to the UML meta-model and the respective notation elements proposed to effectively support the updateable object views modeling. In Section 5, we briefly outline the position of view modeling within the software development process. Section 6 concludes.

2 The Approach to Updateable Views

In this section we shortly present our approach to updateable views [7], which is a motivation for the graphical notation presented in this paper. The view mechanism is based on the Stack-Based Approach (SBA) [13], which assumes that query languages are a special kind of programming languages and can be formally described in a similar manner. SBA defines its own query language – Stack-Based Query Language [14].

A database view definition is not a single query (as it is in SQL), but it is a more complex structure. It consists of two parts: the first one determines the so-called *seeds* (the values or references to stored objects that are the basis for building up virtual objects), and the second one redefines the generic operations on virtual objects. The first part of the view definition is an arbitrarily complex functional procedure. The *seeds* it

returns are passed as parameters for the operations on virtual objects. The operations have the form of procedures that override default updating operations. We identified four generic operations that can be performed on virtual objects:

1. **Updating**, which assigns a new value to the virtual object. A parameter the procedure accepts is the new value to be assigned.
2. **Deletion**, which deletes the virtual object.
3. **Insertion**, which inserts a new object into the given virtual object. The object to be inserted is provided as a parameter.
4. **Dereference**, which returns the value of the given virtual object.

For a given view an arbitrary subset of these operations can be defined. If any operation is not defined, it means it is forbidden (we assume no updating through side effects, e.g. by references returned by a view invocation).

Moreover, a view definition may contain nested views, defined within the containing view's environment. Thus, arbitrarily nested complex objects can be constructed.

When a view is invoked in a query, it returns a set of virtual identifiers (that are counterparts of the identifiers of stored objects). Next, when a system tries to perform update operation with a virtual identifier as an l-value, it recognizes that it deals with the virtual object and calls a proper update operation from the view definition. To enable that, a virtual identifier must contain both a seed and the identifier of the view definition. The whole process of view updating is internal to the proposed mechanism and is invisible to view users, who deal with virtual objects in the same manner as with real objects (this feature is known as a view *transparency*).

3 Available Modeling Notions

3.1 UML vs. Object-Oriented Databases

The UML object model is strongly inspired by programming languages like C++ and Java. Thus, the most straightforward to model are applications written with one of the mainstream general purpose programming languages. For other applications like e.g. data modeling for relational databases, specialized profiles are required [1]. One would expect that for modeling object databases the core UML constructs could suffice. In fact, this is not true: we need to go further and reconsider the issue of object's features accessibility and visibility.

The first problem of modeling such database is object relativism, which allows arbitrarily nested object compositions. This feature differs from traditional programming languages, where the structure of objects is physically "flat", since it does not allow object members to be themselves complex objects. Fortunately, UML is not equally restrictive. Nested objects can be represented as class-typed attributes. The detailed structure can be shown using the composition symbol. However, in this case the visual notation becomes rather inconvenient. One needs to choose between the "nested" composition notation, where the class of the subobject is shown inside the class of its owner, and the regular composition (see Fig. 1). CASE tools seldom support the former construct. UML does not allow to draw associations from the nested class symbol, neither. In this sense, it makes the inner object "encapsulated". The lat-

ter approach provides the necessary flexibility, at the cost of expressiveness, as the nested object's features are no longer shown inside the super-object's class symbol, and may be thus mistreated as an association.

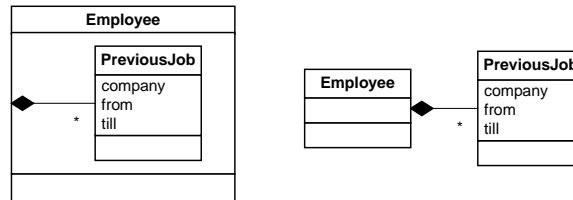


Fig. 1. UML notations available for object composition

Another problem is describing the manipulations allowed for particular object's feature. The visibilities (public, protected, private) definitely do not provide the complete description. Abstracting from the declarations available in current popular programming languages, we find the need for specifying the applicability for the following generic operations (as supported by the previously presented view mechanism): updating, dereferencing¹, inserting and deleting (if a feature's multiplicity allows). The UML metamodel already defines a meta-attribute of similar purpose. This is the *changeability* attribute defined in the *StructuralFeature* metaclass. However its allowed values are enumerated as: *changeable*, *frozen* and *addOnly*, which does not cover all the cases ($2^4=16$) resulting from the presence or absence of a given generic operation. Simply extending this enumeration to 16 values would be rather impractical. Instead, four independent boolean attributes seem to be more suitable.

Although four generic operations appear in the context of virtual objects, it seems reasonable to apply analogous constraints (marking a given feature e.g. read-only, removable etc.) also to regular (concrete) objects, to preserve view transparency.

3.2 Describing Derived Data

The well-known UML “derived” symbol (represented by “/” character) allows to mark any model element as derived from other element or elements and thus serving redundant data. In practice, as suggested e.g. by the *UML Notation Guide* (part of the official specification [11]), the symbol is applicable to attributes and association ends, to indicate that their contents can be computed from other data. This feature is suitable to mark the database features served by object views. On the other hand, due to genericity of the notion, the way of specifying the source features of a given derived attribute or association is not precise, and requires additional comments to be associated. Taking into account the importance of data source traceability in virtual views modeling, we suggest introducing a kind of dependency relationship dedicated for indicating the source data in a more detailed form of class diagrams.

¹ This could be named “reading” as it is intended to return a value representing a given object. However, we chose “dereferencing” to note that even without this operation provided, it is still possible to navigate into a given object (if it is a complex object). For details see [9, 12].

4 Proposed Extensions to the Modeling Notions

4.1 Modeling Virtual Objects

Database Global Objects. It is necessary to decide how the top-level database features² should be shown in class diagrams. In UML, *structural features* designate, where class instances may occur. Thus, relying on the *extent* notion can be avoided.

Following this style would require introduction of e.g. *Database* pseudo-class, in order to “anchor” the global object declarations (as shown in Fig. 2a). In typical cases (but not all) it can be perceived as overly formalistic: global object declarations are the only place where the instances of their classes occur, names of those global objects (*instance names*) are usually fixed by classes. Fig. 2b shows a less formal notation for those situations. The presence of *Employee* concrete objects and *Clerk* virtual objects in database’s global scope is assumed implicitly.³ Also the “/” sign marking the derived feature has been moved from the feature name to the class name compartment. The same can be done with changeability symbols discussed later.

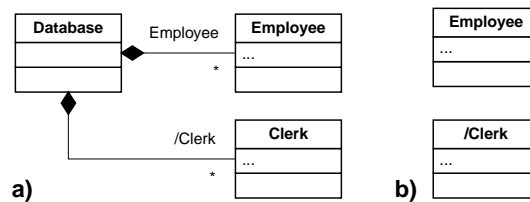


Fig. 2. Traditional (a) and simplified (b) notation for top-level database features

Objects Interfaces. Externally visible features (interfaces) of virtual objects require constructs for proper:

- Distinction of the composition of nested objects and references among objects.
- Marking the derived (virtual) features.
- Showing the changeability allowed for particular (derived or concrete) objects.

The first two problems can be solved with the standard UML notation, provided that there is an agreement on the semantics of the composition relationship. However, changeability flags would need the following symbols (see Fig. 3):

- *isUpdateable* – represented by the exclamation mark (“!”);
- *isDereferencable* – represented by the question mark (“?”);
- *isRemovable* – represented by the caret mark (“^”);
- *isInsertable* – represented by the “greater than” mark (“>”);

The symbols can appear before a feature name (or before a class name in the simplified syntax suggested in Fig. 3). The changeability symbols are shown within the curly brackets in order to allow suppressing changeabilities (by showing no brackets, to distinguish from declaring a feature with none of the changeabilities allowed).

² Usually objects from which we start navigation in queries.

³ Similarly for the lower levels of object hierarchy, that is, for nested objects, we tend to suppress their composition role name, showing only their class name.

Dependency Illustration. For the most detailed diagrams, the notation presented above can be accompanied with the view dependency symbols, based on the generic UML dependency relationship and using the same graphical notation (labeled «view dependency» if necessary). Notice that for pragmatic reasons we simplify the notation. Although the view dependencies span between structural features (as shown in Fig. 3), the dependency arrows are drawn rather between their classes. In contrast to the regular dependency arrow, view dependency can additionally indicate (using keywords within curly brackets, as shown in Fig. 3), the selectivity and aggregation property (*selection* and *aggregation* keywords respectively). To indicate that particular complex view (that is, a view containing other views) preserves the structure of its source object (mapping the features of the latter), we use the *stem* keyword in the properties representing sub-views. Section 4.2 shows and explains notions introduced in the metamodel.

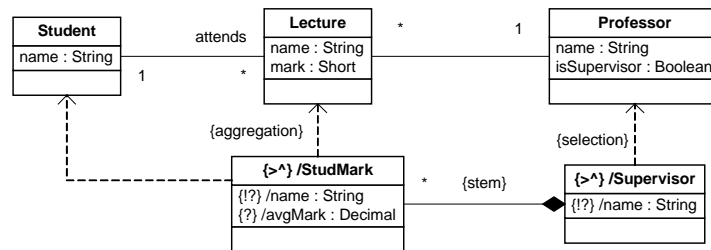


Fig. 3. Exemplary complex view with its data dependency specifications. Assume the data is restructured according to the needs of some external system (e.g. a statistical analysis subsystem), which should not have any access to the identities of the students. *StudMark* and *Supervisor* show also the changeability notation. *Selection* means, that to provide *Supervisor* virtual objects only certain *Professor* source objects are selected. *Aggregation* indicates that a number of *Lecture* objects is used to create a single *StudMark* object. *Stem* label indicates preserving the structure (and dependency) of source objects. Here *StudMark* depends on *Supervisor* as *Lecture* is connected to *Professor*.

Data Integration. Recently one of the most important tasks of object views is data integration. Fig. 4 presents how example integration can be modeled within our notation. Let us assume the following case. Data about students is distributed among three locations: Warsaw, Gdansk, Cracow, and Radom. All students are identified by their IDs. In Warsaw some personal data: students' names and addresses are kept; in Radom – information about students' scholarships; whereas in Gdansk the information about their supervisors is stored. Additionally, we need to incorporate complete data of other students, provided from Cracow. We would like to gather all these information and present them as if they were located in one place.

Merge and *join* labels show relationship between dependencies rather than relationships between virtual and concrete objects. This is another field in which UML must be extended. Clearly, while presenting integration of data from multiple sites one can also use the discussed earlier properties of view dependencies like *aggregation*, or *selection*.

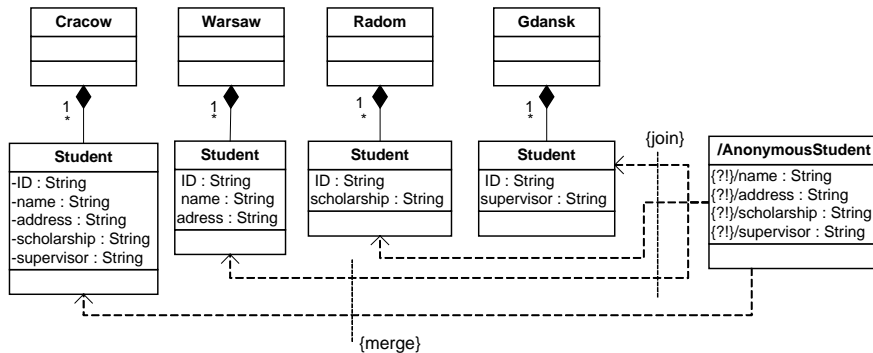


Fig. 4. Integration of distributed data

4.2 Extending the UML Metamodel

In this subsection, we present an extended UML metamodel provided with the features necessary to describe view definitions. The nature of the proposed extension (universal applicability of derived features) seems to justify a modification of the core metamodel.

In contrast to the programming languages, where the class declarations remain independent on their instances (e.g. particular variable declarations), database class (or interface) are often related to particular extent and may determine its name used when referring to the object of that class. In other words, there could be practically one-to-one relation between a class and the place where its instances occur. However, to better align with the UML style and not to limit the object model flexibility, we decided to locate the view-related notions within the feature definition rather than within a class. All the relevant features are shown in Fig. 5.

Modifications of the UML metamodel. The only change into existing UML notions is the replacement of the *changeability* attribute from the *StructuralFeature* class. As explained in Section 4.1, it would be also possible to keep this attribute and only extend the enumeration of values allowed for it. However, with total number of 16 possible changeabilities, we suggest introducing four boolean attributes as a more intuitive solution. As already explained, we use the following names: *isUpdateable*, *isRemovable*, *isInsertable* and *isDereferencable*.

Additions to the UML metamodel. We assume that those structural features, which possess (standard-defined) tagged value “derived” represent virtual objects⁴ and may therefore be the subject of data dependency specifications.

The dependencies point other features to indicate that they are used as sources for virtual object represented by particular feature. Although it is not possible to precisely describe visually how a given virtual object is computed, some information can be easily provided concerning the characteristics of a view dependencies and relations between them, which are in fact data integration patterns.

⁴ In our approach we currently deal only with virtual (not materialized) object views. Thus, a feature marked as derived is assumed to provide virtual objects. A more general approach would require an additional flag to distinguish virtual views from materialized ones.

- **View dependency properties** (mutually orthogonal) modeled by flags in *ViewDependency*: **Selection** (Source data is used to select only the objects meeting a given criteria); **Aggregation** (This property indicates that a given virtual object realizes a many-to-one mapping of the source data).
- **Integration patterns** modeled by binding two or more dependencies (of complex features) with the *formedWith* association link (Fig. 5). We have suggested: **Merge** (integration dealing with horizontally fragmented data) and, **Join** (used with vertically fragmented data (see example in Section 4.1 and Fig. 4).

We also choose to add a *stem* mark to the composition, which makes the dependency graph simpler (as explained in the previous section and Fig. 3). The necessary meta-attribute *isStem* was located within the *Feature*. This is consistent since each nested view belongs to exactly one composition.

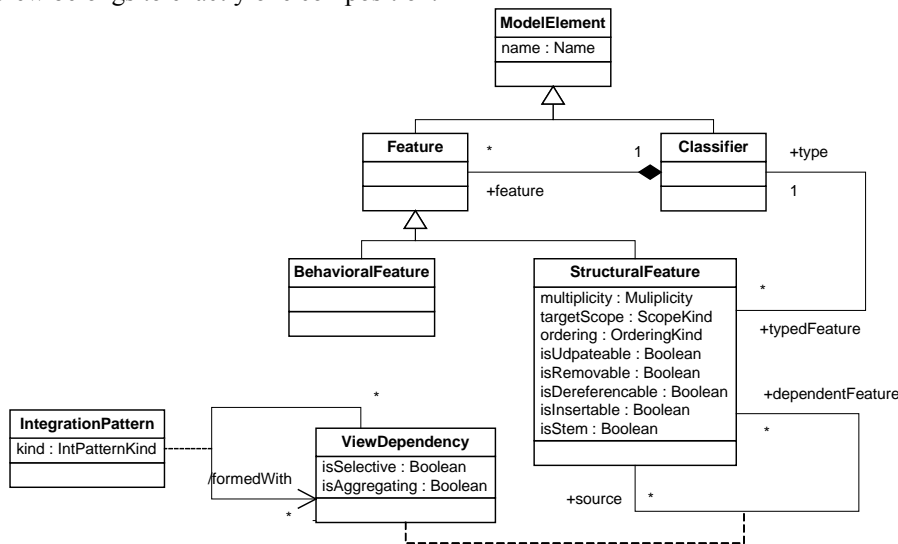


Fig. 5. Fragment of the UML metamodel extended with the notions supporting updateable object views

The proposed dependency properties are not exhaustive, as it is not possible to cover with such notions the whole expressiveness of even the most typical queries that may be used as view definitions. However, it provides some hint concerning the intent of a given view, with the level of detail that is feasible to show on a diagram.

Integration description usually requires choosing an integration key to join the fragmented data. Although it would be possible to specify such key without further extending the metamodel (by introducing yet another *IntegrationPattern* kind), we do not describe it due to the inherent detail limitation of such diagram. Complete specification of view definitions the query language statements are inexplicable.

Note that the information stored using the abovementioned metamodel extensions is very detailed as it may indicate the origin of every elementary data item of a virtual object. However, it is hardly feasible and rather impractical to show such a detailed dependency network on a diagram. Thus, we assume that in most cases, the lowest level of view definitions (that is, the level providing primitive objects) would be “col-

lapsed” using the UML’s attribute notation and in consequence, the dependencies on this level would not be shown.

5 View Modeling in a Development Process

View modeling in our approach can be a subject of different modeling perspectives commonly used by practitioners during software development [4]:

- Basic level (analysis) perspective shows only general idea of data reorganization by description of resulting virtual object properties via interface feature. In this stage of system development, too many details could obscure important ideas.
- In design perspective the number of details may be adjusted by a modeler accordingly to specific needs. Data dependencies can be shown, including indication of some typical transformation kinds applied and changeability flags.

As already stressed, the main concern of this research is modeling data integration. This field of modern system design is not satisfactorily covered by existing techniques. This situation may lead to problems during development and could cause delays or unpredicted complications and thus additional costs. Extended modeling constructs based on the presented metamodel, give advantages for data integration efforts.

Data modeling and software change management. By explicitly documenting dependencies between objects analysts outline required data transformation. Specification of view dependencies may help not only in early estimations of a view complexity, but also in predictions of database change impact. Virtual objects are treated exactly in the same way as normal objects – following the object relativism principle clarifies semantics of a modeling language.

Integration modeling. Dependency links between objects help recognizing necessary data transfer if objects are distributed. Design diagrams uniformly describe data and data integration paths (plus relationships between sources), thus more completely document the system. Extended information about virtual objects supports reflection and may be dynamically used in more advanced applications. For example, design diagrams may generate the templates for view’s implementation.

Integration verification. Extended view dependency links also help in verification of view implementation completeness. Using uniform modeling notions described in the metamodel helps tracking user requirements through all the development stages.

6 Conclusions and Future Work

In our opinion, the current lack of adequate notation for view modeling in UML is a serious drawback. Therefore, in this paper we proposed an extension of UML that supports view modeling. The presented notation seems to be consistent and well fitted into the UML metamodel. It may be a useful notation that supports view modeling at analysis and design stages of application development.

Presented notation allows to model integration of object-oriented or XML data. It supports modeling of (possibly nested) views and allows to describe relationships between stored and virtual data. Some limitations come from the style UML uses for

representing classes and their features, especially in case of nested objects. We tried to propose an optimum solution, providing a valuable description without introducing significant changes into the UML. The notation supports description of access rights to virtual objects at any level of view's hierarchy. The presented notation can be also useful for automatic generation of skeletons of views.

Taking into account that Grid applications are recently in focus of researchers all over the world, we claim that UML should support modeling in this area. Therefore, our future works include development of methodology for data-intensive Grid development that is based on our UML view notation.

References

1. Scott W. Ambler: Agile Database Techniques. Effective Strategies for the Agile Software Developer. John Wiley & Sons 2003
2. Z. Bellahsene: Extending a View Mechanism to Support Schema Evolution in Federated Database Systems. Proc. of DEXA 1997, 573-582
3. R. Cattell, D. Barry. (eds.) The Object Data Standard: ODMG 3.0. Morgan Kaufmann, 2000
4. M. Fowler. UML Distilled, Addison-Wesley Pub Co, ISBN: 0321193687
5. W. Heijenga. View definition in OODBS without queries: a concept to support schema-like views. In Doct. Cons. 2nd Intl. Baltic Wg on Databases and Information Systems, Tallinn (Estonia), 1996.
6. K. Kaczmarek, P. Habela, K. Subieta. Metadata in a Data Grid Construction. Proc. of the 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), Modena, Italy, 2004
7. H. Kozankiewicz, J. Leszczyński, K. Subieta. Updateable XML Views. Proc. of ADBIS'03, Springer LNCS 2798, 2003, 385-399
8. H. Kozankiewicz, K. Stencel, K. Subieta. Integration of Heterogeneous Resources through Updatable Views. Proc. of the 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), Italy, 2004
9. H. Kozankiewicz and K. Subieta. SBQL Views – Prototype of Updateable Views. Local Proc. of ADBIS'04, Budapest, Hungary, 2004
10. E. Naiburg, R. A. Maksimchuk. UML for Database Design. Addison-Wesley, 2001
11. Object Management Group: Unified Modeling Language (UML) Specification. Version 1.5, March 2003 [<http://www.omg.org>].
12. K. Subieta. Theory and Construction of Object-Oriented Query Languages. Editors of the Polish-Japanese Institute of Information Technology, 2004, ISBN: 83-89244-28-4, pp. 522.
13. K. Subieta, C. Beerli, F. Matthes, and J. W. Schmidt. A Stack Based Approach to Query Languages. Proc. of 2nd Intl. East-West Database Workshop, Klagenfurt, Austria, September 1994, Springer Workshops in Computing, 1995.
14. K. Subieta, Y. Kambayashi, and J. Leszczyński. Procedures in Object-Oriented Query Languages. Proc. of 21-st VLDB Conf., 182-193, 1995