Time Travel: A Pattern Language for Values That Change

Massimo Arnoldi, LifeWare
Kent Beck, Daedalos Consulting
Markus Bieri, LifeWare
Manfred Lange, Hewlett-Packard

By splitting business objects for Time Travel and always processing through Period Enumeration, you can write systems with minimal extra complexity that make it possible to flexibly navigate and process changing business objects.

Introduction

You get a raise on 1 March but the system doesn't find out about it until mid-May. In the meantime, who knows what happens? Reconciling these two time lines, world time and system time, is one of the most difficult problems to solve in writing information systems. Many of the actions in business are irreversible. It is complicated and expensive to take an incorrect check back. To "make the check right", you must discover what happened and to fix the situation.

Another significant problem of writing business information systems is that the rhythms of the world are not the rhythms of the system. You would like to make changes, like giving an employee a raise in the middle of a pay period if that made sense. You would then like the system to behave reasonably under these conditions.

The simplest business option for dealing with the difference between world time and system time is just to ignore the distinction and hope nobody notices. We have seen many systems that took exactly this approach. If the customers are unlikely to notice and the consequences are small enough, you might get away with pretending that all changes are instantaneous and perfect, perhaps even forever.

This isn't a very satisfying answer, nor is it an adequate solution when the consequences of calculating the wrong answer are severe or if not being able to fix an answer that later proves to be wrong causes hurts the business.

Solving two problems when time flows by at two different rates-
- navigating through changes,
- processing changes on a time scale dictated by the business and not by the system-

is the focus of this pattern language.

Underlying Value System

Lurking beneath the surface of these patterns is a definite view of how the world should be for people in two groups whose lives are most affected by computers- customers and those who answer the customers' calls. We believe that representatives should be treated as thinking people who are capable of initiative and judgement. The computer should be there to:
- Help the representative notice that intervention is necessary
- Provide the representative with the information necessary to make a decision
- Help the representative implement a decision once it is made
- Automate tedious or routine work that can better be done by the computer, leaving the representative more time and energy to apply to the problem of serving customers.

Copyright © 1999, Massimo Arnoldi, Kent Beck, Markus Bieri, and Manfred Lange. All rights reserved.

These patterns are written with this bias in mind.

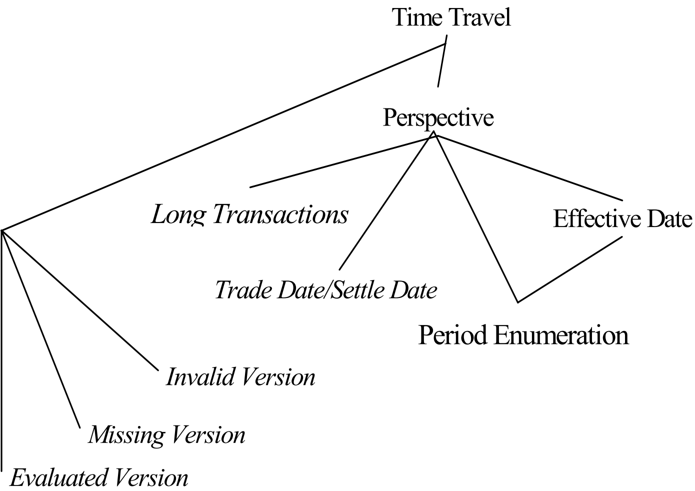
> Besides communicating the Time Travel pattern language, a secondary purpose of this paper is to experiment with a new style of presenting patterns. The patterns here contain less information than is usual, but are supplemented by being embedded in a development narrative, as if we were programming together.
>
> We use the following icons to distinguish the different phases of the development narrative:
>
> - Story
> - Test case
> - Implementation

Patterns

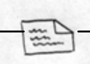
The patterns in the language are:

Pattern	*Summary*
Version History	You have an object whose changes you need to track. Split the object into two parts. One part is immutable. One is a snapshot of the changeable state.
Perspective	Create a Perspective object to represent a point of view into the history of a versioned object.
Effective Date	Separately record the date on which the system should act as if a version is valid.
Period Enumeration	For a Period and a History, iterate over the Versions valid in that Period using #in:do:.
Long Transactions	Explicitly represent long business transactions, and create a special Perspective so partially completed Versions are visible only from inside the transaction.
Evaluated Version	Give Versions containing continuously changing values a chance to update those values during lookup.
Missing Version	Before there are any valid Versions, return a Special Case Version.
Invalid Version	After the History is over, return a Special Case Version.
Trade Date/Settle Date	Record Versions with a Perspective that remembers both trade and settle dates. Navigate with Perspectives that query one or the other date.
Implicit Navigation	Share a Perspective so clients do not need to use Time Travel if they do not wish to.

Here is a map of the patterns (patterns in italics are only covered in outline form in this paper):



Navigation Story



Imagine we are working with a payroll system. An employee is paid based on a daily rate recorded in their contract. The pay rate changes from time to time. Periodically, the employee is actually paid based on the days worked and the pay rates in force when those days were worked.

The first story is that of a payroll employee who has to answer questions about paychecks.

> "Hello, payroll support here."
>
> "I just got my paycheck, and I want to know why I received $1500 this month and only $1000 last month."
>
> "Well, sir, I see that you got a raise on the first of this month."
>
> "I got that raise last month."
>
> "I'm sorry about that. I see that the raise wasn't recorded in our system until this month. We adjusted your paycheck to reflect the increase. You should receive $1250 per month from now on."

The next step is turning these vague ideas into concrete code. We will do this by first presenting a set of test cases that, if they can be satisfied, will enable us to build the system implied by the stories. Then we will show, with patterns and with code, how the test cases can be satisfied.

The simplest possible test case is the one where the contract never changes:

```
NavigationTest>>testNaivePayToday
    | contract |
    contract := Contract pay: 1000.
    self assert: contract pay = 1000
```

The naïve object model of a contract, for example, contains a single object to represent the contract. However, contracts change over time. The business relationship remains the same, somehow, but the details change.

To help answer the questions posed by the employee, the system must keep track of changes and when they happened. The simplest test case that helps reach this goal allows us to look up the pay rate on a particular date. Before we can write the test case, though, we must first introduce the pattern that transforms an immutable Contract to a Contract that can have changing pay rates:

Version History

Things change. You'd like to know what happened when. But people (and processes and organizations) are imperfect. Put these three facts together and you are faced with a challenge. How do you construct objects that can record and compute correctly in spite of imperfect recording of change?

Split one conceptual object into two parts, one part that doesn't change, the History, and one part that does change, the Version. Give the History messages to store and retrieve versions similar to keyed access to a collection.

- History>>versionAt: aTime put: aVersion - Says that from the point of view of aTime, the Version in force is now aVersion
- History>>versionAt: aTime – Returns the version visible from aTime

Using Time Travel, we can refactor the naïve test case into one that can remember changing pay rates. For now we can use simple Dates as the lookup key.

```
NavigationTest>>testPayToday
        | contract version |
        contract := Contract new.
        version := ContractVersion pay: 1000.
        contract versionAt: '2/1/99' asDate put: version.
        self assert: (contract versionAt: '2/1/99' asDate) pay = 1000
```

To make this test case run, first we create Contract (the History part of our Time Travel):

```
Contract
        Superclass: Object
        Instance variables: versions
```

Now we can define the ContractVersion (we omit the Constructor Method #pay:)

```
ContractVersion
        Superclass: Object
        Instance variables: pay date
```

With this in place, we can implement the two visible methods in Contract- #versionAt:put: and #versionAt:. (In some cases you will want to make a Safe Copy of the Version before you store it, but we are just careful not to modify a Version after we have stored it in a History.)

```
Contract>>versionAt: aDate put: aVersion
        aVersion date: aDate.
        self versions add: aVersion
```

Most Recent First

How should you iterate through versions?

Iterate so the most recent versions are seen first. Many algorithms will work out better if you can rely on this ordering.

The instance variable "versions" will be lazily initialized to a SortedCollection of Versions.

```
Contract>>versions
        versions isNil ifTrue: [versions := SortedCollection sortBlock: [:a :b | a date > b date]].
        ^versions
```

Now we can implement Contract>>versionAt:

```
Contract>>versionAt: aDate
	^self versions detect: [:each | aPerspective > each date]
```

The next test case shows that we can look up pay rates even if the pay rate changes:

```
NavigationTest>>testPayLastMonth
	| contract firstVersion secondVersion |
	contract := Contract new.
	firstVersion := ContractVersion pay: 1000.
	contract versionAt:  '1/1/99' asDate put: firstVersion.
	secondVersion := ContractVersion pay: 2000.
	contract versionAt:  '2/1/99' asDate put: secondVersion.
	self assert: (contract versionAt:  '1/1/99' asDate) pay = 1000.
	self assert: (contract versionAt:  '2/1/99' asDate) pay = 2000
```

This test case is already satisfied by the code above.

> "I got that raise last month."
>
> "I'm sorry about that. I see that the raise wasn't recorded in our system until this month. We adjusted your paycheck to reflect the increase. You should receive $1250 per month from now on."

In the next test case we store a pay rate that should have been effective at an earlier date. On 1 March we discover that on 1 February we should have given the employee a raise.

This test case will require two new patterns, Perspective and Effective Date. Perspective replaces the Date as a storage and lookup key with a more sophisticated object. Effective Date is an example of the kind of extra information that can be stored in a Perspective.

Perspective

Values are changing. How do you represent a point of view into that stream of changes?

Borrowing from the metaphor of accounting, we call the basic representation of perspective a "posting date". The posting date is the date on which a piece of data officially entered the system. In the context of Time Travel, this is the date on which a Version is committed to its History.

Create a Perspective object. Give it a Posting Date or Timestamp.

Consider including an Effective Date or Trade Date/Settle Date in the Perspective.

Using this pattern we change all the occurrences of "asDate" in the above test cases to "asPerspective".

Now we need an Effective Date, so we can complete the thought "on 1 March [posting date] we discover that we should have given the employee a raise on 1 February [effective date]."

Effective Date

When you find out a fact is not necessarily when you would like to act like that fact is true. How do you represent the difference between when something you discover a fact and when you would like the system to act like the fact was true?

Add an effective date to the Perspective. Take the effective date into consideration when computing visibility.

The existing test cases use a simple form of Perspective where the posting date and effective date are the same. The new test case uses a retroactive Perspective.

```
NavigationTest>>testWhereIsMyRaise
        | contract firstVersion secondVersion correction |
        contract := Contract new.
        firstVersion := ContractVersion pay: 1000.
        contract versionAt: '1/1/99' asPerspective put: firstVersion.
        secondVersion := ContractVersion pay: 2000.
        correction := Perspective
                posting: '3/1/99' asDate
                effective: '2/1/99' asDate.
        contract versionAt: correction put: secondVersion.
        self assert: (contract versionAt: '2/1/99' asPerspective) pay = 1000.
        self assert: (contract versionAt: ‘3/1/99’ asPerspective) pay = 2000.
        self assert: (contract versionAt: correction) pay = 2000
```

If we look up the pay rate effective on 1 February based on what we knew on 1 February, we get the old (and in retrospect wrong) value. If we look up the value on 1 March, we get the new value. And if we use the perspective of the correction, we get the new value.

We need to teach the Perspective the difference between when something happened in the outside world and when the computer learned about it.

Perspective and Effective Date together represent the 2 dimensions of time.

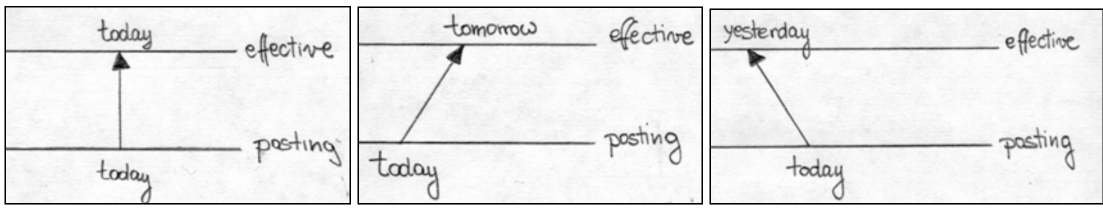


Figure 1 Figure2 Figure3

Figure1 represents the ideal case where posting date is identical to effective date. Figure2 represents a change in the future: today we commit a contract that will go in force tomorrow. Figure3 represents a classical retroactive change: only today do we learn about a change that happened yesterday

The separation of effective date and posting date adds considerably to the complexity of the software. The implementation of the objects becomes more complex. The interface to the objects is also more complex, because you must supply two dates instead of just one. We hide some of the complexity behind the Perspective object.

The simple implementation of #versionAt: using #> to compare Perspectives does not suffice anymore. We have to introduce the notion of visibility between two perspectives.

The rule for one Perspective “seeing” another is:

1. A Perspective can only see another if the viewing Perspective has a posting date the same or after the viewed Perspective. You can only see into the past, not the future.
   Insert a picture of the two lines with an eyeball on posting, black circles to its left and open circles to its right.

2. Given 1., a Perspective can only see another if the view Perspective has an effective date the same or after the viewed Perspective. Even if the system found out about a change that is to happen in the future, if necessary you should act like it hasn’t happened yet.
   Insert a picture of the two lines with an eyeball on effective, black circles to its left and open circles to its right.

Visually, a Perspective can only see another if the viewed Perspective lies completely to the left of the viewing Perspective:

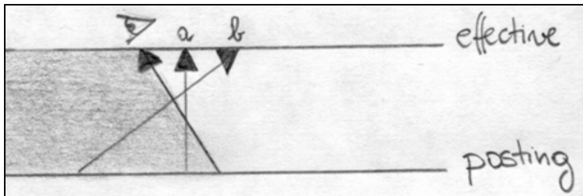


Figure: Cannot see either a or b.

```
PerspectiveTest >>setUp
	today := Perspective
		posting: Date today
		effective: Date today.
	yesterday := Perspective
		posting: (Date today addDays: -1)
		effective: (Date today addDays: -1).
	retroactive := Perspective
		posting: (Date today addDays: 1)
		effective: (Date today addDays: -1).
	proactive := Perspective
		posting: (Date today addDays: -1)
		effective: (Date today addDays: 1)


PerspectiveTest>>testTodaySeesToday
	self assert: (today sees: today)

PerspectiveTest>>testTodaySeesYesterday
	self assert: (today sees: yesterday).
	self deny: (yesterday sees: today)
```

Perspective>>sees: does not impose a total ordering on Perspectives. For example, the today and proactive Perspectives do not see each other:

```
PerspectiveTest>>testTodaySeesProactive
	self deny: (today sees: proactive).
	self deny: (proactive sees: today).
	self assert: (proactive sees: proactive)

PerspectiveTest>>testTodaySeesRetroactive
	self deny: (today sees: retroactive).
	self deny: (retroactive sees: today).
	self assert: (retroactive sees: retroactive)
```

First we define Perspective (we omit the Constructor Method #posting:effective:).

```
Perspective
	Superclass: Object
	Instance variables: posting effective
```

Here is the implementation of sees:

```
Perspective>>sees: aPerspective
	^self posting >= aPerspective posting & (self effective >= aPerspective effective)
```

Using #sees:, we can re-implement Contract>>versionAt:

```
Contract>>versionAt: aPerspective
	^self versions detect: [:each | aPerspective sees: each perspective]
```

To finish the conversion from Dates as keys to Perspectives as keys we have to implement #> to impose a total ordering on Perspectives. We sort first by posting date and then by effective date. Here is the test for Perspective comparison:

```
PerspectiveTest >>testComparison
	self deny: today > today.
	self deny: yesterday > today.
	self deny: yesterday > proactive.
	self assert: proactive > yesterday
```

And here is the implementation:

```
Perspective>> > aPerspective
	^self posting > aPersective posting | (self posting = aPerspective posting & (self effective > aPerspective
effective))
```

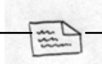
Navigation Implications

Using Contract and ContractVersion above, you can implement a system for payroll support. When you are implementing a user interface to search for the pay rate, be sure to include fields for both posting date and effective date. The field for posting date should default to Future, a Special Case of Date that is later than any other date. The field for effective date should probably default to now, so that the pay rate found by default is the pay rate that would be used at the moment.

You also need to provide a user interface for viewing the entire History. If you use Long Transactions (see below), you will be able to show who put each Version in force and why.

Finally, you will need to provide an interface for adding new versions. The interface should provide a field for effective date. The effective date should default to now. In normal circumstances the posting date should be now, and not be editable.

Enumeration Story



Consider an employee who needs lots of changes.

> "Hello, payroll support here."
>
> "I would like to increase my deductions to 6, effectively immediately."
>
> "Certainly."

Two days later:

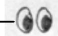
> "Hello, payroll support here."
>
> "I would like to take the additional life insurance coverage we were offered."
>
> "Certainly. That will be in force as of today. You will notice a half-month's deduction in you next paycheck. Thereafter the deduction will be $55."

Three days later:

> "Hello, payroll support here."
>
> "I would like to begin contributing 5% of my wages every month to the United Way."
>
> "Certainly. Since this is a monthly deduction, you will see a full month's deduction on your next paycheck."

At the end of the month, the paycheck is correctly computed automatically.

In the story the employee changed the contract (introduced new Versions) several times in a single pay period. When the time came to process the monthly pay, the paycheck was automatically calculated in spite of the changes.



Here is a simple test case for an employee who worked every day in January at $1000/day (we don't actually recommend working every day for a month, it just makes the arithmetic simpler):

```
EnumerationTest>>testOneMonthSalary
	| contract version month start end |
	contract := Contract new.
	version := ContractVersion pay: 1000.
	start := '1/1/99' asPerspective.
	end := '2/1/99' asDate.
	contract versionAt: start put: version.
	month := Period from: start effective to: end.
	self assert: (contract salaryIn: month) = 31000
```

The test cases require that we implement a Period class. Here is a simple version of Period:

```
Period
	Superclass: Object
	Instance variables: from to
```

In particular, the implementation of #duration simply subtracts the to date from the from date:

```
Period>>duration
	^to subtractDate: from
```

Financial instruments, for example, have many variations on the difference between two dates. However, these distinctions don't matter from the point of view of Period Enumeration.

The test cases also require that we implement Contract>>salaryIn:. An implementation that works for the first test case uses multiplication:

```
Contract>>salaryIn: aPeriod
	| version |
	version := self versionAt: (Perspective posting: aPeriod end effective: aPeriod end).
	^version pay * aPeriod duration
```

Why did we go to all the trouble of recording the intermediate changes with Perspectives if we aren't going to use that information? How can we process part of a month's pay at one rate and part at another?

Here is a test case where the salary changes on the 16th of the month.

```
EnumerationTest>>testChangeSalary
	| contract firstVersion month start end secondVersion change |
	contract := Contract new.
	firstVersion := ContractVersion pay: 1000.
	start := '1/1/99' asPerspective.
	end := '2/1/99' asDate.
	contract versionAt: start put: firstVersion.
	secondVersion := ContractVersion pay: 2000.
	change := '1/16/99' asPerspective.
	contract versionAt: change put: secondVersion.
	month := Period from: start effective to: end.
	self assert: (contract salaryIn: month) = ((15 * 1000) + (16 * 2000))
```

(We have omitted several other test cases of various odd conditions, which will nevertheless be accounted for in the code).

Period Enumeration

How do you process a History in a given period, since there may be more than one Version active in that Period?

Many information systems make the simplifying assumption of a fixed world. When the time comes to process for a period (a month or a year, for example), the world is assumed to have a single fixed state, either the state at the beginning or end of the period. While this assumption makes writing systems easier, it does not reflect the complexity of business interactions.

The consequences of fixing the state of the world during a processing period can be significant. We have seen a case where a large corporation overpaid USD 5.000.000 in salaries when a large subsidiary was sold in the middle of a pay period, but the payroll systems were unable to process a partial month. There are subtler effects also- for example companies that can only sell contracts that begin on the first of the month.

Implement an enumeration method History>>in: aPeriod do: aBlock. The block executes repeatedly with two parameters: a Version and the sub-period of the original Period in which the Version is valid. Do all processing with respect to the Version and the duration of the sub-period. Do not attempt to automatically generate reversals of previously computed results. Notify the user of the problem and give them an interface to manually process individuals.

One useful method we define on Period is the intersection of two periods:

```
Period>> intersect: aPeriod
    ^self class
        from: (self from max: aPeriod from)
        to: (self to min: aPeriod to)
```

With this Period object, we can implement Contract>>in:do:. The implementation strategy is to march backwards in time (recall that versions is sorted most recent Perspective first), creating a sub-period starting at the effective date of each Version's Perspective and ending at the previous sub-period's start date.

```
Contract>>in: aPeriod do: aBlock
    self versions
        inject: aPeriod to
        into:
            [:eachTo :eachVersion || subPeriod |
            subPeriod := Period from: eachVersion effective to: eachTo.
            aBlock value: eachVersion value: (subPeriod intersect: aPeriod).
            eachVersion effective]
```

Using this method, we can implement Contract>>salaryIn: to take changes of pay into account:

```
Contract>>salaryIn: aPeriod
    | sum |
    sum := 0.
    self
        in: aPeriod
        do: [:eachVersion :eachPeriod | sum := sum + (eachVersion pay * eachPeriod duration)].
    ^sum
```

Instead of three lines of code we have six lines, but the resulting method can handle any number of changes to a contract in any given pay period. And it can easily handle pay periods of different lengths, another challenge for many legacy systems.

Enumeration Implications

You may be tempted to automatically generate reversals with an #in:do:. A reversal occurs when, for example:

1. You process January,

2. You add a version in February whose effective date is in January
3. Now you go to process February

At this point it is possible to automatically discover that you have to undo the old Version’s processing for January, and then reprocess January and February with the new Version.

We have succumbed to the temptation to automatically generate reversals from time to time. It never works out in the end. Once you have a situation that complicated, it is unlikely that you have thought through all the combinations and permutations sufficiently so that your program is guaranteed to do what you want. Even if you could think everything through perfectly, all that extra thought is unlikely to be economically reasonable. In that same amount of time you could have added more valuable features.

We observe that:

- Reversals are reasonably rare
- The cost of making reversals by hand is reasonable
- The likelihood of the need for human intervention is high

If this is your situation, then make a handy interface for an educated person to manually fix the damage. If a particular reversal becomes routine, you can always automate it later.

Implicit Navigation

Let’s say you wish to write code that reads current values, ignoring history, you would like to be able to write that code without having to explicitly navigate.

Implicit Navigation

Some clients only care about the current value. How can you preserve Time Travel and still present a simplified protocol to such clients?

Share a Perspective so clients do not need to use Time Travel if they do not wish to.

We want a test case that creates a Router, then changes its configuration from ‘a’ to ‘b’. The Perspective used to change objects and to navigate must come from somewhere. We use a Session object containing the Perspective. (The Session could be stored in thread-local storage to reduce its scope.) All code that wants to implicitly navigate must run within a Session.

In the following, notice that the message #configuration is sent directly to the History (a Router), not the Version. This works precisely because we are implicitly navigating with respect to the current Session.

```
testRouterChange
	session := Session effective: '12/1/1999' asDate.
	session commit: [router := Router new].
	session commit: [router configuration: 'a'].
	session date: '12/3/1999' asDate.
	session commit:
		[router configuration: 'b'.
		self assert: router versions size = 2].
	session commit:
		[session date: '12/1/1999' asDate.
		self assert: router configuration = 'a'.
		session date: '12/2/1999' asDate.
		self assert: router configuration = 'a'.
		session date: '12/3/1999' asDate.
		self assert: router configuration = 'b']
```

In implementing this test case, we will start over from scratch. First, we need the Session object.

```
Object subclass: #Session
	instanceVariableNames: 'date editingVersions '
	classVariableNames: 'Instance '
```

Its Constructor Method sets the Perspective (in this case just a Date) with which subsequent navigation will occur.

```
Session class>>effective: aDate
	| result |
	result := self new.
	result date: aDate.
	^result
```

To further hide navigation, a Session could also be created without an explicit Perspective, in which case a default Perspective seeing the most recent version could be used.

The public protocol of a Session is to commit all the changes that occur within a block. The implementation sets the shared Session to be the receiver, executes the block, then commits all the changes that have occurred.

```
Session>>commit: aBlock
	self begin.
	aBlock value.
	self commit
```

Session>>begin sets the class variable Instance. Session is like a Singleton in that there is at most one current Session, but it is different in that the current Session changes.

```
Session>>begin
	Instance := self
```

We will show the details of the implementation of #commit later. For now, we can say that Histories that change (gain a new Version) during a commit block will register with the current Session, so that when the Session commits, the new Versions are made visible.

We use Version History to create a Router and a RouterVersion.

```
Object subclass: #Router
	instanceVariableNames: 'versions '
Object subclass: #RouterVersion
	instanceVariableNames: 'date configuration '
```

The interesting business starts when we want to modify the Router. *Talk about editing version*

First, the Router must lazily create a RouterVersion which we call an editing version. Every editing message will be forwarded from the Router to the special RouterVersion recording the changes. Editing messages in RouterVersion first invoke #edit to be sure there is an editing version.

```
Router>>configuration: aString
	self edit.
	self version configuration: aString
```

We found it best for Implicit Navigation to store the editing version in the Session, not the History.

```
Router>>edit
	self session editingVersionFor: self
```

The Session stores an IdentityDictionary mapping Histories to editing versions. If a History is not already being edited, a new Version is created.

```
Session>>editingVersionFor: aHistory
	^self editingVersions at: aHistory ifAbsentPut: [aHistory basicEditingVersion]
```

If no Versions have been committed, a new Version is created for editing. If a History has a Version at the date of the Session, a copy is returned to avoid modifying an existing Version.

```
Router>>basicEditingVersion
	self basicVersions isEmpty ifTrue: [^RouterVersion new].
	^(self versionAt: self session date) copy
```

The current version of a Router is found by asking the current Session:

```
Router>>version
	^self session versionFor: self
```

The Session returns the editing Version if it exists, or looks up the Version by Date.

```
Session>>versionFor: aHistory
	^self editingVersions
		at: aHistory
		ifAbsent: [aHistory versionAt: date]
```

As promised, we can now explain how a Session commits its changes. It commits the editing Versions, clears the dictionary of versions, and clears the shared reference to itself.

```
Session>>commit
	self commitEdits.
	editingVersions := nil.
	Instance := nil
```

To commit an editing Version, the Perspective of the Version (in this case just a Date) is set, and the Version is added to the History.

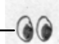
```
Session>>commitEdits
	self editingVersions keysAndValuesDo:
			[:eachHistory :eachVersion |
			eachVersion date: date.
			eachHistory addVersion: eachVersion]
```

To complete the story, the accessor for the configuration is forwarded from the History to the Version.

```
Router>>configuration
	^self version configuration
RouterVersion>>configuration
	^configuration
```

Versioning Associations

A related problem to versioning simple attributes is versioning associations.



For example, we can connect routers to other routers. In this test case we connect a router to router2 on 12/1, and also to router3 on 12/3.

```
RouterTest>>testAddingAssociations
	| router2 router3 |
	session := Session effective: '12/1/1999' asDate.
	session commit: [router := Router new].
	session commit:
			[router2 := Router new.
			router connectTo: router2].
	session commit:
			[self assert: router connectedRouters size = 1.
			self assert: (router isConnectedTo: router2)].
	session date: '12/3/1999' asDate.
	session commit:
			[router3 := Router new.
			router connectTo: router3].
	session commit:
			[self assert: router connectedRouters size = 2.
			self assert: (router isConnectedTo: router2).
			self assert: (router isConnectedTo: router3)].
	session date: '12/1/1999' asDate.
	session commit:
			[self assert: router connectedRouters size = 1.
			self assert: (router isConnectedTo: router2)]
```

The implementations of #connectTo:, #connectedRouters, and #isConnectedTo: are straightforward. However, to avoid changing the collection of connected routers accidentally we must augment the copy code by also copying the collection.

```
RouterVersion>>postCopy
    super postCopy.
    connectedRouters := connectedRouters copy
```

When the second editing Version is created, the first Version is copied. By copying the collection, additions and deletions will only affect the editing Version.

Other Patterns

Long Transactions

Business transactions can take months or years to complete. How can you partially update a Version while continuing to process with the previous Versions?

Create a Task object to represent the business transaction. Create a special Perspective that is only seen by other Perspectives within that Task.

Evaluated Version

How do you model continuously changing values, like the surrender value of an insurance contract?

Extend History>>versionAt: to copy the Version, then give the Version a chance to update its values. Store the Perspective used for look up in the returned Version. This way you don't have to always pass around the Perspective.

Missing Version

How do you avoid having to check whether a Version exists at a given Perspective?

Create a Special Case of the Version called MissingVersion. Return a MissingVersion if the Perspective used in #versionAt: is before the beginning of the History. Give the MissingVersion the same protocol as the Version, but return default values or throw exceptions as appropriate.

Inactive Version

How do you represent the end of a History?

Create a Special Case of the Version called InactiveVersion. Commit an InactiveVersion when the History is over. Give the InactiveVersion the same protocol as the Version, but return default values or throw exceptions as appropriate.

Trade Date/Settle Date

How do you represent the difference between the trade date and the settle date when trading financial instruments?

Instead of a single Effective Date, add two dates to the Perspective, one for the trade date and one for the settle date. Then, create two special Perspectives, TradePerspective and SettlePerspective, that implement #sees: according to one date or the other. Accounting applications navigate with SettlePerspectives. Trading applications navigate with TradePerspectives.

References

Francis Anderson, *A Collection of History Patterns*, University of Washington, Department of Computer Science, Technical Report TR#WUCS-98-25.

Kent Beck, *Smalltalk Best Practice Patterns*, Prentice-Hall 1996 – Many of the implementation patterns that appear here are from this reference.

Andy Carlson, Sharon Estepp, Martin Fowler, *Temporal Patterns*, University of Washington, Deportment of Computer Science, Technical Report TR#WUCS-98-25.

Ward Cunningham, Personal communication, 1999 – The WyCash bond portfolio management system had many of the same features as those generated by Time Travel.

Martin Fowler, *Analysis Patterns – Reusable Object Models*, Addison Wesley, 1997.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elments of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Manfred Lange, *Time Patterns*, University of Washington, Deportment of Computer Science, Technical Report TR#WUCS-98-25.

Frederick Winslow Taylor, *The Principles of Scientific Management*, Inst of Industrial Engineers, 1998 (1st edition 1911) – Taylor laid out the basics of work breakdown and specialization, ideas that we believe have caused enormous human and economic suffering when applied to information systems and information workers.

Acknowledgements

Special thanks to Ward Cunningham, our shepherd, for his insights into structuring this paper. Thanks also to the rest of the LifeWare team- Giovanni Müller, Lara Pfyffer Gianocca, Willi Dürig, and Marco Gianocca.

Manfred would like to thank James Noble for his great support during preparation of my paper “Time Cursor”, which was workshopped at the conference. However, Kent and I thought, that it would be easier to integrate the ideas presented in my paper into “Time Travel”. I also want to thank Kent for the time he spent working with me. The things I learned from him do have an “extreme” impact on my daily work!

In addition we want to encourage other authors with papers that are similar to merge them. This would make it far easier for all of us to learn about the great ideas presented in those papers.