1. Short version of the deliverable

CONTENT

1.1 Introduction

Time is everywhere in the information systems. Even if time is required for every application, the use and the need related to time are different. We identified five classes of time dependent applications :

Temporal database correspond to applications including data and business rules strongly dependent on time. Time is linear and discrete.
Temporal reasoning applications require reasoning on events (often fuzzy events) and their possible temporal ordering. Temporal reasoning is based on intervals and not on periods (where time is not anchored to time line). A particular use of temporal reasoning is Automatic Planning.
Branching time applications: classically time is considered in a linear manner, but some applications deal with activities done in parallel with different time and/or several calendars (versioning in design database, events in parallel systems etc.).
Real-time systems applications are focusing on the duration, the priority and the synchronization of tasks.
Time series management: time series are defined by a set of vectors for different time periods. A vector is a set of values for a time period . In general, time series are based on periodicity requiring a user-defined granularity or a user-defined calendar. This management requires special functionalities on the data management and time management.
Conventional databases are designed to store and retrieve current data. When update operations, data relative to evolving real situations, are erased and replaced by their new values. It means that past data are no longer in the database, only current data are kept. Except if the designer of the application has designed their historical storage.
The importance of time in databases has been revealed from 1975. From this date, applications required more and more to manage past, present and future data. This justifies to add to DBMS, mechanisms dedicated to model, retrieve, store and modify data dependent on time. This trend is confirmed by the introduction of SQL/T (temporal extension of SQL) in SQL3. This new generation of temporal DBMS integrates time into the data model in order to facilitate the management of temporal aspects in applications. Such DBMS have simplified the storage and the request of temporal data. But, they don't solve design problems such as : definition of the data schema, specification of the activities of the application, verification of the coherence of the temporal database and also how to administrate such a database. The temporal extension of the OO method is focusing on these design problems relative to *temporal databases*.
The document is a short version of the Methodology deliverable. It is composed of 4 sections :

- a state of the art section : it gives a synthesis of the comparison of six existing OO methods in terms of ways of modeling and ways of working.
- the basic OO method (OOM) section introduces the basic concepts of the TOOBIS method.
- the section on the temporal extension (TOOM) introduces the customized concepts of the TOOBIS method to deal with the temporal aspects.
- the Unified Method Language (UML) compliance section. It shows the compatibility of the TOOBIS method with UML.

1.2 State of the Art

Object-oriented analysis and design (OOA&D) has raised exceptional interest both in practice and academics. However the state of art in OOA&D is extremely diffuse and it is becoming increasingly difficult to follow the developments in the field. Books introducing new methodologies OMT [Rumbaugh91], OOSE [Jacobson93], OOA&D [Martin92], OOAD [Shlaer88,

92], OOA/OOD [Coad 90, 91] and O* [Lee94, Brunet93, Rolland96] are being published at a tremendous speed and CASE tools and environments are being marketed with an accelerating pace. The methods are somewhat immature both theoretically and empirically. Their theoretical grounds are poorly documented, many of them just push upwards concepts of programming languages and combine object-oriented concepts with more traditional information systems and software engineering concepts such as E/R modeling and data flow modeling. Their practical experiments are very limited and most of them are subject to considerable evolution stimulated by both marketing considerations and practical experience.
This section is an evaluation of the six selected methods. The analysis task has been driven by the requirements of the GlaxoWellcome and Delta applications. It leads to the proposal of a method engineering approach in which the TOOBIS method will be a modular method constructed by assembling fragments of various methods. Based on the analysis of the strengths and weaknesses of methods we also make suggestions for the TOOBIS method.

There exist many different definitions of a method [Brikkemper90], [Lyytinen89], [Wynekoop93]. However most of them converge to the fact that every IS Development Method (ISDM) combines two aspects, namely the product part and the process part. Following [Olle91] we define an ISDM as composed of two parts : a product model and a process model.

1.2.1 Product model evaluation

The product model is composed of a set of concepts and rules to manipulate them. The process model usually prescribes the steps to be followed and the activities to be performed in order to develop the information system. Therefore, a method provides the concepts to describe the product and the methodological guidelines to produce a quality product with a reasonable efficiency. As said in [Olle91], the product is the desired target of an IS development whereas the process is the route to be followed to reach the target. Based on this assumption the framework employs two reference models : one to evaluate the product model of an ISDM and a second one to analyze its process model. These reference models are presented in details in section 2.2 of the long deliverable.
Each selected method has been analyzed according to the two references models and presented with three tables (detailed analysis is provided in section 2.3 of the long deliverable). The analysis of the product part is based upon two assumptions :
first, an information system is inherently a dynamic system including three aspects or perspectives in [Olle91] : structure (data), functions (process) and behavior (control).
second, as a model of the Universe of Discourse (UoD) [Griethuysen82], an information system includes three levels :

- the model of the UoD, which defines the parts of the reality which have to be abstracted in the IS (we will refer to this level as the organizational level),
- the model for the information about the UoD i.e. the conceptual level,
- the model of the technical information system (we will refer to this level as the technical level).

The comparison of the various concepts and notations shows that all methods include the three types of abstractions mentioned in the product reference model : namely, structure, function and behavior abstractions. There is a consensus among the six methods on the structure abstraction. Except for the class concept which originates from OO approaches, almost all the other notions are adaptations of the E/R model concepts. The notions of class, aggregation, association and inheritance are the preeminent concepts. They provide means to structure complex objects, classify them into hierarchies and relate one to another. However significant differences in the way of modeling the system functions and behavior appear among these six methods. These differences are summarized in Table 1 : Different approaches of system functions & behavior.

Integration of the three abstractions seems to show a number of significant advantages: minimality of the set of concepts, orthogonality of concepts, ease of understanding and use, better and quicker validation of specifications. We propose to follow this line in TOOBIS.

The levels (organizational, analysis, design) are inequally treated in the methods, but we can summarize the analysis by the three points of Table 2.

	Behavior abstraction			Function abstraction	
approaches	**event-driven**	**state-transition**	**message passing**	**Data-Flow**	**Encapsulation**
definition	the behavior is described in terms of events and triggering rules. An event is a particular state change of objects or IS environment which plays the role of a stimulus implying the system reaction. The reaction of the system to an event is expressed through triggering rules in terms of operation on objects.	The traditional state transition diagram can express the behavior of an object of a class. It exhibits the possible states of an object in the class and the allowed transitions between these states.	The behavior is modeled in terms of message sent from an object A to an object B.	classical Channel Agency Diagram for modeling the system function.	the OO paradigm is followed : functions are operations defined in the class scheme.
strengths	-global and local study of system behavior, -tighly coupling of abstractions (minimality of concepts) - when, why and how are expressed	-local study of system behavior, -*when*, *how* are expressed	-Only *how* is expressed -integration of the structure and the behavior abstractions		-Integration of the structure and the function abstractions (as it is in the OO programming languages) - OO paradigm is respected.
weaknesses		-No global study of system behavior, -No integration between abstractions	-It is a programming technique, not a conceptual modeling technique. -No explanation about *when* and *why*	-No integration between the abstractions -Encapsulation principle is not followed.	

Table 1 : Different approaches of system functions & behavior.

analysis -> design		organizational level	traceability of specifications
The ***expansion*** mode is applicable when the two levels are based on the same modeling paradigm; for example the Object Oriented paradigm. Design decisions based on technical requirements are made to augment the conceptual product with design features which were not relevant at the conceptual	The ***transformation*** mode assumes a different model at each level. The analysis product needs, therefore, to be transformed into a valid design representation. This mode is required when the two levels are based on different paradigms; for instance a conceptual OO schema and a relational database schema.	Three methods deal with organizational modeling : OMT, OOSE and OOA&D. Use Cases, scenarios and Object Flow diagrams are respectively used by OOSE, OMT and OOA&D to identify the relevant events and activities of the application.	Traceability is recognized nowadays as a key factor of success for handling change in the requirements of a system in an efficient manner. Product traceability can be handled in an ISDM method in a rather simple manner; by keeping track of links between product elements belonging to different levels; for example between an *actor class* at the analysis level and the corresponding C++ *class* at the design level. Another approach

level.			consists in tracing the analysis and design process itself to record the design decisions and their rationale.

Table 2 : Inter-level integration.

1.2.2 Process model evaluation

The evaluation of the process dimension of methods are focusing on three points :
the process modeling approaches,
the steps and heuristical knowledge,
the meta-process i.e. the process of constructing process models.

- Process modeling approach

The analysis of the process part of methods shows that almost all methods (all except O*) follow an activity-oriented process modeling approach. Limitations of this modeling paradigm have been acknowledged and discussed in many papers. Despite its limitations, the use of this paradigm, is rather universal in classical methods. It is de facto the approach used, not by choice but more because of the lack of focus that methods put until recently on the process dimension. Things are changing and the focus is shifting from product modeling to process modeling. O* benefits from this shift and is associated to a decision-oriented approach which is semantically more complete and better adapted to the early stages of system development e.g. requirements engineering and system analysis.

- Steps and Heuristical knowledge

The process model of methods is restricted to a list of ordered steps which are informally described using natural language phrases. They constitute no more than a road map to proceed globally at each level. The granularity of the process steps is too large and descriptions are not detailed enough to provide an explicit guidance. Consequently the analyst has to discover by himself how to proceed in detail. The O* process model is able to handle different levels of granularity with an homogeneous way of describing detailed decisions and large grained decisions. In addition it offers more flexibility because there is not one predefined order in which decisions have to be made but different possible ones. The exact order is chosen dynamically when the process proceeds.

Heuristics are useful and supportive for the analysts especially for novices. However we found that their description in the five methods other than O* was first, spread over the books in various chapters and not necessarily in the chapters dealing with the process model descriptions. Second the heuristics are expressed in various forms : they can be examples only, natural language explanations, very often ambiguous or stories. Third, guidelines do not cover all aspects of the process. For example the Object Flow Diagram of OOA&D is not treated in the process part. Fourth, most often the guidelines are not globally consistent. We reached this conclusion when we gathered all heuristics in one single list and tried to organize the list to group together rules dealing with the same subject.

Current methods seem to be completely disorganized in their way of presenting methodological guidelines. We are in front of a kind of mess which does not really help the analysts who have to work with these methods. Finally, there is no possible automatic support of these methods on the process side. It is obvious that a software tool cannot guide a process with a disorganized list of informal, ambiguous and conflicting guidelines.

As a result of tackling process modeling in the similar way product modeling is tackled, O* uses the same precise formalism for describing global steps and detailed heuristics. In addition, these two aspects are related to each other by refining or decomposing higher decisions to more detailed ones. O* provides, in one hand, a process description guiding analysts from the beginning of their tasks until the end. In the other hand, a single process modeling formalism accommodates a wide range of process granularity in an homogeneous fashion. Moreover, based on the process model description, analysts can be supported, guided by a CASE environment during the performance of the process.

- Meta-process

Traditional information systems methodologies were defined in an "ad-hoc" manner. This has a number of disadvantages among which the fact that the process model is depending of the experience of his designers.

The Information Systems Engineering (ISE) community has been recently paying attention to this problem adopting a method engineering approach [Lyytinen89], [Wynehoop93], [Brinkkemper90] in which the process model is considered as the product of an engineering process. The SOCRATES project [Hofstede90], [Wijers91] and the CAME (Computer Aided Methodology Engineering) approach [Martiin94] are early and promising works dealing with an automated support for creating situation specific process models.

1.2.3 Proposed features for the TOOBIS method

We sum up in this paragraph the proposed features for the TOOBIS method. The analysis of OO methods emphasizes the fact that no method has included specific concept for handling temporal aspects of Information Systems. However, we can sketch the features adopted for the TOOBIS method. Table 3 summarizes features of the product model which will be integrated in the reference manual of TOOBIS method. Table 4 shows features of the process model which will be the concern of the TOOBIS method handbook.

characteristics	proposed features
structure abstraction	concepts of object class, properties, association, composition and inheritance links following the OMT terminology but with a formal semantics borrowed from O*.
behavior abstraction	event driven approach which seems semantically more powerful and particularly adapted to temporal applications.
function abstraction	encapsulation principle : functions are integrated in class definition
intra-level abstraction	one model single model for the analysis level. i.e. full integration of the three types of abstractions : structure, behavior and function abstractions.
inter-level abstraction process	expansion mode i.e the design specification results of the augmentation of the conceptual definition of classes
inter-level abstraction : traceability	the O* approach : the traceability mechanism is provided through process traces.
organizational level	a "scenario" based approach to initiate the analysis process

Table 3 : features of the TOOBIS method.

characteristics	proposed features
process modeling approach	the decision-oriented approach borrowed from O* to provide a flexible process guidance advising on which decision to make in the situation at hand.
process model formalism	formal process description language allowing a modular description of the TOOBIS way-of-working.
heuristical knowledge & granularity	the O* approach in order to provide : - a modular process description at different levels of detail - a complete and well organized set of heuristics, - traceability of design decisions

Table 4 : features of the process modeling approach.

1.3 Overview of the OO method (OOM)

The basic OO method is composed of one model dealing with the three abstractions : structure, function and behavior. The main concepts of this model, that are used to formalize real world phenomena, together with their underlying relationships are shown in Figure 1.

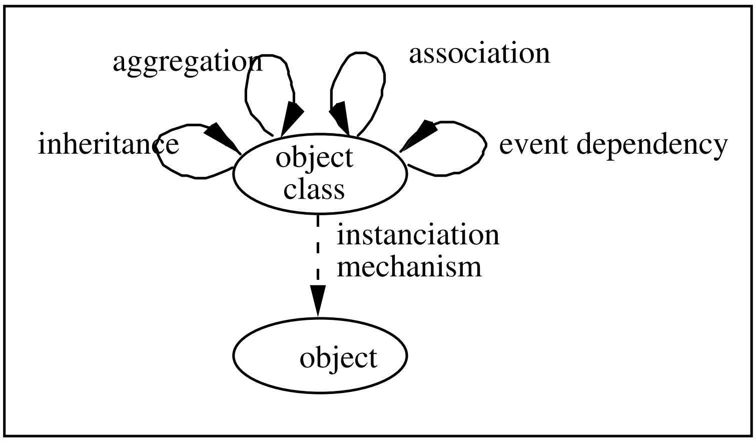

Figure 1: Main concepts of the OO Model.

1.3.1 Object and Object class

Two basic concepts of the model, namely object and object class, are adopted from the object-oriented paradigm. The main aim of the model is to adopt the concept of object from the analysis phase right through the implementation phase.

Objects represent real phenomena of the application. For instance, the patient Smith, the GlaxoWellcome company, the truck number 4477 GB 75. are objects.
Object identity provides a means for differentiating an object from other objects in the same class. It implies two points :

- existence of an object is independent from the value that characterizes the object; and
- equality of two objects in terms of values does not mean equality of the objects.

The object identity is intrinsic to the object and it is insignificant to the user's point of view. Though the identity of an object is invariant across time, the object evolves in terms of its value with respect to time.
When we want to precise and refer exactly one thing, we will use the phrase object. We will use the phrase class to refer to a group of similar things.
The classification mechanism allows objects of the same structural and behavioral characteristics to be classified into object classes.
For instance, the objects patient Martin and patient Smith are classified into the object class PATIENT , which describes the structural and behavioral characteristics of potential patients.
An object class describes and groups a set of objects of the same nature in terms of the structural and behavioral characteristics of the objects. It is identified by a unique name and has an extent consisting of the set of objects at a given time.
An object class has a type that is defined by the description of the structural and behavioral characteristics of its objects.

1.3.2 Structural properties

Moreover, the model also takes into account the life cycles of objects for the determination of structural links (static links) between object classes, besides the inheritance links of the object-oriented paradigm. Other than the inheritance links, two types of structural links can be distinguished: **associations** and **aggregations**, where each link can be defined between two object classes after identifying the dependency between the life cycles of objects in these classes. The methodological guidelines attached to the concept of aggregation and association are provided in the long deliverable (section 3.3.1.5).

Figure 2 shows that *an order is an aggregation of several orderlines and an order is associated to a customer*.

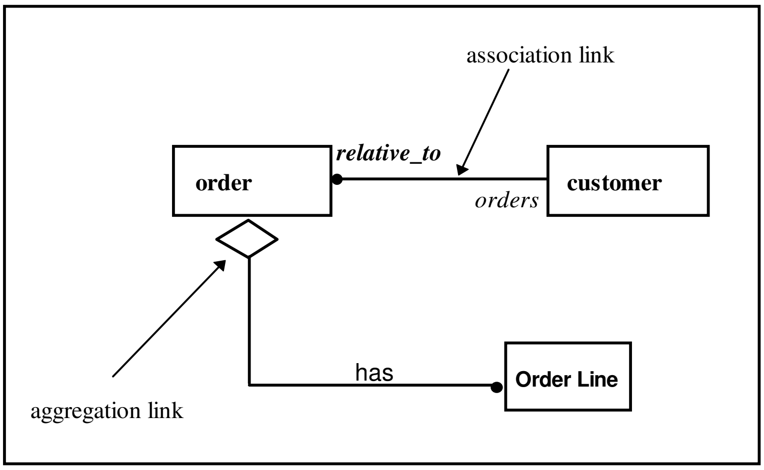

Figure 2 : Association & aggregation links.

An attribute is a characteristic of a class. The value set of an attribute is defined with the notion of domain. For instance, name, address are examples of attributes of the Customer class. STRING is the domain of the name attribute whereas ADDRESS is a structured domain associated to the address attribute.

```
                Customer
properties
    name : STRING
    address : ADDRESS

domain ADRESS    [ no : INTEGER,
                   street : STRING,
                   city : STRING,
                   zipcode : STRING ]
```

Figure 3 : Examples of attribute.

An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have identity. For example, all occurrences of the integer 70 are indistinguishable, as are all occurrences of the string Smith.
Due to the fact that each object has its own identity, explicit object identifiers are not required. But notice that internal identifiers are purely an implementation convenience and have no meaning in the problem domain.
Structural description of an object class is completed by a set of constraints. **Constraints** are inheritance constraints, attributes constraints and uniqueness constraints (cf. long deliverable section 3.3.1.7).

- Partition, covering and disjunction constraints are inheritance constraints applied among sub-classes of the same super class.

- An attribute constraint is a property assertion that restricts either the attribute values of an object, or the cardinalities of the attributes defining related objects or domains according to a certain criterion. It is expressed as a boolean assertion. For instance, the invariant stating that *the weightdate of a patient must be greater than the birth date* is specified in the object class *Patient.*
- Uniqueness constraint is expressed between a set of attributes and a collection of objects. For instance, *the order number is unique in the collection of Orders*.

An object class is implicitly defined by an *extent* that stores objects representing the real world phenomena of the same nature. The extents of all the object classes form the persistent information of the IS. By default, an object of an object class refers to an object from the extent of the object class, which in turn, represents a persistent object of the IS. For specification purposes, the extent of object class is alternatively referred to as *persistent collection*, or more precisely, *persistent set*.

1.3.3 Behavioral properties

Other than the structural aspects, the model also considers the concepts of ***event*** and ***operation*** to take into account the behavioral aspects of ISs at the conceptual level. The causal dependency of events is related to the cause and effect in the occurrence of events. It can be explicit by the fact that the occurrence of an event is due to the previous occurrence(s) of some event(s).

Three groups of events are distinguished: *external events*, *internal events* and *temporal events*. External events correspond to the events occurring in the environment outside the IS. Internal events correspond to the internal state changes, or rather, the system responses to external changes. For instance, *out of stock* is an internal event detected when *an object of the stock class becomes less than 10*. Temporal events that depend on the description of time, correspond to time occurrences. For instance, *each day all the unpaid orders are refused*. External events interact with the IS through messages initiated by an actor of an *actor class.* For instance, *arrival of an order* is an external event coming from a *customer*.

Each external event is attached to an actor class, while all temporal events are attached to the *calendar class*. Internal events are attached to object classes. Each event is defined by an *event predicate* that indicates when the event occurs. The calendar class is simulated by the system's internal clock.

The concept of event is derived from the REMORA approach [Rolland82]. In this approach, propagation of state changes in the IS by basic operations is normally initiated by internal events as a result of the interception of an external event occurred in the environment outside the IS. Besides this, state changes can also be initiated by the occurrence of a temporal event. The concept of external event provides a more powerful way of communication than message passing mechanism by providing a mean for explaining when a certain phenomenon occurs. In addition, the IS can react to an event by sending a message to an actor via an *external operation* encapsulated in the actor class definition.

Operations are defined in object classes. They are classified under *basic operations* and *queries.* A basic operation defined in an object class represents an action that will act on an object of this class, as a result, causing a state change to that object. The execution of *basic operations* is triggered by events. Queries merely provide access services to other objects. Derived attributes are realized as queries without input parameter.

Triggering conditions and/or triggering factors can be defined in the operation triggering of a trigger. They will be activated when the event occurs. A triggering condition is the condition that must be verified for activating a basic operation, while a triggering factor evaluates the set of objects on which the basic operation will be executed.

Figure 4 summarizes the integration of the different abstractions (structure, function & behavior).

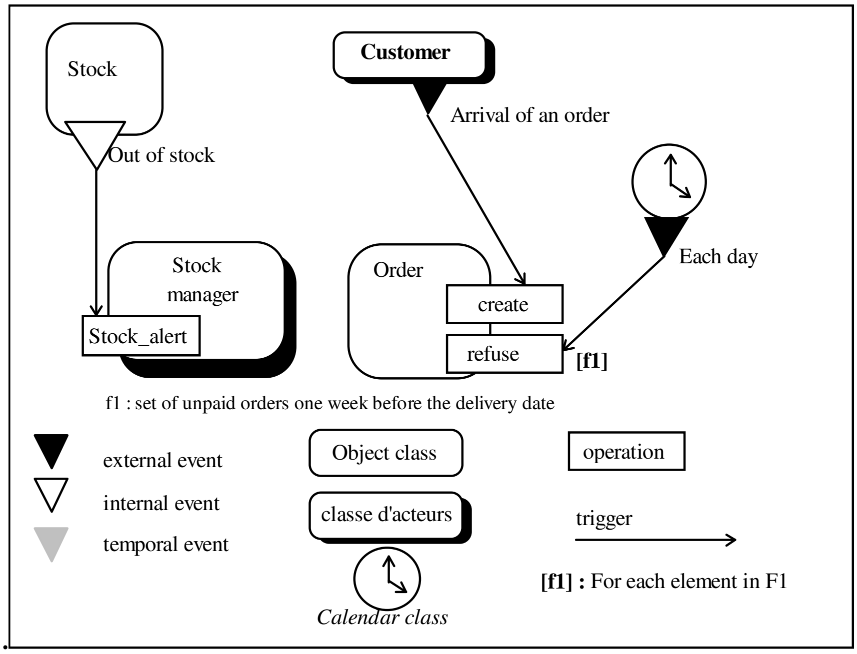

Figure 4 : Example of dynamic schema

A dynamic transition [Rolland82] is made up of an event, a set of basic operations triggered by the event and a set of persistent objects manipulated by these operations. A dynamic transition is analogous to an elementary transaction of databases, that is, a unit of atomicity, consistency and integrity. In a wider perspective, a dynamic transition may be defined globally and explicitly by the sequence of event occurrences caused by the occurrence of an external or a temporal event. The execution of a dynamic transition must lead the database from a coherent state to a coherent state.
In a dynamic transition, the basic operations that manipulate persistent objects are triggered in parallel by the event. Therefore, any creation, update or deletion of persistent objects within the dynamic transition must be completely done before any other process can be carried out by/on any of these persistent objects. In this sense, a dynamic transition can easily be mapped into a run-time elementary transaction.

Concurrence and induction between events are advanced features treated in two graphs : event induction graph and event concurrence graph. These two graphs are introduced in section 3.3.3.3 & 3.3.3.4 of the long deliverable.

1.3.4 Views

The analyst can use during the specification phase three views :

- global structure view,
- global behavior view,
- local class view.

The global structure view gives to the analyst the general map of the persistent information of the system. The graphical notation is introduced by Figure 2.

The global behavior graphs describe the dynamic of the application in terms of events, triggers and operations embedded in classes such as object classes, actor classes and calendar classes. This global behavior graph allows to see how an event is ascertained and what are the effects of an event. The graphical notation is mentioned in Figure 4.

The local class views describe the static and the dynamic aspects of a class of any kind (object class, actor class and calendar class). The local view of an object class describes its attributes, its constraints, its operations or queries as well as its internal events. The local view of the object class COPY is graphically shown in Figure 5.

COPY
properties reference : STRING purchase date: DATE loan date : DATE state : COPY-STATE loans : inverse (zeromany, LOAN.copy) notices : inverse (zeromany, NOTICE.copy)
constraints unique(Copies,reference)
operations COPY purchase(c:structure(COPY)) reservation() availability() check out() wear out() loss() deletion()
queries boolean exist_copy_available()
internal-events $availability of copy

Figure 5: The local class view of the object class COPY

1.4 Overview of the temporal extension (TOOM)

The temporal extension aims at introducing at the analysis level, the concepts required to manage the temporal aspects of a database application.

In addition information systems become more and more dataware houses which have to manage basic information of the enterprise but also derived information such as statistics, decisional supports , management reports etc. This new trend is taken into account at the analysis level, by providing in the OO method the notion of derived class.

The temporal extension of the OO method described in section 2 of the long deliverable is structured in 8 points :

- definition of one or more calendars for an application,
- extension of basic domains to temporal domains,
- derived class : specialization of object class,
- extension of object classes to time dimensions,
- extension of static links between object classes to time dimensions,
- extension of constraints applied on object classes,
- extension of events to time dimensions,
- database administration policy.

1.4.1 Definition of one or more calendars

A calendar is a metric system to apply on the time line (cf. section 4.4.1.2 of the long deliverable). The OO method proposes the concept of calendar class for defining temporal events of the application (cf. section 2.3.1.4 of the long deliverable). In the basic OO method, the calendar is the Gregorian one. At the object level, the calendar class has only one instance : the clock of the Gregorian calendar.

We extend the definition of the calendar class for defining any calendar useful for the application.

A calendar is described by :

- a name,
- an origin (instant),
- a basic unit (granule),
- a set of granules . Each granule is described by the upward and downward mappings. If the mapping is regular, the mapping is described with a coefficient whereas an irregular mapping is associated to a function name.
- three operations have to be redefined for the calendar :
    - addition of an interval to an instant,
    - addition of an instant to an interval,
    - extraction of an interval between two instants.
- a set of external formats. An external format is a temporal format used by the application which has to be mapped into the defined calendar. They are managed by the application and the mapping functions are explicitly invoked by the application.

The *Gregorian* calendar is described in Figure 6[1]. The chronon of the system is a second. As we can see that month and minute are granules with irregular mappings (cf. section 4.4.1.2 of the long deliverable).

[1] The irregular mapping is denoted by a function : minute_second().
The three temporal types are instant (point on the time line), interval (unanchored quantity of time) and period (quantity of time between two instants).
Instant<Gregorian,hour> means an instant at the *hour* level of the *Gregorian* calendar.
*Instant <Gregorian, *>* means an instant with one of the granularity of the *Gregorian* calendar.
*Interval<Gregorian, *>* means an interval with one of the granularity of the *Gregorian* calendar.

```
Calendar class    Gregorian
origin    "01/01/0001:00:00:00"  basic unit =chronon
granules
second :    {1 basic unit, basic unit=1second}
minute :  {minute_second(), second_minute()}
hour :  {60 minutes, minute=1/60hour}
day :  {24 hours, hour=1/24day}
month :  {month_day(),day_month()}
year :  {12 months, mois=1/12 year}
operations
instant<Gregorian, *> operator+ (instant<Gregorian, *>, interval<Gregorian, *>)
instant<Gregorian, *> operator+ (interval<Gregorian, *>, instant<Gregorian, *>)
interval<Gregorian, *> operator- (instant<Gregorian, *>, instant<Gregorian, *>)
external formats
TimeZone= "instant<Gregorian, hour>, TZ, DST"
              {instant<Gregorian, hour> input(TimeZone),
               TimeZone output(instant<Gregorian, hour>)}
EndCalendarClass
```

Figure 6 : The Gregorian Calendar Definition (defined by the system).

The Gregorian calendar as it is defined in Figure 7 is provided by default.

How to proceed if the application requires extra granules such as week, quarter etc. ?

You have to analyze why the application needs such granules. Two cases are possible :

- the granularity is helpful for giving and retrieving information. No temporal manipulation is done with this granule. Thus, the granule can be viewed as user-defined external formats of the Gregorian calendar. For each granule, you have to define the input and the output functions. Figure 7 shows how to add the external format *week* to the Gregorian Calendar.

```
Calendar class    Gregorian

External formats

week="year/week" {instant<Gregorian, day> input(week),
              week output(instant<Gregorian, day>)}
endCalendarClass
```

Figure 7 : Overloading of the Gregorian Calendar.

- the granularity is used to measure and refer to one instant on the time line. It means that time manipulation and physical storage require this granularity. Thus, a new calendar should be defined. Figure 8 explains how to define new calendars such as *MyWeek* and *MyHalfYear*.

Calendar class MyWeek
origin ="01/0001" , **basic unit** = <Gregorian, day>
granules
week : {day_week(), week_day() }
year : {53 weeks, week=1/year},
operations :
instant<MyWeek,*> operator + (instant<MyWeek,*>, interval<MyWeek,*>)
instant<MyWeek,*> operator + (interval<MyWeek,*>, instant<MyWeek,*>)
interval<MyWeek,*> operator - (instant<MyWeek,*>, instant<MyWeek,*>)
End Class

Calendar class MyHalfYear
origin ="01/0001" , **basic unit**=<Gregorian,month>
granules
half : {6 months, month=1/6half}
year : {2 halfs, half=1/2year},
operations :
instant<MyHalfYear,*> operator + (instant< MyHalfYear,*>,
interval< MyHalfYear,*>)
instant< MyHalfYear,*> operator + (interval < MyHalfYear,*>,
instant< MyHalfYear,*>)
interval < MyHalfYear,*> operator - (instant< MyHalfYear,*>,
instant< MyHalfYear,*>)
End Class

Figure 8 : MyWeek and MyHalfYear calendars.

1.4.2 Extension of the basic domains to temporal domains

Three temporal domains are introduced in section 4.4.1.1 of the long deliverable : instant, period and interval. An instant represents a point on the time line (*1996-12-12*). A period represents the quantity of time between two instants *([1996-12-12, 1996-12-15)*[2]). By contrast an interval is an unanchored quantity of time (*3months*).

These three types are expressed with one granularity of one calendar. Consequently, the granularity and the calendar are their parameters. For instance, an instant expressed at the *week* level of *MyWeek* calendar is described as INSTANT<MyWeek, week>. An interval expressed at the *hour* level of *Gregorian* calendar is defined as follows : INTERVAL<Gregorian, hour>. And finally, PERIOD<Gregorian, day> defines a period at a *day* level of *Gregorian* calendar. Figure 9 shows different temporal attributes.

Temporal attributes :
birthday : INSTANT<Gregorian,day>
charge : INTERVAL <Gregorian, hour>
week : INSTANT <MyWeek, week>
dates : PERIOD <Gregorian, day>

Figure 9 : examples of temporal attributes

[2] [1996/12/12, 1996/12/15) expresses a period including the beginning of the period and excluding the end of the period.

These generic temporal domains *(INSTANT<calendar, granule>, PERIOD<calendar, granule> and INTERVAL<calendar, granule>)* replace the classical types : *date*, *time* and *datetime* found in the classical DBMS. The two domains *INSTANT<calendar, granule> and PERIOD<calendar, granule>* permit to define temporal attributes which are absolute or relative times (cf. section 4.4.1.3 of the long deliverable). These generic domains are specialized in two categories for absolute and relative times respectively.

The domains *INSTANT-A<calendar, granule>, PERIOD-A<calendar, granule>* are used to specify <u>absolute time</u> domains.

The <u>relative time</u> is introduced by two new temporal domains : *INSTANT-R<calendar, granule> and PERIOD-R<calendar, granule>*. Applications require to manage and store relative time for planning reasons. For instance, the end of the project is equal to *"J0+30months"*; *J0* is a temporal mark. In addition, the *task1* of the project should be the relative period *["begin of the project+7days", "begin of the project +2months")*. The relative time appears as a temporal derived attribute. The temporal expression of a relative time is limited to *<temporal mark>* + *<interval>*. Instead of storing the temporal value, the temporal expression used to derive its value is stored. The characteristics of a relative time is to have no value until its temporal mark is valued. The relative time becomes absolute time since the valuation of its temporal mark has occurred. The temporal mark of a relative time attribute is a temporal attribute which can be itself a relative time or an absolute time. In the previous examples, *J0 and begin of the project* must be temporal attributes of the application.
These new temporal domains are used to define the domain value of temporal attributes. Three time dimensions are defined in the temporal DBMS : valid time, transaction time and user-defined time (cf. section 4.4.4.4 of the long deliverable). The valid and the transaction time are managed by the temporal DBMS. The analyst defines explicitly in his specification the temporal attributes belonging to the user-defined times (semantically dependent on the application).

1.4.3 Derived class Vs Basic class : specialization of object class

Information systems become more and more data warehouses or Executive Information Systems (EIS). Such systems may be projected in three distinct spaces [Toobis96a] : data, aggregation and variation spaces. The data space deals with basic information managed by the enterprise. The aggregation space results of aggregate computation on data elements, such as *the amount of sales per vendor and per week*. Finally, the variation space deals with the observation of changes of derived data belonging to the aggregation space.

Executive Information Systems (EIS) include basic information about the enterprise but also derived information such as statistics, decisional supports, management reports etc. The specification of such Information Systems requires to take into account the exploitation of basic information as well as derived information.

The persistent and relevant information of an application is designed in the OO method by the notion of *object class* (cf. section 3.3.1.2 of the long deliverable).

The extension specializes this concept in two classes : *basic class and derived class*. A basic class is an object class describing basic objects of the enterprise (not derivable) or information belonging to the data space. A derived class is an object class derived from one or more object classes. It means that a derived object can be created or updated through its derivation function. A derived class is used to structure the aggregation space.

In Figure 10, the *Order* class is a basic class whereas the *WeeklySale* class is a derived class representing the amount of sales for a vendor in one week. The objects of the class WeeklySale are created from the orders. In the global static diagram, derived and basic classes are graphically represented by a black box with the name of the class and its type : « basic class » or « derived class ».

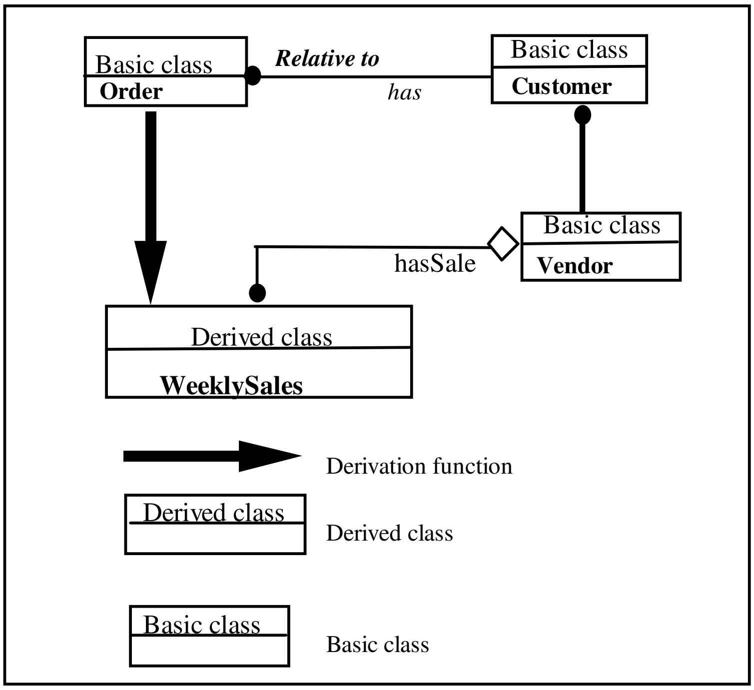

Figure 10 : Examples of basic & derived classes

A *basic class* is defined as an object class (cf. section 3.3.1.2 of the long deliverable). A *derived class* is similar to an object class except it is defined by a set of derivation functions instead of operations.

1.4.4 Extension of object classes to time dimensions

A temporal DBMS is able to manage different kinds of objects : snapshot, historical, rollback and temporal (cf. section 4.4.4.4 of the long deliverable). A snapshot object is managed in non-temporal DBMS (without time management). Historical, rollback and temporal objects are specific to temporal DBMS (with a specific time management). Valid time is managed in historical and temporal objects whereas transaction time is included in rollback and temporal objects.

This section presents how the valid and the transaction times are introduced in object classes either basic or derived. Three sub-classes are introduced *: snapshot class, temporal class and temporal dictionary class*.

- A ***snapshot class*** is a class which does not require time management. The properties of such classes are not related to time. For instance, The *Order class* is a snapshot class because the evolution of its properties is not relevant.

- A ***temporal class*** is a class with time management (cf. historical, rollback and temporal objects). It means the properties are associated to the valid or/and transaction times. The *salary of an employee* is a temporal class because its evolution has to be kept in the database.

- A ***temporal dictionary class*** is a specific class where data are accessed only via the valid time key. For instance, *the inflation-rate of a country* can be defined as a temporal dictionary if each stored value is indexed by its valid time period. The semantic of the value depends on its valid time period.

1.4.4.1 The **snapshot** *class*

The snapshot class is used to model objects which do not require time management. Object classes described in section 3 of the long deliverable are always snapshot classes.

For instance, the *Order* class is a snapshot class because the evolution of its properties is not relevant to keep track of.
A snapshot class can be a basic or a derived class.

- Properties, constraints, methods and events in snapshot class.

  A *property* of a snapshot class can be a relationship to another class or an attribute as it is defined in the object class (cf. section 3.3.1.2 of the long deliverable). The relationships between classes are aggregation, association and inheritance links.
  A *constraint* defined for a snapshot class can be an attribute constraint or an inheritance constraint as it is described in the object class (cf. section 3.3.1.7 of the long deliverable)
  Methods of a snapshot class can be queries or basic operations, and they can be applied at the object or class level. (cf. section 3.3.2 of the long deliverable). These features are defined on the object class concept and are available for the snapshot class. The types of basic operations are create, update, delete or erase. Because of the temporal consistency of the temporal database, the delete operation applied on a snapshot object is a logical deletion. The physical deletion is materialized by the erase operation.
  And finally, *internal events* can be expressed on snapshot classes (see internal events, section 3.3.3.1 of the long deliverable).

1.4.4.2 The **temporal** *class*

The temporal class is a class with time management. It means that the properties are associated to the valid or/and transaction times. The *function of an employee* is a temporal class because its evolution has to be kept in the database.
The temporal class permits to integrate the valid or/and the transaction times management into the class definition. This concept represents the way to define historical, rollback and bi-temporal objects found in the temporal DBMS literature.

The temporal class is characterized by the time dimension to manage. For each time dimension, two possibilities are offered : *last state* or *history*. The possible combinations from the *temporal class* concept are summarized in Table 5. Note that a class with no time dimension to manage is not handled by the *temporal class* concept but by *the snapshot class* concept.

		transaction time		
		none	state	history
valid time	none	no time management (snapshot class)	last database state of an object	database history of an object
	state	last state of an object with its valid time	last bi-temporal state of an object	database history of an object and its last valid state
	history	valid history of an object	valid history of an object and its last database state	bi-temporal history of an object

Table 5 : time management defined in a temporal class.

- The semantic of a temporal class

A temporal class is used to group properties which evolve at the same time and on which the time management required is the same. For instance, the *salary* and the *function* of an *employee* should be two temporal classes if the *salary* and the *function* are not evolving at the same time.

A temporal class describes one temporal aspect of objects identified in a snapshot class. For instance, *the function of an employee* is a temporal aspect of the *employee* class. In the OODBMS, we consider that it exists at least one property which does not change : the identity of the object. Thus, a temporal class is a temporal variation of one snapshot class. By contrast, a snapshot class can have several temporal variations. The relationship of a temporal class to its snapshot class is called « temporal aggregation » (cf. Figure 11). A snapshot class is graphically mentioned with a black box whereas a temporal class is a grey box.

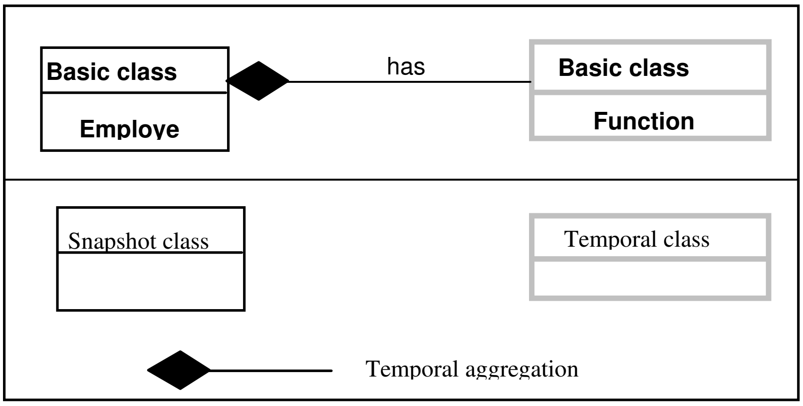

Figure 11 : Temporal class of the class Employee.

- Definition of the valid time (if it is required) :

  Several characteristics for the valid time have to be determined :
    - the nature of the valid time (instant or period),
    - the granularity and the calendar of the valid time,
    - the type of management (state or history),
    - the completeness of the history (complete or partial),
    - the extrapolation function if the history is partial.

- The nature of the valid time can be an instant or a period. For instance, the *weight* of a patient is measured at each visit of a *patient* to his doctor. As his weight can evolve between two visits, its validity is instantaneous.

  The valid time is a period when the application can control the evolution of a property or when the evolution of a property is known « a posteriori ». For instance, the *salary* and the *function* of an *employee* can be defined with a period as valid time because their evolution is controlled by the enterprise. The position of the « enemy » unit in the military system is updated « a posteriori ». We know that the *Panzer* unit was inside the *town hall* during the period *[1940/8/4 12:56:30, 1940/8/5 10 :25 :30).*
- The granularity and the calendar of the valid time have to be determined according to the application. For instance, the *salary* and the *function* of an *employee* are associated to a valid time expressed at the *day* level in the *Gregorian* calendar.
- The type of management is defined by two options : state or history. The *state* option allows to manipulate the last valid state of an object whereas the *history* option permits to have all the valid states of an object. For instance, the *salary* of an *employee* should be an *history* whereas the last valid state of the *family status* of an employee is sufficient for his/her enterprise.
- The completeness of the history is a characteristic applied only on an history managing the period of validity for each state. This characteristic expresses if the history is complete or partial.
  - A complete history implies that the union of all periods of validity forms a period ; this period represents the life span of the object. For instance, the *salary* of an *employee* is a *complete history*.

  - A partial history implies that the union of all periods of validity forms a set of periods (no adjacency). The *weight of a patient* is a partial history, because the weight between two visits is not known.

  The completeness of an history is managed as a constraint applied on the valid timestamp.
  - The extrapolation function can be defined if a partial history must be viewed as a complete one. The extrapolation function is used to determine the state of an object at a given instant which are not part of the history. The query operation called *get_value()* must be redefined as an extrapolation function. For instance, *the weight* of a *patient* is a partial history on which the extrapolation function must be defined.

- Definition of the transaction time (if it is required) :

  One characteristic for the transaction time has to be determined : the type of management (state or history). The *state* option permits to manipulate the last database state of an object whereas the *history* option permits to have all the database states of an object.

- Temporal class on a basic or derived class.

A temporal class is defined on a derived class or on a basic class. For instance, *Weight* of a *patient* is a *basic & temporal* class (cf. Figure 12) whereas *WeeklySales* of a *vendor* is a *derived & temporal* class (cf. Figure 13).
The specific characteristics of a temporal class are the definition of the time dimensions to manage. The two temporal classes require the valid time management, but *Weight* is a partial history with extrapolation whereas *WeeklySales* is a complete history. Note that the temporal aggregation link (between a temporal class to a snapshot class) in the textual specification is introduced by the keyword **of**.

```
Temporal-class for basic class Weight of patient extents Weights
time dimensions VT INSTANT-A<Gregorian, day> as history partial with extrapolation
```

properties weight : REAL ;

constraints 0 < weight < 200 kilos ;

operations

create_weight (w : REAL, t : INSTANT<Gregorian, day>) **type create**

update_weight (w : REAL, t : INSTANT<Gregorian, day>) **type evolution**

correct_weight (w : REAL, t : INSTANT<Gregorian, day>) **type correction**

queries

REAL **get_value** (t : INSTANT<Gregorian, day>)

events ...

endtemporal-class

Figure 12 : Textual specification of a basic & temporal class.

Temporal-class for derived class *WeeklySales* **of** *vendor* **extents** *Wcollection*

time dimensions VT PERIOD-A<MyWeek, week> **as history complete**

properties

sales : REAL **derived**

derivation

weeklysales_derivation (Vendor, PERIOD-A<MyWeek, week>)

queries

real $averageSaleVendor(v :Vendor)

real $averageSale (s :PERIOD-A<MyWeek, week>)

events

decreasebytwo

endtemporal-class

Figure 13 : Textual specification of a derived & temporal class.

1.4.4.3 *The* temporal dictionary class

The dictionary is a particular collection class in ODMG-97 allowing to access to data via a key. A dictionary class is a collection of pairs (key, data).
For example, The storage unit should be designed as a dictionary where each file (data) is accessed via its address (key).

The TOOM method introduces a temporal form of the ODMG-97 *dictionary* collection called *temporal dictionary class*, where the key is the valid time period of the corresponding data. For example, the *inflation rate* of a *country* can be a *temporal dictionary class* of the snapshot class *country*, because the semantic of an *inflation rate* value depends of its valid time period.

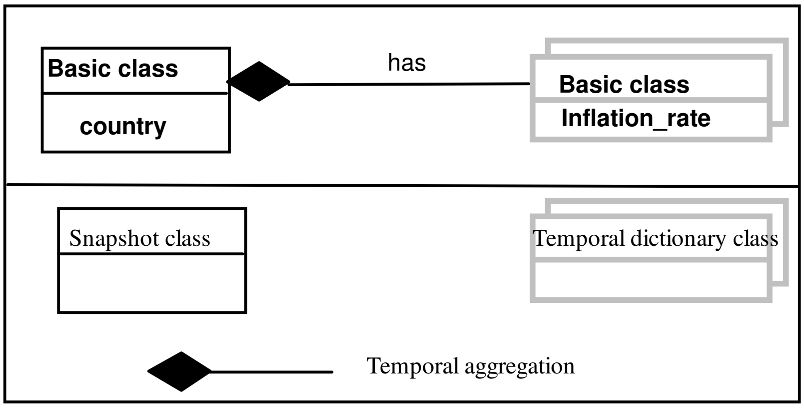

Figure 14 : Example of temporal dictionary class.

Consider a class *country*, that has a temporal variation called *Inflation_rate* class which is a temporal dictionary class. *Inflation_rate* may have entries with overlapping valid time periods as long as the exact period cannot be used with two different values. This means that for a specified country, say Greece, we may have different inflation rate values according to their valid time period (cf. Table 6).

Inflation_rate	**key :** **Valid Time Period**
7.8%	[1995-01, 1996-01)
8.8%	[1995-01, 1995-07)
6.8%	[1995-07, 1996-01)
9.5%	[1996-01, 1997-01)

Table 6 : Inflation rate of Greece as a temporal dictionary class.

Note that the periods that refer to the inflation rates are overlapping. This is acceptable, since the inflation rate depends on the time period on which it is measured as a whole, and has no meaning for subperiods. However, a value for the inflation rate equal to, say, 9.9% and valid time period equal to [1995-01, 1995-07) would be unacceptable, since a value for this time period is already defined.

- Difference between *temporal dictionary class* and *temporal class*

  In the temporal dictionary, the valid time periods can overlap because the valid time is only used to access the value.
  The *inflation rate* of a *country* was not considered as a temporal class because the different values can have different semantics depending of their valid time and all the values do not build the evolution of the *inflation rate*.

- Time dimensions definition

  - Definition of the valid time: the key of the temporal dictionary is the valid time period. The valid timestamp is a period, several values can be managed but are not considered as an history. The granularity and the calendar of the valid time has to be determined. The *inflation_rate* temporal dictionary class exemplified in Table 6 has a valid time expressed at the *month* level of the *Gregorian* calendar.

- Definition of the transaction time (if it is required) :

The transaction time can be managed if the database history or the last database state for each key is required. A database history is managed for each key. It permits to know the successive values stored in the database with their transaction time.

1.4.5 Extension of static links between classes to time dimensions

The static links between object classes are aggregation, association and inheritance (cf. section 3.3.1.5 of the long deliverable).

- Aggregation

  The aggregation link existing in the OO method is used to express the composition of objects of snapshot classes. For instance, the *order* class is composed of *order lines*. This aggregation is time independent.

  The *temporal aggregation* is a kind of aggregation between a snapshot class and a temporal class or a temporal dictionary class. For instance, an *employee* has a *salary* and a *function* which are evolving during the career of the employee. The temporal variation is denoted graphically with a black diamond (cf. Figure 15).

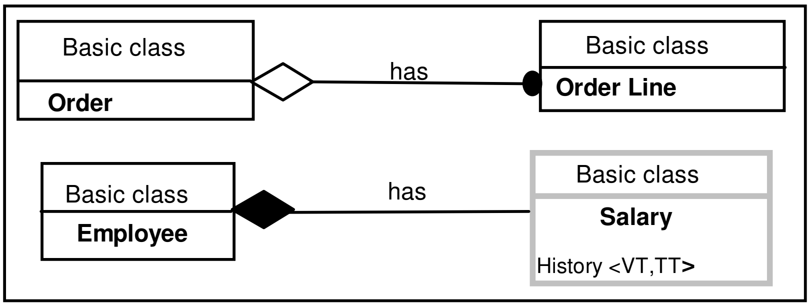

Figure 15 : Examples of aggregation.

- Association

  The association is defined in the OO method. It is applied on snapshot or temporal or temporal dictionary classes. Two kinds of relationship exist : an association referring to an object and association referring to a state (state of an object at a given instant).

  A person owns a car (a) and *an order is related to a customer* (b) are examples of association on objects. But, *a client has a version of a software* (c) is an association referring to a state of an object, such associations are *called state associations* (cf. Figure 16).

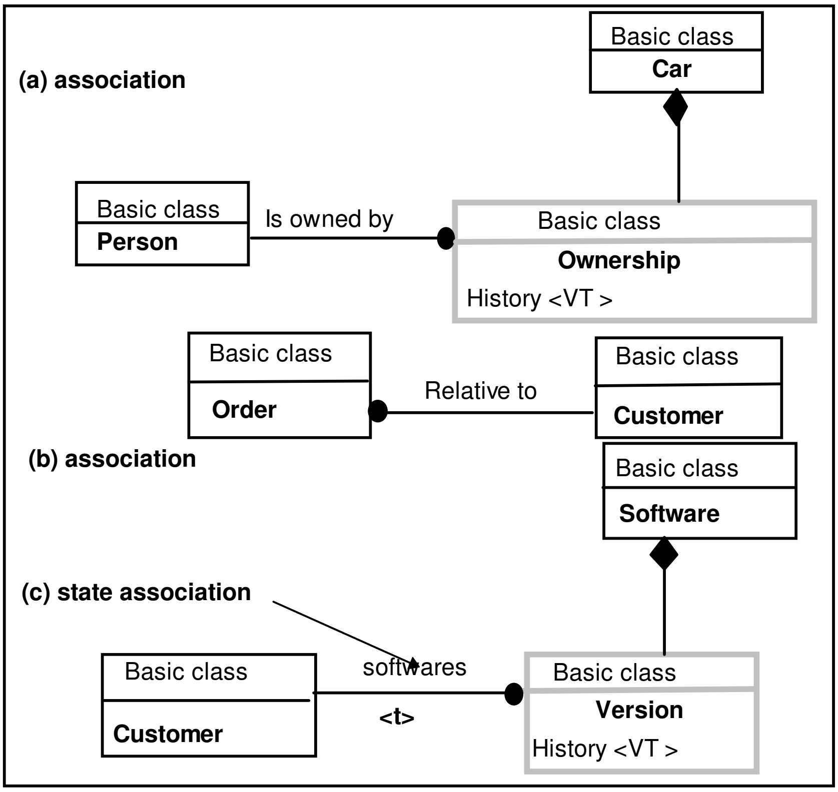

Figure 16 : Examples of association.

An <u>association</u> between a snapshot class and a temporal class expresses a link between objects. As the target objects are time dependent, the link provides the whole history of the target. For instance, the *association between person and ownership classes* expresses that a *person owns a car*, and *a car stored all its owners over time in the temporal class ownership*.

A <u>state association</u> expresses the fact that an object is linked to a specific state of an other object. It means that the time of the state to refer to is mentioned in the association state. Consequently, the identity of the referred state of an object is the OID of the object and the time of the required state (temporal OID). For instance, *a customer has a particular version of a software* ; it means a state of a software object. Figure 16 illustrates the graphical notation of a state association (case c).

- <u>Inheritance</u>

  The inheritance mechanism can be applied only between snapshot classes. For instance, the *employee* class *inherits from* the *person* class. It means that an employee inherits from the person class all the structural properties as well as the behavioral properties ; in particular all the temporal variations, e.g. the *family_status* class.

  The inheritance is prohibited with temporal and temporal dictionary classes.

1.4.6 Extension of constraints applied on object classes

The Extension of constraints consists of :

-defining a framework for classifying constraints,

-introducing constraints specific to temporal features.

The framework is not introduced in the short presentation but is fully explained in section 4.3.6.1 of the long deliverable.

In the OO method introduced in Chapter II, the constraints are defined firstly on object classes : uniqueness constraint, inheritance constraint and attribute constraints and are also embedded in the definition of aggregations and associations such as cardinality constraint and referential constraints. A definition of each of these types of constraint is provided in section 4.3.6 of the long deliverable.

The Extension of these constraints to temporal aspects affects only referential, uniqueness and attributes constraints.

- The ***referential or temporal existence constraint*** is embedded in the aggregation and association definitions. But, three different referential constraints are introduced according to the time to consider.
    - The ***non temporal referential*** constraint is applied on snapshot classes. For instance, the attribute *relative to* in the *Order* class refers to an object of *Customer* class.
    - The ***temporal existence*** constraint is applied on temporal or temporal dictionary classes. For instance, the attribute *is owned by* in the *Ownership* temporal class with History<VT> refers to an object of the *Person* class which should exist at the valid time.
    - The ***temporal referential*** constraint is defined on state-association which refers to the state of an object at a given instant. For instance, the *customer* refers to a *version of a software at a given instant*.

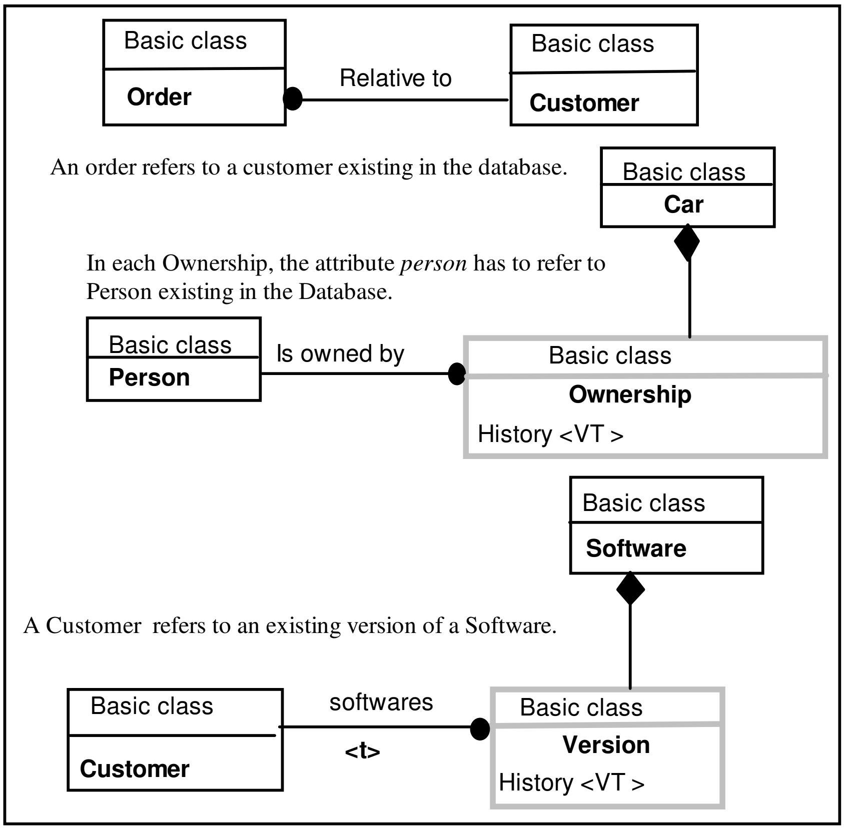

Figure 17 : Examples of referential constraints.

- The ***uniqueness constraint*** is also defined for temporal class and temporal dictionary class. According to the time integration in the collection to consider, 3 different uniqueness constraints are introduced :

- The ***uniqueness*** constraint defines on the non-temporal collection. For instance, the attribute *SocialSecurityNumber of a person* must be unique in the set of *Persons* (collection associated to the class *Person*).
- The ***local temporal uniqueness*** constraint is applied inside an object. For instance, *each instant belonging to the VT period* must *be unique* inside the *ownership* (temporal class with history<VT>) of a *car*.
- The ***global temporal uniqueness*** constraint is applied on a temporal collection relative to several objects. For instance, the *immatriculationNumber of a car at a given instant* must be unique *in the set of cars at every instant*.

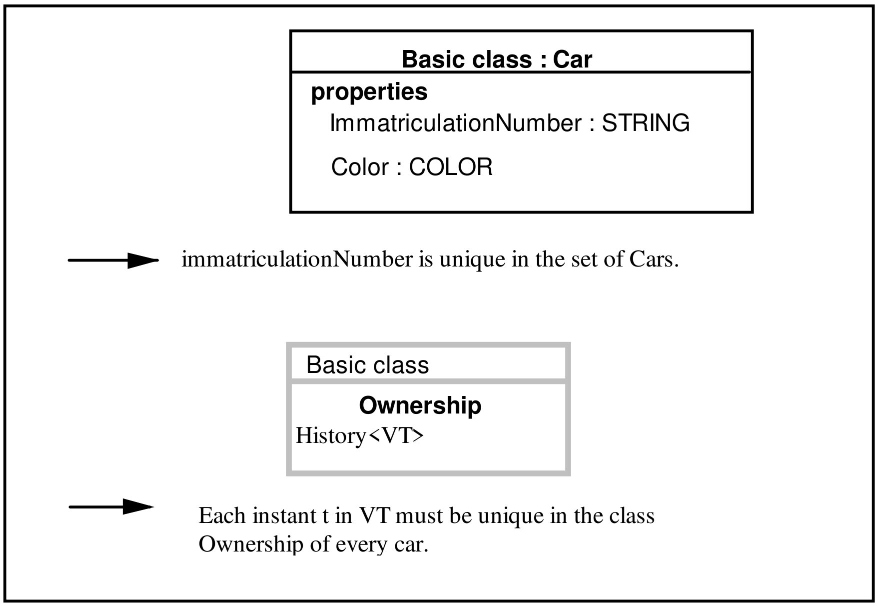

Figure 18 : Examples of Uniqueness constraints.

- The ***attribute constraint*** is defined for any kind of classes. They can refer to one or more objects and they can use one or more times. An example for each possible category is provided (cf. section 4.3.6.2 of the long deliverable).

1.4.7 Extension of events to time dimensions

Event is the concept used in the OO method to define the dynamic of the application (cf. section 3.3.3 of the long deliverable). Three kinds of events are defined :

- an external event materializes the emission of a message by an actor,
- an internal event acknowledges a noteworthy state change of an object,
- a temporal event is a particular state of the clock.

The introduction of the time dimensions in the database leads to take into account the valid and the transaction times into the event concept.

The occurrence of an event happens at a given instant. This instant is the transaction time of this event. The valid time of an event depends on the kind of event.

1.4.7.1 External event

The emission of a message by an actor is an external event. The trigger part of this event represents the treatment to perform when such messages arrive to the system. This treatment is the answer of the IS to this message. The valid time of an external event is the valid time associated to the message. *An event in time* is an event whose valid time is the present time. *An « a priori » event* has a valid time in the future whereas an *« a posteriori » event* has a valid time in the past.

The events described with the basic OO method (cf. section 3.3.3.1 of the long deliverable) are event in time. The « *a priori* » and « *a posteriori* » events are exceptional cases, which have to be taken into account in temporal databases. These two kinds of events are managed in the TOOM method as follows :

- An « a priori » event allows to trigger treatment in advance. In order to avoid problems relative to the management of post-actions, the acquisition of the message is done in advance but the treatment associated to the event is executed in time.
- An « a posteriori » event is limited to exceptional cases aiming at correcting the modeled real world stored in the database. For instance*, the correction of the customer's account related to a registered cheque in a bank* is an example of « a posteriori » event. The treatment associated to this event has to correct the situation by erasing the treatment made with the « wrong » account and apply the treatment with the « right » one. In the same manner, the event *« input the salary of an employee with three months delay* » requires to redo the payment treatment for these three months. The « a posteriori » events are exceptional situations of the application requiring exceptional business rules to apply. The corrective treatment is in charge of the application layer. The temporal DBMS does not provide retro-action functionalities.

1.4.7.2 Internal event

The valid time of an internal event is restricted in the TOOM method to the « event in time ».

Internal events based on snapshot classes can only be ascertained on the change of the current state of an object. Thus, the event is considered in time.

For the temporal classes with history, the « a priori » and « a posteriori » events are forbidden. Only change of the current state can trigger internal events. This limitation avoids the problem of automatically triggering retro-actions and post-actions raising the problem of consistency assurance.

Classically the predicate of an internal event expresses the state change of an object ; for instance, *if the stock of a product becomes less than the replenishment level*, a message is sent to the stock manager. This event is called *event on object* because it is based on the state change of the stock object. The extension of object classes to temporal classes with history implies a new class of events based on the state of the history instead of the state of an object. For instance*, If an employee has increased his salary four times during the last three years*, the staff department is alerted. The predicate of this event is based on the historical state changes.

history or temporal dictionary with history<TT>.

1.4.7.3 Temporal event

A temporal event is a temporal expression based one of the calendars of the application. If the application requires several calendars, temporal events are local to a calendar. Each calendar class is characterized by the definition of the calendar explained in section 4.1.3.1 of the long deliverable and the set of temporal events based on the clock of this calendar.

Each calendar class is realized at the instance level by a clock. By default, a DBMS provides the clock of the *Gregorian* calendar. Each clock of the application is synchronized on the system clock.

Three types of temporal events are defined : absolute, relative and periodic events. For instance, the *«25th of December 1996»* is an absolute event, *« each 25th of December»* is a periodic event and *«1 month after the expiration of the loan»* is a relative event.

The transaction time of a temporal event is the instant when the event is acknowledged. The valid time of the temporal event is a period during which its predicate is true. In a temporal database, a time occurrence of a temporal event belongs to its valid. If the period of validity attached to temporal events has a duration less than *the minute granularity*, the application is considered as a characteristic of real time system application.

In the temporal extension TOOM, the relative event is refined according to the relative time domains defined in section 4.1.3.2 of the long deliverable).

The relative event has a relative time expression as predicate. Two kinds of temporal marks are possible : *temporal attribute* or *time occurrence of event*.

- *«1 month after the expiration of the loan»* is an ***event relative to an object*** of the loan class and the temporal mark is the attribute *end of loan*. The valid time of the event is derived from the attribute used as temporal mark.

«15 days after the arrival of an order » is an ***event relative to the event*** *« arrival of an order »*. The valid time of this event is derived when the event *« arrival of an order »* occurs. The temporal mark used in this case is the time occurrence of *the « arrival of an order »* (transaction time).

1.4.8 Database administration policy

Temporal databases aim at keeping all data about the real world which represent present, past or future knowledge. Such histories increase the volume of the database and also decrease its performance in terms of queries. These two aspects are

essential for preserving the optimal usage of the database. The analyst has to model the application but has also to take into account the policy to apply for cleaning the database by physically deleting all past information about the real world which is no longer used. But, these vacuuming activities must keep the data consistent. For instance, an object or a state of an object at a given instant should not be deleted physically if an association refers to it.
The coherence of the temporal database is difficult to preserve if the policy is always local to a class. We advise the analyst to consider the administration of the temporal data as a global problem, and to design the set of vacuuming activities as a particular case of the database behavior.

The vacuuming rules are described through a dynamic schema in terms of events and operations on objects. Thanks to the event concept vacuuming rules can be classified as follows :

	how to specify it	example
Number maximum of states	internal event on history	The *salary* temporal class with history<VT> is limited to 10 states
Fixed interval	temporal event relative on object	The *salary* temporal class with history<VT> is limited to the interval *2 years*
Relative time limit	temporal event relative on object	The inferior limit of the *salary* temporal class with history<VT> is : « now » - 2 years (now-relative time)
Periodic triggering	temporal event with periodic time	Each year
Manual triggering	external event associated to the *database administrator* actor	Database transaction called « Salary vacuuming »

Table 7 : Kinds of vacuuming rules

1.5 UML compliance

The Unified Method Language (UML) is owned by Rational Company and it is issued from three well known OO methods :

- OMT designed by James Rumbaugh,
- OO method designed by Grady Booch,
- OOSE of Ivar Jacobson.

UML is a method for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. By unifying these three leading object-oriented methods, UML provides the basis for a *defacto* standard in the domain of object-oriented analysis and design founded on a wide base of user experience.

Version 0.91 of UML is a proposal for an OO method submitted to the OMG group. The final decision is not made but the proposal has a great chance to become a standard in the OO methods area. The UML compliance is an important issue for the TOOM method.

1.5.1 UML notation

UML has defined the external format of diagrams. A diagram is a projection upon the elements of a model; the same elements may appear in multiple diagrams and a given diagram may present some - but not necessarily all - of the details of any particular element.

- ***Class diagram***. It is the core of an UML model. This view shows the logical static structure of a system: its contents and their relationships to each other. In its most fundamental form, it shows the elements composing the state of a system (those elements that must be remembered from moment to moment).
- ***Use case diagram***. It is a generic description of an entire transaction involving several objects. It can also describe the behavior of a set of objects, such as an organization.
- ***Sequence diagram***. It shows the interactions among a set of objects in temporal order. A text dialogue can accompany or replace such a diagram. It emphasizes the temporal flow of behavior.
- ***Collaboration diagram***. It is a scenario diagram that shows the sequence of messages that implements an operation or a transaction.
- ***State diagram***. It describes the temporal evolution of an object of a given class in response to interactions with other objects inside or outside the system. Each class may have a state diagram to describe its dynamic behavior, although not all classes have interesting dynamic behavior.
- ***Components diagram***. It represents the physical modules that provide the defining occurrence of these logical elements.
- ***Deployment diagram***. It specifies and visualizes the physical topology upon which a software system executes.

1.5.2 Mapping from OOM and TOOM to UML

Behavior in temporal applications is an important aspect to study. This is why we choose for OOM an event oriented approach and a strong integration of behavioral modeling with the two other perspectives.

In the OOM approach, the behavior of the application is described in terms of events and triggering rules. An event is a particular state change of objects which plays the role of a stimulus implying the system reaction. The reaction of the system to an event is expressed through triggering rules in terms of operations to perform on objects. This is a causal approach of dynamics in the sense of cybernetics and causal theory of organizations. It has numerous advantages among which are (a) the possibility of both a global and a local study of the system behavior and (b) a tight coupling of the three perspectives. The important principle of minimality (minimum set of concepts for a maximum representability) which is a well established principle in science is then, fully followed.

On the contrary, UML uses states models. A states model is only local to a class; it exhibits the possible states of an object in the class and the allowed transitions between these states. This approach lacks the representation of the global system behavior; this makes for example, the validation of the specification more difficult. In addition, the behavioral modeling is disconnected from structure and function modeling, which is bad.

The UML method is an open method which can be extended with the concept of stereotype. A stereotype represents the metaclassification of an element. It represents a class within the UML metamodel itself, i.e., a type of modeling element. It is a way to allow users (methodologists, tool builders, and user modelers) to add new kinds of modeling types. Every element in the UML may have at most one stereotype; this stereotype is omitted when the normal predefined UML modeling semantics suffice. There is a separate list of stereotypes for each kind of UML element. Some stereotypes are predefined, but users are allowed to define their own. Stereotypes have semantic implications which may be specified for every specific stereotype value. UML recommends that tools allow every stereotype to be further distinguished by style (color, line thickness, fill pattern, font) as well as by icon. The purpose of this improvement is to permit tools builders and users to tailor the UML's graphical syntax in controlled ways, and to permit model to have special visual cues.

The detailed mapping of the OO Method and its temporal extension is provided with case studies in section 5 of the long deliverable.

1.6 References

[Brinkkemper90] : S. Brinkkemper, *Formalisation of Information Systems Modelling*, PhD Thesis, Thesis Publishers, Amsterdam, University of Nijmegen, 1990.

[Brunet93] : J. Brunet, *Analyse Conceptuelle orientée-objet*, PhD Thesis, University of Paris 6, 1993.

[Coad90] : Y. E. Coad P., *Object Oriented Analysis,* Prentice Hall, englewoods Cliffs, New Jersey, 1990.

[Coad91] : Y. E. Coad P. , *Object Oriented Design,* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Griethuysen82] : J. J. V. Griethuysen, *Concepts and Terminology for the conceptual Schema and the Information Base*, ISO TC97/SC5/WG3, 1982.

[Hofstede90] : T. F. V. Hofstede A.H.M., G.M. Wijers , S. Brinkkemper, *The SOCRATES project*, proc. of NGCT'90, (Noordwijkerhout, The netherlands), pp. X1-1-X1-7, 1990 (April).

[Jacobson93] : I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Le genie logiciel orienté-objet, une approche fondée sur les cas d'utilisation,* A.-W. France, Eds., Bordas, 1993.

[Lee94] : S. P. Lee, *Formalisation et aide outillée à la modelisation conceptuelle*, PhD Thesis, Paris 1, 1994.

[Lyytinen89] : S. K. Lyytinen K., Tahvainen V-P., Modeling CASE *Environments in systems Work*, CASE'89 conference papers, (Kista, Sweden), 1989.

[Martiin94] : P. Martiin, *Methodology Engineering in CASE shells : Design Issue and current Practice, Computer science and information systems reports* PhD thesis, Technical report TR-4, 1994.

[Martin92] : J. Martin, J. J. Odell, *Object-Oriented Analysis and Design ,* P.-H. I. Editions, Eds., 1992.

[Olle91] : H. J. Olle W., Macdonald I.G., Rolland C., Sol H.G. Assche F.J.M.V., Verrijn-Stuart A.A., *Information Systems Methodologies,* Addison-Wesley, 1991.

[Rolland82] : Rolland C., Richard C. : *The REMORA methodology for Information systems Design and management*, Information Systems Design Methodologies : a comparative Review, Olle T.W, Sol H.G., Verrijn-Stuart A.A. (eds), North-Holland (pub), 1982.

[Rolland96] : Rolland C., *Using Generic Method Chunks to Generate Process Models Fragments*, 3rd Int. Conf. on Requirements Engineering (ICRE'96), 1996.

[Rumbaugh91] : J. Rumbaugh, M. Blaha, W. Premerlani, F.Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* P.-H. I. Editions, Eds., 1991.

[Shlaer92] : S. Shlaer, S. J. Mellor, *Object lifecycles. Modeling the world in states ,* Y. Press, Eds., 1992.

[Shlaer88] : M. S. J. Shlaer S., Eds., *Object Oriented Systems Analysis ,*Yourdon Press, 1988.

[Toobis96a] : deliverable « User Requirements », TOOBIS project, June 1996.

[Wijers91] : G. M. Wijers, *Modeling Support in Information Systems Development*, PhD Thesis, PhD thesis, 1991.

[Wynekoop93] : R. N. Wynekoop J., *System Development Methodologies: Unanswered Questions and Research-Practice Gap*, 14th Int. Conf. Information Systems, (Orlando), pp. 181-190, 1993.