Undertaker

FINAL DRAFT SUBMISSION EPLOP-97, June 6 1997.

Björn Eiderbäck and Jiarong Li
IPLab, NADA, KTH
S-100 44 Stockholm, Sweden
email: {bjorne, li}@nada.kth.se
Phone: +46-8-790 6277
Fax: +46-8-790 0930

Abstract

In this paper we will describe the pattern *Undertaker* aimed at handling dangling references to objects which not are recognized as garbage by the environments ordinary reclamation facilities. The context of the work is an environment which comprises a garbage collector, as Smalltalk or Java, but we believe that this pattern more generally could be applicable to environments without such services, as C++.

1 Introduction

This particular work was originally driven by efforts to utilizing the VisualWorks (ParcPlace 1992, ParcPlace-Digitalk 1995a) Smalltalk environment with distribution facilities (Eiderbäck 1993). The development of a distributed applications is well known to be a difficult task. The debugging, modification maintenance of such applications are even harder. Another difficulty is that the interface building techniques not are matured enough yet and even the most spread Interface Builders are likely to be changed and improved. One way to overcome some of the difficulties is to develop one's own distributed interface builder—but if developing such tools not is your premiere issue then this certainly is a too big endeavour and if we also want to get benefits from standardised components this approach is even more insufficient.

In this paper we will describe a pattern which we have named the *Undertaker pattern*. This pattern is an important component in our strive to add distribution facilities to the VisualWorks environment in general and in efforts to augment the interface and application builder with distribution and persistence mechanisms in particular.

Throughout the paper we assume that the readers are familiar with the patterns in the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al 1995).

1.1 Background

We have developed a distribution package, MultiGossip (MG) (Eiderbäck 1993), on top of VisualWorks. The package is completely written in Smalltalk and our aim is to provide programmers with seamless and reflexive distribution mechanisms for all kinds of objects. A particular goal was to enable experimentation of *Computer Supported Cooperative Work* (CSCW) applications. This latter goal made it particularly admirable to integrate distribution facilities into the application builder of the environment.

1.2 The Environment

Smalltalk is a language with a common root class, `Object`, therefore we can augment all objects' behaviour by adding methods to `Object`. VisualWorks comprises a wide set of basic patterns and the publisher-subscriber pattern (also known as the observer pattern) is even defined in the root class—and thereby ready for use by all objects. The adapter pattern—another important ingredient in the Undertaker pattern—is as all other objects in VisualWorks available with all its source code and thereby ready for inspection and rapid extension (or specialization) by subclassing.

1.3 The Problem

Smalltalk comes with a reclamation facility which includes a scavenger and garbage collector. The intention of them is to automatically reclaim space occupied by objects that are no longer accessible. Thereby the programmers are liberated from the burden of explicitly freeing the memory.

Those reclamation facilities rely on strong pointers, defined as follows:

Definition: strong pointer

A *strong pointer* is a reference to an object that not could be broken by any of the virtual machine's garbage collection mechanisms. The closure of strong pointers is transitive, i.e. no object reachable through strong pointers from the system roots is garbage collected.

This kind of referencing mechanism is preferable in most situations but sometimes one needs to reference an object through a reference anonymous to the garbage collection facilities, for instance to inspect the behaviour of objects in an application (without affecting any strong pointers whatsoever). In Smalltalk it is possible to benefit from such *weak pointers* by indirectly referring objects through a special purposes container—a *weak array*.

A benefit with Smalltalk's weak arrays is that as soon as one of the referred objects is reclaimed (as a result of strong pointers to it having expired) all objects in an "subscriber position" of the weak array are informed.

In the pattern description we demonstrate how we can use the above described features of a weak array, i.e. *weak pointers* and *information to subscribers* upon object reclamation, in combination with an extended value holder (ParcPlace-Digitalk 1995a) and a new *change manager*, see the implementation discussion of the observer pattern in (Gamma et al 1995).

2 The Undertaker Pattern

Name

Undertaker

Intent

- Provide publishers with facilities to reclaim references to subscribers that have ceased to exist without informing the publisher.
- Enable reuse of components that not are constructed with the possibility to survive while any of its subscribers disappears.
- Free programmers from the burden of explicitly freeing memory from objects that not are considered garbage by the reclamation facilities but still have turned into obsolete ones.
- Another very important intention is to provide mechanisms which let designers and programmers focus on the structure of their application domain. This is essentially achieved by encapsulating specialities in special purpose objects and modules.

Motivation

In our work in utilizing an environment comprising a wide set of classes and several tools aimed at designing both conventional and interactive graphical applications with distribution and persistence facilities we have encountered several problems. An important one, especially if we require seamless integration of the new capabilities, is that the system's ordinary reclamation facility under certain circumstances is unable to recognise some references as obsolete even if the referred object has ceased to exist.

In context of VisualWorks the problem can be formulated as follows:

PROBLEM: *the problem with dangling references*

Visual components in applications built with the application builder of VisualWorks are related to the models as subscribers to vital changes of them. On expiration of the application the visual components are not released from the models' (publishers') lists of dependants (subscribers). This is not a problem if the model and visual component simultaneously cease to exist (as in conventionally fabricated applications). But in a situation where the models alternatively are persistent or distributed we need to reclaim objects from the list of subscribers whenever this happens. Otherwise update-messages will be transmitted to non existing or obsolete objects, with a high risk of damaging the whole environment.

■

One solution is to force the programmers to explicitly release all dangling references upon termination of applications but this is not a particularly satisfactory approach—it is well known that the risk for memory leaks increases if the burden of freeing objects and memory is devoted to the programmer! Further the requirement of fabricating fancy garbage collection mechanisms is neither a central issue in application development and will most likely distract the programmer from focusing on important application issues. By the undertaker pattern on the other hand we can reuse existing components and still enjoy automatic and proper release of subscribers from the publishers' lists of subscribers.

In the following situations we would also benefit from the Undertaker pattern since it could hide the details from the applications' code:

- Sometimes it is hard or even impossible to know exactly when a referred entity, such as an object located in a shared file or one situated in a remote (passive) domain, has become obsolete without continuously polling all external units.

- To maximize flexibility and adaptability of applications, e.g. as in one situation use a replica of shared entity but not in another.

- In some situations it is not important where the referred object is located—e.g. in memory or at a remote host—but still it is desirable to interact with the object in a way that is independent of it actual location.

As another example graphical presentation objects (i.e. views and in some sense controllers) in applications built with the application builder of VisualWorks are related to the components (i.e. to their models) as subscribers to important changes. In applications built in the ordinary (intended?) way the scope of the components is within a single domain and the programmer has to explicitly define if a component is to be shared among several interfaces. By this approach the programmer should be relatively aware of dependencies among interfaces and he/she is also responsible to break references to parts of interfaces that are not valid any longer.

In Smalltalk objects that are not referred to by any object in the running context are garbage collected. This means that a “final” break always takes place at the end of an application since both interface presentation objects and their models are within the same scope, i.e. they are not reachable any longer and considered garbage. But in applications built with the application builder in the usual manner there is no easy way of defining objects as shared or persistent—the programmer must provide code for such situations explicitly. If we still decides to use such facilities we want to reuse as many as possible of the widgets and essential parts of the code that the framework provides us. The main problem with this solution is that when an application ceases to exist the publishers are not informed about their subscribers' decease, this is due to inherited behaviour of the widgets, and since the publishers are persistent we have a problem with dangling references to presentation components. One solution to this problem is to change some basic components. This is not attractive since we risk that it would be hard to use new components or harmonize them with new features and releases of the system. So we want a solution that changes as little as possible and reuses as much as practical of the existing environment.

Applicability

This pattern can be used in a variety of situations where some objects require to be informed of other objects' expiration. It is especially useful in combination with the publisher-subscriber (or observer) pattern or if it is impossible (or unsuitable) to change the involved objects' interface or behaviour to better fit the actual situation.

The pattern is also applicable in the following situations:

- While adding reclamation of subscribers facilities to publishers where the subscribers not inform the publisher upon reclamation.
- In utilizing the environment with basic mechanisms that simplifies migration and distribution of objects with minor impact on the applications' code.
- To provide bookkeeping of temporal entities which are to be freed if not referred any longer. A temporal entity could for instance be a socket connection or a reference to a file.

Structure

In VisualWorks\Smalltalk there is a class `ValueHolder` which is devoted to "taking care" of a value and its changes. The value holder could be said to take the same role as an active value found in some other systems. Objects interested in changes to the particular value declares themselves as subscribers to the value holder—which thereafter takes the role of a publisher.

The principles of the ordinary relation between a value holder and visual components is depicted in figure 1. Brighter colored smoother arrows and comments are used to specifically emphasize the *publisher*, the *concrete subscribers* and the *receiver*. Especially note that the visual component is explicitly dependent on the publisher—through a model-view relation inherited from the Model-View-Controller concept (Lewis 1995, Buschmann et al 1996)—and that typically an appli-

cation is implicitly dependent on the publisher—through an instance of `ChangeManager`.

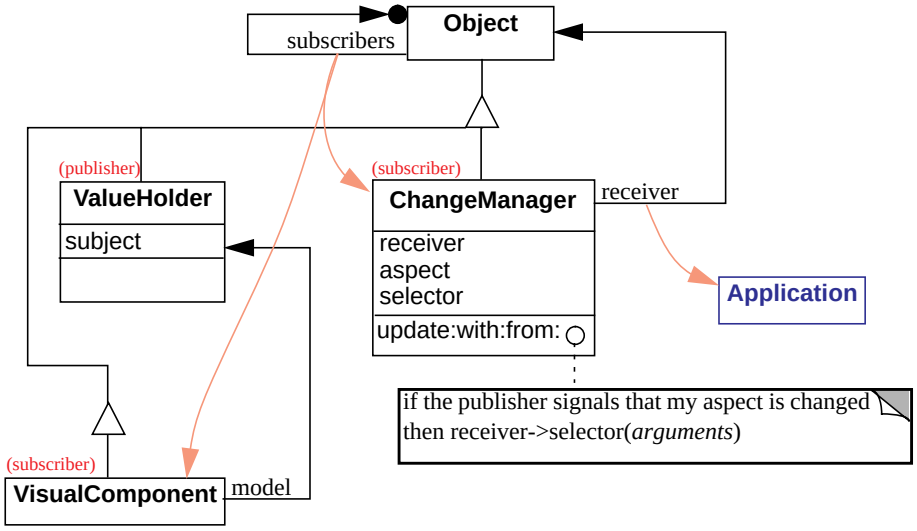

FIGURE 1 The value holder and visual component relation of VisualWorks

In figure 2 we have depicted a typical scenario involving an application, a value holder and its visual component, and the change manager intended to filter and mediate changes to the application.

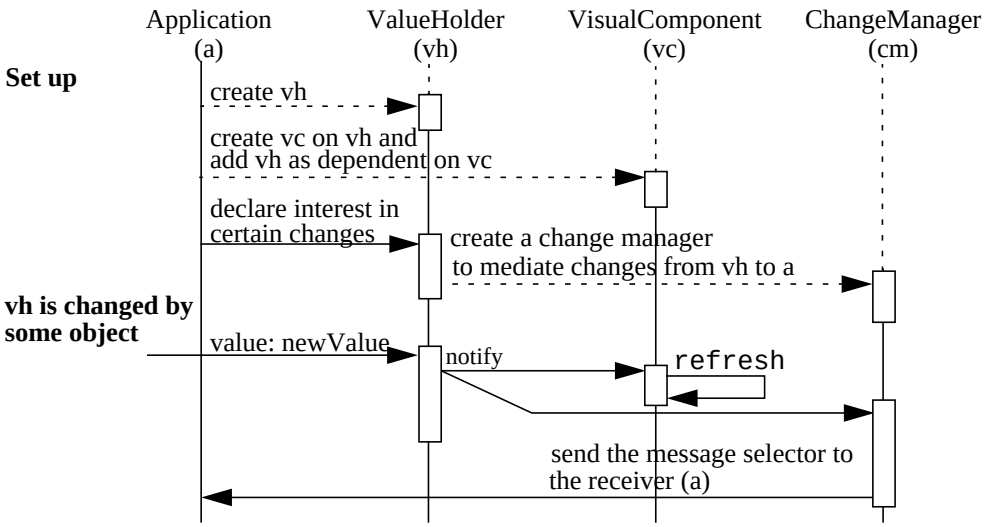

FIGURE 2 Message trace diagram for the collaboration between a value holder, a change manager and an application

For a complete description of this “pattern” we refer the interested reader to (ParcPlace-Digitalk 1995a) for details.

The enhanced (undertaker-) structure look pretty much the same as the one of the previous figure and is depicted in figure 3. For brevity we have collapsed the diagram and only included the most essential parts.

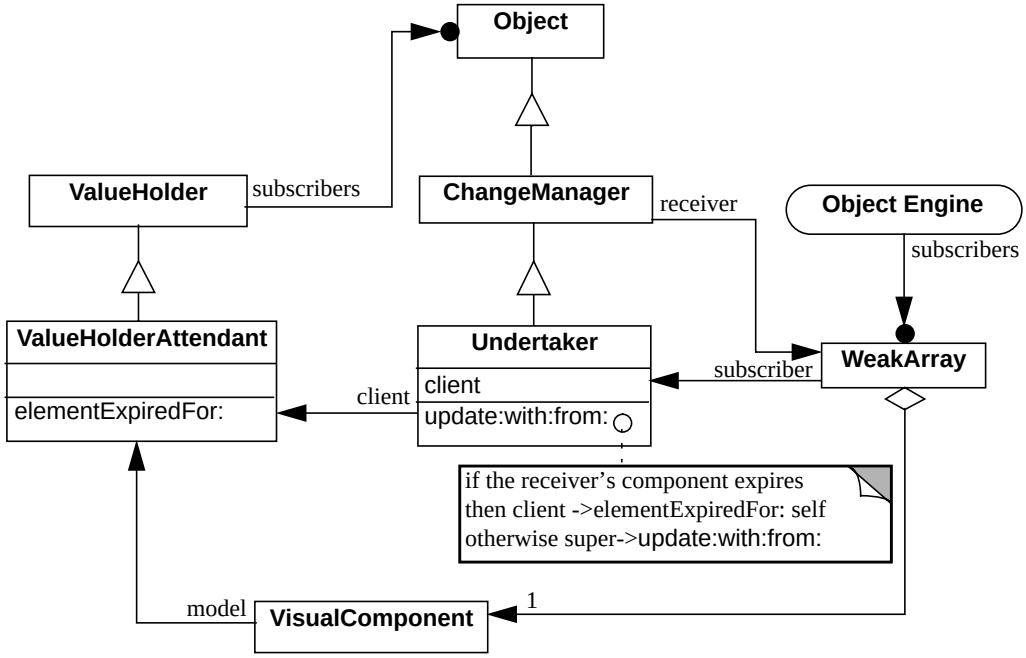

FIGURE 3 Undertaker structure

Participants

- Publisher-subscriber, or observer—the undertaker pattern inherits services from the Publisher-subscriber pattern in general and the `ChangeManager` in particular.
- `Undertaker`—subclass of a `ChangeManager` mediates the coupling from the publisher (`ValueHolderAttendant`) to the subscriber (`VisualComponent`).
- `ValueHolderAttendant`—encapsulates the subject (i.e. the value) and knows how to handle dependencies and updates. It knows in particular how to react upon expiration of the visual component.
- `WeakArray`—is automatically dependent (subscriber) on the Object Engine and is informed if any of its elements expires.
- Object Engine—a system component that controls the lives of the objects.

Collaborations

The components makes themselves dependent on `ValueHolderAttendant` and it will in turn use the undertaker to take care of its subscribers.

The subscriptions on the value holder is indirect through the `Undertaker`, the undertaker creates a `WeakArray` assign the `VisualComponent` to it and makes itself subscriber to changes of it. The `WeakArray` will automatically be dependent on the `Object Engine`.

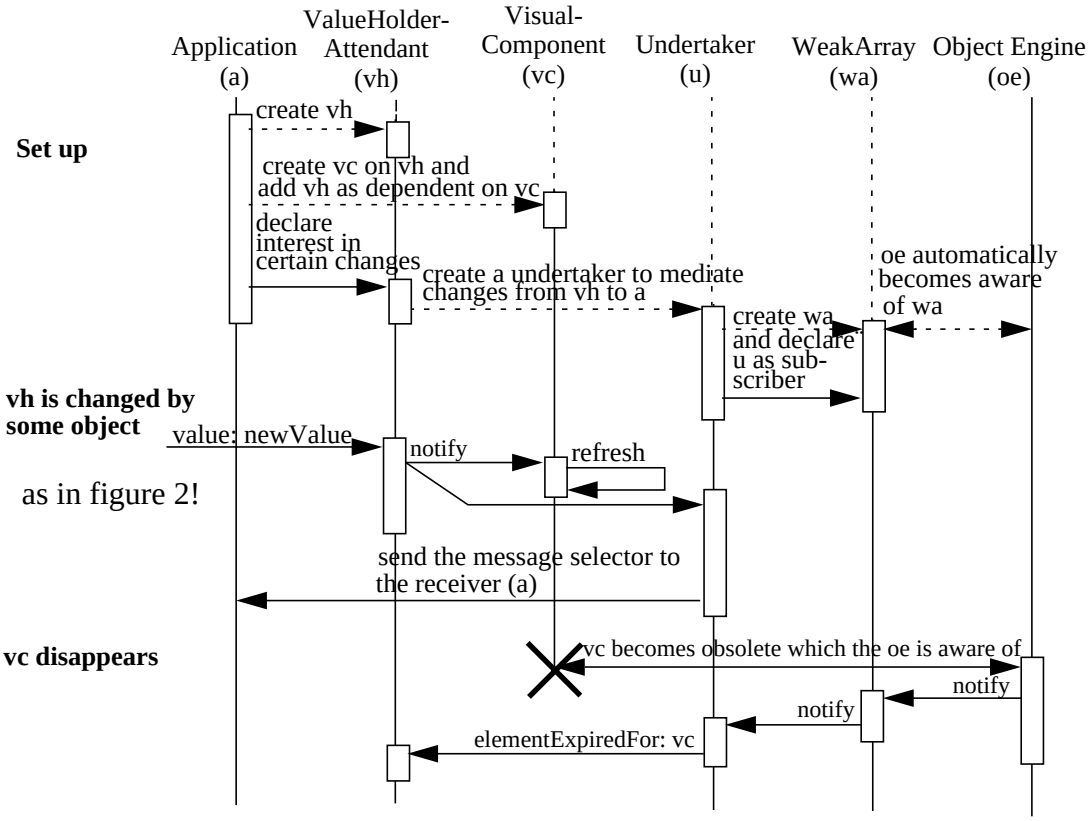

FIGURE 4 Message trace for the `Undertaker`

If the `VisualComponent` ceases to exist the `Object Engine` notifies all `WeakArrays` that have the `VisualComponent` as one of its elements. Since the `Undertaker` in turn is subscriber to changes of the `WeakArray` it is also notified. Then, finally, the `ValueHolderAttendant` is informed and is able to take appropriate actions.

Consequences

All the consequences of the publisher-subscriber pattern, i.e. *abstract coupling between publisher and subscriber*, support for *broadcast communication* and a risk for *unexpected updates,* are also valid for the undertaker pattern.

Further benefits and liabilities of the Undertaker pattern include the following:

1. *solves the dangling references problem*.
2. *supports remoteness of objects*. Since the "real" objects only are implicitly dependent on each other we could more easily, than in for instance the publisher-subscriber pattern, hide the location of the publisher, e.g. by replacing the `ValueHolderAttendant` with a proxy.
3. *delayed notification of object finalization*. In some systems there could be a delay between the time we deference the concrete subscriber and the time the undertaker is informed (usually when the next garbage collection occurs). Then their is a risk that the publisher list of subscribers will contain obsolete objects.

Implementation

Given the publisher-subscriber pattern with a `ChangeManager` the undertaker is quite straight forward in a system that provides a specific object finalization facility, as VisualWorks (ParcPlace-Digitalk 1995a) and VisualSmalltalk (ParcPlace-Digitalk 1995a).

In this particular implementation we have decided to use one `WeakArray` for each pair of `Undertaker` and `VisualComponent` (see figure 3—i.e. the arity of the aggregation). With minor modifications we could instead use the same `WeakArray` for several pairs of `Undertaker` and `VisualComponent`.

Sample code

We define the `ValueHolderAttendant` as subclass of `ValueHolder` (to enjoy all the superclass' benefits).

```
ValueHolder subclass: #ValueHolderAttendant
```

As an alternative to inheritance an object adapter could be used.

The following message `elementExpiredFor:` will be sent from the subscriber if it ceases to exist. Here we simply remove the subscriber from the register of dependants.

```
elementExpiredFor: aDep
   self removeDependent: aDep
```

The following two methods only differs from the methods in the superclass in that they instead of handling a DependencyTransformer, i.e. the inherited change manager, also takes care of cases where the subscriber is a Undertaker. The first of the methods takes care of objects that declare interests in specific changes to the receiver.

```
expressInterestIn: anAspect for: client sendBack: aSelector
   "Arrange to receive a message with aSelector when anAspect
   changes
   at client"
   | undertaker |
   undertaker := Undertaker new.
   undertaker
      setReceiver: client
      aspect: anAspect
      selector: aSelector
      for: self.
   self addDependent: undertaker
```

And the second method retracts a specific interest.

```
retractInterestIn: anAspect for: anObject
   "Undo a send of expressInterestIn:for:sendBack:"
   | deps |
   deps := self myDependents.
   deps == nil ifTrue: [^self].
   ((deps isKindOf: DependencyTransformer) and:
        [deps matches: anObject forAspect: anAspect])
     ifTrue: [^self removeDependent: deps].
   (deps class == DependentsCollection) ifFalse: [^self].
   1 to: deps size do:
     [:i | | dep |
     dep := deps at: i.
     ((dep isKindOf: DependencyTransformer) and:
          [dep matches: anObject forAspect: anAspect])
       ifTrue: [^self removeDependent: dep]]
```

The enhanced change manager—Undertaker—is defined in the following way:

```
DependencyTransformer subclass: #Undertaker
   instanceVariableNames: 'client '
```

The instance variable client will point to the original publisher (i.e the value holder) and the inherited instance variable receiver will be assigned to a weak array only containing aReceiver (which is the visual component). In DependencyTransformer the receiver is pointing directly to a aReceiver and the knowledge about the original publisher is absent.

```
setReceiver: aReceiver aspect: anAspect selector: aSymbol for: aClient
   receiver := WeakArray with: aReceiver.
   receiver addDependent: self.
   client := aClient.
   … assign call-back method and check if the arity of it is valid …
```

The next method takes care of both ordinary changes and expiration of the receiver.

```
update: anAspect with: parameters from: anObject
   (anObject == receiver and: [anAspect = #ElementExpired])
      ifTrue: [^client elementExpiredFor: self].
   "The rest of the method is in principle equivalent to the
   superclass' update:with:from:"
   aspect == anAspect ifFalse: [^self].
   … Send the chosen call-back method to (receiver at: 1) …
```

Finally to get everything working we had to override the following method which is used while retracting interests from the value holder.

```
matches: anObject forAspect: anAspect
   ^(receiver at: 1) == anObject and: [aspect == anAspect]
```

Since the interest in changes to a value holder is so common the ValueHolder provides the following method, where the client not is forced to remember the value holder's particular change-trigger #value.

```
onChangeSend: aSymbol to: anObject
   "Arrange to receive a message with aSymbol when the value aspect
   changes on anObject."
   self
     expressInterestIn: #value
     for: anObject
     sendBack: aSymbol
```

By inheritance this more convenient message is immediately applicable for the undertaker's ValueHolderAttendant.

Known Uses

We have used the undertaker to enhance the application builder of VisualWorks with persistence and distribution services. Here we used the undertaker to reclaim obsolete references from persistent or remote objects to applications developed by the application builder (normally caused by the termination of the application).

Besides using this pattern in utilizing the application builder of VisualWorks we have used it for handling references to external units, as files and in servers to for instance inform interested participants if an client connection is released, expired or by some other reason becomes corrupted. This approach is used in MultiGossip (Eiderbäck 1993) to both control the system as such, e.g. in browsers aimed at controlling connections, and in more mundane services as transparently managing objects situated in files.

Another useful type of situation is while several clients share a specific unit and this unit requires that only one client at a time uses its services. If it is important that each client not unnecessary possesses the unit we must in some way arrange for the client to “give the unit back”. We have successfully used the undertaker in such a situation for developing a sound manager at a UNIX platform (where the audio port only could be owned by one client at a time).

Related Patterns

Publisher-Subscriber—the undertaker is an enhancement of the publisher-subscriber pattern.

Adapter—a specific adapter could be made to behave like an undertaker.

Mediator—also posses a loose coupling between the involved objects.

3 Summary and Conclusions

We have described the pattern undertaker and outlined how it can be used to solve problems with dangling references. By using this pattern in utilizing the VisualWorks environment with distribution and persistence facilities we have been able to hide implementation details from the application programmer and thereby let him/her focus on problems in the application's domain instead.

In languages with no reclamation facility built into the environment it would certainly be straight forward to incorporate the Undertaker pattern by first adding an idiom such as the *counted pointer*—described in (Buschmann et al 1996)—to the environment.

4 Future Work

This work has taken place as an effort to solve a particular problem within a specific domain, i.e. within VisualWorks. Although we believe that it would be of interest to generalize upon the description of the pattern, by for instance, investigate what it takes to implement it by means of Java with in many senses enjoys similar semantics and class library as Smalltalk's.

References

Eiderbäck 1993	Eiderbäck, B. and Hägglund, P. *MultiGossip – a General Distribution Package in Smalltalk*, in Proc. of the First International Conference on Multi-Media Modelling, Singapore, Nov 9-12, 1993, pp. 293-307.
Buschmann et al 1996	Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern - Oriented Software Architecture A System of Patterns,* Wiley, 1996.
Gamma et al 1995	Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.
Lewis 1995	Lewis, S. *The Art and Science of Smalltalk,* Prentice Hall, 1995.
ParcPlace 1992	ParcPlace Systems Inc. *ParcPlace VisualWorks\Smalltalk-80, ver. 1.0,* 1992.
ParcPlace-Digitalk 1995a	ParcPlace-Digitalk Inc. *VisualWorks\Smalltalk-80, ver. 2.5*, 1995.
ParcPlace-Digitalk 1995b	ParcPlace-Digitalk Inc. *VisualWave, ver. 1.0*, 1995.
ParcPlace-Digitalk 1995c	ParcPlace-Digitalk Inc. *VisualSmalltalk Enterprise, ver. 3.1,* 1995.