Temporal Patterns

Andy Carlson, Sharon Estepp, Martin Fowler

3 August 1998

Abstract:

Objects do not just represent objects that exist in the real world; they often represent the memories of objects that once existed but have since disappeared or projections of how they are expected to exist in the future.

This presents a particular modeling challenge because something which appears straightforward at a point in time becomes far more complicated when the model must consider how objects change over time.

This paper presents 3 patterns which show how this problem can be addressed by elaborating the object model and how the resulting model can support clients which are not concerned with the temporal aspects.

Pattern Name: *Temporal Property*

Aliases: Historical Mapping

Context

You are building a complex information system in which a property of an object must be able to change over time. You need to be able to track how this property has changed or is expected to change (or both). The nature of the property is such that it holds a number of discrete values for intervals of time (as opposed to properties such as temperature which can change continuously). You may also need to implement the system using some form of database, possibly relational.

Problem

When considering how an object changes over time, we are usually concerned with how its properties are changing. These properties may be attributes, relationships or query operations.

For example, consider the model shown in Figure 1.

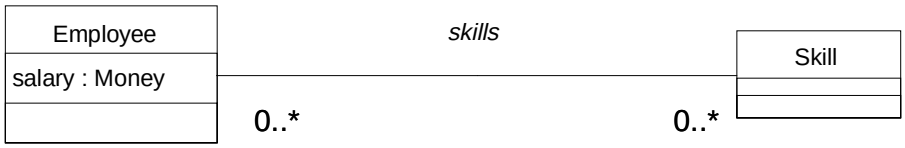

Figure 1 Basic Employee model.

In this model we are concerned with two properties of the employee, namely his salary and his skill set. The resulting Employee interface would look like Figure 2

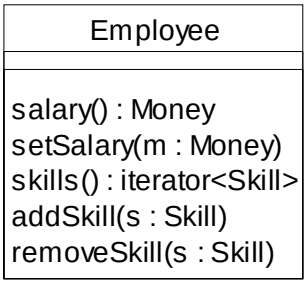

Figure 2 Basic Employee interface.

This does not preclude a similar interface in the Skill class if that is useful.

Now we will look at how this will need to change in order to allow us to record past salaries and skill sets or represent future salaries and skill sets.

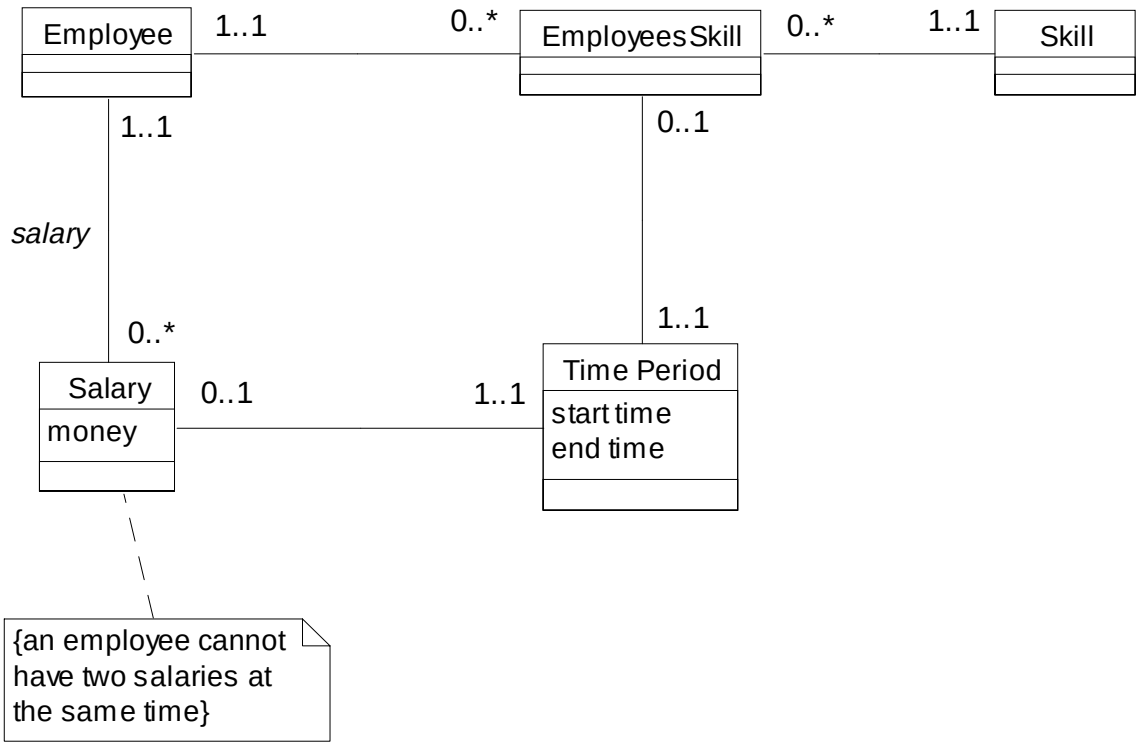

Figure 3 Employee model enhanced to represent changes over time.

As shown in Figure 3, we have now attached Time Periods to the properties of Employee. In the case of Salary, we have had to create a separate class to hold the amount with an associated Time Period to indicate the period of validity. We have thus allowed for multiple Salaries to be associated with the same Employee (at different points in time). In the case of Skill, this is a class whose instances may be shared by many Employees, so it is necessary to introduce an intermediate object to hold the Time Period information.

Using this model we can represent the following information:-

- Dinsdale Piranha starts with a salary of $75,000 on 1st January 1998 and is promoted on 1st April 1998 which results in a salary of $85,000.
- Dinsdale attends an assertiveness course on 1st March 1998 and as a result, adds this to his skill set.

Now imagine that we wish to know what Dinsdale's salary was on 1st February 1998. We can do this by examining all of Dinsdale's Salary objects to find the one which includes the date which we are interested in. We can do something similar for the EmployeesSkill objects to find out what Dinsdale's skills were on 1st February. In this case, we are likely to find more than one Skill as a result.

We can also assign Dinsdale a future salary (say, for his annual pay review on 1st January 1999) simply by creating a new Salary object with a time period beginning on 1st January 1999.

We have now got the expressiveness that we need but at a cost:

- We have now gone from 2 classes, an association and an attribute to 5 classes, 5 associations, and a constraint.
- In order to answer the question for any point in time (including 'now'), we now need to retrieve and process a lot more objects.
- We have lost the clarity of the original model, particularly where cardinality is concerned : we can no longer tell at a glance whether an Employee can have one or multiple salaries at a given instant.

In summary, the temporal aspects of the model have overwhelmed the other aspects. In reality, systems will have far more temporal relationships than we have shown here. We leave it as an exercise for you to decide on the number of classes, attributes and associations required to support this.

Forces

- **Simple models are more appealing** - Developers often have a desire to 'keep things simple' when creating a design by minimizing the number of classes and relationships and/or not concerning themselves with the temporal aspects.
- **Simple models are sometimes not enough** - There is often a requirement for change tracking (past or future)
- **Complex models may be less clear** - The essence of the model may become hidden by the temporal aspects.
- **Impact of changes** - The representation of the relationship may change, incurring a cost in changing any clients which depend on knowledge of the representation.

Solution

For properties for which you need to track changes over time, build a model (such as the one in the Problem section) which can represent the validity period of each property value. There are a number of alternative ways to implement this pattern so it is not possible to include a single abstract structure diagram without the use of a stereotype (as shown later). The implementation section contains some alternative models.

Having decided on a model, make it easier for clients to navigate it by hiding it behind a convenient interface of the class in whose properties we are interested. Provide support for clients which are not concerned with the temporal aspects by adding methods which do not require a time parameter and assume a default of the current time. This interface is shown in Figure 4.

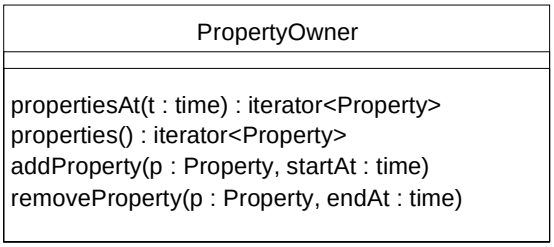

Figure 4 PropertyOwner interface

If you are using a notation which allows it (such as UML as shown here) the model structure can now be made more concise by adopting a stereotype which we will call «temporal» as shown in Figure 5.

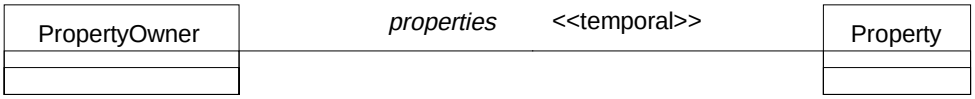

Figure 5 *Temporal Property* structure with stereotype.

In the case of our example, the salary and skills properties would be accessed through an interface like the one shown in Figure 6.

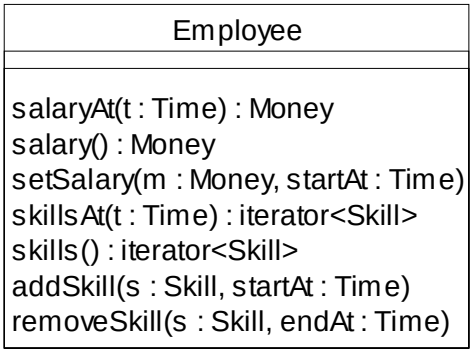

Figure 6 Employee Interface using Temporal Properties.

As in the problem section, this does not preclude a similar interface in the Skill class if that is useful.

By adopting the «temporal» stereotype, the complex diagram shown in the problem section then becomes much simpler as shown in Figure 7

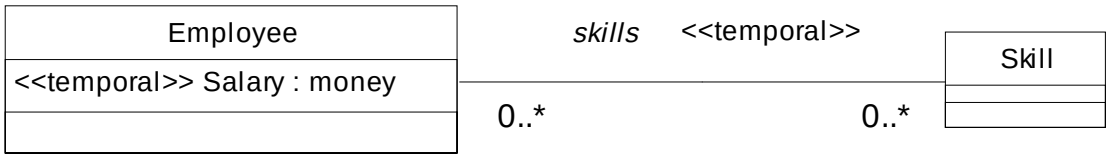

Figure 7 Employee Model using stereotype.

Consequences

- ✓ We have added the capability to represent changing properties over time to the original model
- ✓ By using the «temporal» stereotype to express those parts of the solution which have been added to support the representation of time we have regained the simplicity of the original model.
- ✓ Along with the simplicity, we have also regained the expressiveness of the original model – we can now tell from our example model whether an Employee can have more than one Salary at once.
- ✓ Placing the majority of the knowledge about the relationship's representation in the PropertyOwner class ensures that other clients do not become tightly coupled to this representation.
- ✓ Using this pattern implies more regularity in the modeling of objects which have temporal properties.
- ✓ Future values will become current values and later on turn into historical values simply by the passage of time and without the need to update any stored information.
- × If you need more information abut the intermediate object in the relationship (for example, a skill level in EmployeesSkill class), this indicates the use of a first class object for the intermediate object and would indicate using the *Temporal Association* pattern

Implementation

The simplest implementation for this pattern involves creating a relationship object (PropertyAllocation) with a Time Period between the PropertyOwner and the Property as shown in Figure 8.

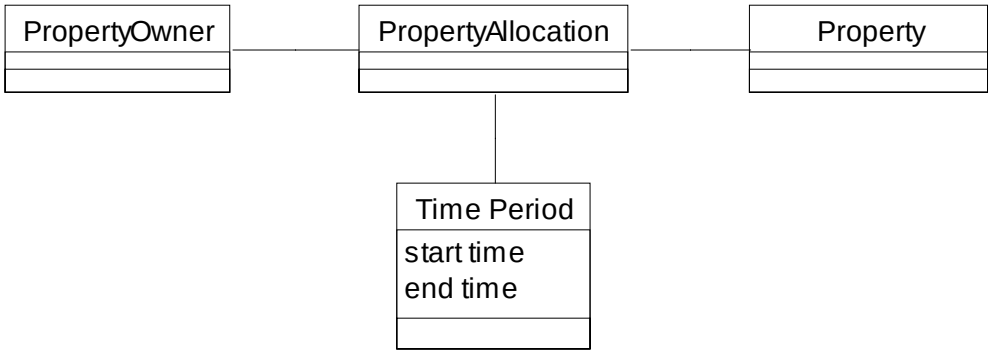

Figure 8 *Temporal Property* implementation using PropertyAllocation object

Where you are sure that the instances of the Property class cannot exist without their PropertyOwner (and also that there is only one PropertyOwner class, the Time Period may be attached directly to the Property object (DedicatedProperty) as shown in Figure 9.

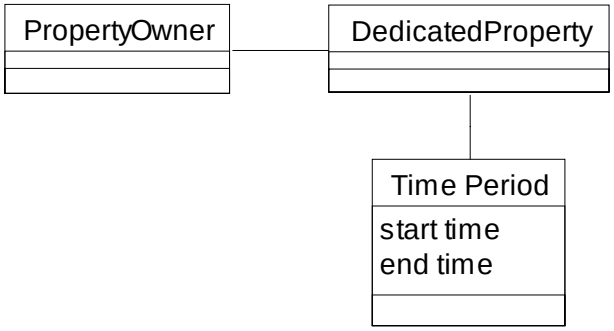

Figure 9 *Temporal Property* implementation for dedicated properties.

Alternatively, you may find it useful to implement this pattern using a Temporal Collection class specially written for the purpose. This type of implementation is best suited to situations where relational databases are not involved.

For relational database implementations you will often need to provide a class and corresponding table to hold the information required to represent the relationship. This can be avoided in situations where dedicated property implementation can be used. In this case you can include the time fields in the property class/table. A further simplification is possible where the cardinality between owning object and a dedicated property object at any point in time is one-to-one. In this situation you can assume the end time of one (older) property to be the start time of the next newer one and dispense with storage (and maintenance) of the end time.

Known Uses

The current implementation of the AT&T Rialto™ suite uses temporal patterns to represent several relationships with more to follow in future as the model expands into new areas.

The relationship between User and User Identity illustrates the dedicated form of the *Temporal Property* pattern as shown in Figure 10.

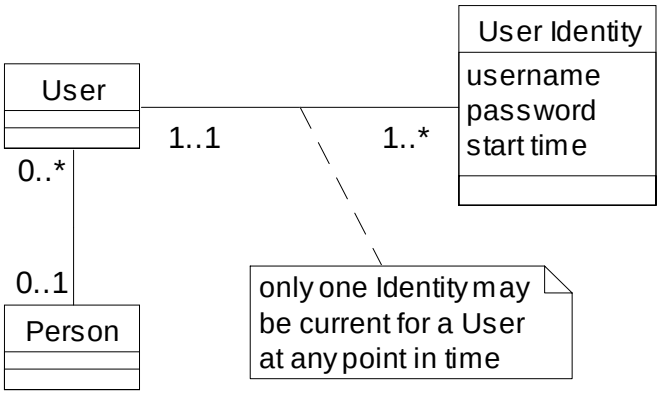

Figure 10 User and Identity model.

Holding User Identity in a separate class from User allows, for example, a User to change his or her username without the system losing track of the fact that he is still the same user and therefore that any information attached to the user (e.g. accountability or privilege information) remains unaffected. This is an example of the simplification mentioned in the Implementation section as the User Identity is assumed to have no end date until another is created with a later start date.

By adopting the Stereotype this becomes the model shown in Figure 11.

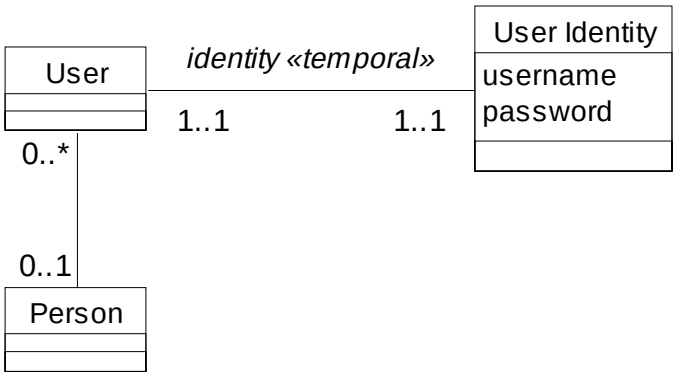

Figure 11 User and Identity model with stereotype.

The model is now expressing the information which required a note in the first diagram.

The representation of the validity of a particular Tariff as the current pricing information for a Price Plan also illustrates the dedicated form of this pattern as shown in Figure 12.

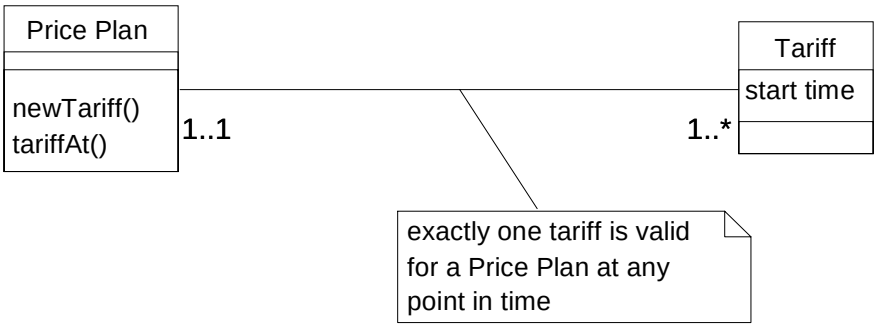

Figure 12 Tariff Model

By adopting the stereotype, this becomes as shown in Figure 13.

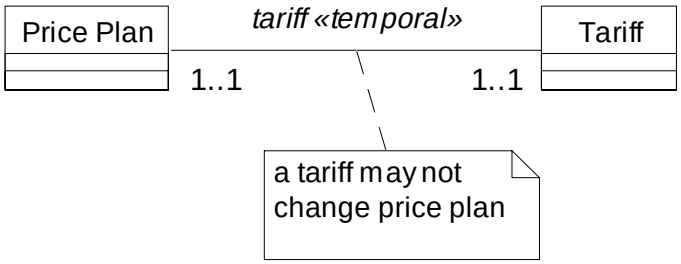

Figure 13 Tariff Model with Stereotype

Note that in this case, the need for a note does not go away but its content does change. We can now see the situation at any point in time by looking at the cardinality of the relationship. The need for a new note arises because of the bidirectional nature of association. Without the note, we cannot tell that the *temporal property* does not work in both directions.

Related Patterns

This pattern is a development of Martin Fowler's *Historical Mapping* pattern [FOWLER97].

The Time Period abstraction is an example of the use of Martin Fowler's *Range* pattern [FOWLER97].

This pattern is closely related to *Temporal Association* and when you find yourself faced with a problem of this nature you should consider carefully which is the correct one to use. *Temporal Association* is concerned with situations where we are interested (from a modeling perspective) in the intermediate object in the relationship as opposed to the situation here where we it is part of the implementation (if it is needed at all). Note that, as in the example here, mere presence of an intermediate object does not preclude considering the use of *Temporal Property* as the intermediate object may be concerned only with implementation (for example representing a 'cross reference' table in a relational database)

It is quite possible that you will find situations where there are multiple relationships and/or properties which change over time. In these situations you may find that both *Temporal Association* and *Temporal Property* are needed together.

When using the *Temporal Property* you should consider the use of the *Snapshot* pattern if you wish to view one or more properties with the time and historical information removed. This may be particularly applicable if there are multiple *Temporal Properties* and/or mixtures of *Temporal Association* and *Temporal Property*.

Before using *Temporal Property* to represent future changes in value of a property, you should ask yourself how likely it will be that the property will actually hold the indicated future values. There are (at least) two possible ways to look at the future values of a property. The first is by considering that the future values will cause the property to take a new value at the appropriate moment in time, or put another way, that the future values can be used to determine a schedule of forthcoming property changes. The second way is to consider that the future values are an estimate or plan of what will happen. In this situation, there may be several alternative plans for the same property. The *Temporal Property* pattern is intended for the first alternative, that is that the future values will determine how the property changes over time. If you are faced with an estimating or planning problem, you should instead consider using patterns intended for that purpose like Martin Fowler's *Plan* [FOWLER97].

Pattern Name: *Temporal Association*

Aliases: Association Object

Context

You are building a complex information system in which relationships between some of your objects must be able to change over time. You need to be able to track how the state of these relationships have changed or are expected to change (or both). You may also need to implement the system using some form of database, possibly relational.

Problem

The problem addressed by this pattern is similar to that addressed by *Temporal Property*, namely how to add the representation of time to an existing model. In this case, however we are interested in more complex situations where there is always an intermediate object representing the relationship. This may be either a 'first class' object or a result of using the concepts known in OMT [RUM91] as 'Association as Class' and 'Link Attribute' and as 'Associative Type' in Analysis Patterns [FOWLER97].

For example, consider the model shown in Figure 14.

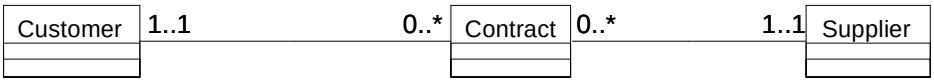

Figure 14 Contract Model

If we now wish to know, for example whether a contract has been terminated or has not yet come into force, we can achieve this by adding temporal information to the Contract itself as shown in Figure 15.

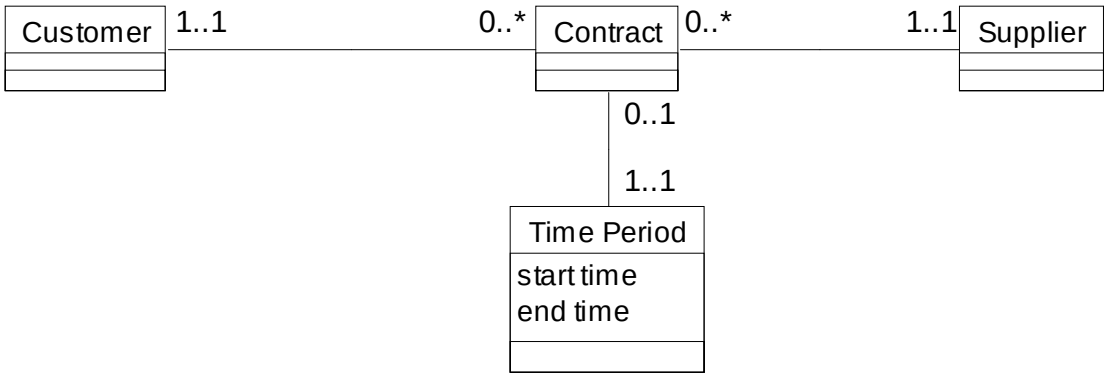

Figure 15 Contract Model with temporal information

At first sight this appears identical to the Employee-EmployeesSkill-Skill relationship from the example in *Temporal Property*, so it would seem reasonable to ask whether that pattern can be applied here. Clearly this would not work as the result would be the elimination of the Contract class from the model (when the stereotype is used) and its replacement by a direct relationship between Customer and Supplier. It is very likely that Contract carries other interesting information (such as billing data) and cannot reasonably be removed from the model.

The importance of the Contract class is reflected in the interface of the Supplier class as shown in Figure 16.

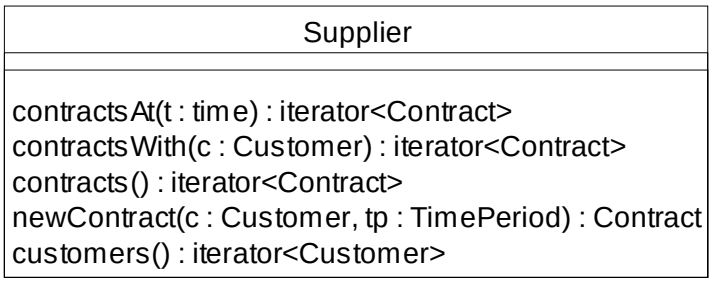

Figure 16 Supplier Interface

Presumably the Customer class would have a similar (but possibly less capable) interface.

In contrast with *Temporal Property*, we have not had to complicate the model to any great extent by adding the structure necessary to represent temporal information.

Forces

- **Simple models are more appealing** - Developers often have a desire to 'keep things simple' when creating a design by minimizing the number of classes and relationships and/or not concerning themselves with the temporal aspects.
- **Simple models are sometimes not enough** - There is often a requirement for change tracking (past or future)
- **Complex models may be less clear** - The essence of the model may become hidden by the temporal aspects.
- **Impact of changes** - The representation of the relationship may change, incurring a cost in changing any clients which depend on knowledge of the representation.
- **Symmetry is attractive** - It would be nice if there was a similar modeling convention to that adopted for *Temporal Property*

Solution

In contrast with *Temporal Property*, there is already a pre-existing intermediate object in the relationship for which we wish to track changes over time. There is therefore no need to introduce new classes or relationships, so add Time Period information directly to the intermediate object.

The resulting structure is shown in Figure 17

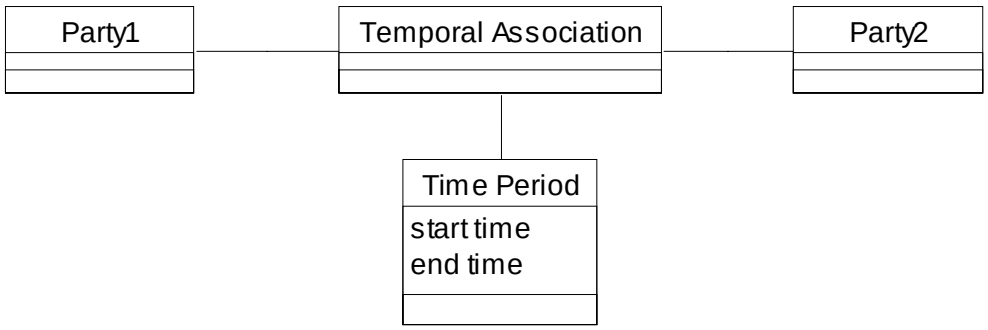

Figure 17 *Temporal Association* Structure

As the Association object has importance of its own, simplifying the model by adopting a stereotype is not useful.

Consequences

- ✓ We have added the capability to represent changing properties over time to the original model
- ✓ By combining the attributes of the Time Period object with the intermediate object in the relationship, the temporal aspects can be represented in a relational database.
- ✓ The 'ownership' of the time information is kept with the object to which it relates
- ✓ Knowledge of the relationship's representation need go no further than the objects at either end.
- ✓ The intermediate object retains its importance in the model by being exposed via the interfaces of the objects at the ends of the relationship.
- ✓ Future relationships will become current relationships and later on turn into historical relationships simply by the passage of time and without the need to update any stored information.
- × Less implementation flexibility than *Temporal Property*
- × There is no parallel with the stereotype adopted for *Temporal Property*

Implementation

In a relational database implementation, keeping Time Period as a separate object (and hence table) is unlikely to be useful. You can reduce the number of tables by combining the Time Period attributes with those of the intermediate class. Contract would then be as shown in Figure 18. This also has the effect of reducing the number of classes to that of the original model.

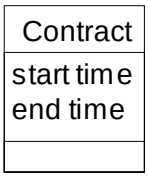

Figure 18 Contract with Time Period attributes combined.

In a relational database implementation where the start time is used as part of the key information, it is likely that start time must also be introduced to any entities which are dependant on the main table (e.g. where a contract is made up of a number of items) in order to ensure referential integrity. An alternative approach is to provide the main table with a column populated with an identifier number unique to each (main table) row and include this column as part of the key of the subordinate tables.

Known Uses

The Contract example quoted in the Problem section is taken (in simplified form) from the current implementation of the AT&T Rialto™ suite.

The Premium and Client Applications within S3+™, a property and casualty insurance product developed by Policy Management Systems Corporation (PMSC) uses the *Temporal Association* pattern for its underlying relational database. Effective date and expiration dates are included in intermediate objects such as policy, policy client association, policy insurance lines, and policy coverage.

Related Patterns

This pattern is a development of Andy Carlson's *Chronology* pattern which was discussed in a writer's workshop at ChiliPLoP 98 [CARLSON98A]. It is also similar to Lorraine Boyd's *Association Object* [BOYD97] pattern.

This pattern is closely related to *Temporal Property* and when you find yourself faced with a problem of this nature you should consider carefully which is the correct one to use. *Temporal Property* is concerned with situations where we are not interested (from a modeling perspective) in the intermediate object in the relationship either because there isn't one or because its presence is merely a result of considering a particular implementation.

It is quite possible that you will find situations where there are multiple relationships and/or properties which change over time. In these situations you may find that both *Temporal Association* and *Temporal Property* are needed together.

When using the *Temporal Association* you should consider the use of the *Snapshot* pattern if you wish to view one or more relationships with the time and historical information removed. This may be particularly applicable if there are multiple *Temporal Associations* and/or mixtures of *Temporal Association* and *Temporal Property*.

Pattern Name: *Snapshot*

Aliases: none

Context

You are building clients to make use of a complex information system in which properties of some of your objects and/or relationships between them must be able to change over time. You need to be able to track how the state of these things have changed or are expected to change (or both). Some clients are interested in how information changes over time while others are not.

Problem

There are many ways to introduce a representation of time into an object model. This may involve tracking discrete changes in properties (e.g. salary) or relationships (e.g. contracts) or continuous changes in properties (e.g. temperature). See the Related Patterns section for some examples. Having given ourselves the ability to represent time in our information model, we now wish to make use of it.

When considering how we make use of the model, it is convenient to separate those parts of the system with an interest in the information from the information storage (and representation) itself. A convenient way to do this is to consider the users of the information as 'clients'. These clients might be:-

- Complex information processing applications
- Other systems which require a periodic information feed from or interface to the model.

Let us consider again the Contract example from the *Temporal Association* pattern as shown in Figure 19. That pattern allowed us to represent the validity period of a Contract and answer questions like 'What contracts did supplier X have on 1st January 1998?'.

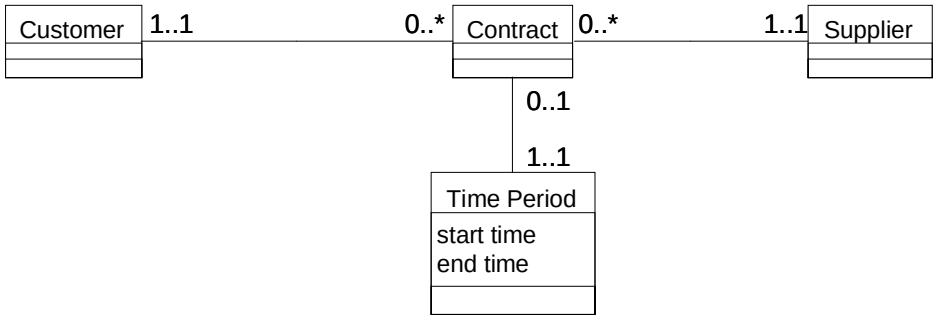

Figure 19 Contract Example

Let us now consider that we are writing a billing system. Obviously, ability to obtain information about contracts is an important requirement for the implementation of such a system. Typically, a billing system will need to refer to the contract at several points in its operation for a variety of information. Another important attribute of a good billing system is that it should not continue to send invoices to customers whose contracts have ended (or have not yet begun), so we would be wise to incorporate something like *Temporal Association* in our design.

Now assuming that we have implemented a rich temporal model for our Contract information and probably other properties and relationships (like the ability to change the billing address over time), we have also made life considerably more difficult for our billing system. Does the introduction of a temporal model now require our billing system to navigate through the temporal relationships to find the valid contracts for the point in time in which it is interested? Of course we can encapsulate this complexity behind convenient interfaces in our model (e.g. Supplier.contractsAt(time)) but the navigation processing must still be done, we have just moved the problem. If, as is often the case, different stages of the billing processing require

access to the contract information, we may find ourselves repeating the same complex navigation over and over again, despite the fact that in a single invoicing run, we are probably only interested in the state of all relationships at the same point in time, namely the billing period end.

There are many situations where the introduction of the time dimension complicates matters not only for the designer and developer of the model itself but also for the clients of the model. As we saw in the example above, one or more clients may wish to perform some complex information processing based on a particularly important point in time (for example billing period end or year end). Alternatively, it is possible that some clients (especially external systems) have a specific 'view' that they require of the information which may not include the time dimension (for example requiring a periodic 'feed' containing just the current information rather than a full history of changes).

Forces

- **Simple models are sometimes not enough** - There is often a requirement for change tracking (past or future)
- **Complex models may be less clear** - The essence of the model may become hidden by the temporal aspects.
- **Impact of changes** - The representation of the relationship may change, incurring a cost in changing any clients which depend on knowledge of the representation.
- **Performance** - There is a cost (which we would rather avoid) of repeatedly navigating temporal relationships to discover their state at the same point in time.
- **Unsophisticated clients** - One or more clients or external systems may be unable to receive historical data.

Solution

Modify the interface normally used by clients, by adding methods to allow provision of a new object called a *Snapshot* for a specific point in time. The *Snapshot* object provides a 'view' of one or more properties or relationships with the time dimension and historical information removed. The *snapshot* should provide the time to which it applies as one of its visible properties. The *snapshot* object may then be used over and over again by the client or passed on to other clients.

The structure of the *Snapshot* pattern for a *Temporal Property* is shown in Figure 20. Note that this pattern is equally applicable to other temporal patterns (see Related Patterns).

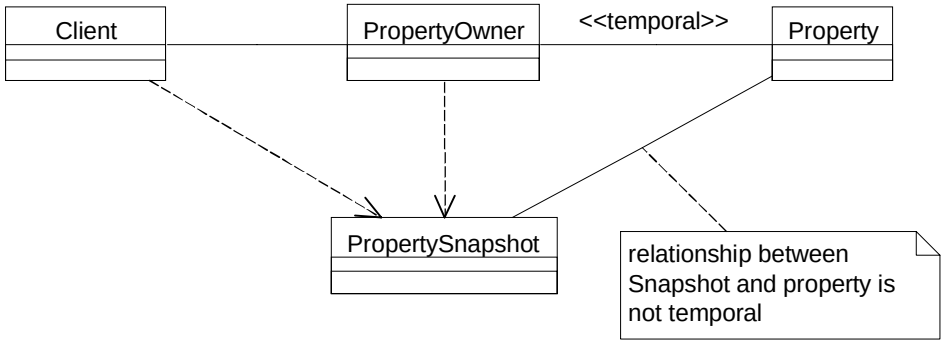

Figure 20 *Snapshot* structure

The collaborations are shown in Figure 21.

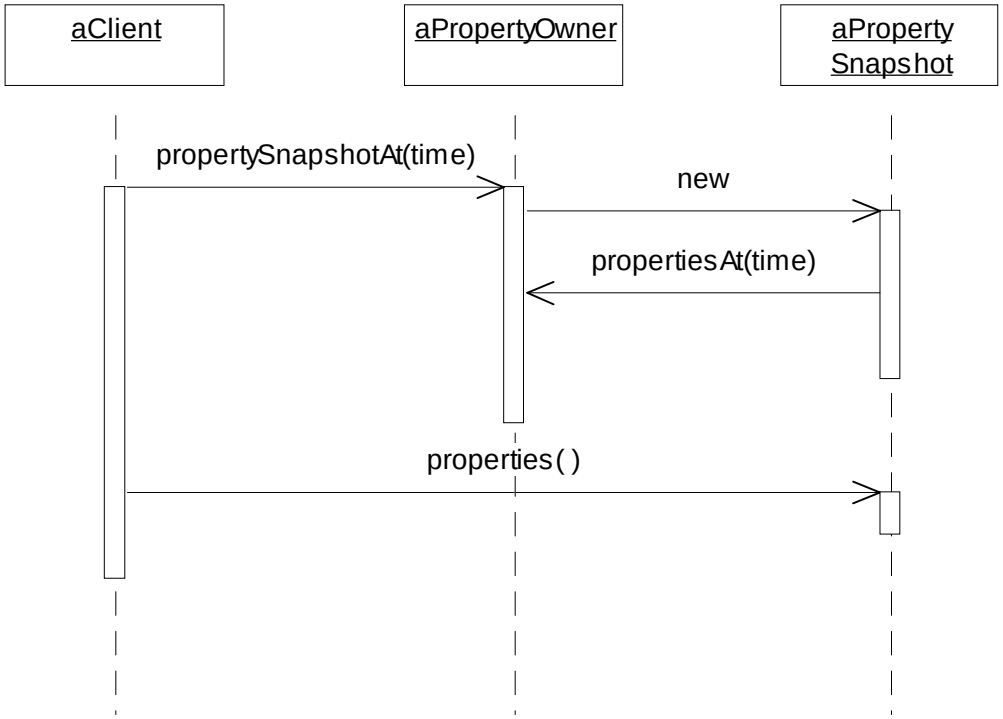

Figure 21 *Snapshot* collaborations.

The client first asks the owner for a PropertySnapshot, specifying the time. If the Property Owner does not already have a *Snapshot* for the requested time it creates a new one. The PropertySnapshot then requests the property values for the relevant time from the property owner. The PropertySnapshot object is returned to the Client which can then retrieve the properties repeatedly without the implementation needing to recalculate the valid ones. Alternatively, the client could pass the PropertySnapshot to another client which need not be aware of the temporal aspects.

In the case of a *Temporal Property*, the PropertyOwner interface would be changed by the addition of the propertySnapshotAt() method as shown in Figure 22.

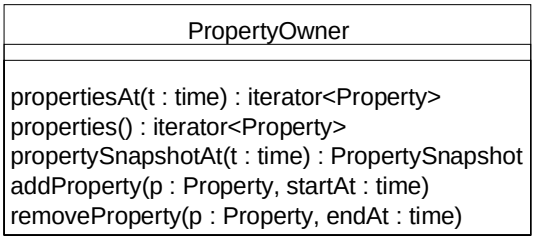

Figure 22 PropertyOwner interface with Snapshot

The PropertySnapshot interface would look like Figure 23.

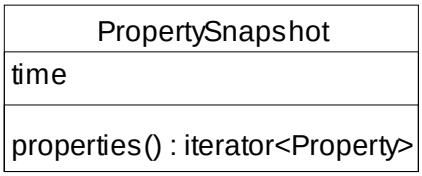

Figure 23 PropertySnapshot interface.

The PropertySnapshot interface is not providing any new operation signatures. There are, however, two important differences between these operations and their equivalents in the PropertyOwner class:-

- The implementation need not incur the cost of navigating the temporal information to determine the result.
- The PropertyOwner interfaces always apply to a default time (probably the current time) whereas a PropertySnapshot can be obtained for any time specified by the client.

To use a more concrete example (from *Temporal Association*), the Supplier interface would become as shown in Figure 24.

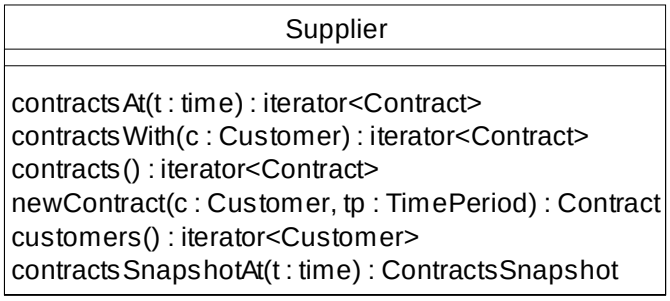

Figure 24 Supplier interface with Snapshot

And the ContractsSnapshot class might look like Figure 25. Note that in this case the *Snapshot* is supporting multiple query operations.

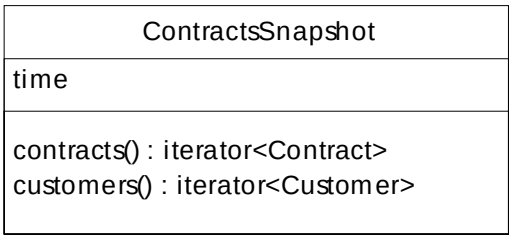

Figure 25 ContractSnapshot interface

Consequences

- ✓ Using *Snapshot* eliminates the cost of repeated navigation of a temporal model.
- ✓ The temporal model can be used by clients which do not understand or have no use for historical information.
- ✓ Simple interface signatures can be used by clients without restricting the meaning of these interfaces to some default time.
- ✓ Use of *snapshots* is optional – the same or other clients can still have access to the temporal information in the original model.
- ✓ If *snapshots* are saved, they can provide a permanent record of the information on which a client's processing was based independent of any later changed to the original temporal information.
- × An additional class has been introduced, partly duplicating existing functionality.

Implementation

If it is useful, you may also wish to implement some refinements to the basic *Snapshot* mechanism:-

- Caching of *snapshots* may optionally be carried out as part of the implementation of the interface which provides them to clients if it is likely that multiple independent clients will be expected to request *snapshots* for the same point in time.
- For properties or relationships which change in a discrete fashion, rather than having the *snapshot* store just its time, it may also be desirable to calculate a period of validity for the *snapshot*. This could dramatically improve the hit rate in the cache, depending on the typical duration between changes in the state and the chronological distribution of request for *snapshots*.

Known Uses

The AT&T Rialto™ Rater uses this pattern to 'freeze' complex temporal relationships at billing period ends. The main use at present is in the Tariff subsystem, which has multiple temporal relationships which allow entire Tariffs or component parts of them to change over time.

The Rialto™ suite also has a real-time network user authentication component called Access Control, which is designed with speed of access as a major goal. This receives information from a repository which has a rich temporal model. The historical information in the repository is of no interest to Access Control and navigating it in real time would reduce user authentication throughput considerably, so data is fed in *Snapshot* form to Access Control to ensure that Access Control always has a current picture of the authorized users.

The Claims Application within S3+™, a property and casualty insurance product developed by Policy Management Systems Corporation (PMSC) uses the *Snapshot* pattern for its underlying relational database. A *snapshot* of the policy related information such as policy coverage and insured object is taken as of the date of the loss specified on the claim. The Claims application then uses this *snapshot*; thus eliminating the need to navigate via the temporal relationships to access the policy related information that was effective at the date of the loss.

CyberLife® Administration, a life insurance policy administration product developed by CYBERTEK, a PMSC company, uses the *Snapshot* pattern for its underlying relational database. A *snapshot* of policy fund value information is taken as of the monthliversary date.

Related Patterns

This pattern is a development of Andy Carlson's *Snapshot* pattern which was discussed in a writer's workshop at ChiliPLoP 98 [CARLSON98B].

This pattern can be applied to most patterns which add representation of temporal information. Examples include *Temporal Property* and *Temporal Association.*

References

[BOYD97] L Boyd. *Business Patterns of Association Objects* from *Pattern Languages of Program Design 3*. R Martin, D Riehle, F Buschmann (eds.). Reading, MA: Addison Wesley Longman, 1998.
[CARLSON98A] Andy Carlson, *ChronologyPattern.* http://www.attlabs.att.co.uk/andyc/patterns/chronology-pattern.html
[CARLSON98B] Andy Carlson, *Snapshot Pattern.* http://www.attlabs.att.co.uk/andyc/patterns/snapshot-pattern.html
[FOWLER97] Martin Fowler. *Analysis Patterns : Reusable Object Models*. Menlo Park, CA: Addison Wesley Longman, 1997.
[RUM91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

Acknowledgements

We would like to thank our shepherd, Dirk Riehle for his helpful comments on this paper.

Trademarks

Keywords: time, temporal, history, relationship, property, snapshot