A Collection of History Patterns

Francis Anderson (francisa@altinet.net)

Abstract

Over time, events bring about changes of state in a domain. These events may originate external to the domain, or be generated by the domain itself. It is frequently necessary to either provide an audit trail as to how a domain object reached a particular state, or to enable operations on a domain object in the state it was at a previous point in time. This paper presents a collection of patterns that document techniques for recording the history of domain objects, by using an Edition to associate the changed state with the event that caused it. The sequence of the patterns reflects an increasing scope in the change of state: from the changing of a simple value of a variable (ChangeLog) through to the capturing of an entire composite structure at a point in time (HistoryOnTree).

Contents

Figures

Introduction

Business systems record the state of (a subset of) the Enterprise. Naturally, this state changes over time, in response to events that either originate external to the system, or that the system itself generates. Frequently, it is necessary to remember, for a significant period of time, the state of the system as it was prior to the recording of an event. The historical state may be read-only, for audit trail purposes, or may be updateable, for the recording of subsequent backdated transactions.

This paper describes a series of patterns that outline techniques for this recording of history.

Having described some of the basic concepts of time, the following patterns are documented:

- Edition associates a changed state with the event that caused the change.
- ChangeLog records the previous values of simple variables of a domain object.
- HistoryOnAssociation records the values of a complex variable as it changes over time.
- Posting records the detailed changes that an event contributed to a domain object which is responsible for providing aggregated totals.
- HistoryOnSelf records the previous states of the domain object itself, as significant events cause complex changes of state.
- MementoChild leaves a historical copy of a domain object in the children collection of a previous parent, in order to correctly handle backdated transactions.
- HistoryOnTree recursively applies HistoryOnSelf to a tree of nodes, for those significant events that require the state of an entire tree structure to be saved.

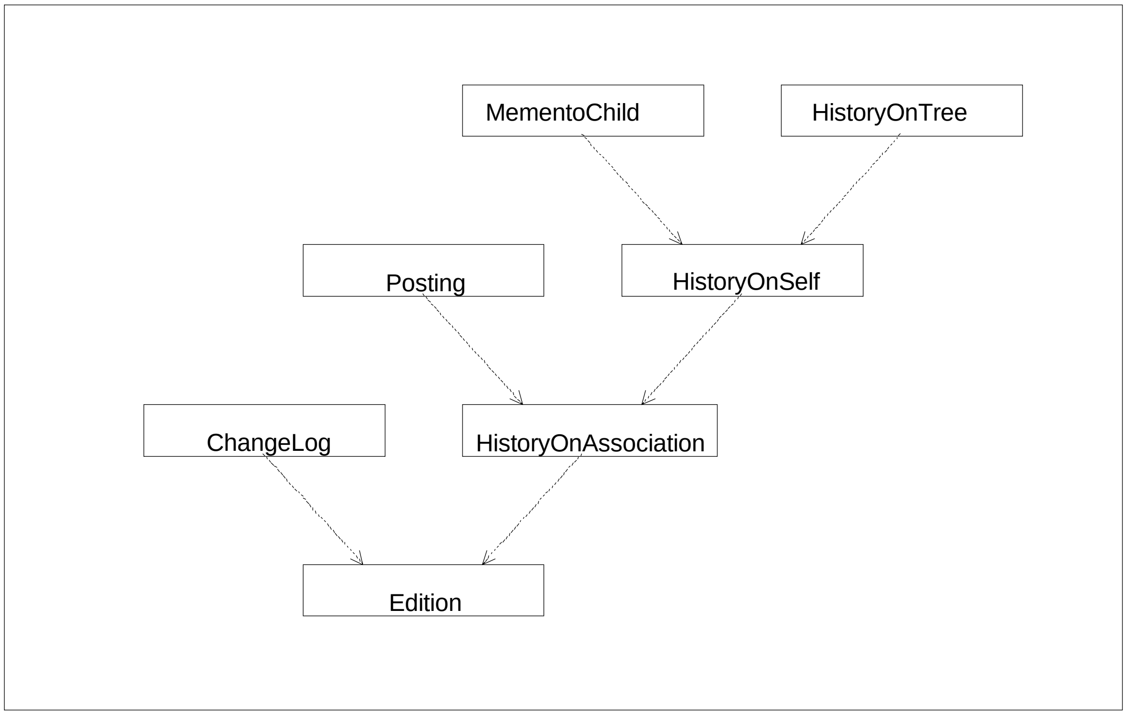


Figure 1: Pattern Dependencies

Some of these patterns have been documented elsewhere ([Fowler97], [GoF95], [JO]), but are repeated since their aggregation in one place is required to tell a better story.

It was Jean Piaget [Piaget46] who performed the pioneering for a pattern language of time, by addressing the problem of how a child is programmed. In *The Child's Conception of Time*, he wrote:

> "… the construction of time concepts are … operations that do not involve classes of objects, relations between invariable objects or numbers, but bear exclusively on positions, states, etc., i.e. on transformations rather than on constant states"

With Object-Oriented techniques, however, we only have classes of objects, their relations and their operations to deal with. The representation of a single transformation is recorded by capturing either the pre- or post-condition state of the action, and associating it with the triggering event, as an Edition. The overall history of the domain object is maintained by adding each Edition to a chronological collection. The precise transformation brought about by an event is derived by the comparison of the values of consecutive editions.

Those patterns that have concentrated on this transformation through time include:

- Memento [GoF95], which only concentrates on change of state within a transaction, so there is no association between the state and the event that caused its change, and
- HistoricMapping and TwoDimensionalHistory [Fowler97], which, as analysis patterns, do not really go into implementation details.

Known Uses

The number of potential uses for these patterns is enormous. It is frequently a requirement for most kinds of systems (business, embedded real time, source code control systems, CAD/CAM, etc.) to be able to trace changes of state to the events that caused them. The author's experience in other than business systems is, however, limited.

All the patterns in the paper are implemented in the Objectiva Architecture, a black-box Smalltalk framework for telecommunications billing and customer care, from which the sample code and instance diagrams are drawn. In Objectiva, these patterns reside at the core of the architecture, and are implemented alongside Composite [GoF95], TypeObject [WJ96], and Observation (a.k.a. ValueObject) [Fowler97], among others. This provides a high level of reuse, but does not make for ease of their standalone explanation.

Where possible, the patterns draw parallels with source control systems and database management systems, but these systems perform configuration management at a much higher level than an individual object. Most of the known uses are drawn from *ENVY*, a software configuration management tool from OTI, for Smalltalk and Java code. It is an integral part IBM's Visual Age for Java Professional. *ENVY* components form an aggregation: Configuration Maps contain Applications, which contain SubApplications, which contain SubApplications; both Applications and SubApplications contain Classes, which contain Methods. The Method is the leaf component, and a new Edition is generated for every change in code. Editions of the composites are either open for work in progress, or, as work increments are completed, they are versioned and released to their containers. Versions are immutable Editions.

1) Basic Time Concepts

1.1) Event

An event may trigger a change of state in a system. Events are of two kinds:

- Ad-hoc events usually originate external to the system. Examples include "Change of Customer Name" and "Completion of Phone Call".
- Periodic events are usually initiated by the system itself based on an Operational Calendar. Examples include "Closing of Billing Cycle Period" and "Renewal of Contract" (see Recurring Events [Fowler97a]).

The minimum responsibility of an event is the recording of the timestamp at which it occurred. A transaction mechanism is usually used to bring about the desired change of state. Transactions are entered in two basic manners:

- If entered via Online Transaction Processing (OLTP), an event is responsible for recording the User Id of the individual performing the transaction.
- If entered as part of a batch of transactions (e.g. a flat file), an event responsible for recording the Batch ID.

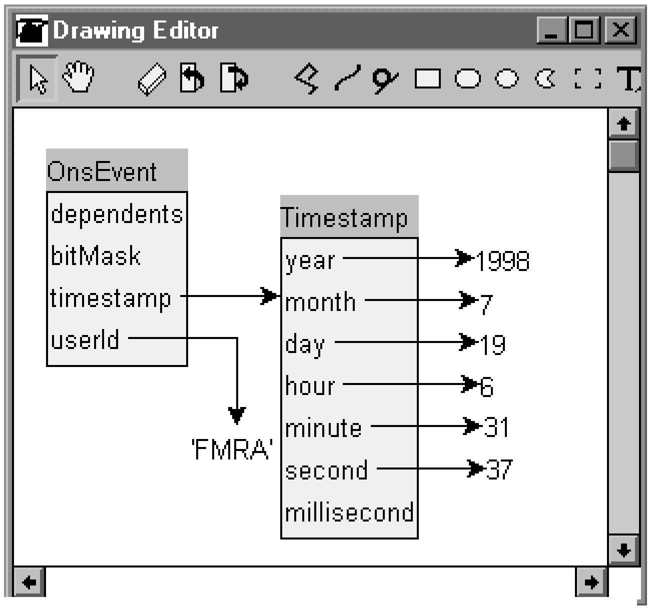


Figure 2: Instance Diagram of Event

1.2) Time Interval

A Time Interval has a specific start event and end event; its magnitude is expressed in terms of duration.

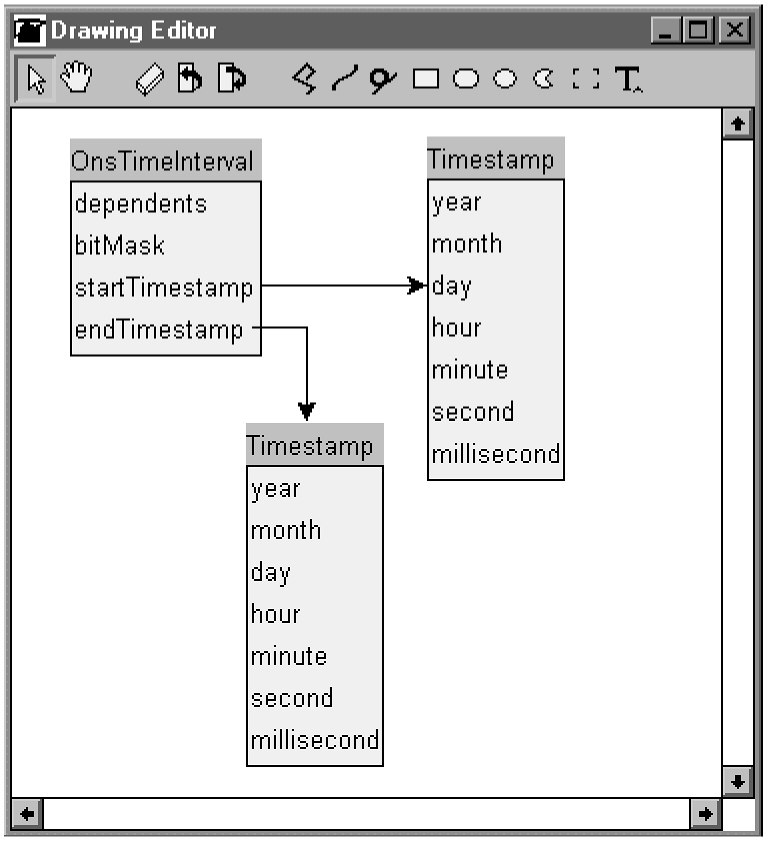


Figure 3: Instance Diagram of TimeInterval

1.3) Duration

Duration is a Quantity [Fowler97] that expresses the difference in time between two events. The Units of Duration include seconds, minutes, minutes to one decimal place, etc.

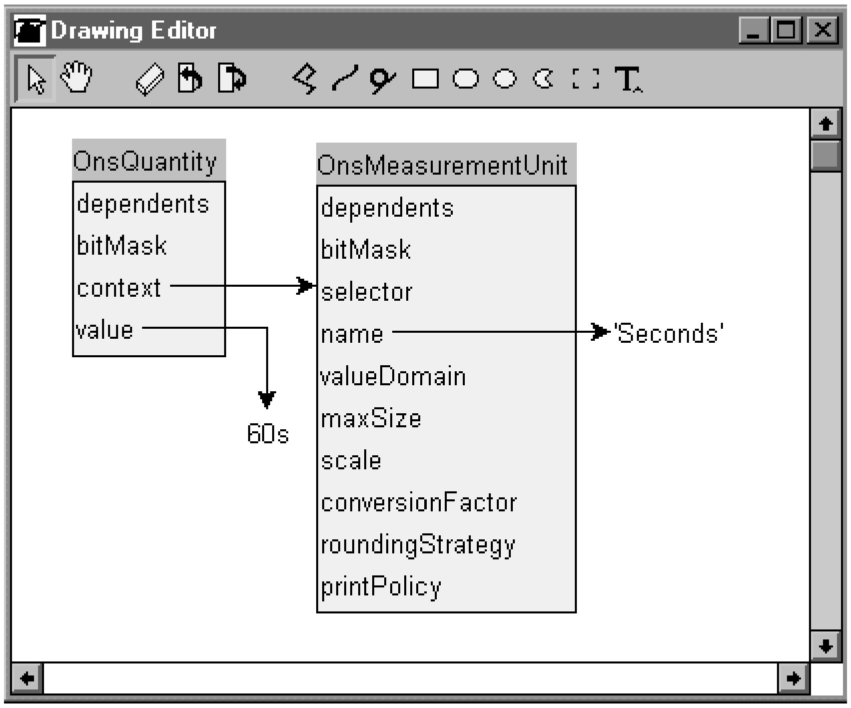


Figure 4: Instance Diagram of Duration

2) Edition

2.1) Context

An event has resulted in a domain object changing its state. We wish to track the changes of the state of the object over time. At any point in time, the variable may only have one value.

2.2) Examples

A developer checks out a component from a configuration management system (e.g. *ENVY*) in order to make changes to it.

A customer changes his mailing address.

The exchange rate for a currency changes.

2.3) Problem

How to represent that the change of state was related to a specific event?

2.4) Forces

There should be a loose coupling between the event and the values that it affected; neither the event nor the values should directly reference the other.

A value should not know whether it is current or historical or whether it even has history recorded on it.

Becoming a historical value of a variable should not cause a complex object to change its class.

2.5) Solution

Extend Memento [GoF95] by collapsing the Caretaker role into the Originator, using Edition to create the association between the Originator and the Memento. The key of the Edition is the Event causing the change of state; the value of the Edition is Memento. The choice of whether to store the pre- or post-action state is the choice of the Originator, and depends on whether the current state of a variable is recorded separately.

2.6) Diagram

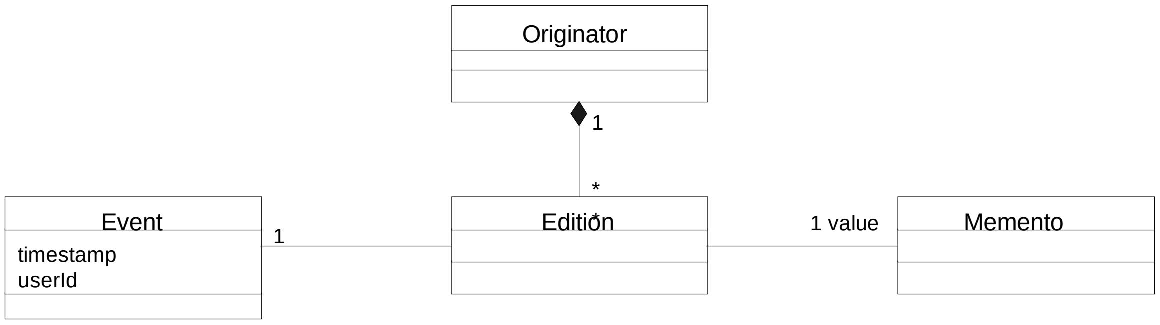


Figure 5: Class Diagram of Edition

2.7) Sample Code

```
Edition class>>newFor: aMemento

        ^self new initializeFor: aMemento
```

```
Edition>>initializeFor: aMemento

        event := Event new.
        value := aMemento
```

2.8) Resolution of Forces

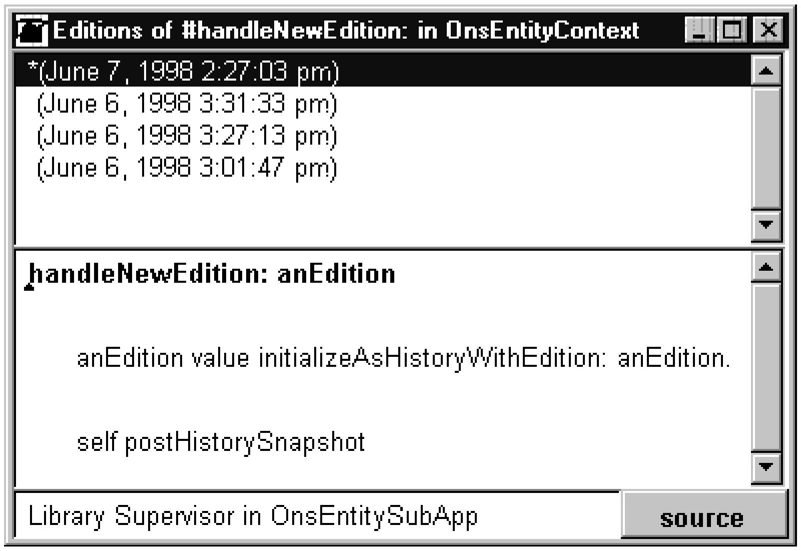
Edition “wraps” the Memento non-intrusively.

If the value of a variable is discontinuous (i.e. there may be none or many at any point in time) the key may be an interval, rather than an event.

2.9) Related Patterns

- Memento [GoF95], in that the Originator and Caretaker roles are played by the same instance, and the Memento role is played by the value of the Edition.
- AssociationClass, in that the ternary relationship between Client, Event, and Value of a variable is resolved.
- The subsequent patterns in this paper all build on Edition for handling increasing scope of the change of state.

2.10) Known Uses



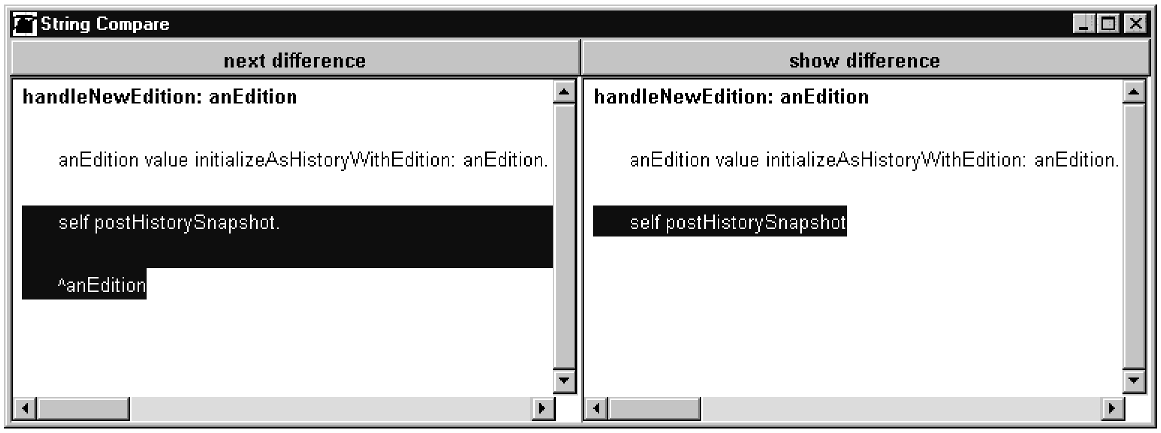
```
*(June 7, 1998 2:27:03 pm)
 (June 6, 1998 3:31:33 pm)
 (June 6, 1998 3:27:13 pm)
 (June 6, 1998 3:01:47 pm)
```

```
handleNewEdition: anEdition

    anEdition value initializeAsHistoryWithEdition: anEdition.

    self postHistorySnapshot
```

Library Supervisor in OnsEntitySubApp

Figure 6: Editions of an *ENVY* Method

As shown in Figure 6, saving a change to the code in a method causes a new edition to be created. The creation of the edition adds the timestamp and developer, the differences between editions can be browsed, identifying the actual changes that took place (see Figure 7).



```
handleNewEdition: anEdition

    anEdition value initializeAsHistoryWithEdition: anEdition.

    self postHistorySnapshot.

    ^anEdition
```

```
handleNewEdition: anEdition

    anEdition value initializeAsHistoryWithEdition: anEdition.

    self postHistorySnapshot
```

Figure 7: Differences between the Editions of an *ENVY* Method

3) ChangeLog

3.1) Context

The value of a simple variable (one that can be represented as a string, and whose identity is not required to be maintained) of a domain object has changed. The previous value of the variable is to be recorded, for reference / audit purposes.

3.2) Examples

- Customer changes name.
- The credit limit on an account is increased.
- The exchange rate of a currency relative to a base changes.

3.3) Problem

How to store the previous value of a changed simple-valued variable and have the state subsequently accessible as of a previous point in time?

3.4) Forces

In most database management systems, simple-valued variables (strings, numbers, etc.) are locally stored with the object they describe. C++ tends to refer to this situation as "has-by-value" rather than "has-by-reference", although I personally to do not like these labels since they lead to the overloading of "has". Wrapping these simple values in an Edition, and directly making this Edition persistent, would lead to a proliferation of complex objects (has-by-reference), which is not an effective usage of persistent storage.

3.5) Solution

Assign a ChangeLog to those objects for whom the values of simple variables are to be tracked over time. A ChangeLog is an OrderedCollection of ChangeLogEntry. New entries are added at the beginning of the ChangeLog (LIFO).

A ChangeLogEntry is the collapsing of the Edition and Memento roles into one, i.e. both the event and the value data are stored explicitly. The Event data is stored as its timestamp and userId; the Memento data is stored as a field name (aspect) / string value pair.

In addition to the standard set of variable accessors, those variables of the Originator which require the accessing of a previous value also provide an accessor of the form: *variableName*AsOf: effectiveDate. This either returns the current value or the last value for the variable in the ChangeLog subsequent to the specified date.

3.6) Diagrams

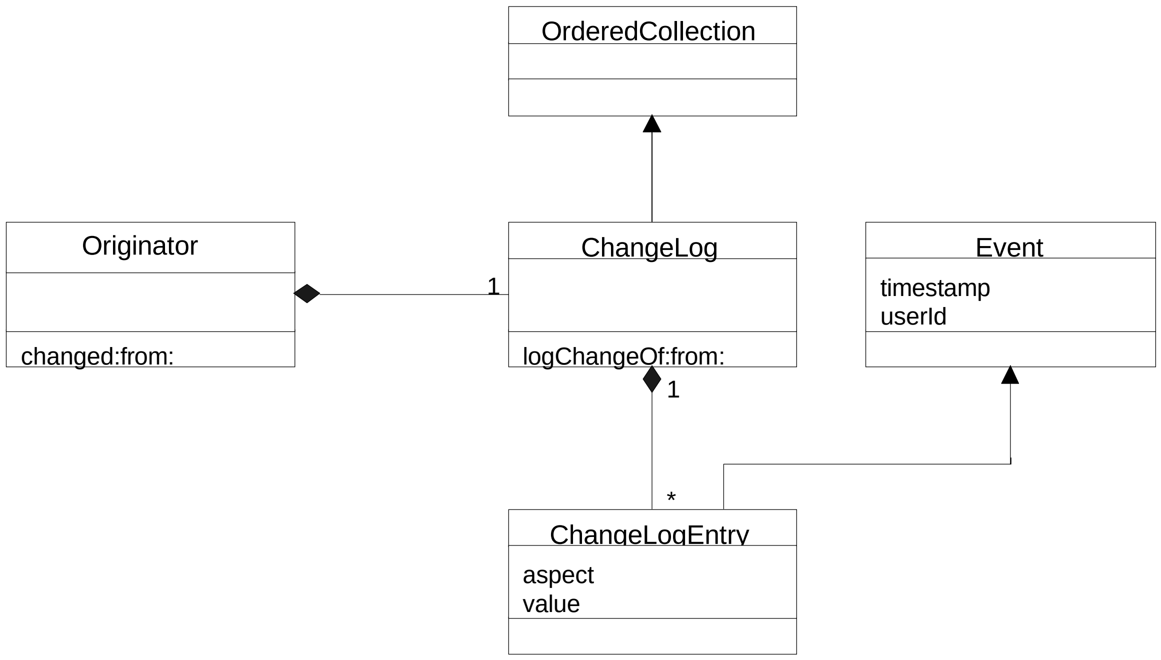


Figure 8: Class Diagram of ChangeLog

Figures 9 and 10 below demonstrate the problem of object proliferation. ChangeLog is specifically targeted at recording the historical values of simple-valued variables, i.e. those in which object identity is not an issue, and can be represented as a string.

The very point of simple values is that they can be stored persistently as part of the object they describe. There is a 0-to-Many relationship between each variable of an object and ChangeLogEntry. If we do not want to store the value of a simple variable as a reference, we certainly do not want to store its ChangeLogEntries that way.

Figure 9 shows the transient objects for a changeLog. Figure 10 shows the changeLog transformed by serialization so that it can be stored locally in the Originator / Caretaker as a ByteString. A Tab character is used to separate each field in a changeLogEntry, and a carriage return is used to separate each entry. Timestamp transforms into number of seconds from a fixed point in time, and quantities transform into their unit and magnitude.

.

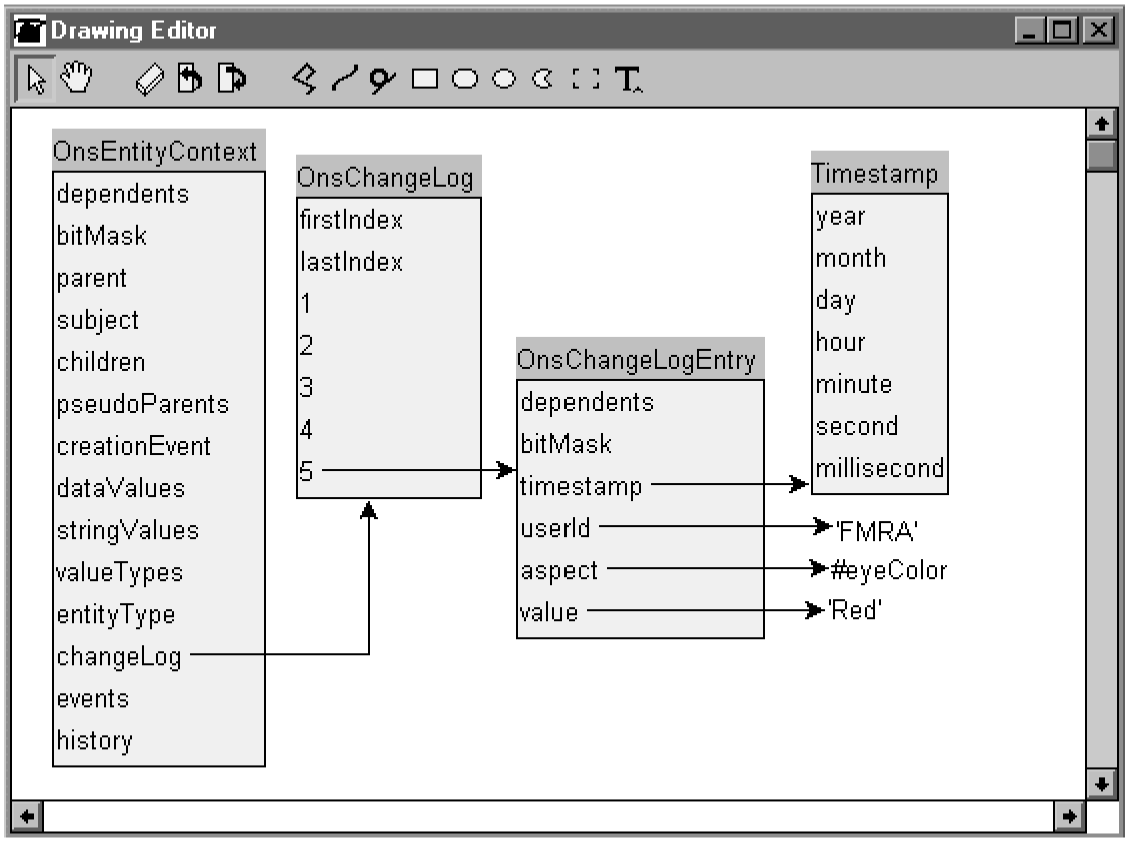
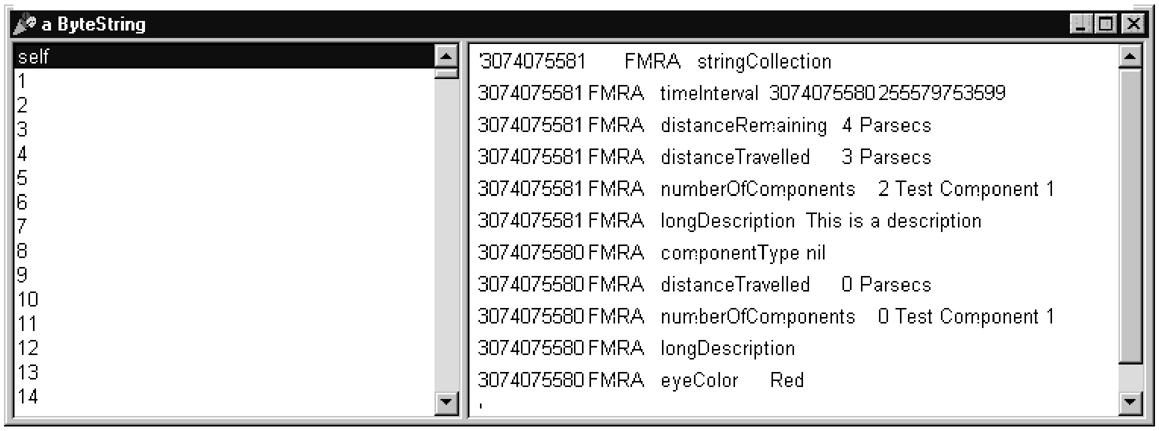


Figure 9: Instance Diagram of ChangeLog



```
'3074075581      FMRA  stringCollection
3074075581 FMRA  timeInterval  3074075580 255579753599
3074075581 FMRA  distanceRemaining  4 Parsecs
3074075581 FMRA  distanceTravelled  3 Parsecs
3074075581 FMRA  numberOfComponents  2 Test Component 1
3074075581 FMRA  longDescription  This is a description
3074075580 FMRA  componentType nil
3074075580 FMRA  distanceTravelled  0 Parsecs
3074075580 FMRA  numberOfComponents  0 Test Component 1
3074075580 FMRA  longDescription
3074075580 FMRA  eyeColor     Red
'
```

Figure 10: ChangeLog serialized as String for Persistent Storage

3.7) Sample Code

```
Originator>>name: aString

        self changed: #name
                from: name.
        name := aString
```

```
Originator>>changed: anAspect from: previousValue

        self changeLog logChangeOf: anAspect
                from: previousValue.
        self changed: anAspect
```

```
Originator>>logChangeOf: anAspect from: previousValue

        self addFirst: (ChangeLogEntry newChangeOf: anAspect
                from: previousValue)
```

```
ChangeLogEntry class>>newChangeOf: anAspect from: previousValue

        ^self new initializeChangeOf: anAspect
                from: previousValue
```

```
ChangeLogEntry>> initializeChangeOf: anAspect from: previousValue

        aspect := anAspect.
        value := previousValue asString
```

```
Currency>>exchangeRateAsOf: aTimestamp
        | value |

        ^(value :=
                self changeLog get: #exhangeRate
                        asOf: aTimestamp) isNil
                        ifTrue: [ exchangeRate ]
                        ifFalse: [ value asNumber ]
```

```
ChangeLog>>get: anAspect asOf: aTimestamp

        ^self entries do:
                [ :e | | value |
                e timestamp < aTimestamp ifTrue: [ ^value ].
                e aspect == anAspect ifTrue: [ value := e value ].
                value ]
```

3.8) Resolution of Forces

Note that ChangeLogEntry is a subclass of Event rather than Edition. This means that only simple events and values may be written to the ChangeLog. Since all variables in ChangeLogEntry can be resolved to strings, the ChangeLog can be serialized and stored persistently in its client DomainObject.

3.9) Related Patterns

- Observer [GoF95], in that, in this implementation the change notification interface has been altered so that the change is logged prior to notifying any dependents.
- VariableState [Beck97] is used to store the Memento value in the ChangeLogEntry as an field name / value pair.
- Serialization enables the ChangeLog to be stored as a simple value rather than a collection of complex objects, and occurs at three levels: the field name / value pair, the ChangeLogEntry, and the ChangeLog.

3.10) Known Uses

The PVCS software configuration management tool from INTERSOLV recreates a previous version of a file by applying reverse deltas. In this case, the file as a whole corresponds to the Originator, and the lines of code to the simple variables. Any change to a line of code is applied to the file, and its previous state is recorded in the reverse delta log.

4) HistoryOnAssociation

4.1) Also Known As

- HistoricMapping [Fowler97]
- History [JO98]

4.2) Context

ChangeLog concentrated on maintaining and accessing history for simple variables (strings, numbers, etc.). HistoryOnAssociation addresses the case of a domain object requiring history on a variable that references a complex object.

4.3) Examples

As an individual changes residence, the values of all previous addresses should be retained.

As a customer changes the role it plays (lead, prospect, subscriber), the details of the previous roles should be retained.

4.4) Problem

How do we maintain the historical values of a complex variable?

4.5) Forces

The same complex object may be referenced by a number of contexts, so the historical information should not reside in the target object.

4.6) Solution

Replace the pointer to the complex object with an instance of History, which is a SortedCollection of Edition. When the current value is changed, a new Edition is added at the start of the History (LIFO).

Historical values are obtained using *variableName*AsOf: aTimestamp.

4.7) Diagram

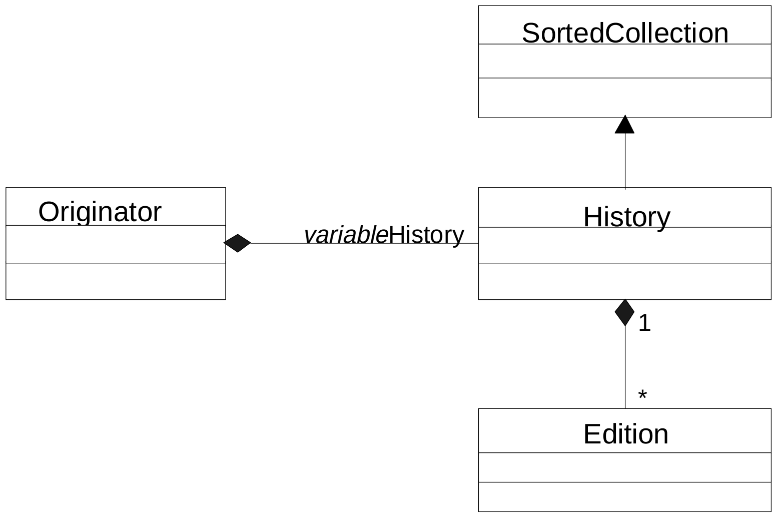


Figure 11: Class Diagram of HistoryOnAssociation

4.8) Sample Code

```
Individual>>address: anAddress

        addressHistory newEditionFor: anAddress
```

```
History>>newEditionFor: anObject

        self addFirst: (Edition newFor: anObject)
```

```
Individual>>address

        ^addressHistory currentValue
```

```
History>>currentValue

        ^self notEmpty
                ifTrue: [ self first value ]
                ifFalse: []
```

```
Individual>>addressAsOf: aTimestamp

        ^addressHistory valueAsOf: aTimestamp
```

```
History>>valueAsOf: aTimestamp

        self do: [ :e | e timestamp <= aTimestamp ifTrue: [ ^e value ]].
        ^nil
```

4.9) Resolution of Forces

The proposed solution places all the responsibility for history on the Originator and the Edition, not on the Memento. This allows an object to be referenced in multiple historical contexts. It does, however, introduce a new persistent object – the Edition. This is a relatively trivial class, in effect, just an association, which may end up with a high number of instances, thus wasting space.

4.10) Related Patterns

- HistoryOnSelf (see below) applies this pattern, except that the Memento is the previous state of the Originator, rather than a previous value of one of the Originator's variables.

4.11) Known Uses

The Hartford Insurance Company User Defined Product Framework [JO98] uses this form of History for recording the values of attributes. In this case, although the value of the Edition may be a simple value, the key is a complex transaction, thus the serialization used by ChangeLog is not possible.

5) Posting

5.1) Also Known As

- Entry [Fowler97]
- Item

5.2) Context

The context is described by:

- The Account pattern [Fowler97], whose problem is "Recording a history of changes to some quantity" (e.g. balance). The solution is: "Create an account. Each change is recorded as an entry against the account". Note that term "Account" is used here with a generic definition, not specific to bookkeeping or monetary amounts.
- The Transaction pattern [Fowler97], whose problem is "Ensuring that nothing gets lost from an account". The solution is "Use transactions to transfer items between accounts".

Although the account balance is a simple value, the ChangeLog is not the appropriate mechanism for tracking its history, since:

- The transactions that bring about a change in the value of the balance are not simple events.
- Multiple measurements may result from the application of posting rules to a transaction. It is the amount of each individual measurement that requires recording, not just the change in balance.

5.3) Examples

A telephone subscriber makes a call, which is to be charged to the appropriate accounts. If the call is from a mobile phone to a mobile phone, the one event (the call) can result in multiple charges (e.g. airtime and landline, peak and off-peak) being posted to two accounts (the originator and the receiver).

The closing of a contract period results in the posting of discounts and monthly recurring charges.

The closing of an accounting period results in the posting of federal, state and local taxes.

The receipt of a shipment causes the stock levels of a warehouse to be increased by the number of items in the shipment.

In double entry bookkeeping, a transaction results in entries to a number of accounts. The total of the monetary amounts in the entries must balance to zero.

5.4) Problem

A domain object's current state is the result of a number of transactions, and frequently the totals of the measurements of each transaction are accumulated. In turn, a transaction may affect multiple domain objects, with the measurement amounts being calculated by PostingRules [Fowler97] specific to each. How is the contribution of each transaction to each domain object's totals recorded?

5.5) Forces

The same Event can be posted to a number of accounts. This is not only due to double entry bookkeeping; in the mobile-to-mobile phone call example, the call is to be debited to two customer accounts. Thus the source event should not contain any information relating to its association to one specific domain object.

Any measurement associated with the event's posting to an account should reside in the association between the event and the account. Measurements are frequently calculated using PostingRule [Fowler97], which is usually a function of the event and the target account.

Two distinct events have occurred: the source event (e.g. the customer phone call) and its recording in the appropriate domain object. This problem is documented in TwoDimensionalHistory [Fowler97]. Both these events must be recorded, since the domain object has the responsibility of selecting the period in which to record the totals. In accounting, this is always the current period; for insurance claims and other agreement-based transactions, it is frequently the date on which the source event actually occurred.

5.6) Solution

Resolve the many-to-many relationship between Account and Transaction with a Posting subtype of Edition. This provides a "historical wrapper" around the transaction. There are therefore two events involved:

- The source event (e.g. phone call or period closing), which is responsible for the source information for billing (effective date and price determinants, such as duration of call) and is wrapped in a Posting, whose key is…
- The posting event, which records when the posting was made to the account (knowledge date).

The Posting not only decorates its source event with the Posting Date information, it also includes the charges that result from any billing calculations triggered by posting rules that govern the event.

Unlike Edition, the value of a Posting (the SourceEvent) knows the Postings that it generated, and the Posting knows its account.

5.7) Diagrams

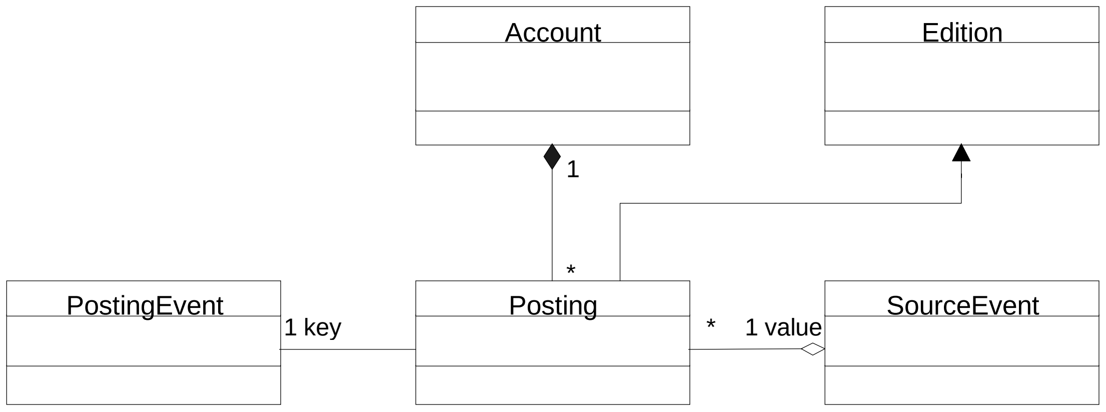


Figure 12: Class Diagram of Posting

5.8) Sample Code

```
Account>>rateAndPostEvent: anEvent

        ^(self rateEvent: anEvent) do: [ :e | self post: e ]
```

```
Account>>rateEvent: anEvent

        ^self pricePlans inject: OrderedCollection new
                into:
                        [ :postings :e | | posting |
                        (posting := e rateEvent: anEvent) hasDataValues
                                ifTrue: [ postings add: posting ].
                        postings ]
```

```
PricePlan>>rateEvent: anEvent
        | posting |

        self ratePosting: ( posting := anEvent createPostingWithPricePlan: self)
                inCurrency: self currency.

        ^posting
```

```
Event>>createPostingWithPricePlan: aPricePlan

        ^OnsPosting newOn: self
                withPricePlan: aPricePlan
```

```
Posting>>get: aSelector

        ^self get: aSelector
                ifNone: [ self sourceEvent get: aSelector ]
```

```
Account>>post: aPosting

        ^aPosting postTo: self
```

```
Posting>>postTo: anAccount

        self sourceEvent addPosting: self.
        anAccount addPosting: self.
        self postingAccount: anAccount
```

5.9) Resolution of Forces

The posting provides the association between a transaction and an account. Except for the addition of postings to its collection, the transaction itself should remain unaltered by any posting rules applied.

The posting provides the context upon which posting rules operate. For example, a telephone call's charges are frequently dependent on its duration; i.e. duration is an input price determinant. The posting rule that calculates the airtime charge of a telephone call operates upon the posting. Posting uses VariableState [Beck97] to store the result of previously triggered rules. If the price determinant requested by the rule is not available in the posting, the request is forwarded to the transaction. Posting is thus a Decorator [GoF95] on the transaction. The result of the calculation of the airtime charge is stored in the posting, and thus may itself become an input price determinant to a subsequent calculation.

The TwoDimensionalHistory pattern [Fowler97] is implemented by the posting being responsible for the "knowledge date" and the transaction being responsible for the "applicable (or effective) date". This is why two events are necessary, to store when the transaction actually occurred, and when it was posted to an account.

5.10) Related Patterns

- Many of the Analysis Patterns [Fowler97] play a part here, including Account, Transaction, PostingRule and TwoDimensionalHistory.
- Decorator [GoF95] is applied since Posting must enable a PostingRule access to its transaction's price determinants.
- VariableState [Beck97] is an appropriate means by which Posting stores the results of the PostingRules.

5.11) Known Uses

Database management systems that support forward recovery use log files to record changes to individual records (objects, rows, or segments) within a database. If the database should become corrupt for any reason, a previous backup is restored, and the transactions replayed up to the specified point in time.

6) HistoryOnSelf

6.1) Context

In the patterns above, the focus has been on the change of state of a single variable. History on Self handles a complex transaction that results in the change of state of a number of variables.

To continue with Piaget's analysis:

> "In fact, space and time result from operations just as do concepts (classes and logical relations) and numbers, but in their case, the operations take place within the object itself" [Piaget46]

Thus, it is not the previous state of a domain object's variable on which we need to record history; it is the state of the object itself, as the sum of its variables' states, i.e. the variable "self".

6.2) Examples

When an electric meter is changed, history of the location, state, seal number and reading must be recorded.

When a change is made to a method, previous version of the class must retain their appropriate edition of the method.

6.3) Problem

How do we associate all the changes of state that may have occurred on a domain object as the result of the occurrence of a single event.

6.4) Forces

Use of History and ChangeLog on the individual affected variables would result in a number of editions from a single event. Reconstruction of the state of the domain object prior to the event may be extremely complex.

A new class should not be introduced for each class that requires history.

External object references should not require updating as the result of a history snapshot.

Queries on a unique identifier should only return a single object.

The minimum of additional structure should be introduced in order to track historical states.

6.5) Solution

Instead of applying History to individual variables, it is applied to the object as a whole (the sum of the values of its instance variables). An additional variable, history, is introduced. Prior to changing any state as a result of a significant event in the life cycle of the object, a new edition is written to history, with the event as the key and a (shallow) copy of the Originator as the value This is an example of the HistoryOnAssociation pattern with the roles of Originator, Memento, and Caretaker all played by instances of the same class; the relationship between the Caretaker and the Mementos is implemented by a History collection.

Both the Originator and the Memento play the role of Caretaker. It would be redundant for both the Originator and Memento to be the Caretaker of all history up to the moment when the Memento was created. On taking a new history snapshot of the Originator, its changeLog and postings are initialized. On the creation of a new Memento, its history is initialized with the its start and end Editions. Thus, a Caretaker knows whether it is a Memento by checking if its history values include itself, and a Memento can derive the TimeInterval during which it was current.

6.6) Diagrams

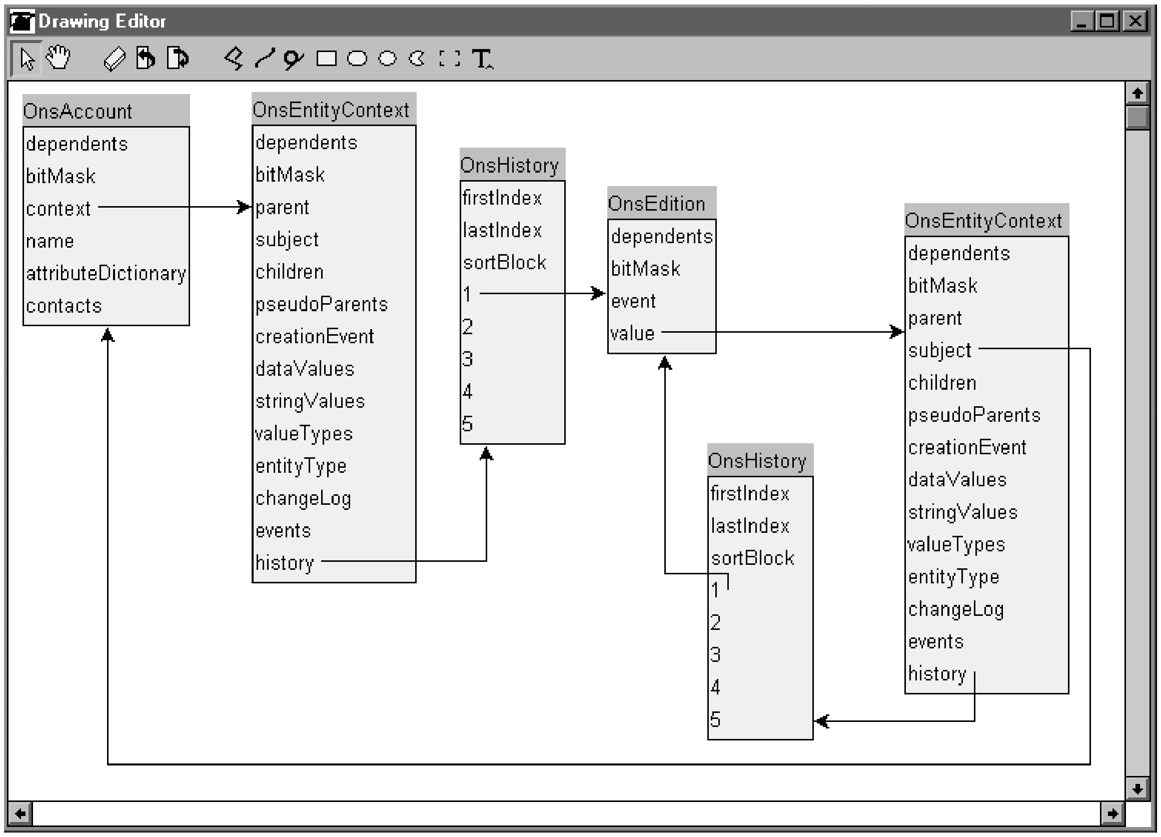


Figure 13: Instance Diagram of HistoryOnSelf

In Figure 13, OnsAccount delegates the recording of History to its OnsEntityContext. The current state records the balance brought forward, and its changeLog is initialized. The creation of a new Edition was caused by an “Account Period Closure” event. This event also caused a Posting to the Memento, which tracks totals at the charge type (i.e. Landline Charge) level.

6.7) Sample Code

```
Originator>> nodeHistorySnapshotWithEvent: anEvent
        | edition |

        edition :=
                history newValue: self copy
                        withEvent: anEvent.
        self handleNewEdition: edition.
        ^edition value
```

```
Originator>>handleNewEdition: anEdition

        anEdition value initializeAsHistoryWithEdition: anEdition.

        self postHistorySnapshot
```

```
Memento>>initializeAsHistoryWithEdition: anEdition
        | newHistory |

        newHistory := OnsHistory with: anEdition.
        self hasHistory ifTrue: [ newHistory add: history latestEdition ].

        history := newHistory
```

```
Originator>>postHistorySnapshot

        changeLog := OnsChangeLog new.
        events := OrderedCollection new.

        self initializeAccumulators
```

6.8) Resolution of Forces

Since the Memento is a copy of the Originator, there is no reconstruction of state involved; the Caretaker (the latest state) either returns itself, or the Memento that was current at the requested time.

The Memento and Originator are of the same class, since copy is used to create the Memento. The Caretaker is the current state, and holds onto its Mementos in the history variable. Thus no new classes are introduced.

One of the key decisions involved in this pattern, is whether the new copy becomes the Originator or the Memento. In the approach described above, the newly created object is the Memento, which, in general, is only obtained through the Caretaker, using an "asOf:" accessor. There is no need to change external references, since they refer to the current state, i.e. the Originator / Caretaker.

The problem of only one object of a class possessing a unique key value is somewhat tricky. The following alternatives are available:

- The query mechanism could be tuned to only return the current state.
- The key could be mutated on initialization as a memento.
- The Bridge[GoF95] pattern could be used to separate the state requiring history from that (e.g. a key) that should be unchanging. This is the approach used in Objectiva, in which the EntityContext class has a number of responsibilities in addition to history.

The only additional instance variable introduced is the history collection itself.

6.9) Related Patterns

- Memento since this pattern corresponds to the specialization in which one class plays all three roles (Originator, Memento, and Caretaker), and the Memento is obtained by a shallow copy.

6.10) Known Uses

When a database backup is performed, the copy becomes the Memento.

When using a generation data group [Bod96], an IBM MVS batch job reads a previous version (generation –1) of a file, and writes a new one (generation 0). In this case the copy becomes the current state. The Catalog plays the role of Caretaker.

Each time an *ENVY* Method is changed, a new edition is created. In this case, all previous editions of the containing class need to retain a previous edition of edition of the method, so again, the copy becomes the current state.

7) MementoChild

7.1) Context

Tree structures are common in business systems. In the telecommunications industry, services, agreements, and accounts are all composites, as are policies in the insurance industry. Postings to a child object rolls up to the balance of its parent. The closing of account and contact periods follows a regular cycle, but it is frequently necessary to change the structure of the composite part of the way through the time interval of a period.

7.2) Examples

The North American Numbering Plan is composed of Numbering Plan Areas (NPA), identified by area code, which are composed of Central Office Codes (NPA/NXX), which are assigned within Area Code. Occasionally an “NPA split” is required, in which NPA/NXXs are moved to a new NPA. Recently in Texas, the 972 area code was split off the 214, so the phone number (214) 618-0000 is now (972) 618-0000. For a certain period of time it is necessary for the old NPA (214) to remember that it once included the moved NPA/NNX’s (214 618).

Responsibility for an account or agreement may change part way through an accounting period, due to takeover or acquisition of an asset (e.g. a ship on which satellite service is provisioned). It must be possible for transactions prior to the transfer to appear on the invoice of the original account, without having to produce a full special invoice for the partial time period.

7.3) Problem

How to change the tree structure such that transactions that were effective prior to a certain event roll up to one parent, and those subsequent roll up to another?

7.4) Forces

The entire tree structure, which may be large, should not be affected, only those parents and children that whose relationships are changed.

7.5) Solution

Apply HistoryOnSelf, prior to changing the Originator’s parent. Replace the Originator with the Memento in the Memento’s parent. After a predetermined duration, remove Mementos from the children collection.

7.6) Sample Code

```
Originator>>parent: anObject

        parent == anObject ifTrue: [ ^self ].

        self nodeHistorySnapshot.
        parent notNil ifTrue: [ parent removeChild: self ].

        parent := anObject.
        self changed: #parent
```

```
Originator>>removeChild: aChild

        self keepsChildMementos
                ifTrue: [ self useMementoForChild: aChild]
                ifFalse: [ super removeChild: aChild].

        ^ aChild
```

```
Originator>>useMementoForChild: aChild
        | index |

        (index := children indexOf: aChild) = 0 ifFalse:
                [ children at: index
                        put: aChild latestHistory.
                self changed: #children ]
```

7.7) Known Uses

When changing an *ENVY* method, the copy (in this case the new edition) replaces the original in the work in the open edition of the containing class. Other editions of the class retain a previous edition of the method.

8) HistoryOnTree

8.1) Context

Tree structures are common in business systems. In the telecommunications industry, services, agreements, and accounts are all composites, as are policies in the insurance industry. HistoryOnSelf takes a snapshot of a single component object. For some major business transactions it is necessary to treat the component as the sum of its parts, and create a snapshot of an entire graph of nodes.

8.2) Examples

On close of the billing cycle of an account, the balances and invoice detail of all subaccounts must be stored.

This occurs in numerous business examples such as telecom products, for the calculation of discounts, and insurance, for the historical settlements of claims.

8.3) Problem

This is an extension of a composite's lifetime responsibility for its children. In addition to cascading copying and deletion, some significant events may also require a composite structure to cascade the recording of history. How should the treatment of the whole as the sum of its parts be implemented for historical purposes.

8.4) Forces

The structure of a tree at a point in time must be easily re-creatable, and not require the referencing of subsequent events.

8.5) Solution

Recursively apply HistoryOnSelf to the descendants of the subject node. A parallel graph is created in which a historical parent node has historical children, i.e. a child's history is a history's child.

8.6) Diagrams

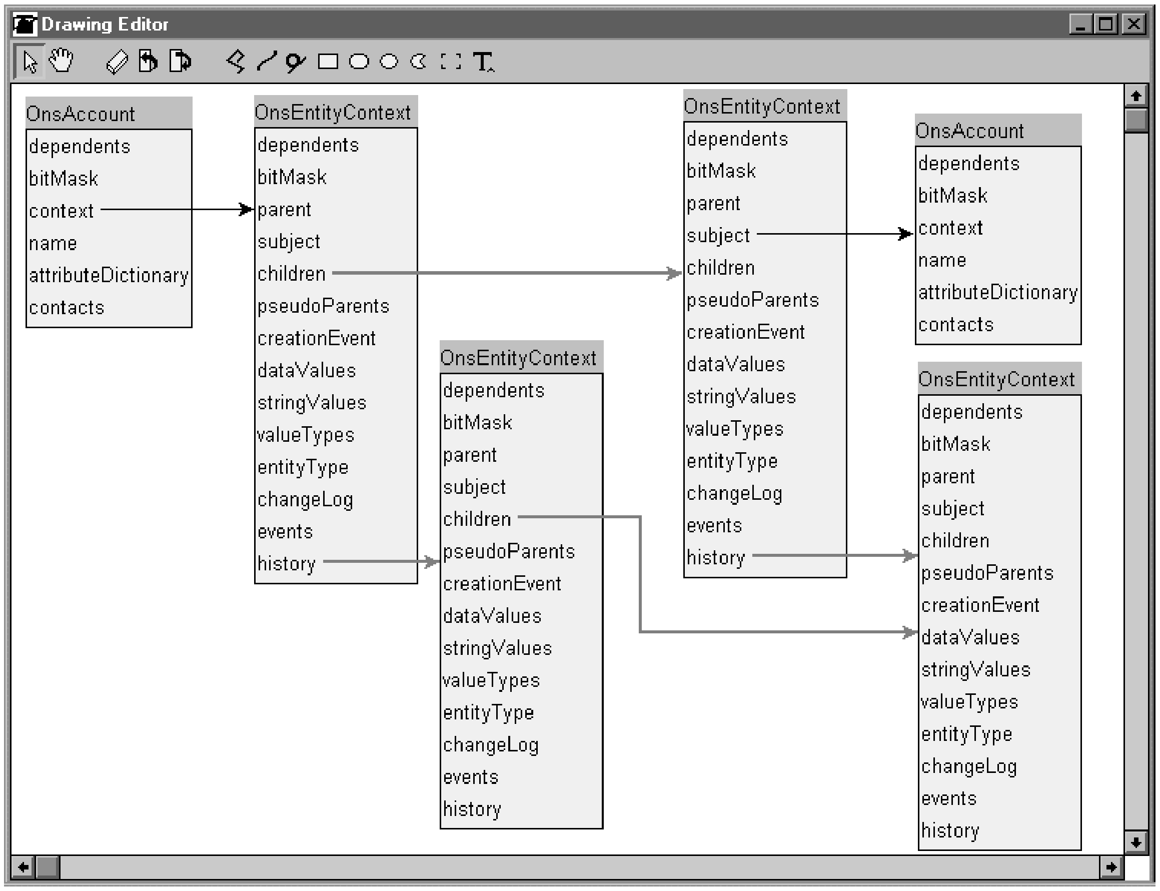


Figure 14: Instance Diagram of HistoryOnTree

8.7) Sample Code

```
Originator>> treeHistorySnapshotWithEvent: anEvent
        | memento |

        memento := self nodeHistorySnapshotWithEvent: anEvent.
        memento propagateTreeHistorySnapshotUsing:
                [ :e | e treeHistorySnapshotWithEvent: anEvent ].
        ^memento
```

```
Memento>>propagateTreeHistorySnapshotUsing: aBlock
        | currentChildren |

        currentChildren := children.
        children := OrderedCollection new.

        currentChildren do: [ :e | self addChild: (aBlock value: e) ]
```

8.8) Related Patterns

- Composite [GoF95], since in this case the whole is being treated as the sum of its parts for historical purposes.

8.9) Known Uses

In the solution above, it is assumed that all subcomponents will have activity during the current time interval. In *ENVY*, the assumption is that they will not, so editions are created in a bottom up manner, and only those components that have changed or whose subcomponents have changed need new editions. In order to version an edition of a parent component, all of its subcomponent editions must be versioned.

There are times, however, during development when it is beneficial to open new editions on all the applications and subapplications in a configuration map. This allows a team of developers greater visibility to each other's changes, enabling earlier integration testing. This approach follows the HistoryOnTree pattern, since the editions are opened from the top down; versioning still occurs from the bottom up.

Afterword

To conclude the Piaget thread,

> "... the operations (of time) take place within the object itself, and by the colligations* of its parts, play a direct part in the transformation of that unique object which is the universe of time-space."
>
> * colligate – 1. To tie together. 2. To bring (isolated observations) together by an explanation or hypothesis that applies to them all. [American Heritage Dictionary]

The patterns of time are fundamental to building accurate models that reflect the real world. They are not easy, but without them, we are practicing, in the wrong sense, a timeless way of building.

Acknowledgements

Many thanks to Ralph Johnson for his collaboration on a previous paper, Tree with History, and for his emphasis on examples over abstractions; and to Robert Martin, for his shepherding expertise to get this paper PloP-ready and for pressing on regardless of its initial Smalltalk-centricity.

References

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998.

[AJ97] Francis Anderson and Ralph Johnson, *Tree with History*, http://www.sound.net/~jchace/patterns/twh.html, paper submitted to PloP, 1997.

[Beck97] Kent Beck, *Smalltalk Best Practice Patterns*

[Bod96] Mark Bodenstein, *Data Rotations: Using Generation Data Groups to Manage Successive Copies of Related Data*, http://cornellc.cit.cornell.edu/datarot.html, 1996

[Fowler97] Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[Fowler97a] Martin Fowler, *Recurring Events*, http://www2.awl.com/cseng/titles/0-201-89542-0/events2-1.html, 1997

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[JO98] Ralph Johnson and Jeff Oakes, *The User Defined Product Framework,* http://www-cat.ncsa.uiuc.edu/~yoder/Research/metadata/udp.pdf, 1998.

[Piaget46] Jean Piaget, *The Child's Conception of Time*. English Translation – Routledge and Kegan Paul, 1969.

[WJ96] Bobby Woolf and Ralph Johnson, *TypeObject.* Published in *Pattern Languages of Program Design 3* (Robert Martin, Dirk Riehle, Frank Buschmann, eds.), Addison-Wesley, 1997.