Time Patterns

Author: Manfred Lange
Hewlett-Packard GmbH
NSL (Network Support Lab)
Herrenberger Str. 130
71034 Boeblingen
e-mail: Manfred_Lange@hp.com

Abstract

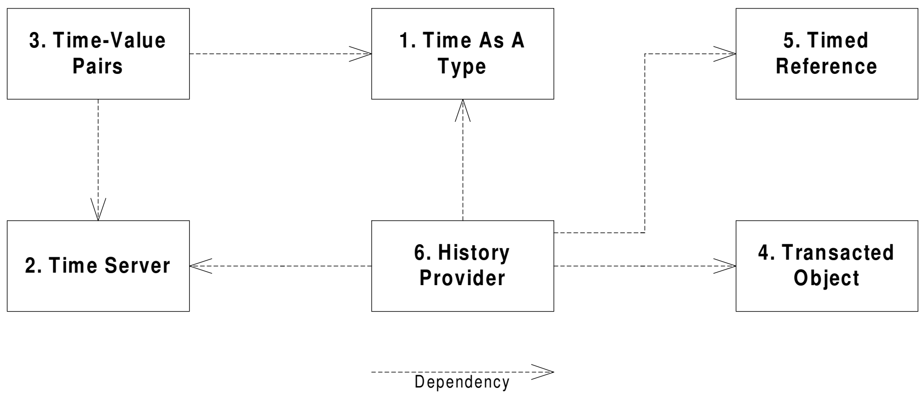
Many systems need to deal with time. Time can be just an additional type, such as for the date of birth, or it can be an additional dimension, e.g. when a history of data is needed. This paper summarizes some of the patterns that help to design these systems.

Overview

Christian August Crusius (1715-1775), a German philosopher wrote that each existing thing has its very own place in space and time [CRUS1744]. Objects as they are used in software industry also represent existing things. Objects represent also concepts and ideas. There are only few papers regarding time, but a lot more dealing with space, assuming that a particular object can exist only once.

Adding time to a system model is like adding another dimension. Many systems exist that need to deal with time. An accounting system needs timestamps in order to track, when each entry was made. A system for managing resources needs time periods for ensuring that the same resource is not used twice at any time. A revision control system needs to track the changes of files over time.

All these cases have some aspects in common, e.g. having time as a type, or tracking the state of object over time. The following figure shows the patterns described in this paper and also how they relate to each other.



This paper contains a couple of examples. All examples are formatted the same way as this paragraph using a border and a light gray shading.

Disclaimer

In general, there are not a lot of patterns that specifically focus on time related design issues. This paper tries to describe some solutions that have worked in practice, but the paper does not claim to be a final set of time related patterns.
If the paper happens to become a starting point in the community, it has well served its intent!

Common Forces

Whenever time comes into play, this adds another level of complexity. Not only is it difficult to understand, which implications "time" has, but it is also hard to explain and sometimes even harder to model.
So the common force is additional complexity, especially when several of the techniques described in this paper are applied at the same time.

Known Uses

All the patterns described in this paper are used in proprietary network support software currently in use by Hewlett-Packard. Some of the patterns, e.g. "Time As A Type" and "Time Server" are also used in a medical product HP Series 50 TraceVue.

In one of our network support software we take snapshots of the customers network. The support engineers need to know, when the snapshot was created and when the snapshot was transferred to the database containing the customers network data. The C++ class representing the snapshot looks as follows:

```
class CSnapshot
{
   … // other members
private:
   // Attributes:
   CTime   m_snapShotDateTime;
   CTime   m_transferDateTime;
};
```

Other possible uses include, but are not restricted to revision control systems and data monitors such as any technical devices that need to track values over time.

1. Time As A Type

Motivation

Time and time periods are used frequently. So it is helpful to be able to use it as a built-in type.

Forces

1. Some programming languages are not strongly typed. This allows for errors that are not discovered until runtime.
2. Some programming languages don't support definition of types at all.
3. In some programming languages a whole bunch of different libraries with different implementations of classes for time and time periods exist. These types are often not compatible. This is even worse, when the platform on which the software runs requires the use of different time types, depending on the API to be used.

Solution

Define and implement a class for both time and time period[1].
Provide for a rich set of conversion operations between the newly implemented classes and the already existing ones.

The actual data is kept in data members. The number and types of the data members depends on the requirements. A good tradeoff for both classes is, that both have data members for year, month, day, hour, minute, second and milliseconds.
If the type must be optimized for space, then as many data members as possible should be omitted, e.g. in some cases milliseconds may not be required.
If the type must be optimized for precision, then additional members can be added to the classes, e.g. a float member for milliseconds to represent fractions of milliseconds instead of a 2-byte value representing just the milliseconds.
For both classes a number of methods should be implemented, e.g. conversion from/into other types of the development environment.

On a Win32 platform conversion operators would be provided from/into FILETIME, COleDateTime, time_t etc.

Consequences

+ Easy use of time as a type as the new classes can be used the same way as built-in types.
+ Use of only one type throughout the software. Only this class needs to be learned.
+ Better portability, as in principle only this class needs to be ported.
+ Precision can be chosen depending on applications requirement.
– Introduction of yet another time class and yet another time period class. This problem can be softened if one of the existing time classes can be reused, e.g. in the form of delegation.

Implementation Issues

1. Depending on the accuracy required by the target application, the private member(s) representing the time or timespan can be anything from 4 byte integers to sophisticated structures having a member for each part of the time value, e.g. milliseconds, seconds, minutes, etc.
2. If the programming language supports it, provide operators for conversion from/into other types, assignment, comparison, addition, subtraction etc. CTime and CTimeSpan objects can then be used the same way as built-in types, which makes their use more convenient.
3. For the implementation of conversion operators the following typical problem arises: What if an object with a higher precision is converted into a CTime or CTimeSpan object? The following options are available:
   - Silently round the value, e.g. fractions of milliseconds.
   - Throw an exception (if supported by the programming language), which must be handled by the application.
   - Write a warning to the trace log during runtime.
4. The function for retrieving the current system time can be a method of the CTime class, or it can be a standalone function.

[1] The terms time period and timespan are used interchangeably in this document.

2. Timeserver

Motivation

Several reasons exist why the system time of a machine can change. Daylight savings time and low battery are just two of them. On the other hand some applications need to have a reliable time series, that means it must be guaranteed that for each call of the time function the resulting value is new than all values for preceding calls.
A possible solution to the problem is additional hardware that receives signals from an atomic clock. The system time is then synchronized with the transmitted time. Here a software only solution will be discussed, that proved to be helpful in some environments. But even when an atomic clock signal is available, this does not solve all problems as shown in the sample.

Sample: Assume you have an application where notes of different types are sorted by time. Further the first note is entered at 05:00 PM actual time and a second note is entered at 07:00 PM actual time. At 06:00 PM actual time the user finds out that the system clock needs to be adjusted and sets it back 3 hours, resulting in a system time of 06:00 PM. This means that the first note - which was entered at 05:00 PM actual time - was entered at 08:00 PM system time, too. This is then the same system time as for the second note. If the application now would sort the notes just by a system date, the order of the notes would not be correct. At first sight, the application may decide to have some kind of serial number attached to each note, so that the system time does not play a role. But what if the user wants to add a third not that is associated with 07:30 PM actual time? This third note then the position of the third note should be between the other two, which then causes problems with the serial number.
Extension to the sample: Assume the application has to keep track on when the system time has changed and by what amount for legacy reasons. This, too, can't be solved using a serial number, and there, a timestamp is need again. But which timestamp should be used? The system does not know about the actual time. It has the system time and it can keep track of the changes that occur because of user intervention or because of adjustments caused by the signal of the atomic clock signal receiver.

Forces

1. Application needs to track all user-initiated changes to the system clock.
2. Application needs a series of timestamps that are guaranteed to be sorted in ascending order.

Solution

Run a service permanently while the system is up and running. Alternatively make sure that the service is running while a process runs that changes the system clock programmatically or while a user is logged in who can change the clock using the console.
While running the timeserver maintains a list of time-time pairs. Each pair consists of a timestamp that represents the actual system time and the other timestamp represents the time extrapolated of the previous times.

Consequences

+ It is possible to track the changes in the system clock. The track record contains information on when the clock was changed, by which amount it was changed and what time the system clock would have if the change would not have occurred.
+ If the application always gets the timestamps from the timeserver, it is guaranteed that they are generated in ascending order.
– It must be ensured that the service is running all the time. This might be a problem, if the workload of a system is already high.

Implementation Issues

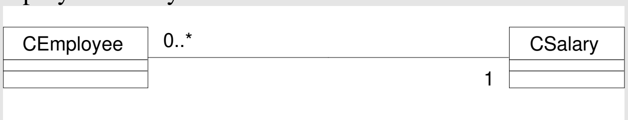
The Timeserver should be implemented as a service. This means it should all the time, when the system is up and running. Depending on the operating system the Timeserver may use different mechanisms to find out whether the time has changed: Under Win32 Windows Messages are being sent to processes in case the time setting has been changed the user.

3. Time-Value Pairs

Motivation

Some systems need to track the history of one single attribute in an object.

To track the evolution of an employee's salary the association could be modeled like this:



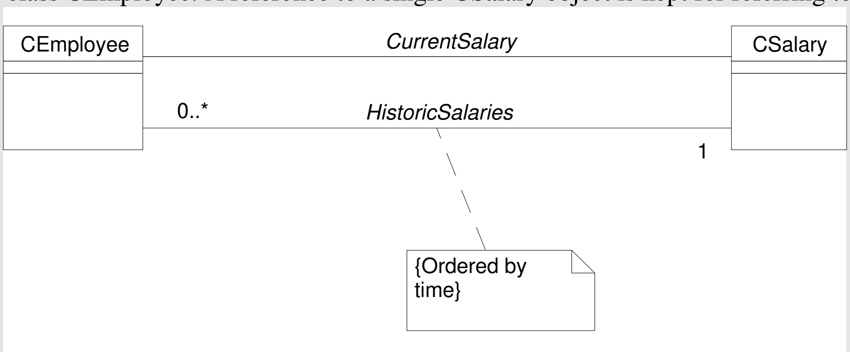
Forces

The system has to provide a means for storing the different salaries over a certain period of time.
When querying the object, the caller has to specify which time and/or time period he wants to retrieve.

Solution

Implement the attribute as an ordered collection with time-value pairs. The pairs are ordered by time. When a change to the value occurs, a new pair is created and added to the tail of the ordered collection.

Each employee typically has different salaries over time. To keep track of the different salaries a list of previous salaries is added to class CEmployee. A reference to a single CSalary object is kept for referring to the current salary.



The client now has the possibility to retrieve the value, which is associated with a specific time.
Some (error-handling related) problems pop up here:

1. What happens, if a collaborator of an employee object tries to get a value for a time that is older than the oldest entry in the HistoricSalaries collection?
   This can be solved in several ways, e.g. by returning a Null Object (See "Related Patterns" section for reference material regarding Null Object.), or by throwing an exception (if the programming language supports it).

2. What happens, if a collaborator of an employee object tries to get a time-value pair that does not exist?
   Again, a Null Object may be returned here or an exception may be thrown. Another possibility is, that the closest time-value pair or the time-value pair with the previous timestamp is returned, thus assuming, that each time-value pair is part of that state of the employee object and that the employee object not only has a state at specific times but continously.

   An employee is hired on January 1st. His salary is 5,000 US-$. At July 1st, the employee gets a salary increase of 500 US-$. If somebody wants to know what is the salary as of April 1st, then correct value would be 5,000 US-$, although no time-value pair for April 1st exists.
3. What happens, if a collaborator of an employee object tries to get a time-value which is newer than all existing time-value pairs?
   Again this can be handled the same way as described in 2.

Consequences

+ All changes to the value can be undone.
+ Only changes are stored, thus saving resources.
– Interface must provide an in-parameter for specifying the time. The interface becomes a little bit more difficult to use.
– The implementation has to deal with the additional parameter. The implementation becomes more complex, too.

Implementation Issues

If the current value is used frequently, it may be cached in order to improve the performance. In this case, the code for changing the salary must also make sure that the cached value is updated.
The ordered collection of historic salaries may be implemented such that the collection contains the values themselves or it may be implemented, so that the collection holds references to the value objects. Depending on how the classes are used one of the two may be faster.

Related Patterns

Historic Mapping

In his book [FOW1997, p. 305] Martin Fowler describes a pattern named "Historic Mapping". He also describes a variety of derivations of that pattern. He also suggests having a list of previous values, each of them having a time period associated with them, ordered by time, and having the constraint that time periods must not overlap.
The difference between the proposal of Martin Fowler to the pattern presented here is minimal. Instead of associating a time period as Martin Fowler proposes, this pattern uses a time. The advantage is, that the constraint regarding overlapping time periods is not needed no more.

Null Object

A "Null Object", as described by Bobby Wolf in [PLOPD98, page 5], may be used to return a value indicating, that the requested time-value pair does not exist.

Observation

The "Observation" pattern [FOW1997, page 42] can be used, if additional requirements for tracking attribute values comes into play, e.g. also monitoring who made a change.

In contrast to the time-value pairs an observation keeps track of a measurable value, which consists of a quantity and a category. Time-value pairs do not have this restriction.

Timeserver

The Timeserver pattern is used for creating a series of timestamps that is guaranteed to be sorted in ascending order.

4. Transacted Object

Motivation

Time-Value pairs may not be sufficient for some systems, or it may not be feasible to track several attributes using the time-value approach. Changing more than one attribute one after the other leads to intermediate objects state where only some of the attributes are changed. These intermediate states are of no interest to the application.

> Sample: A CNetworkInterface class may represent a network interface card (NIC) in a network device[2]. Attributes from that NIC can be done through some network management protocol, e.g. SNMP.
> Retrieving the attributes from a physically existing network interface card may consist of getting the number of bytes read on that interface card and the packets discarded there. If at a later point in time the same two attributes are read again, there might be two different methods to modify the state of the NIC, e.g. SetBytesRead() and SetPacketsDiscarded(). It does not make sense to set one and then to set the other value. Also, it does not make sense to have methods that set both at the same time. A NIC can easily have dozens of attributes associated with it. So if we want to set all or nothing, then that single method would take dozens of attributes, which makes it very hard to use.

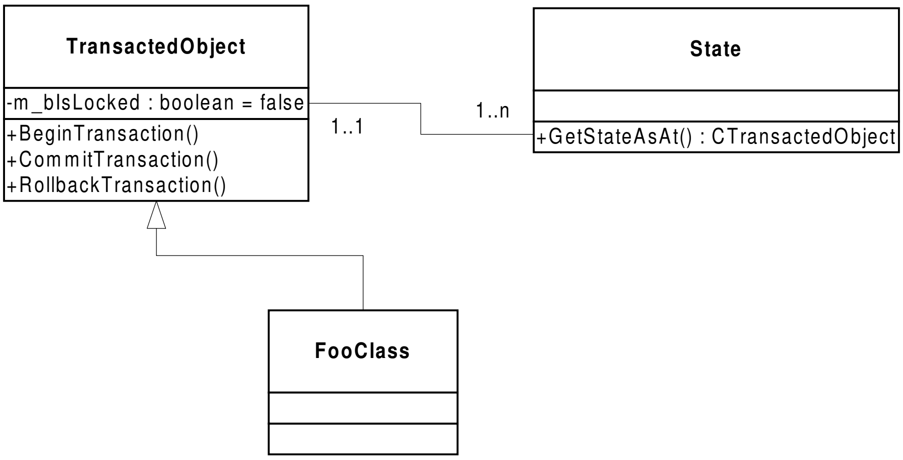
Forces

1. Track the state of an object with a generic solution.
2. Maintain encapsulation of the tracked object.
3. Maintain consistency of the object's state.

Solution

Add a transaction mechanism to the object. All transactions are serialized and obey the ACID[3] principle of transactions. Each collaborator accessing a transacted object starts a transaction, calls methods on that object and commits the transaction. Only one transaction per object can be active at any given time.

[2] A network device can be a computer, a router, a switch, a bridge, a router, a printer and so on.
[3] ACID: See appendix for further details.



Beginning a transaction means locking the object for both, read and write access. The lock is removed when the transaction is committed or rolled back.
If a second collaborator tries to access an object for which a transaction is already active, then the second call to BeginTransaction() is blocking until the first collaborator commits its transaction.
A transacted object may have references to other transacted objects. When a transaction is started the transacted object is locked. Referenced objects are not automatically locked. This decision is left to the implementation of the methods. If a method wants to access a transacted object, the method implementation has to start a transaction, use the referenced object and finally commits the transaction.
The solution can be enhanced by adding a parameter to the BeginTransaction()-call which indicates, whether the object will be accessed for read or write access. Allowing other concurrency models, e.g. multiple readers or one writer/multiple readers, can enhance parallelism.

Consequences

- \+ Objects always have consistent states, even when used by callers executing from different threads
- \+ Use of transacted object can made very easy by hiding the transaction-related code in a smart pointer implementation.
- \+ System resources only limit the number of state changes.
- \+ As long as a transaction is active a client of the object can be sure that the state of the object does not change.
- \+ If a Transacted Object refers to other Transacted Objects, then these objects are automatically locked in a transaction.
- – Classes of Transacted Objects are harder to implement than normal classes.
- – Transacted objects perform slower because of the additional overhead.
- – All accesses to a transacted object must be done when a transaction is active. This also applies to read access.
- – Transaction handling is very tough to implement. If possible the transaction mechanism of a database system or a transaction monitor should be used instead of implementing a proprietary solution.
- – The proposed solution does not work for long transactions (check-out, check-in model).

Implementation Issues

1. The transaction mechanism can be implemented in a common base class:

2. When implementing the TransactedObject class a locking scheme must be implemented too. A simple locking scheme could be that only one client could have a transaction on an object, e.g. by using a semaphore or a mutex. If a transaction on a particular object is already active, then a successive call of BeginTransaction() would be a blocking call.
3. The current state can be stored upon BeginTransaction() or upon CommitTransaction(). The former stores the state of the object even if the state does not change. The latter prevents this, but the current state is duplicated.
4. For multithreaded applications or applications that consist of components running in several processes a mechanism must be implemented to detect deadlocks. A simple mechanism could be to define a timeout for BeginTransaction(). If the time out occurs, the function would return with an error state.

Related Patterns

Memento

[GAM1995] lists a pattern named memento: "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later." [GAM1995, p. 283]
Memento is a straightforward solution good in situations where it is not necessary that several clients possibly executing on different threads access the same object at the same time. Therefore the memento provides no means for transactions.

5. Timed Reference

Motivation

Generally an object is accessed through a reference to it. Methods accessing its private data can be used only with a reference at hand[4]. Having an object with possibly more than one state, it is also necessary to specify which state/version of the object should be used.

> Sample: Given a network management application that retrieves data about a network interface card (NIC) on a regular basis, e.g. every hour. Assume that the following attributes for the NIC are retrieved: "bytes in" and "packets discarded". An object represents the NIC. The NIC class is a transacted object, as we want to keep the history of the object's states.
> The values for "bytes in" and "packets discarded" are part of the state of the NIC object. If, at a later time, these values are retrieved again and the state of the NIC object is updated, the NIC object stores its previous state, as it is a transacted object. Its methods modify the state of the NIC object.
> Now two or more states for the same object are available. With traditional designs, a reference simply knows which object it refers to, but it does not care for any specific state. The object just has one state.
> With history at hand, it is not sufficient for a reference to know only the identity of the object, but it also must know which state of the object it refers to. The reference must refer to a combination of object identity and state.

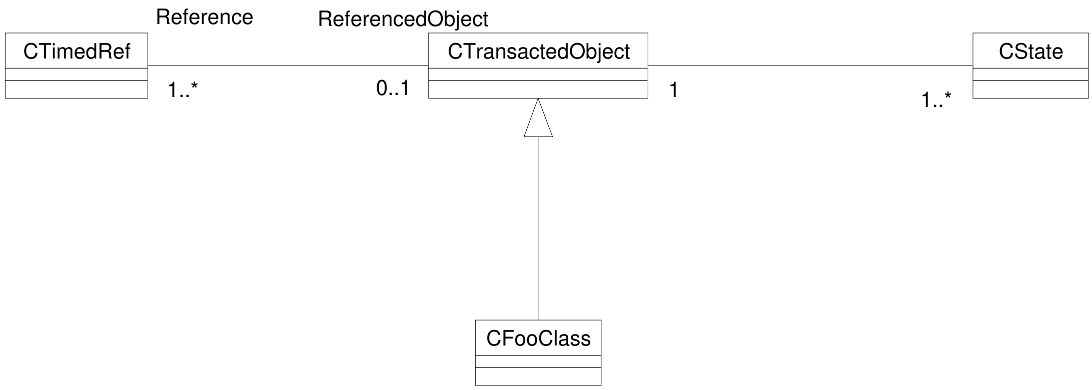
Forces

1. Usage of a Timed Reference should be consistent with the usage of a normal reference.
2. Some programming languages don't support operator overloading.

[4] Except, if class methods are used, which do not need to have an instance of that class.

Solution

Implement a class that represents a reference. Provide for a means to parameterize each instance of that class with a timestamp during runtime:
Use instance of the TimedRef class to refer to objects that have multiple states in a history list. By looking at the TimedRef object, the referenced object can decide which state it has to load when being accessed.



Consequences

+ TimedRef can be used the same way as normal references. The learning curve is flat.
– TimedRef adds a level of indirection, which adds overhead. This overhead is similar to the overhead of a virtual function versus a statically linked function in C++.

Implementation Issues

1. Some programming languages such as C++ support operator overloading. In this case the TimedRef class would be implemented as a smart pointer. Compared with a traditional smart pointer a TimedRef needs an additional timestamp value upon instantiation. When calling functions through a TimedRef object FooClass will be told which state to load.
2. It might be possible to derive the TimedRef class from an already existing smart pointer class.
3. If the programming language supports it, the TimedRef class can be implemented as a template, with the class of the referenced object as a template parameter.

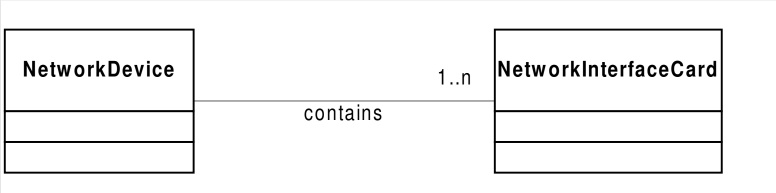
6. History Provider

Motivation

So far, all patterns in this paper described mechanisms for single objects. Additional problems arise when objects are referring to other objects, e.g. with associations or aggregations.

Sample: Assume that an application continuously monitors a network infrastructure. It detects devices, collects data and finds out how they are interconnected. Such a discovery/topology cycle usually takes a longer period of time (up to several hours) depending on the size and the characteristics of the network.

A router, for instance, contains two or more network interface cards (NIC). Using a management application an interface of a router can be turned off during runtime, replaced and initialized again. This can be modeled using the following class diagram:



If both classes were derived from TransactedObject, then updating their attributes, e.g. removing an interface (this is a modification of the attribute of attribute representing the collection of references to the NICs of the router), changing the values for "BytesIn" and "PacketsDiscarded" would be done in transactions. Each object would keep their own history as an ordered collection of states.
First the application detects that an interface has been removed. The application starts a transaction for the NetworkDevice object, calls the function to remove the interface, and finally commits the transaction. Next, the application starts a transaction on the NIC object, calls the methods in order to change the attributes "BytesIn" and "PacketsDiscarded", and finally commits the transaction.
Both transactions almost always have different time stamps, let's say 04:00:00 and 04:00:02. If a client wants to have the router objects at a time of 04:00:01, the status of the NIC objects have not been updated yet, eventually leading to an inconsistent state of the complete model.

As the example shows, having a history per object is not sufficient. Independent transacted objects cannot solve the problem of ensuring the consistency of the history of several objects related to each other.

Forces

1. Objects referring to each other build a network of objects (cluster). A change of the state of one or more objects may also include a change in the set of references between these objects.
2. Two access methods are required to access the objects and their states: The traditional approach by having root or anchor objects, which are then used for further navigation. This is at the same time the logical view of the problem domain (if the time aspect is left out). The other access method is to look at a set of objects and see how they have changed over time. This is the history aspect.
3. An object A refers to a different object B. During a change of the state of object A, the reference to B is removed. Object B will be "deleted" if the object A was the only object referring to object B. Object B is deleted and not deleted at the same time.

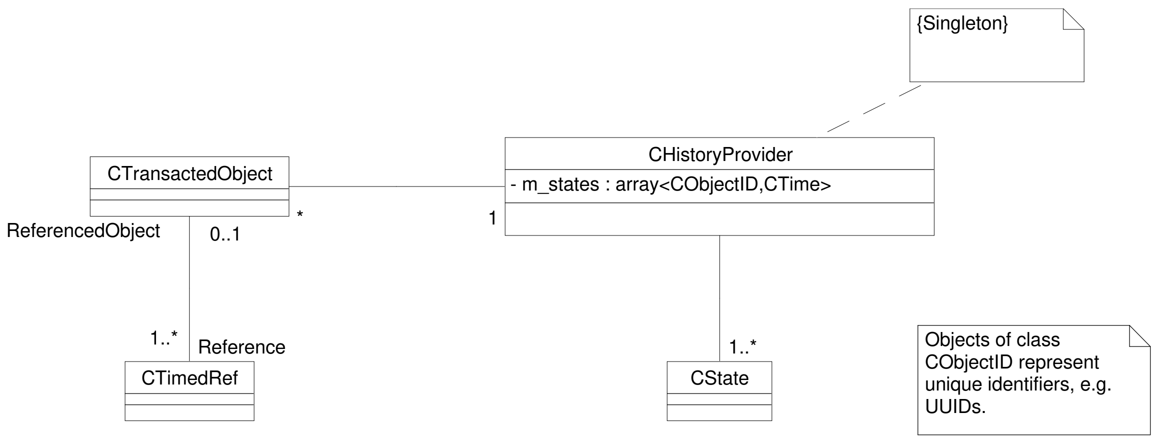
Solution

The solution consists of several components. The basic idea is to record all state changes of each object. The history provider does this. It keeps track of objects and their states.
On the logical level this is done in an array:

The array has two dimensions: One dimension is the object identifier, the other dimension is time. In other words: each row in the table represents the history of the object's state, each column represents the set of states that is the result of a transaction.
A gray square indicates that a state for the object with that ID exists at that time. Letter "D" in a gray square indicates that the object has been deleted at that time.
Note that it is not necessary, that the time periods between the columns are the same.

The sample array shows four objects with different curriculum vitae. Object 0x091 exists all during the complete period. Object 0x653 has been deleted at 00:25. Object 0x0438 was created at 00:15. Finally, object 0x705 was created at 00:15 and it was deleted at 00:25.

How to handle the gaps between the times (00:10, 00:15, etc) in the table, will be covered later.
The structure of the patter looks as follows:



Consequences

1. It must be carefully considered whether to apply this pattern, as it is very complex. It is not only difficult to understand, but also difficult to implement.
2. Just storing modified states of objects reduces the resource requirements for the array. If the objects' state has not changed a placeholder would be introduced into the array, e.g. referring to the last state that has been stored.

Related Patterns

Singleton

History Provider is implemented as a singleton [GAM1995].

Object Recovery

António Rito Silva, João Dias Pereira and José Alves Marques present their paper [PLOPD98, p. 261] a pattern called "Object recovery" which shows how to keep a log of objects. The authors state themselves that the pattern generates an overhead as a copy is generated each time the object is altered even, when its state is not changed at all. Additionally, their paper does not contain any hint on how to solve the problem, when several objects are interconnected. "History Provider" treats the instantiated associations as part of the state of the objects and keeps track of them, too. By using Timed References it is still possible to navigate within a consistent state.
The intent of "Object Recovery" is different. "Object Recovery" tracks states in case these states are needed later. Most of the time the old states will not be used at all. In contrast "History Provider" makes equal-rights objects out of the old states and also provides access mechanisms in two dimensions:

- Within a consistent state of a network of objects.
- Within the history of all states of a particular object.

Timeserver

The Timeserver pattern is used to generate a series of timestamps that is guaranteed to be sorted in ascending order.

7. Appendix

Literature

[FOW1997] Martin Fowler, "Analysis Patterns - Reusable Object Models", Reading Massachusetts, 1997, Addison-Wesley, ISBN 0-201-89542-0

[CRUS1774] Christian August Crusius, "Entwurf der nothwendigen Vernunft-Wahrheiten, wiefern sie den zufälligen entgegen gesetzet werden.", Leipzig 1745

[GAM1995] Erich Gamma et al., "Design Patterns - Elements of Reusable Object-Oriented Software", Reading Massachusetts, 1995, Addison-Wesley, ISBN 0-201-63361-2

[MSDN1998] Microsoft Developer Network: http://premium.microsoft.com/msdn/library/default.htm

[PLOPD98] Robert Martin et al., "Pattern Languages of Program Design 3", Reading Massachusetts, 1998, Addison-Wesley, ISBN 0-201-31011-2

ACID Properties

"Transactions provide the ACID properties.

Atomicity

A transaction either commits or aborts. If a transaction commits, all of its effects remain. If it aborts, all of its effects are undone. For example, in renaming an object, the new name is created and the old name is deleted (commit), or nothing changes (abort).

Consistency

A transaction is a correct transformation of the system state. It preserves the state invariants. For example, by adding an element to a doubly linked list, all four forward and backward pointers are updated.

Isolation

Concurrent transactions are isolated from the updates of other incomplete transactions. These updates do not constitute a consistent state. This property is often called serializability. For example, a second transaction traversing the doubly linked list mentioned in the consistency example will see the list before or after the insert, but it will see only complete changes.

Durability

Once a transaction commits, its effects will persist even if there are system failures. For example, after the rename in the atomicity example, the object will have the new name even if the system fails and is restarted right after the commit completes." [MSDN98]

8. Acknowledgements

Without the feedback of my colleagues at HP, this paper wouldn't have been possible. Special thanks go to Gerhard Eujen and Thomas Titze for their inputs and constructive comments.
I would also like to thank my shepherds for the EuroPLoP 98 conference Massimo Arnoldi and Kent Beck who provided me with very valuable feedback and suggestions. Their comments not only helped me to improve the quality of the contents and the appearance of this paper, but also to view my paper from different perspectives, which helped me a lot.