2D History

Versioning in the Presence of Retroactive and Future Changes

Andreas Rüping

Sodenkamp 21 A
D-22337 Hamburg, Germany

andreas.rueping@rueping.info

www.rueping.info

Introduction

Think of a family who have a health insurance contract. A new-born child is automatically covered by the contract from the moment he or she is born. But since parents have better things to do than call their insurance company the very moment the child is born, the company doesn't learn about the new child immediately. This should not represent a problem to the insurance company: they must be able to cope with a change that has already been in effect for a while at the time they are informed of it.

Or think of a discussion with your boss after you completed a difficult project successfully in mid February. Your boss might agree to give you a pay raise for the entire year. Congratulations! Hopefully your company is able to deal with retroactive pay raises, so that you get the increased pay for January.

Or imagine you are moving to a different place. You want to make sure that your mail reaches you safely, so you inform your bank well in advance that you are going to relocate. Your bank must be able to store your new address but must not use it yet, so as to make sure that no letters are sent to your new address before you even live there.

We can see from these examples that in some application domains, mostly in financial information systems, we need to distinguish between the moment a change becomes effective and the moment we learn about the change. This is what 2D history, and the patterns in this paper, are about.

Guidelines for the Readers

This paper present five patterns on two-dimensional history, ranging from concept to implementation level. The context of these pattern is defined by an information system (object-oriented or not) on top of a relational database.[1]

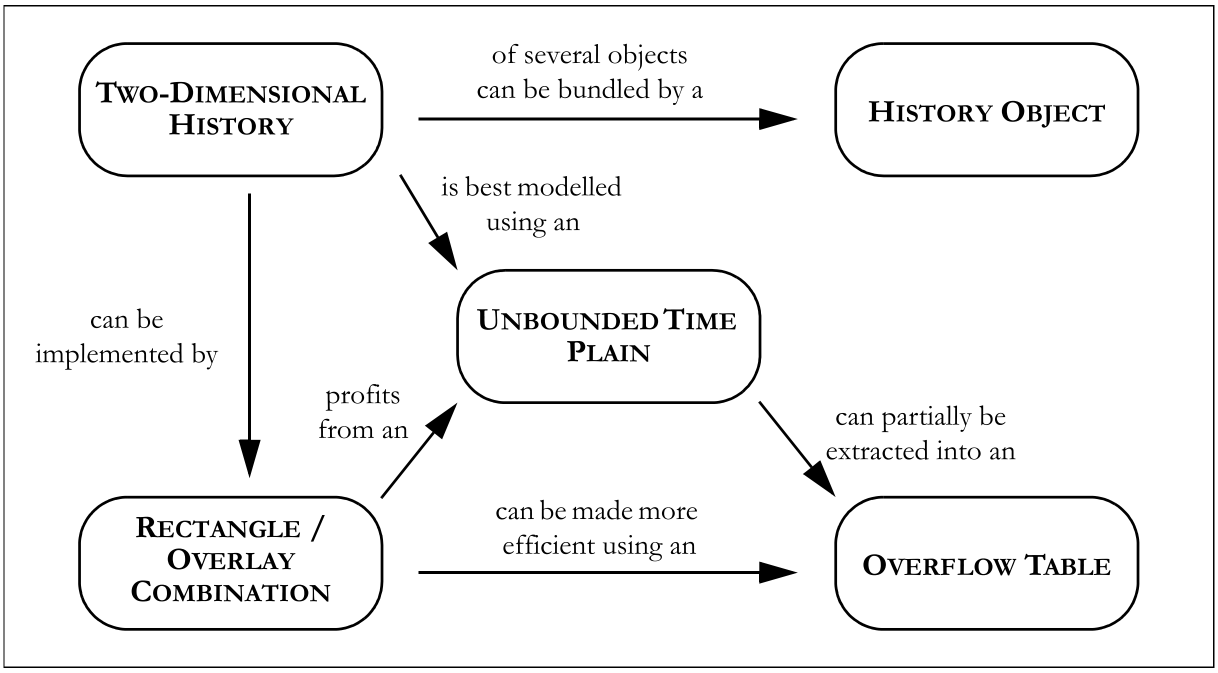
The pattern form used in this paper begins with a problem section followed by a discussion of forces. Next the problem as well as the driving forces are motivated with an example. Then the solution is presented, and explained in the example resolved section. We conclude with relationships between patterns and additional aspects in the discussion section.

The concepts of two-dimensional history seem rather mathematical and abstract at first. However, the examples will get you on the right track, and introduce you into the concepts quickly. It is therefore recommended you read the paper from beginning to end, including the example and example resolved sections.

The examples, however, cannot explain all the algorithmic details involved in two-dimensional history. The most important algorithms are therefore sketched as pseudo-code in the Appendix.

Once you have familiarised yourself with the basic ideas of two-dimensional history, you can use the problem section and the first paragraph of the solution section as thumbnails; they are printed in bold face for convenience.

The patterns are closely connected. The overview in Figure 1 shows how they are related through various kinds of relationships.



TWO-DIMENSIONAL HISTORY
of several objects can be bundled by a
HISTORY OBJECT
is best modelled using an
can be implemented by
UNBOUNDED TIME PLAIN
profits from an
can partially be extracted into an
RECTANGLE / OVERLAY COMBINATION
can be made more efficient using an
OVERFLOW TABLE

Figure 1 Roadmap to the patterns

1. The idea behind two-dimensional history is independent of any implementation technique. Most patterns in this paper, however, are influenced by implementation techniques and have to assume a relational database.

1 Two-Dimensional History

Context We design an information system in which modifications made to business objects will not necessarily become effective the moment they are entered into the system. There might be retroactive changes as well as changes that become effective only in the future.

Problem **How can time be modelled so that a distinction is made between the time an object becomes effective, and the time we learn about the object becoming effective?**

Forces Time-dependent data is often modelled using so-called time value pairs — tuples that associate a value (or an object version) with a certain period of time [1][2][3][4]. However, the time component in a time value pair merely describes when the value is in effect. Modelling with time value pairs tacitly assumes that the moment an object is entered into a system is the moment the object becomes effective.

However, this isn't necessarily the case, and time value pairs sometimes aren't an adequate means to describe time-dependent objects. Many financial information systems require to differentiate between effectiveness and knowledge.

This, however, leads to more data that will have to be stored, as well as to more complex algorithms for entering or retrieving versions. This additional complexity shouldn't be spent unnecessarily. Yet when the differentiation between effectiveness and knowledge is indeed necessary, we must find a way to model time appropriately.

Example Let's assume that a family buys a health insurance on January 1 which also becomes effective January 1. On February 1 the insurance company computes new insurance premiums which will be effective March 1. On March 1 the insurance company is informed that a child was born on February 1 that was covered by the contract since that day. The premium that had been calculated didn't take the child into account, so it is now void. Instead, when the inclusion of the child is being processed, the company calculates a new premium which is going to be effective March 1, based on the new information that the child is now covered by the contract as well.

This is a typical scenario, including both a retroactive change and a future change.

Solution **The core idea is to associate to an object not one time attribute, but two distinct time attributes whenever retroactive changes or future changes need to be modelled. One attribute represents the effective-at-time, the other the known-at-time of an object version.**

When we visualise this concept, object versions no longer sit on the time axis. As time becomes two-dimensional, there are now two axes representing effectiveness and knowledge.

An object version is now represented by an area on the time plain. Data access now looks as follows:

- To access an object version we need two time parameters: the effective-at-time and the known-at-time of the version we would like to retrieve.
- To add a version, we only need to know its effective-at-time. We assume that the version is known from the moment it is entered into the system, so its known-at-time will be assigned automatically.
- Given a certain known-at time, we can also retrieve a so-called journal — the entire series of versions effective in the past, in the present, and in the future.

As for the level of granularity:

- It is often sufficient to store the effective-at time as year-month-day, and let several changes that become effective the same day start simultaneously. Sometimes legal requirements demand a particular order for changes that become effective the same day based on the type of business process they represent. In this case the year-month-day format can be refined accordingly.
- The known-at time normally needs to be more fine-grained, as we cannot rule out that several changes are made to a contract the same day. A good idea is to use a time stamp down to milliseconds that can be uniquely associated to the business process which causes the change.

Example Resolved

Figure 2 demonstrates the changes that are made to the health insurance contract according to the principles outlined above. Scenario 1 describes the original contract, Scenario 2 the future premium change, Scenario 3 the retroactive change concerning the child's birth, and Scenario 4 the final premium.

Let's see how this collection of versions helps us with possible use cases.

What happens if, due to legal requirements, the insurance company must provide an account of that contract, given the knowledge of March 10. What is requested here is a journal of the contract with known-at-time March 10. In Figure 3, Scenario 5 this journal is indicated by the dotted line. The journal includes three versions: the original contract that was effective in the past, the contract after adding a child which is effective now, and the contract which includes the new child as well as the modified premium and which is going to be effective in the future. Retrieving these three versions, the insurance company can report all contract details as they have evolved over time.

Or perhaps calculations of future premiums require that the premium that was effective on March 15 be known, given the knowledge back on February 10. Retrieving the appropriate version is illustrated in Figure 3, Scenario 6. The intersection of the two dotted lines determines the version that on February 10 we thought would be effective on March 15: it is the contract with only the premium modified that, in this instance, will serve as a basis for future calculations.

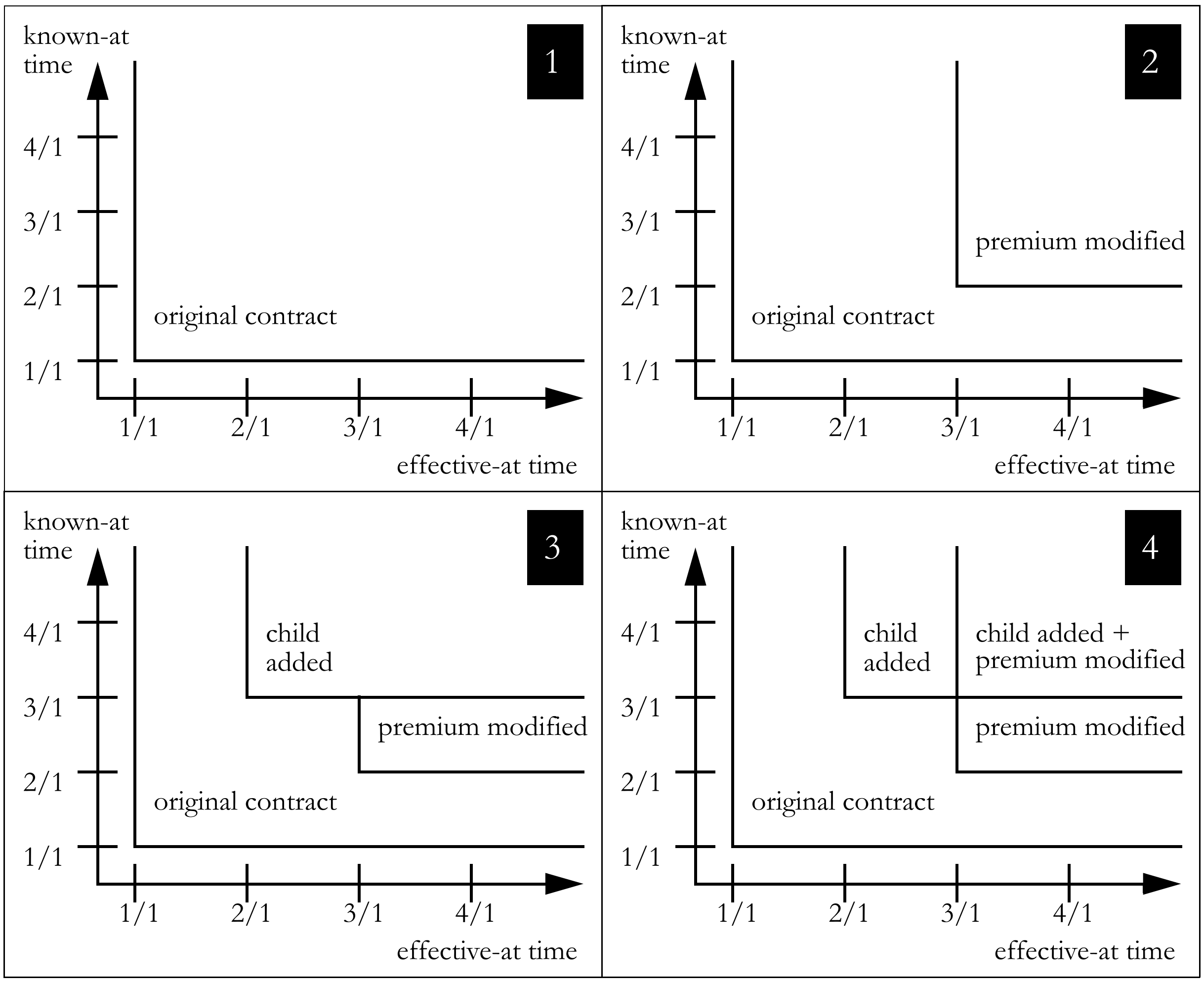


Figure 2 The 2D history of an insurance contract

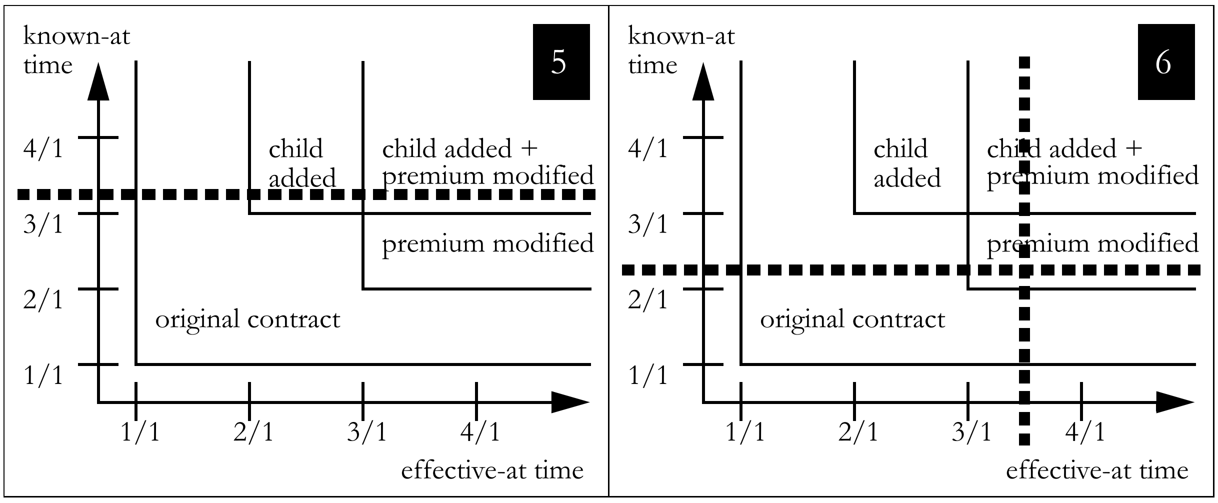


Figure 3 Version retrieval from a 2D history

Discussion Our model makes sure that new versions are added from the bottom of the time plain to the top. Obviously, there is no strict order for new versions as far as the horizontal dimension is concerned; retroactive changes make a new version further to the left than existing versions necessary. However, we are strict about the vertical dimension. If we added a version below an existing version, we would pretend we knew something when we actually didn't, and worse, we could confuse the order in which we learned about changes. It's the effective-add time that allows for the retroactive effect, not the known-at time. Knowledge is incremental.

So far we have not mentioned the deletion of versions. This will be an issue when we motivate our goal of obtaining an UNBOUNDED TIME PLAIN. Neither have we talked about implementation issues yet. Our model of two-dimensional history is a logical model so far, but we will address implementation issues when we talk about the RECTANGLE / OVERLAY COMBINATION and about the OVERFLOW TABLE.

2 Unbounded Time Plain

Context We have decided to use TWO-DIMENSIONAL HISTORY for our information system. We have an idea of how to add versions, retrieve versions, and retrieve a journal according to that model. Before we can think of a concrete implementation, we must make sure on a conceptual level that an efficient implementation will be possible.

Let's take a look at the evolution of a business object. Its initial version extends to the future infinitely, both as far as effectiveness and as far as knowledge is concerned. The version forms an infinite plain bounded to the bottom and the left, and open to the top and the right. As we add more versions, the area covered with versions remains a plain unbounded to the top and to the right.

However, adding versions isn't all we do. Imagine customers ask to have their contract cancelled. This would mean the object representing the contract has to be deleted. There are different ways to do this, with different implications for the implementation of two-dimensional history.

Problem **How can we delete an object in two-dimensional history without making the implementation difficult and inefficient?**

Forces A simple idea for the deletion of a version is to limit its effective-at interval and its known-at interval. Depending on how we choose to implement our model of two-dimensional history, we could find a way to do this. The consequence would be that the area inside the time plain that is covered by actual versions was now bounded to the top and the right.

This, however, had a disastrous effect on obtaining a journal. To obtain a journal, we have to begin with the version that lies in the future most, and travel backwards step by step. (We can't obtain a journal travelling forwards, since the object history doesn't extend into the past indefinitely, so we wouldn't know where to start.) If, however, the time plain was bounded to the right and the top, we couldn't even obtain the journal travelling backwards.

Worse yet, what happens if a version that once was deleted should be active again, as it can happen when a cancellation itself is to be cancelled? We would end up with a non-solid area of versions — an area with gaps in it that represent the deleted versions. Obtaining a journal would become impossible even if, for some reason, we knew where to begin travelling backwards, since the different version wouldn't connect any more.

Example Assume our family receive an offer for a cheaper health insurance and have their contract cancelled on April 1, to be effective May 1. A month later they figure this was a mistake as the benefits their current contract offers are better than the benefits offered by the new one. They ask the insurance company to have their contract reinstated. The company is happy to have their clients back, and cancel the cancellation.

Again, this is a typical scenario, as cancellation and reinstatement are rather common in financial information systems.

Solution **Deleted objects should be represented by pseudo-versions. This allows the area covered by object versions to be unbounded to the top and to the right.**

A special attribute needs to be added to each object; this attribute is used to mark the pseudo-versions. This way the pseudo-versions can easily be recognised and are otherwise treated as ordinary versions. A pseudo-version acts as a NULL OBJECT [6].

Because the area covered by versions is unbounded to the top and to the right as well as solid (without any gaps that would represent deleted objects), we have a reliable way to obtain an object's journal (see the algorithm in the Appendix for details):

- We retrieve the version that lies in the furthest possible future, given the desired known-at time.
- We retrieve the previous version — the one that was effective directly before the current version's effective-at time. We continue to travel backwards through the object's history until we find no previous version.
- Whenever we come across a deleted versions we either ignore it or include it in the journal, whatever the application requires.

Example Resolved After the cancellation, a pseudo-version is introduced for the original contract as illustrated in Scenario 7 in Figure 4. Scenario 8 describes the reinstated contract.There is no hole in the time plain even after the contract was deleted and reinstated.

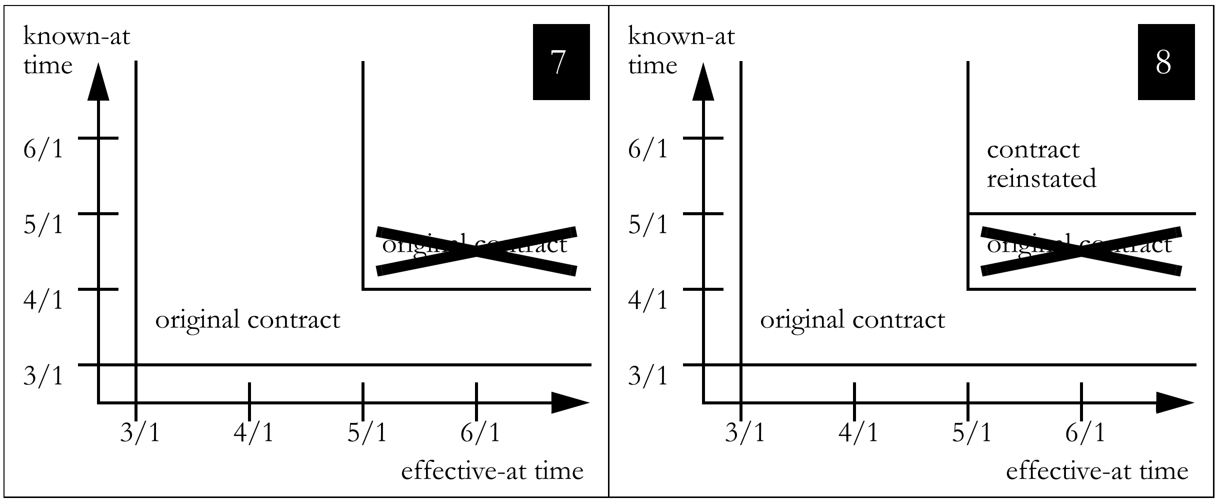


Figure 4 Version deletion and reinstatement

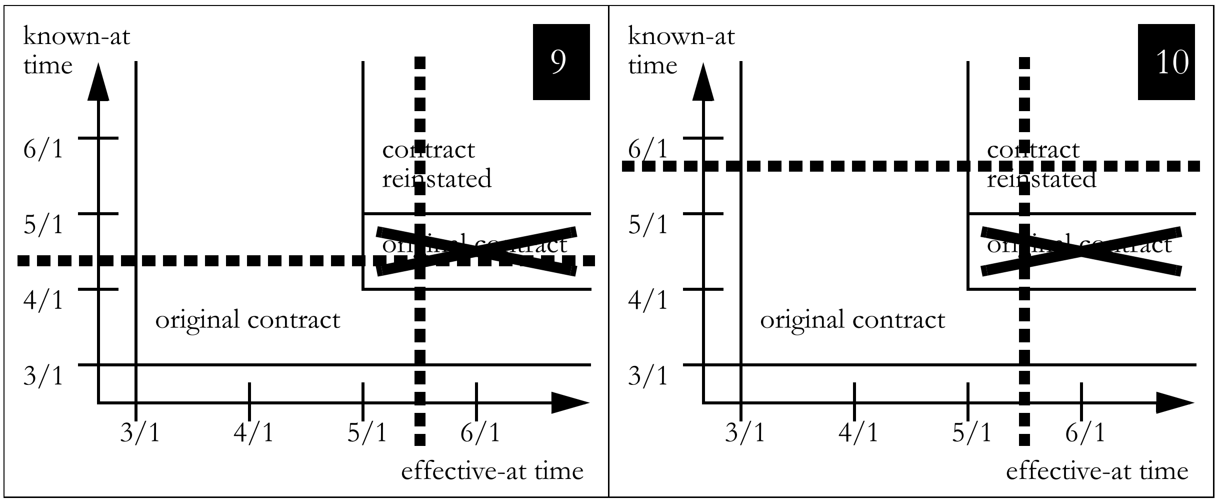


Figure 5 Version retrieval in the presence of deleted objects

Let's see what happens if we retrieve the contract version effective May 10, given our knowledge of April 10. Figure 5, Scenario 9 shows that we obtain the deleted version — a version that does have the standard attributes, but whose special deletion attribute informs us that at that particular time no contract was in effect.

If, however, we use the knowledge we had May 20, we obtain the reinstated contract, as Figure 5, Scenario 10 shows.

Discussion We have seen that a solid time plain is necessary for obtaining journals reliably, but it's not restricted to this. Next we will see that a solid time plain is also the precondition for an efficient implementation of two-dimensional history, namely for the RECTANGLE / OVERLAY COMBINATION implementation technique.

3 Rectangle / Overlay Combination

Context We have introduced the idea of adding attributes for the effective-at time and the known-at time to all objects that require a TWO-DIMENSIONAL HISTORY. Pseudo-version represent deleted objects so as to ensure an UNBOUNDED TIME PLAIN. Now it comes to implementing this concept.

When we take a look at the shape of the versions we notice they're not necessarily rectangular. L-shaped versions come from ordinary changes, and more zigzag-shaped versions come from retroactive changes. We need to store these versions in the database in such a way that version retrieval is unambiguous: for each pair of effective-at time and known-at time we want to retrieve at most one version — the one determined by the intersection of lines as in Figure 3, Scenario 6 or Figure 5, Scenarios 9 and 10.

Problem **How can two-dimensional history be implemented efficiently?**

Forces There are two fundamentally opposed implementation techniques.

One technique is to break the shapes of all versions down into rectangles. Storing such versions is easy, it simply requires two more attributes for the end of the effective-at interval and the end of the known-at interval.

This technique, however, would increase the number of versions rather dramatically. We would end up with up to twice as many physical versions as there are logical versions, and with two additional attributes per version. This technique uses up a lot of additional space in the database.

The other technique is to not store the ends of any intervals, and to accept the fact that versions overlap. The end of the effective-at interval and the end of the known-at interval of a version are represented by the effective-at time and the known-at time of a new version that is drawn on top of it. When we retrieve a version we may thus receive a list of candidates; the version we're looking for is the one with the highest known-at time, and if that is ambiguous, the one with the highest effective-at time.

This technique has two disadvantages. First, when we retrieve a version we have to iterate through the results in order to find the correct version which may represent an efficiency penalty. Second, the individual versions don't carry any information about whether there is a limit to their effective-at interval or their known-at interval, and if so, what that limit is. Most applications, however, require this information. We could obtain this information, but this would require travelling backwards through the object's entire history which would certainly prove inefficient.

In short, these two techniques force us into a choice between storage efficiency on the one side and the efficient version access on the other, both with respect to retrieving versions and to the calculation of the end of the effective-at interval.

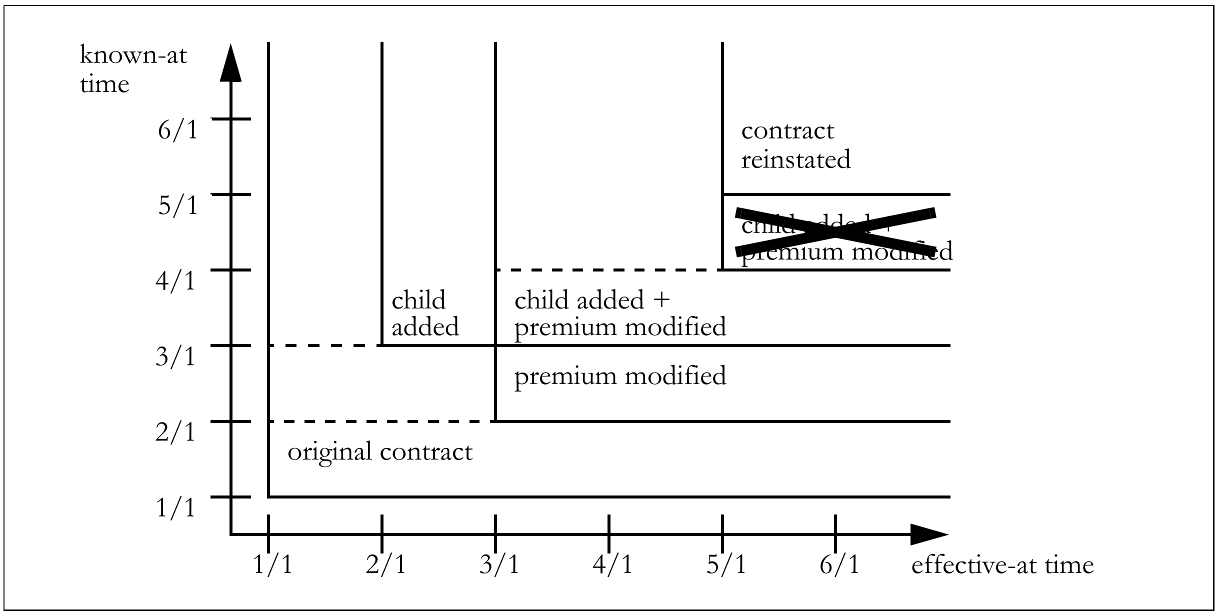


Figure 6 Rectangle versions

Example Let's take a look at how the two implementation options look like in our health insurance example. Figure 6 describes all versions introduced so far, with dotted lines representing the breaking down of versions into rectangles. There are 6 logical versions, but 9 physical versions. The original contract is stored three times; the contract that includes the child and the modified premium is stored twice.

Figure 7 describes the overlay technique. The 6 versions all overlap. When we retrieve the version that was effective on June 1, given the knowledge of June 1, we receive no less than all 6 candidates. The version that we are actually looking for is the reinstated contract, which can be identified by its known-at time of May 1, which is higher than the know-at times of all other candidates.

Still in Figure 7, when we retrieve the version that was effective on January 15, given the knowledge of June 1, we receive the original contract, and in order to find out that its effective-at interval ended February 1, we must travel backwards through all future versions, three in this case.

Solution **A combination of the overlay and the rectangle technique offers the best mix of time and storage performance.**

In detail, this combination looks as follows:

- Basically, versions overlap. Only the effective-at time and their known-at time are used to specify a version. As a consequence, a database query normally yields a set of candidates when a version is retrieved.
- A database index is defined so that when a version is retrieved, the database returns the candidates sorted in order of their known-at time as the primary criterion, and their effective-at time as a secondary criterion. This way, no inefficient iteration is necessary to find out the correct version.

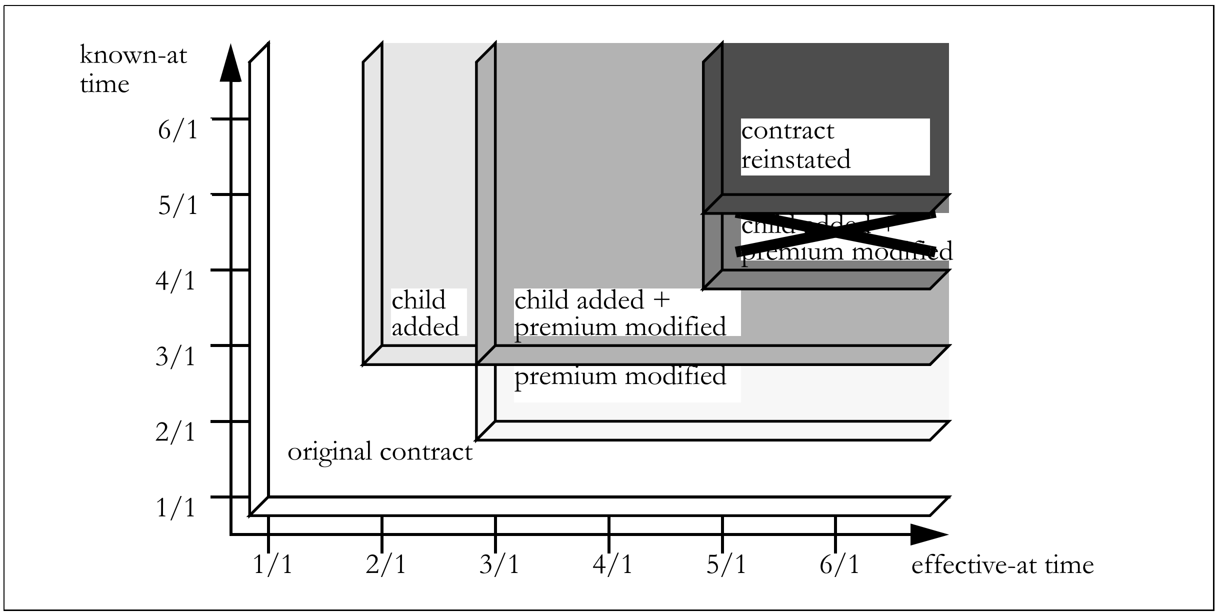


Figure 7 Overlay Versions

- We do introduce attributes for the end of the effective-at interval and the end of the known-at interval. These attributes, however, don't represent exact points in time but only upper limits. These values aren't considered when retrieving versions from the database, but merely serve informational purposes. They tell us whether the effective-at interval or the known-at interval of a version might be bounded, and if so, what the upper limit for its end is. The advantage is that determining the end of a version's effective-at interface doesn't require travelling backwards through the entire journal, but only through a few versions — one plus the number of retroactive changes made to the version).

Whenever a version is added, the attributes for the end of the effective-at interval and the end of the known-at interval of other version may have to be updated (see the algorithm in the Appendix for details):

- The version that so far was valid at the new version's effective-at time and known-at time must be marked as bounded with respect to both effectiveness and knowledge unless it is already.
- All versions known at the new version's known-at time, relative in the future with respect to effectiveness, are assigned the new version's known-at time as the end of their known-at intervals.

Example Resolved Using this implementation technique, the situation that we find in Figure 6 and Figure 7 is mapped onto the database table given in Figure 8, with the columns for known-at time and effective-at time representing the key.

known-at time	upper limit for the known-at interval	effective-at time	upper limit for the effective-at interval	deleted?	version
1/1	3/1	1/1	3/1		original contracts
2/1	3/1	3/1			premium modified
3/1		2/1	3/1		child added
3/1	4/1	3/1	5/1		child added + premium modified
4/1	5/1	5/1		x	child added + premium modified
5/1		5/1			contract reinstated

Figure 8 Database table for the combination of rectangle and overlay techniques

The values for known-at time and effective-at time of each version are clear; they follow directly from the logical model.

The values for the end of the known-at interval and the effective-at interval can be explained with the version that represents the original contract. The value given for the end of its effective-at interval is March 1. This means that, first, given the knowledge of some point in time, (actually, anything after February 1) there is an end to that version's effective-at interval and, second, that the upper limit is March 1. (The upper limit is actually met by the knowledge interval from February 1 to March 1.) Similarly, the end of the known-at interval is March 1. This means that, first, there is an end to the known-at interval of that version with respect to some effective-at time (actually all effective-at times beyond February 1) and, second, that the upper limit for this end to the known-at interval is March 1. (The upper limit is actually met with respect to any effective-at time between February 1 and March 1.)

This implementation techniques offers the following advantages:

- There is exactly one physical version for each logical version, six versions in our case, which saves disk space.
- When we need to know the end of the effective-at interval of the version effective on January 15, given the knowledge of June 1 (the original contract), we don't need to travel backwards through all three future versions, but can begin our backwards journey with the version effective on March 1, which means we only have to take two future versions into account. (In this example the advantage is only two over three, but it can be huge in real cases.)

Discussion The combination of the overlay and the rectangle presented above is in many cases a good compromise between the advantages and disadvantages each technique alone represents. However, it does depend on the situation what exactly the best solution is. If disk space isn't critical, and read access must be highly efficient, the rectangle technique might be best. If there is no need to know the end of a version's effective-at interval and the end of its known-at interval, the pure overlay technique is fine. However, in most cases disk space does matter and the information on the end of the intervals is required; in this case the combination of both techniques is the best choice.

The implementation technique we have just introduced assumes that there is an UNBOUNDED TIME PLAIN, and that deleted objects are represented by pseudo-versions. Holes in the time plain simply couldn't be implemented, as the values for the end of the known-at interval and the effective-at interval are only upper limits but no exact values.

4 Overflow Table

Context We build an information system that requires TWO-DIMENSIONAL HISTORY and we choose the RECTANGLE / OVERLAY COMBINATION for the implementation. We must expect the objects in our system to undergo a long series of changes. This is fairly common in many applications, financial information systems in particular.

An object's history can therefore become rather lengthy. Each version needs to be stored and, for legal reasons, must in many cases not be deleted for many years.

Problem **How can using two-dimensional time be made efficient in the presence of a large number of versions?**

Forces Due to the potentially large number of versions, two-dimensional history can require a significant amount of disk space. As a consequence there are often efficiency problems with accessing the data. Batch runs in particular can suffer from the amount of versions; batches that are supposed to run overnight can have trouble staying inside their time frame.

However, much of the old information is rarely used. While retroactive changes are possible as far as contracts, address databases, etc. are concerned, retroactive changes don't reach in the past unboundedly. Much of the old data is accessed only rarely. Access to this kind of data need perhaps not be highly efficient.

On the other hand, access to the most recent data is common and must be fast, so as to ensure the application's efficiency.

Example Assume the family in our example have kept their health insurance contract for over two years now. Each year a new premium has been calculated, and perhaps they had a special bonus package included at some point. The changes to the contract from more than two years ago as described in Figures 2 and 4 are not so important any more. Neither do they play a role for the calculation of the current premium, nor for any kind of payment the family may receive.

Solution **Versions representing old information can be extracted to an overflow table. Versions can be considered old either when they haven't been in effect for a long time, or when they represent knowledge that has long been invalid.**

The overflow table makes the typical access to rather recent versions more efficient. It works as follows:

- All database tables are duplicated; one table for the most recent versions, one for all other versions.
- The table for the most recent versions is kept rather small so that it works as a cache and provides fast access.
- Versions should go in the overflow table if either the end of their effective-at interval lies in the past more than a threshold time span, or the end of their known-at interval lies in the past more than a (perhaps different) threshold time span.
- Versions get older as time goes by. The cache can be checked regularly for old versions which are then removed from the cache to the overflow table.
- When versions are retrieved, the appropriate table is accessed depending on the parameters for effective-at time and known-at time.

Example Resolved Figure 9 shows the changes made the contract over more than two years. The grey area covers those versions that in mid 2003 are hardly ever used any more, either because their effectiveness lies more than a year in the past or because they're outdated by more than a year. This includes all those versions that we earlier discussed in detail (Figures 2 and 4).

It therefore makes sense to keep the more recent versions in the cache for fast access, and to store the versions in the grey area in the overflow table. Any version retrieval that is successful after just accessing the cache table need not be concerned with the secondary table at all.

Discussion Introducing an overflow table is relatively independent from the implementation technique chosen. If a RECTANGLE / OVERLAY COMBINATION is applied (or a pure rectangle technique), the values for the end of the effective-at interval and the end of the known-at interval can be used to determine whether a version should go into the overflow table.

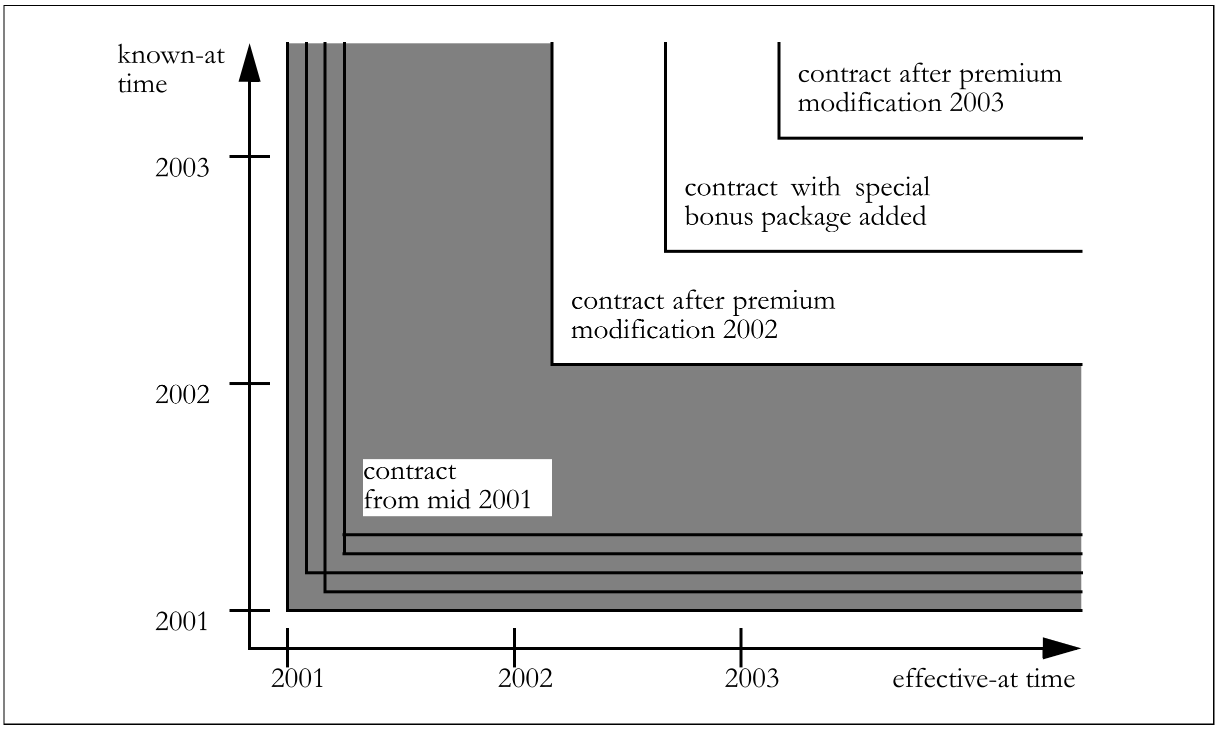

known-at time
2003
2002
2001
contract after premium modification 2003
contract with special bonus package added
contract after premium modification 2002
contract from mid 2001
2001
2002
2003
effective-at time

Figure 9 Overflow table

The use of overflow tables is well-known from database design and is, of course, not restricted to the implementation of two-dimensional history. This pattern is included here because of its particular importance in the given context: two-dimensional history often accumulates such an amount of data that the introduction of an overflow table becomes necessary. For further questions, however, such as how exactly entries from the overflow table can be accessed, we refer you to the literature on database design.

5 History Object

Context So far we have only considered the TWO-DIMENSIONAL HISTORY of individual objects. However, when we set up the data model for an application we introduce many objects that can be related in different kinds of ways, perhaps through foreign key relationships. The common history of a group of semantically related objects can therefore become relevant.

Distributing information over several objects is normal anyway, but it can be motivated by two-dimensional history itself. Because a new version becomes necessary even if only one attribute changes, it is reasonable to define objects in such a way that their attributes are likely to change at the same time. In particular, it is common to avoid objects with a large number of attributes in the presence of two-dimensional history.

Problem **How can two-dimensional history be applied to groups of objects?**

Forces To a certain extent, we can calculate the two-dimensional history for a set of object from the set of histories for the individual objects. The effective-at interval of a group of objects is the intersection of the effective-at intervals of the individual objects, and likewise the known-at interval of the group is the intersection of the known-at intervals of the individual objects.

However, obtaining the history for a group of objects becomes very difficult when the group is dynamic, that is, when new objects can be added to the group. In this case we don't know of how many objects the group might consist at some point in time. The consequence is that for a dynamic group of objects, the end of the effective-at interval and the end of the known-at interval of the group cannot be calculated efficiently. This information, however, is required by most applications.

Example The entire information associated with a health insurance contract certainly isn't stored in one object. Let's assume that in our example there is the contract object, one object for each insured person, and one object for each address associated with the contract.

If we pick up our example from Figure 9 (beginning mid 2002) then there's the contract object as well as objects for the three insured persons (the parents and the child). Let's say there is one address that has been valid for a while.

Let's assume that in August 2003 the family informs the insurance company of two things: an additional address to be effective from September 2003 onward, and an update on the particulars of the child to be effective on March 2004. The contract as a whole consists of all these objects. Figure 10 describes this scenario.

Let's now retrieve the compound object for the entire contract (the set of objects consisting of the contract itself, all insured persons, and all addresses), given the knowledge of December 2003, as it was effective on July 2003. We correctly receive the contract object after the premium modification, the objects for the three insured persons — in the case of the child the version before the update —, as well as the first address. The second address didn't exist at the time. The intersection of the effective-at intervals of all these objects yields March 2003 to February 2004.

The assumption that the compound object's effective-at interval ends February 2004, however, is not correct, as in September 2003 the additional address became effective, so the compound object did change in September 2003. The correct effective-at interval for the compound object is March 2003 to August 2003. Yet there is no way we could tell this from the objects we retrieved that were effective in July 2003.

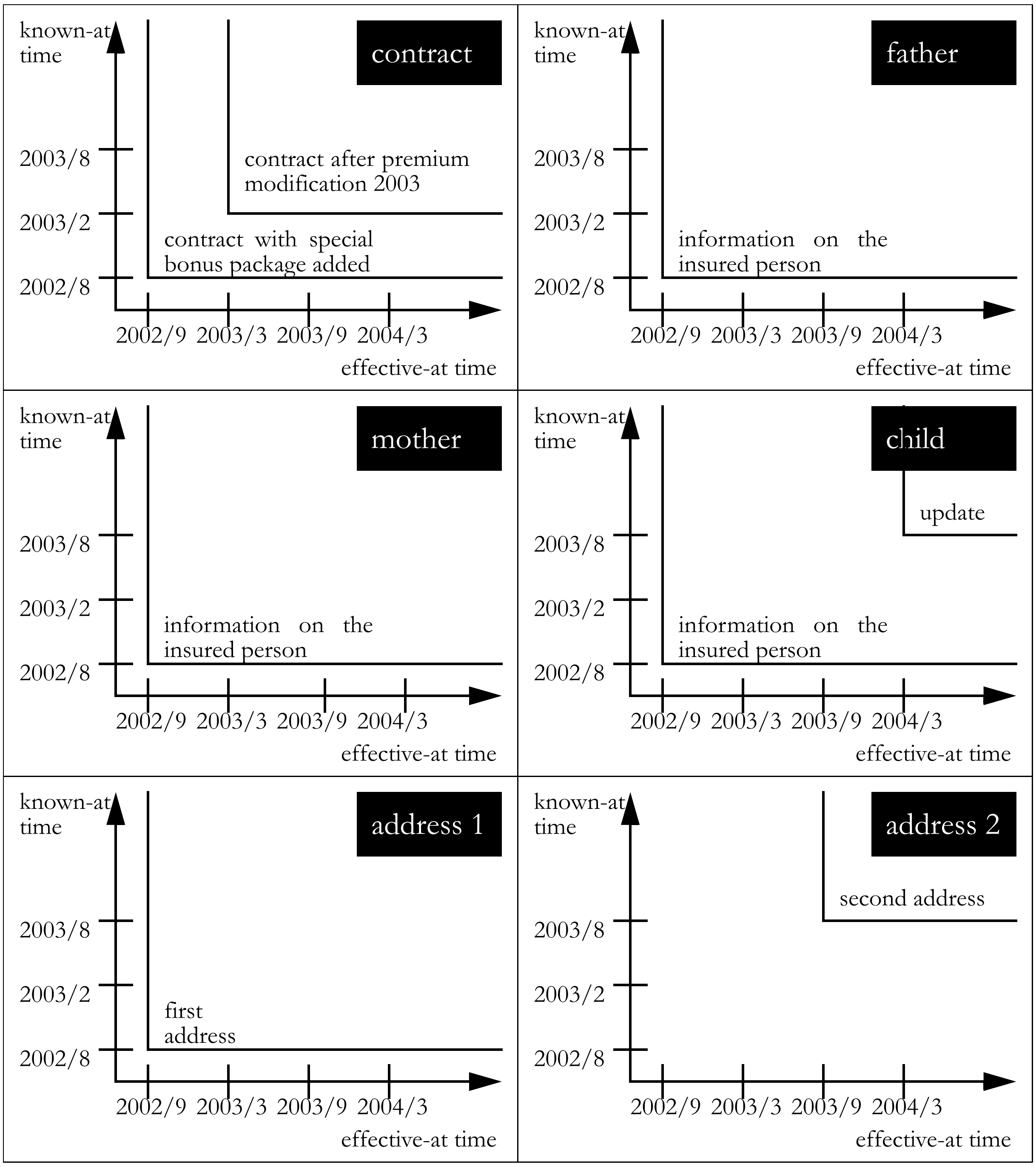

contract
known-at time
2003/8
2003/2
2002/8
contract after premium modification 2003
contract with special bonus package added
2002/9 2003/3 2003/9 2004/3
effective-at time
father
information on the insured person
mother
information on the insured person
child
update
information on the insured person
address 1
first address
address 2
second address

Figure 10 The 2D histories of related objects

Solution **Applications are often interested in the common history of a group of related objects. A special history object can be defined to keep track of the history of this set of objects.**

This history object is stored in the database as all other objects. It doesn't have much to do; its mere purpose is to keep track of modifications performed on any object within the group. It has no attributes except those associated with two-dimensional history.

The history object is updated:

- whenever any of the individual objects change, or
- when a new individual object is added to the group.

We make no assumptions on the implementation of the relationships between the objects of which the compound object consists. Perhaps these objects hold references to each other, perhaps there is an aggregation object which consists of exactly these objects. In the latter case, it may not be necessary to introduce a special history object, since the aggregation object can take its role instead.

Example Resolved The data model for our insurance contract is described in Figure 11. There are 1:n relationships that hold both between contract and insured person, and between contract and address.[2]

We introduce a history object which changes whenever a modification is made to the contract as a whole, whether an individual object is modified or added.The history object therefore has dependencies to all individual objects.

Introducing this history object solves our problem, as a new version of the history object is created when the new address is added, and the history object correctly informs us of the effective-at interval of the compound object, which ends August 2003, as illustrated in Figure 12.

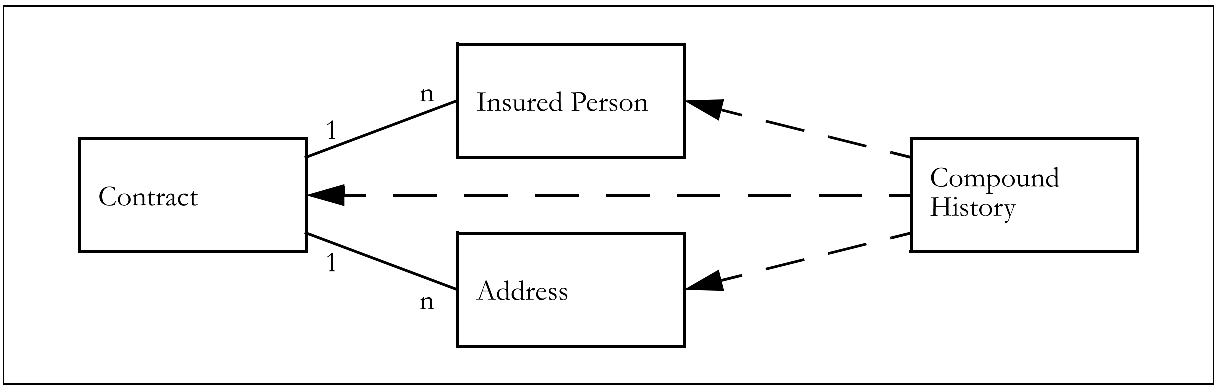


Figure 11 A set of related classes and their history object class

2. For the sake of simplicity we assume that no person has more than one contract.

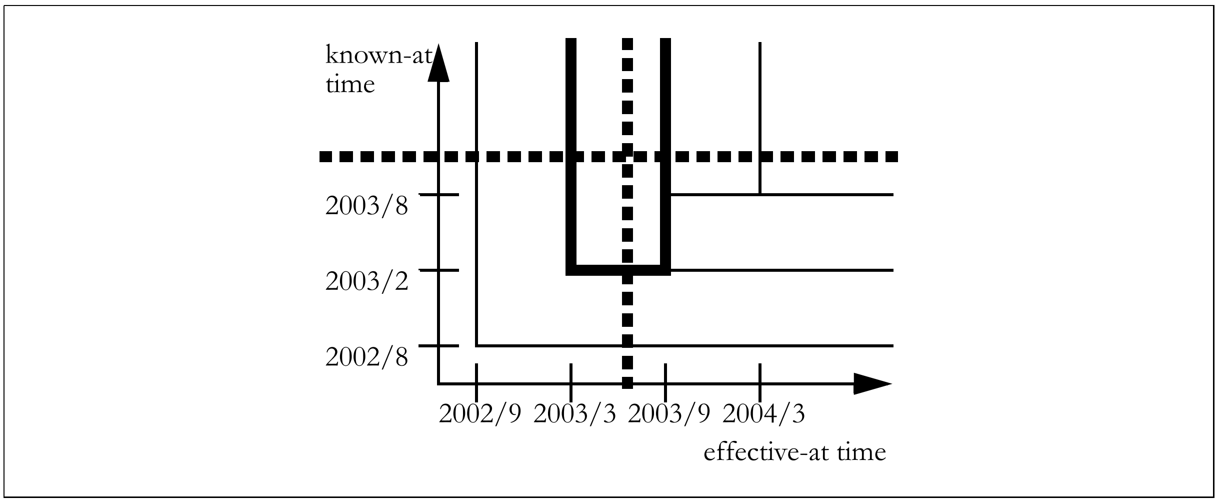


Figure 12 A history object

Discussion We have discussed objects that are added to a group of objects, and the problems that this creates for their common history. It's a common phenomenon, as the common history of objects in a 1:n relationship is often required. But what about objects that are deleted from a group of objects? Does this represent a similar problem? Fortunately enough, deleting objects is no problem as long as we use pseudo-version so as to make sure the UNBOUNDED TIME PLAIN remains intact, and no special care needs to be taken.

Known Uses

Two-dimensional history, and the principles associated with it, are well-known from many applications for financial information systems.

I was able to mine the patterns described in this paper mostly from a project carried out jointly by sd&m and Versicherungskammer Bayern, München, Germany. This project saw the development of a data access layer framework that included two-dimensional history. Several other projects developed insurance applications and used this framework for obtaining database access with integrated two-dimensional versioning. Many patterns were applied when the framework was implemented.

Moreover, I could observe several of these patterns during a consulting project at Generali, Vienna, Austria. The goal of this project was to analyse the application landscape consisting of a large number of insurance systems. Many systems featured several of the patterns described in this paper.

The German insurance association (Gesamtverband der Deutschen Versicherungswirtschaft) has defined a standard architecture for insurance systems (VAA) which includes two-dimensional history [5].

Acknowledgements

I'd like to thank the colleagues from the data access layer framework project, with whom I had many discussions on 2D history and who provided feedback on an internal report on the same topic: Hans Zierer, Achim Kugler, Rudi Simson, Robert Riemann, Winfried Kärtner, and Thomas Foitzik of sd&m, and Uli Schwab of VKB. Thanks also to innovas AG, Hamburg, Germany, for contributing several concepts on 2D history to this project.

Next, I'd like to thank Wolfgang Keller for sharing his views on 2D history.

Thanks to Alistair Cockburn for some hints on the terminology specific to the concept of 2D history.

Particular thanks go out to Klaus Marquardt who, as the EuroPLoP 2002 shepherd for this paper, provided many thought-provoking questions and suggestions for improvement.

Thanks also to Jens Coldewey for a thorough review of this material and for helpful feedback.

Last not least, I'd like to thank all participants the EuroPLoP 2002 writer's workshop in which this paper was discussed. The workshop provided much useful insight.

References

[1] Francis Anderson. A Collection of History Patterns, in: Neil Harrison, Brian Foote, Hans Rohnert (eds.), *Pattern Languages of Program Design, Vol. 4*. Addison Wesley, 2000.

[2] Massimo Arnoldi, Kent Beck, Markus Bierl, Manfred Lange. Time Travel: A Pattern Language for Values That Change, in: Paul Dyson, Martine Devos (eds.), *EuroPLoP '99 — Proceedings of the 4th European Conference on Pattern Languages of Programs, 1999*. Universitätsverlag Konstanz (UVK), 2001.

[3] Andy Carlson, Sharon Estepp, Martin Fowler. Temporal Patterns, in: Neil Harrison, Brian Foote, Hans Rohnert (eds.), *Pattern Languages of Program Design, Vol. 4*. Addison Wesley, 2000.

[4] Martin Fowler. *Analysis Patterns*. Addison-Wesley, 1996.

[5] GDV (Gesamtverband der Deutschen Versicherungswirtschaft). *VAA — Versicherungsanwendungsarchitektur*. www.gdv.de (German language).

[6] Bobby Woolf. Null Object, in: Robert Martin, Dirk Riehle, Frank Buschmann (eds.), *Pattern Languages of Program Design, Vol. 3*. Addison Wesley, 1998.

Appendix

The following pseudo-code explains the algorithms for retrieving a version, adding a version, and retrieving a journal, as introduced in Patterns 2 and 3. We assume that the versions are stored in a database table that features attributes named "effective_at", "effective_upper_limit", "known_at", and "known_upper_limit". If there is no limit to the effective-at interval or the known-at interval, the pseudo-date "9999-31-12" is used. The pseudo-code does not include any declarations or exception handling.

```
retrieve_version (effective_at, known_at):

// retrieves the desired version and the end of its effective-at interval,
// given the specified known-at time

   // determine the desired version
   select version from db where:
      version.effective_at <= effective_at and
      version.known_at     <= known_at
   sorted by (version.known_at, version.effective_at);

   if version.effective_upper_limit = 9999-31-12
   then
      // no limit to the effectiveness for any known-at time
      return (version, 9999-31-12)
   else
      current_effective_at := version.effective_upper_limit;
      current_effective_limit := 9999-31-12;

      select future_version from db where:
         future_version.effective_at <= current_effective_at and
         future_version.known_at     <= known_at
      sorted by (future_version.known_at, future_version.effective_at);

      if version = future_version
      then
         // no limit to the effectiveness given the specified known-at time
         return (version, 9999-31-12)
      else
         while future_version <> version
         do
            current_effective_at := future_version.effective_upper_limit - 1;
            current_effective_limit := future_version.effective.at;

            select future_version from db where
               future_version.effective_at <= current_effective_at and
               future_version.known_at     <= known_at
            sorted by (future_version.known_at, future_version.effective_at);
         done
         // limit to the effectiveness determined by the nearest future version
         return (version, current_effective_limit)
      end
   end
end
```

```
add_version (version, effective_at):

// adds the new version to the database and limits the effective-at and known-at
// intervals of the current and any future versions

  // prepare the new version
  version.effective_at := effective_at;
  version.known_at := today;

  current_effective_at := 9999-31-12;
  select current_version from db where:
    current_version.effective_at <= current_effective_at and
    current_version.known_at     <= known_at
  sorted by (current_version.known_at, current_version.effective_at);

  if current_version = nil or
    current_version.effective_at <= effective_at
  then
    // limit effectiveness and knowledge of the version that is now replaced by
    // the new version
    if current_version <> nil
    then
      current_version.effective_upper_limit :=
                max (current_version.effective_upper_limit, effective_at);
      current_version.known_upper_limit := today;
      update current_version in db;
    end
  else
    whilecurrent_version <> nil and
        current_version.effective_at > effective_at
    do
      // limit the knowledge of all future versions
      current_version.known_upper_limit := today;
      update current_version in db;

      current_effective_at := current_version.effective_at - 1;
      select current_version from db where
        current_version.effective_at <= current_effective_at and
        current_version.known_at     <= today
      sorted by (current_version.known_at, current_version.effective_at);
    done

    // limit effectiveness and knowledge of the version that is now replaced by
    // the new version
    if current_version <> nil
    then
      current_version.effective_upper_limit :=
                max (current_version.effective_upper_limit, effective_at);
      current_version.known_upper_limit := today;
      update current_version in db;
    end
  end

  // once the limits have been set for the version that is to be replaced
  // as well as for possible future versions, insert the new version
  insert version in db;
end
```

```
journal (known_at):

// computes the journal backwards (in reverse chronological order)

   current_effective_at := 9999-31-12;
   i = 0;

   select current_version from db where:
      current_version.effective_at <= current_effective_at and
      current_version.known_at     <= known_at
   sorted by (future_version.known_at, future_version.effective_at);

   if current_version = nil
   then
      // no journal available
      return (version_list, i)
   else
      while current_version <> nil
      do
         i := i + 1;
         version_list (i) := current_version;

         current_effective_at := current_version.effective_at - 1;

         select current_version from db where
            current_version.effective_at <= current_effective_at and
            current_version.known_at     <= known_at
         sorted by (current_version.known_at, current_version.effective_at);
      done

      // return the list of versions and the number of entries
      return (version_list, i)
   end
end
```