Time Patterns

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

Abstract

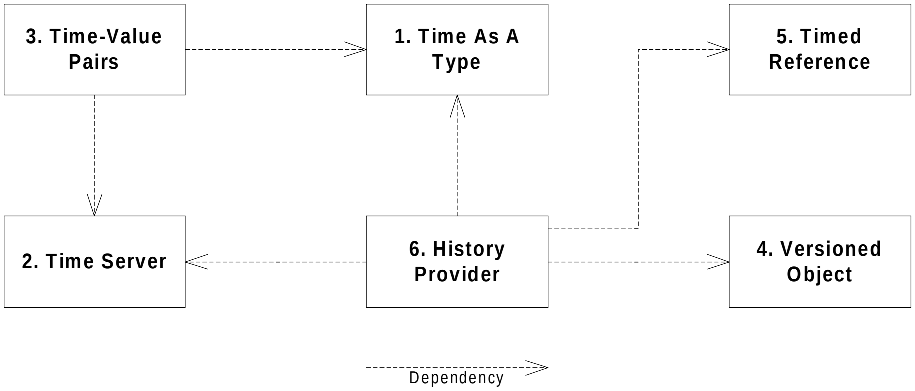
Many systems need to deal with time. Time can be just an additional type, such as for the date of birth, or it can be an additional dimension, e.g. when a history of data is needed. This paper summarizes some of the patterns that help to design these systems.

Overview

Christian August Crusius (1715-1775), a German philosopher wrote that each existing thing has its very own place in space and time [CRUS1744]. Software industry uses objects to represent existing things, too. In addition, objects may also represent 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[1].

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 the entry time of each entry. A system for managing resources needs timespans 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, which are described in this paper, and 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 light gray shading.

[1] The term "space" here includes, that no two objects can exist at the same place (address) within the reachable memory space.

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 for discussion in the community, it has well served its task!

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.

1. Time As A Type[2]

Motivation

Time and time periods are used frequently. Some development environments provide multiple types for time and timespan, which may or may not be compatible with each other. In other cases the APIs[3] of the development platform require you to use certain types.

Forces

1. Your programming language has no built-in type for time and timespan.
2. Your programming environment has multiple types for representing time and/or timespan. You want to reduce the number of explicit conversions.
3. The types for time and timespan that come with your development environment support a different set of operations and functions each, but none of them provides the functionality you want to have.
4. You want to minimize the number of time and timespan types you have to learn.
5. 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.
6. The precision of the available types is not sufficient, e.g. don't support timespans of 1 year or more. Or the type representing the time does not support year representations with more than 2 digits[4].

Solution

Define and implement a class for both time and time period[5]. Provide for a rich set of conversion operations between the newly implemented classes and the already existing ones.

CTime
CTimeSpan

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. (Particular programming languages may support parameterized classes, see "implementation issues".)

[2] For the course of this paper the terms class and type are used interchangeably.
[3] API = Application Programming Interface
[4] This is one of the causes of the year 2000 problem.
[5] The terms time period and timespan are used interchangeably in this document.

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.

Another possibility for supporting different precision is to provide different classes, e.g. CTime and CTimeHighPrecision and CTimeSpan and CTimeSpanHighPrecision.

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

Consequences

+ The new data type can be used the same way as built-in data types.
+ Only one type for time and timespan is used throughout your applications source code. This decreases the maintenance and learning costs.
+ The new data type can be used wherever a different time/timespan parameter type is needed, as you have implemented conversion routines.
+ Porting of your software is easier, as in principal only one time/timespan class needs to be ported.
+ The precision of the type can be adapted to the applications requirements.

– The solution works only for languages that allow definition of new types e.g. C++ and Java.

– A minor problem is that you introduced another time class and another time period class. You can soften this problem 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.
   For languages support parameterized classes, the implementation may provide for a parameter that allows configuration of the precision.
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. For C++ this must be carefully decided, as a conversion from long to short does not throw an exception, but just cuts off the last digits. So throwing an exception would not be consistent with the behavior in similar cases.
     A different alternative to implementing cast operators would be to have explicit conversion routines that return objects of the desired type or a NULL object, if the conversion was not successful.
   - At least you should 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. If it is a class method of the CTime class, then the return value should be an object of the class CTime. In principal this class method is nothing more than a shortcut to retrieving the system time.
5. When adding or subtracting years the implementation must take into account that there are leap years[6].

[6] A year is a leap year, if it is dividable by 4 but not by 400.

Sample code

C++

In C++ the class declaration of a simple time class may look like this[7]:

```
class CTime : public CObject {
public:
   // Semantics:
   CTime();
   CTime(const CTime& time);
   CTime(CTime* cTime);
   CTime(const CTime& cTime);
   CTime(SYSTEMTIME* systemTime);
   CTime(const SYSTEMTIME& systemTime);
   CTime(FILETIME* fileTime);
   CTime(const FILETIME& fileTime);
   CTime( int year, int month = 1, int day = 1, int hour = 0, int minute = 0,
          int seconds = 0, int milliseconds = 0);
   virtual ~CTime();

   // Conversion into other types:
   operator SYSTEMTIME();
   operator SYSTEMTIME*();
   operator CTime();
   operator FILETIME();

   // Operators:
   CTime&       operator =  (const CTime& time);
   CTime&       operator =  (const FILETIME& fileTime);

   CTime&       operator += (const CTimeSpan& timeSpan);
   CTime&       operator -= (const CTimeSpan& timeSpan);
   CTime        operator +  (const CTimeSpan& timeSpan);
   CTime        operator -  (const CTimeSpan& timeSpan);

   CTimeSpan    operator +  (const CTime& time);
   CTimeSpan    operator -  (const CTime& time);

   BOOL operator == (const CTime& time) const;
   BOOL operator < (const CTime& time) const;
   BOOL operator > (const CTime& time) const;
   BOOL operator != (const CTime& time) const;
   BOOL operator <= (const CTime& time) const;
   BOOL operator >= (const CTime& time) const;

   // Access - services:
   int GetYear();
   int GetDay();
   int GetMonth();
   int GetHour();
   int GetMinute();
   int GetSecond();
   int GetMilliSecond();

   // Other services:
   int GetTotalDays();                 // Returns the number of complete days.
   int GetTotalHours();                // Returns the number of complete hours.
```

[7] In order to save space this paper does not contain the sample code for the implementation of the CTime class and the class declaration and the implementation of the CTimeSpan class.

```
   int GetTotalMinutes();              // Returns the number of complete minutes.
   int GetTotalSeconds();              // Returns the number of complete seconds.
   int GetTotalMilliSeconds();         // Returns the number of milliseconds.
   static CTime GetSystemTime();       // Retrieves the current system time.
   BOOL IsLeapYear() const;

private:
   // Attributes:
   short       m_year;
   char        m_day;
   char        m_month;
   char        m_hour;
   char        m_minute;
   char        m_seconds;
   short       m_milliSeconds;
};
```

Smalltalk

Here is the sample code for the class definitions in Smalltalk[8]:

```
Magnitude subclass: #Time
   instanceVariableNames: 'hours minutes seconds '
   classVariableNames: 'EraClockValue LastPrimMillisecondClockValue MillisecondsInEra
                        RolloverProtect '
   poolDictionaries: ''
   category: 'Magnitude-General'

Magnitude subclass: #Date
   instanceVariableNames: 'day year '
   classVariableNames: 'DaysInMonth FirstDayOfMonth MonthNames SecondsInDay WeekDayNames '
   poolDictionaries: ''
   category: 'Magnitude-General'

Object subclass: #Timestamp
   instanceVariableNames: 'year month day hour minute second millisecond '
   classVariableNames: 'FirstDayOfMonth '
   poolDictionaries: ''
   category: 'UIBasics-Support'
```

Known Uses

Time as a Type can be found in numerous class libraries and runtime libraries such as MFC[9], ATL[10], C-runtime library, just to name a few. The C-runtime library for instance defines time_t which is simply a long integer representing the number of seconds elapsed since midnight January 1, 1970.

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, which means it must be guaranteed that for each call of the time function the resulting value is newer than all values for preceding calls.

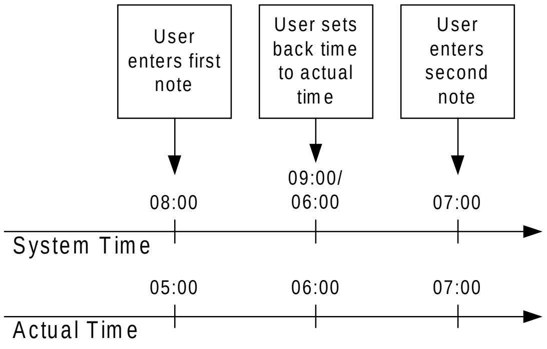
[8] Thanks go to Stéphane Ducasse for providing the Smalltalk source code.

[9] MFC = Microsoft Foundation Classes

[10] ATL = Active Template Library (another Microsoft product)

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. Nevertheless, 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 system time is set to the wrong value (3 hours later) and the first note is entered at 05:00 PM actual time (= 08:00 system time) and a second note is entered at 07:00 PM actual time.



Item	System Time	Actual Time
1st note	08:00	05:00
time change	09:00/ 06:00	06:00
2nd note	07:00	07:00

At 06:00 PM actual time (= 09:00 system 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. This is then a later system time as for the second note (See table in figure).

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 note that is associated with 06:30 PM actual time? 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 that for legacy reasons the application has to keep track on when the system time has changed and by what amount. This, too, can't be solved using a serial number, and there, a timestamp is needed 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. You need to sort items by time despite the possibility that the user may manipulate the system time.
2. Your application needs a series of time stamps that is 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

+ Using the TimeServer you are now able to sort items by time, independent of changes in the system clock. The TimeServer provides you with a list of delta values between the system clock associated with the item and actual time then.
+ 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 been 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 run 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:

Sample: Under Win32 Windows Messages are being sent to processes in case the time setting has been changed by the user.

Known Uses

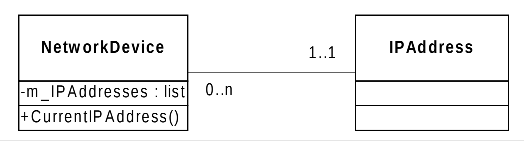
The Time Server pattern is used in the HP TraceVue Series 50 product. This system is used during the birth of a child and tracks among other parameters, the heart rate of the fetus.
For network support purposes HP has a number of applications in production and development that make use of the Time Server as well.
History Provider uses Time Server to create a series of timestamps, which are guaranteed to be sorted in ascending order.

3. Time-Value Pairs

Motivation

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

Sample: When a network device such as a notebook is attached to a LAN, it may have a different IP address depending on to which network it has been connected.
To track the current IP address of such a device, the following model may be employed:



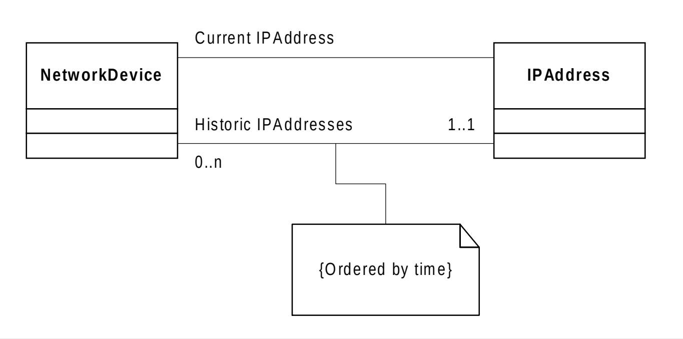
Forces

1. You need to track the history of one or more attributes of an object.
2. You want to consume as less memory as possible for the history.
3. You want to undo changes to a value of an object attribute.

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 collection, which is ordered by time.

> Sample: Each network device may have different IP addresses over time, e.g. when it is moved from one subnet to a different subnet.
> To keep track of the IP address a list of IP addresses is added to class NetworkDevice. The list is ordered by time. The last element in the list is the most current one.
> The variable CurrentIPAddress (see following figure) always contains the value of the last element in the list.



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 a network device object tries to get a value for a time that is older than the oldest entry in the HistoricIPAddresses 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 a network device object tries to get a time-value pair that does not exist?
   Two major solutions are possible here. Which to choose depends on the requirements of the application:
   First, a Null Object may be returned here or an exception may be thrown. This solution might be appropriate for instance for a weather application that has the temperature of a location for two non-consecutive days. The application cannot simply return the previous value for the dates that are between the two days, as that value normally is not correct.
   The second possibility is that the object returns the closest time-value pair or the time-value pair with the previous timestamp.

   > Sample: A network device has an IP address of 192.13.14.15 on February 1. At July 1, the IT department moves the network device to a different subnet and the IP address has to be changed to 192.13.200.54.
   > If somebody wants to know what the IP address was as of April 1, then the correct value would be 192.13.14.15, although no time-value pair for April 1 exists.
   > If the same client queries the IP address for January 1, the result would be a Null-Object or an Exception. Whether you return a Null-Object or an Exception depends on the programming language you use and on the idioms you prefer.
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, the object can handle this the same way as described in item 2 of this list.

Consequences

- + You can access the current value in the same way, as before, so no change is necessary to the code of collaborating objects.
- + You can undo all changes to the value.
- + You save resources, as only changes are stored.
- + You can retrieve the value of any time in the past, even for a time, where no time-value pair is available. In the latter case, you would just return the previous value, e.g. the address at 08:00 was 1.1.1.1, and the address at 09:00 was 2.2.2.2. When querying for the value at 08:30 the return value would be still 1.1.1.1, as this value is valid from 08:00 to 09:00.
- – You have to provide an additional interface with an in-parameter for specifying the time. The interface of the class becomes a little bit more difficult to use.
- – You have to implement the list and the correct adding and retrieving of time-value pairs.

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 update the cached value.

You may implement the ordered collection of historic salaries such that the collection contains the values themselves or, so that the collection holds references to the value objects. Depending on how the classes are used one of the two may be faster.

Known Uses

Several HP network support applications use this pattern.

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 span associated with them, ordered by time, and having the constraint that time spans must not overlap.

The difference between the proposal of Martin Fowler to the pattern presented here is minimal. Instead of associating a time span as Martin Fowler proposes, this pattern uses a time. The advantage is, that the constraint regarding overlapping time spans 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 come into play, e.g. also monitoring who changed an attribute.

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 creates a series of timestamps sorted in ascending order.

4. Versioned 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 several attributes one after the other may lead to intermediate object states where only some of the attributes are changed. These intermediate states are of no interest to the application.

Assume that a class provides two functions f1 and f2. Function f1 changes only attribute a1, and f2 just changes the attribute a2. Assume that at the beginning, the attribute a1 has the value v11 and a2 has the value v21. After the client has called both functions a1 has the value v12 and a2 has the value v22. f1 is called first. f2 is called next. The order of the states of a particular object can be represented by the following series (read s(a,b) as "state consists of values a and b"):

s1(v11,v21) • s2(v12,v21) • s3(v12,v22)

If the object would be persistent, all three states may still be valid. However, from your applications point of view only the states s1 and s3 are interesting. The client does not want to keep the intermediate state s2.
Generally, when calling a method on an object, the object makes a transition from one consistent state to another.

S1 • S2 • S3 • S4 • S5 • S6 • … • Sn

For your application, you are only interested in some of these consistent states, e.g.

S1 • S3 • S4 • S6 • … • Sn

Sample: A network device has two attributes associated with it: IP address and hostname[11]. The object is persistent in a database. The database schema allows both the IP address and the hostname to change independently, as both may happen in practice.
The network device (assume it is a workstation) is moved to a different department within the company. There it is set up from scratch using a new IP address and a new hostname. For the application, it is still possible to identify the machine by using its serial number.
The implementation allows setting both attributes independently, but after the workstation has been set up, both attributes have been changed, although two function calls are necessary to reflect this in the application.
It does not make sense to store only one of the changes to the database although the database may allow it (as it is allowed by the schema). Only the application knows about the fact that the intermediate state (only one of the attributes has changed), is not a valid one.

Forces

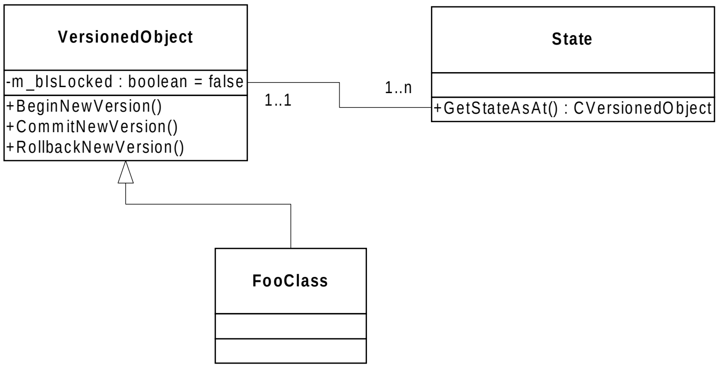
1. You want to keep only select states of an object in your history.
2. You want to keep consistency of your object even when multiple attributes are changed.
3. You want to track select states of objects of different classes.
4. You want to maintain encapsulation of the tracked objects.
5. You want to add the tracking "after the fact".

[11] Note, that hostname in this context does not refer to mainframes. For networks it stands for a device that runs a particular network protocol. The hostname a human readable name for that device.

6. You cannot use the transaction mechanism of the underlying database system, as it does not know about the semantics of your application.

Solution

Note: The solution presented here is orthogonal to transaction mechanisms of database systems.
Add a versioning mechanism to the object. The solution serializes all versions. The process of creating new versions obeys the ACID principle of transactions. ACID is the abbreviation for Atomicity, Consistency, Isolation, and Durability.
Atomicity means, that for each transaction either all or none of the changes in the transaction remains, when the transaction ends. The transaction can end either successful or it may be aborted by the user or the system, e.g. because of a deadlock.
If a transaction changes the database then the new state must be a correct state of the database. This is called *consistency*.
If none of the transactions is aware or affected by any of the other transactions then the transaction is *isolated*.
When a transaction has completed all of its changes should persist even in the case of a system failure. This property is called *durability*.
Each collaborator accessing a versioned object starts a new version by calling BeginNewVersion(), calls methods on that object and commits the version. At any time, only one collaborator can create a new version of an object.[12]



Beginning a new version means locking the object for both, read-access and write-access. The implementation of CommitNewVersion() or RollbackNewVersion() removes the lock.
If a second collaborator tries to access an object for which a new version is currently under construction, then the second call to BeginNewVersion() is blocking until the first collaborator commits its new version.
A versioned object may have references to other versioned objects. When a client starts a new version the versioned object is locked. The client does not automatically lock referenced objects. It is up to the implementations of the versioned objects methods whether and when to lock other versioned objects. If a method wants to access a versioned object, the method implementation has to start a new version, use the referenced object and finally commits the new version.
The solution can be enhanced by adding a parameter to the BeginNewVersion()-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.
Deadlocks can occur, if multiple threads try to create new versions of the same object at the same time. This can be solved with a simple solution: Implement BeginNewVersion() as a blocking call with a timeout. When the time out happens, the client can try to create a new version again, if he decides to do so. Although this approach does not

[12] Allowing the creation of multiple versions at the same time creates the problem of merging these versions into one consistent version. The solution described here does not cover this extension.

guarantee that a particular client will successful create a new version, for most cases it is sufficient. If this approach is not sufficient, you need to implement a mechanism for coordinating locks, e.g. by having a lock manager class. This latter solution is not covered is this paper.

Consequences

+ Objects always have consistent states, even when used by callers executing from different threads
+ You can use smart pointers to ease the use of versioned object. The smart pointer hides the versioning-related code in its implementation.
+ System resources only limit the number of state changes.
+ While a client creates a new version of a versioned object, the client can be sure that no other client changes the state of that object.
+ If a Versioned Object refers to other Versioned Objects, then the Versioned Object locks the referenced objects as well, if the Versioned Object wants to do this.
– Classes of Versioned Objects are harder to implement than normal classes.
– Versioned objects perform slower because of the additional overhead.
– A client can access a versioned object only in the context of a new version. This also applies to read access.
– The proposed solution does not work if creating a new version takes a long time (check-out, check-in model).

Implementation Issues

1. A common base class can implement the versioning mechanism.
2. When you implement the VersionedObject class, you also have to implement a locking scheme.
   A simple locking scheme could be that only one client could create a new version for an object, e.g. by using a semaphore or a mutex. If a new version for a particular object is currently being created, then a successive call of BeginNewVersion() on this object would be a blocking call.
3. The current state can be stored upon BeginNewVersion() or upon CommitNewVersion(). The former stores the state of the object even if the state does not change. The latter prevents this, but duplicates the current state.
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 BeginNewVersion(). If the time out occurs, the function would return with an error state.
   If the time-out solution is not sufficient, a lock manager class may implement a more sophisticated locking scheme. This paper does not cover the lock manager.

Known Uses

Multiple HP network support applications use this pattern.

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 versioning.

5. Timed Reference

Motivation

In general, an object is access through a reference[13]. A client can call public members of the object through this reference. In other words: Associated with the reference we have an object with a particular (the current) state (this also encompasses the identity).

If multiple states of the same object are kept in a history, then a traditional reference is not sufficient as from a conceptual point of view it always points to the current version of the object (previous states are discarded). How must a reference be extended, that it also supports referencing of previous states of the object it refers to?

> 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 application retrieves the following attributes for the NIC: "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, later, the application retrieves these values again and updates the state of the NIC object, the NIC object stores its previous state, as it is a versioned 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

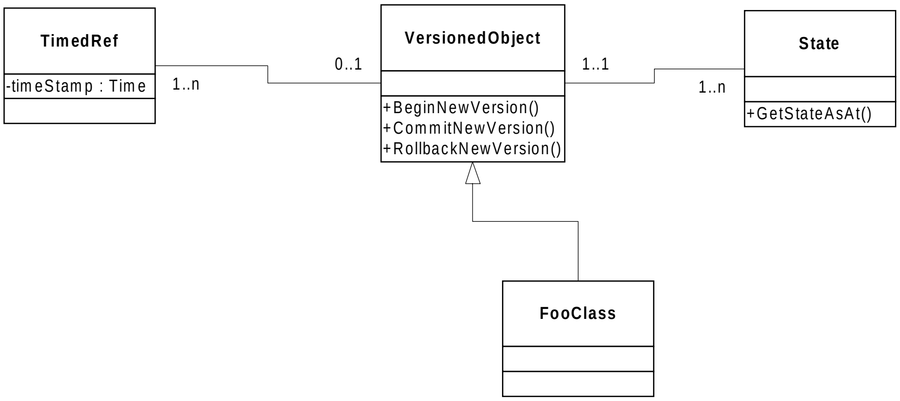
1. You want to use a Timed Reference similar to a normal reference.
2. You want the reference to also support historic states of a particular object.

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 instances 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.

[13] In order to avoid language specific terms, the term 'reference' refers to language specific terms such as reference, pointer, handle, etc.



Consequences

- + You can use TimedRef the same way as normal references. The learning curve is flat.
- + You can navigate to older states of an object.
- – TimedRef adds a level of indirection, which adds overhead. By adding a further level of indirection, TimedRef adds overhead to your implementation.

Implementation Issues

1. Some programming languages such as C++ support operator overloading. In this case, you can implement the TimedRef class 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.
   If the implementation language does not support operator overloading you can achieve a similar effect by implementing a class, that delegates the execution to a different object.
2. It might be possible to derive the TimedRef class from an already existing smart pointer class.
3. If the programming language supports it, you can implement the TimedRef class as a template, with the class of the referenced object as a template parameter.

Known Uses

Multiple HP network support applications use this pattern for their implementation.

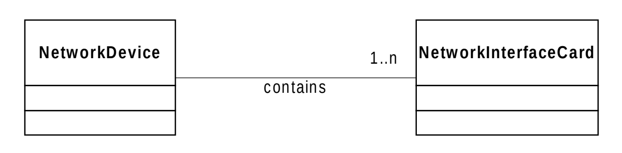
6. History Provider

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 (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, a support engineer can turn off an interface of a router during runtime, replace it, and initialize it again. The following class diagram shows a possible model::



If both classes were subclasses of VersionedObject, then a client would have to update their attributes in the context of creating a new version of the object. Modification of attributes may include, removing an interface (this is a modification of the attribute representing the collection of references to the NICs of the router), and changing the values for "BytesIn" and "PacketsDiscarded". Each object would keep it’s own history as an ordered collection of states.

First, the application detects a removal of an interface. The application starts a new version for the NetworkDevice object, calls the function to remove the interface, and finally commits the new version. Next, the application starts a new version on the NIC object, calls the methods in order to change the attributes "BytesIn" and "PacketsDiscarded", and finally commits the new version.

Both state transitions usually 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 versioned objects cannot solve the problem of ensuring the consistency of the history of several objects related to each other.

Forces

1. You want to track changes not only of objects but also of associations between them.
2. You want to navigate also back in time for a particular object. You want to switch between the normal navigation and the navigation back in time.
3. You want to track, when an object was deleted.

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, an array represents the objects and their states:

Time ObjectId	00:10	00:15	00:25	00:27
0x091	▨	▨	▨	▨
0x653	▨	▨	D	
0x438		▨	▨	▨
0x705		▨	D	

▨ Object State

The array has two dimensions: one dimension is the object identifier, and the other dimension is time. In other words: each row in the table represents the history of the object's state, and each column represents the set of states that is the result of a transaction.

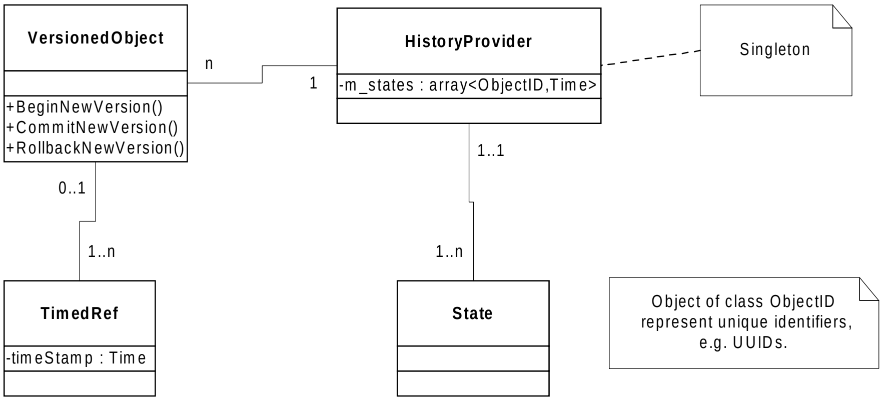
A gray square indicates that a state for the object with that ID exists at that time. Letter "D" in a gray square indicates that the object has been deleted at that time.

Note that it is not necessary, that the time spans 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.

The solution handles time gaps as follows. If the client requests the version of an object for a time, that is not in the table, then the history provider will return the version that was the last one before that requested time. Here, the assumption is, that all objects change their state continuously. There is no point in time, where they don't have a state at all, and each state remains the same until the object transitions to a new state.

The structure of the pattern looks as follows:



Consequences

+ You can navigate traditionally, but also back in time for any of the Versioned Objects.
+ You can easily change between the two navigation methods (traditional versus back in time).
+ You can use the history provider (almost) transparently, when you also use Timed References.

+ You can access your objects in a normal way, if you are not interested in the history feature.
+ The implementation overhead is of minor influence, if the implementation also uses transactions of a database. Here the underlying mechanics become the bottleneck very soon.
+ The history provider also handles time gaps, meaning that for times between two versions, the history provider returns a valid state.
– You must carefully consider whether to apply this pattern, as it is very complex. It is not only difficult to understand, but also difficult to implement.

Implementation Issues

The solution shows just the conceptual view. In order to reduce the amount of memory needed, you may implement the history provider in a way so that states of objects are stored only, when they have changed. In other words: If there is no delta between two successive states of an object, the older state will be stored only.
The overhead created by this pattern is of minor influence, when you apply it to persistent objects. The reason is that the underlying transaction mechanism of the database in conjunction with the physical access of secondary storage is using a considerable amount of processing time. If you apply the pattern to in-memory classes, the overhead may have a significant effect on the performance.

Known Uses

Multiple HP network support applications make use of this pattern.

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

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

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, Kent Beck, and my shepherd for the PLoP 98 conference Christine "Christa" Schwanninger 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.

Thanks go also to the participants of the writers workshop at EuroPLoP 98, who generated many additional suggestions for improving the paper.

In addition I want to thank Stéphane Ducasse for providing me with the Smalltalk sample code for the "Time as a Type" pattern.