Object Recovery Pattern

António Rito Silva, João Pereira and José Alves Marques
INESC/IST Technical University of Lisbon
R. Alves Redol n°9, 1000 Lisboa, PORTUGAL
Tel: +351-1-3100287, Fax: +351-1-3145843
Rito.Silva@inesc.pt

Conference on Pattern Languages of Programs (PloP '96), Allerton Park, Illinois, September 4-6, 1996.

Abstract

The Object Recovery pattern defines a generic object recovery algorithm. It decouples the recovery part from the object's functional part. Different policies can be supported: update-in-place, deferred-update, copy and compensating. The Object Recovery pattern isolates recovery from persistence and object synchronization issues allowing the recovery of transient objects to a previously defined state. Moreover, the recovery pattern can be used to implement user undo/redo models of interaction.

1 Intent

The Object Recovery pattern abstracts several object recovery policies. It decouples object recovery policies from object-specific functionality.

2 Also Known As

Log.

3 Motivation

3.1 Example

Consider the design of a Distributed Drawing Application allowing cooperative manipulation of draw documents. Users at different terminals can simultaneously access the same draw document and changes made to a local view are immediately propagated to other local views.

A possible architecture (Figure 1) for this application has several client applications having their own objects (application space) and sharing a set of domain objects (domain space) which may be kept in a data store (persistent space). The application space has interface objects, e.g scrollbars and shapes; the domain space has shared objects, e.g. shape's data; and, the persistent space stores documents, e.g. a draw document including its shape's data.

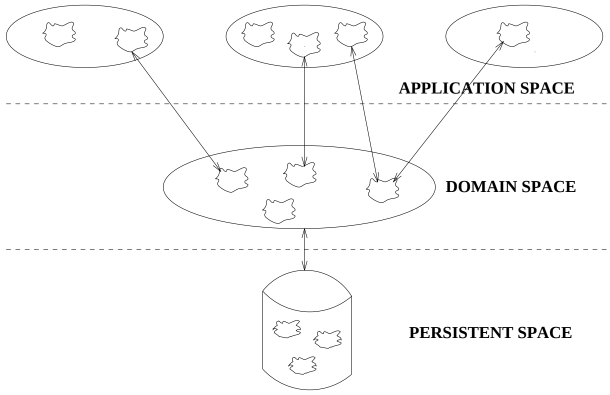

Figure 1: Distributed Drawing Application.

The architecture uses several design patterns from [1]. Users interact with the application space and the ***Command Pattern*** is applied to encapsulate users requests. Commands invoke operations on both application space objects and on domain space objects. Typical commands are create, delete, and update of a shape object. The ***Observer Pattern*** is applied to shapes in the application and domain spaces. A domain shape is shared by several application shapes which are the observers. Domain shapes are proxy objects of persistent shapes (***Proxy Pattern***). Proxies know how to read/write themselves from/to the persistent space. A proxy object caches data read from the persistent space and invocations are executed on cached data.

3.2 Problem

The distributed drawing application may need to undo operations for three different reasons:

1. The user decides to undo a command and thus it is necessary to recover all the invoked objects to a previous state. The user uses a undo/redo model of interaction.

2. Some operations done by client applications invoke methods on shared domain objects. Invocations to domain objects use a synchronization policy to avoid data corruption. In some conditions sinchronization policies need to retract operation effects.

3. A system failure may happen when shape objects are being written to the persistent space so that it is necessary to restore the persistent space to a consistent state.

Sometimes, different solutions are constructed for each of these three problems without reusing the similar parts. Another approach is to treat the three problems as a single problem (the worst problem) and use a single solution (the more complex solution): recovery is done at the persistent space's level.

3.3 Forces

An object-oriented solution for the above problem has to solve the following forces:

- Encapsulation requires the object recovery part to be placed within the object itself rather than spread out among its clients. The object is responsible for its own recovery because the recovery policy may need to access the object's internal representation.

- Modularity requires separation of recovery from persistency, synchronization, failure recovery and object functionality. For instance, recoverable objects should be persistent, but sometimes volatile objects also need to be recoverable, e.g. the user expects the scrollbar to be undone. This orthogonality enhances reuse and allows for policy switching with no repercussions on other components.

- Extensibility requires abstraction of recovery policies. It is not possible to find an optimal policy for all situations: policies should be customized. The more suitable policy depends on the domain object and its operations semantics.

4 Applicability

Use the Object Recovery pattern when:

- *Operation effects may need to be rollbacked* - effects of invocations on objects may need to be retracted due to an external decision.

- *It is premature to decide on which recovery policy to use* - it may be early to decide on a specific policy at a given development process stage. It should be possible to test the performance of several policies before choosing. A predefined default recovery policy can be reused to test the application's functionality.

- *It is necessary to recover transient objects* - recovery should be independent of persistence. The decision to recover is not only due to a crash but also to application semantics as user's undo and synchronization.

5 Structure

The Booch[2] class diagram in Figure 2 illustrates the Object Recovery pattern's structure:

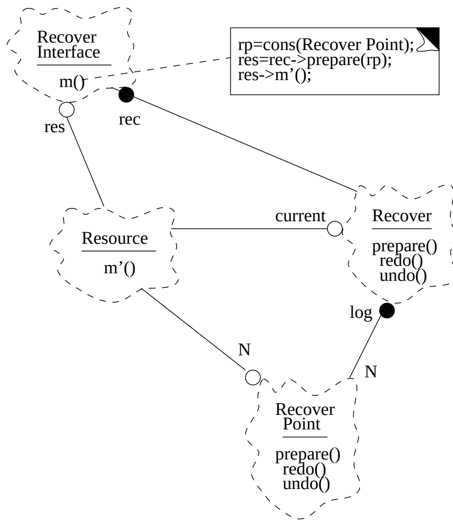

Figure 2: Structure.

6 Participants

The main participants in the Object Recovery pattern are:

- **Recover Interface.** It is responsible for intercepting invocations to the `Resource` object encapsulating it. Intercepted invocations are directed to a `Resource` object returned by the `prepare` operation. Effects of invocations made using this interface are recoverable.

- **Recover.** Defines the object-independent part of the recovery policy. It refers the current `Resource` object, which state, depending on the policy, may reflect the last changes. The current `Resource` object is used as the initial point for redoing and undoing invocations effects. Contains and manages a list (`log`) of

Recover Points which are used to update the current Resource. Recover integrates operations prepare, redo and undo such that a consistent policy is implemented. Particular policies are implemented by reusable Recover subclasses.

- **Recover Point.** Defines object-specific semantics which are used by the generic algorithm. It is responsible for preparing, redoing, and undoing Resource object invocations. Operations prepare, redo, and undo may have to know the internal resource's data and which are the effects of invoking the Resource object.

- **Resource.** Encapsulates data that is invoked and which may be recovered. It may define an internal interface to be used by Recover Point objects. To support recoverability several Resource objects may coexist.

7 Collaborations

The Booch[2] interaction diagram in Figure 3 illustrates collaborations between objects in the Object Recovery pattern:

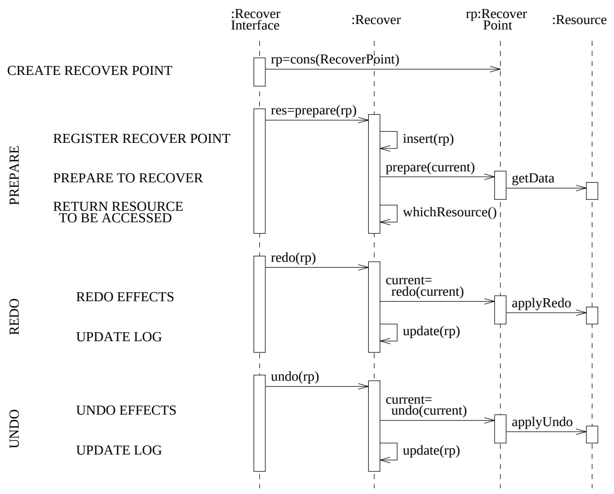

Figure 3: Collaborations.

The diagram describes four phases of collaboration:

1. *Create Recover Point.* This phase creates the Recover Point object, which identifies the operation invocation during the other phases.

2. *Prepare.* The Recovery Interface object sends a Recover Point object to the Recover and receives as reply the Resource object that should be invoked. During this phase the Recover Point is inserted in the log list and data needed for recovery actions may be copied. The Recover object may return, to the Recover Interface either the current Resource object or a copy of it, depending on the policy being implemented (operation whichResource).

3. *Redo.* The redo phase sets current to a Resource object which state is the result of the invocation identified by the Recover Point object. In some policies the Resource object is already up-to-date because the operation invocation was done on the current Resource object, while in others the current Resource object needs to be updated because the operation is executed in a copy. Finally the list of Recover Point objects is updated[1].

4. *Undo.* The undo phase sets current to a Resource object which state is the same as before it was invoked by the operation identified by the Recover Point object. In some policies the current Resource state does not reflect the last changes, because the invocation was done in a copy, while in others the current Resource needs to be restored. Finally, the list of Recover Point objects is updated as in the redo phase.

8 Consequences

This pattern has some advantages:

- *Separates the recovery code from the resource code*, allowing separate development, test and reuse.

- *Separates the recovery policy from persistence*, allowing the recovery of transient objects. This way, the pattern can be used for both persistent and transient objects. The writing of the Recover and Recover Point objects on the persistent space depends on failure recovery policies, e.g. write-ahead.

- *Supports several recovery policies*, using the same generic interface. It is possible to replace policies (see section 9) by replacing Recover and Recover Point objects.

- *Separates the object-independent part of the recovery policy from the object-specific part*, such that it is possible to reuse the generic part of algorithms. Objects Recover and Recover Point hold, respectively, the object-independent and object-specific parts of the algorithm.

[1] It can remove the Recover Point object from the log, which is policy dependent.

This patterns has some disadvantages:

- *There is an overhead* when a straightforward use of the pattern is made. Objects of class `Recover Point` may need to be created[2] and deleted per each invocation, thus penalizing performance.

9 Implementation

The Object Recovery pattern may be implemented in several ways. This section discusses some issues and possibilities:

- *Definition of the object-independent part of policies.* Recovery policies can use two different generic algorithms: update-in-place, when the current `Resource` object reflects the last changes, and deferred-update, when the invocation's updates do not occur on the current `Resource` object.

  - *Update-in-place policies* make changes in the current `Resource` object that, if it is necessary to recover, must be undone. To implement an update-in-place policy the `Recover` object should return the current `Resource` object at the end of the prepare phase. Afterwards, if a undo happens it is necessary to undo the changes on the current `Resource`.
  - *Deferred-update policies* do not change the current `Resource` object so that, when the invocation commits, changes must be redone on it. To implement a deferred-update policy the `Recover` object should return a copy of the current `Resource` object at the end of the prepare phase. Afterwards, if a redo happens it is necessary to propagate the changes to the current `Resource`.

- *Definition of the object-specific part of policies.* This definition is supported by `Recover Point` objects. There are two main solution groups: either copying state, when recovery is done using copies of the resource's state; or compensating operations, when recovery is done using operations that know how to return to a previous state based on the current state.

  - *Copy policies* use copies of the resource's state to redo and undo effects. Operation `prepare` of `Recover Point` object copies the state of the current `Resource` object. Operations `redo` and `undo` use the copied state to change the current `Resource`.
  - *Compensating policies* use the current state instead of doing recovery based on data copies. In this policy it is not necessary to copy the resource's data during the prepare phase. The `Recover Point` object holds operations that know how to redo and undo invocations effects without using a previous resource's state.

  Special care is necessary if a compensating policy is used with a deferred-update policy because the `Recover Point` object needs to return a `Resource` object's copy for the invocation to proceed, even if the copy is not used afterwards.

- *Integration with object synchronization.* Object synchronization may need to recover the resource's state due to synchronization conflicts. There are two generic kinds of synchronization policies: pessimistic, when the object is expected to have high contention, and optimistic, when the level of contention is supposed to be low. The integration of recovery with synchronization is not straightforward: some combinations of policies are not possible and other penalize performance. In [3] it is proved that some operation compatibility relations require a particular kind of recovery policy. Common combinations are optimistic synchronization with deferred-update recovery policies and pessimistic synchronization with update-in-place policies. The designer must be aware of allowed and best combinations. The Customizable Object Synchronization pattern described in [4] supports several object synchronization policies.

- *Integration with user undo/redo models of interaction.* A single-user undo/redo model maintains an history of executed commands which is traversed forward and backward by the user doing respectively redo and undo. The undo (redo) of a command should be propagated to all the objects invoked during its execution. Since a command can be undone and redone several times, the `Recover Point` objects should not be discarded (operation `update` of the `Recover` object) and the log will need to be updated. To support user undo/redo a update-in-place policy is used. The log represents the object's history.

[2] When the invocation does not change the `Recover` object, it is not necessary to create a `Recover Point` object.

- *Integration with failure recovery.* This pattern can be used in conjunction with other patterns to achieve object robustness. Two problems inherent to crash recovery are: disk atomicity, which is different from object atomicity (writing an object on disk may not correspond to writing a single atomic block); and, disk accesses should be delayed until enough data is available to be written to disk (main memory is much faster than disk and the application should not be delayed). To solve these problems it is necessary to write the set of recover points in a single block which will be atomically written to disk. Furthermore, it is necessary to define a special kind of recover point to mark successful or unsuccessful invocations. These marks are used to know if an update to disk should be recovered or not (a crash may happen when writing recover points to disk). The use of write-ahead technique is required when writing objects on disk (a crash may happen when writing objects to disk). Note that as the persistent space uses the recovery pattern, the log on disk is a list of `Recover Point` objects and a `Recover` object is needed to recover the persistent object from the persistent log.

10 Sample Code

This section describes a framework supporting several policies of recovery, and the integration of recovery with synchronization in the context of the Distributed Drawing Application.

10.1 Policies Implementation

A set of abstract classes are defined, as well as their specializations for synchronization deferred-update, synchronization update-in-place, user undo/redo update-in-place and copy policies. Due to the emphasis on reuse, the implementation of Compensating policies is not shown since they can not be made resource-independent.

- *Definition of abstract classes.* Abstract classes define a framework which must be redefined according to the chosen policies. This framework forms a skeleton that enforces reusability. To improve reusability classes are parameterized by `RESOURCE`.

  Class `Recover` defines the interface of the recovery policy's object-independent part. The implementation of virtual methods `whichResource`, `redo` and `undo` are policy specific. Method `prepare` is implemented as a template method [1].

```
template <class RESOURCE>
class Recover
{
public:
  RESOURCE *prepare(RecoverPoint<RESOURCE>*);
  virtual void redo(RecoverPoint<RESOURCE>*) = 0;
  virtual void undo(RecoverPoint<RESOURCE>*) = 0;
protected:
  virtual RESOURCE *whichResource(RESOURCE*) = 0;
private:
  RESOURCE *current_; // current resource
  Queue<RESOURCE> log_; // queue of recover points
};
```

The following template method sets the generic part of the policies while the specific part will be defined in subclasses.

```
RESOURCE *Recover<RESOURCE>::
prepare(RecoverPoint<RESOURCE> *rp)
{
  log_.insert(rp);
  RESOURCE *res = rp->prepare(current_);
  return whichResource(res);
}
```

Class `RecoverPoint` defines the interface of the recovery policy's object-specific part as follows:

```
template <class RESOURCE>
class RecoverPoint
{
public:
  virtual RESOURCE *prepare(RESOURCE*) = 0;
  virtual RESOURCE *redo(RESOURCE*) = 0;
  virtual RESOURCE *undo(RESOURCE*) = 0;
};
```

- *Definition of an update-in-place policy for synchronization.* Class `UPCCRecover` implements this policy. Method `whichResource` returns the current resource because the invocation is going to proceed in it. Method `redo` does not update `current_` because changes were done in the current resource while method `undo` needs to update it. Both, remove the `Recover Point` object from `log_`.

```
RESOURCE *UPCCRecover<RESOURCE>::
whichResource(RESOURCE *res)
{
  return current_;
}

void UPCCRecover<RESOURCE>::
redo(RecoverPoint<RESOURCE> *rp)
{
  log_.remove(rp);
}

void UPCCRecover<RESOURCE>::
undo(RecoverPoint<RESOURCE> *rp)
{
  current_ = rp->undo(current_);
  log_.remove(rp);
}
```

- *Definition of a deferred-update policy for synchronization.* Class `DUCCRecover` implements this policy. Method `whichResource` returns a copy of the current resource because the invocation is going to proceed in it. Method `redo` changes the current resource while method `undo` do not since the invocation was done on a copy.

```
RESOURCE *DUCCRecover<RESOURCE>::
whichResource(RESOURCE *res)
{
  return res;
}

void DUCCRecover<RESOURCE>::
redo(RecoverPoint<RESOURCE> *rp)
{
  current_ = rp->redo(current_);
  log_.remove(rp);
}

void DUCCRecover<RESOURCE>::
undo(RecoverPoint<RESOURCE> *rp)
{
  log_.remove(rp);
}
```

- *Definition of a update-in-place policy for user undo/redo.* Class `UPURRecover` implements this policy. A `Recover Point` object can be undone and redone several times and it is not discarded (methods `redo` and `undo`).

```
RESOURCE *UPURRecover<RESOURCE>::
whichResource(RESOURCE *res)
{
  return current_;
}

void UPURRecover<RESOURCE>::
redo(RecoverPoint<RESOURCE> *rp)
{
  current_ = rp->redo(current_);
}

void UPURRecover<RESOURCE>::
undo(RecoverPoint<RESOURCE> *rp)
{
  current_ = rp->undo(current_);
}
```

- *Definition of a Copy policy.* The implementation of a resource-independent copy policy follows. The `Recover Point` objects can be used with any kind of recovery policies: deferred-update and update-in-place. This flexibility has costs on performance since copy objects can be unnecessarily created. The new instance variable `cpy` holds a resource's copy.

```
template <class RESOURCE>
class CopyRecoverPoint
    : public RecoverPoint<RESOURCE>
{
public:
  RESOURCE *prepare(RESOURCE*);
  RESOURCE *redo(RESOURCE*);
  RESOURCE *undo(RESOURCE*);
private:
  RESOURCE *cpy;
};
```

This policy can be reused by any type of resource if a `clone` method is provided by the `RESOURCE` object. Note that in both methods, `redo` and `undo`, the instance variable `cpy` is updated to allow sequences of redo/undo.

```
RESOURCE *CopyRecoverPoint<RESOURCE>::
prepare(RESOURCE *cur)
{
  cpy = cur->clone(); // make a copy of current
  return cpy;
}

RESOURCE *CopyRecoverPoint<RESOURCE>::
redo(RESOURCE *cur)
{
  RESOURCE *res = cpy;
  cpy = cur;
  return res;
}

RESOURCE *CopyRecoverPoint<RESOURCE>::
undo(RESOURCE *cur)
{
  RESOURCE *res = cpy;
  cpy = cur;
  return res;
}
```

Note that this is a very generic copy policy implementation, other implementations which are object-dependent are possible.

10.2 Distributed Drawing

In the distributed drawing application invocations to shared rectangles need to be synchronized. We adopted an optimistic synchronization policy which allows concurrent invocations by conflicting operations. Therefore, a deferred-update (copy) policy was chosen.

The class `RectShape` represents a rectangle. It knows its coordinates, origin and extent, and how to update them when the rectangle is moved or resized. The `Shape` class is an abstract class from which the several drawing classes are derived.

```
class RectShape : public Shape
{
public:
  RectShape(const Point& ori, const Point& ext){
    origin = ori; extent = ext; }

  void MoveBy(Point delta) {
    origin += delta; }

private:
  Point origin, extent;

  friend class RectRecPoint;
```

```
  void SetRect(const Point& ori, const Point& ext) {
    origin = ori; extent = ext; }

  Point GetOrigin() {
    return origin; }

  Point GetExtent() {
    return extent; }
};
```

The class `RectRecPoint` is a subclass of `RecoverPoint<RectShape>` class. This subclass is tuned for a deferred-update policy in the context of synchronization (operation `undo` is empty!).

```
class RectRecPoint : public RecoverPoint<RectShape>
{
public:
  virtual RectShape *prepare(RectShape *cur) {
    RectShape *rect = cur;
    cpy_ = new RectShape(rect->GetOrigin(),
                         rect->GetExtent());
    return cpy_; }

  virtual RectShape *redo(RectShape *cur) {
    return cpy_; }

  virtual RectShape *undo(RectShape *cur) {
    // empty }

private:
  RectShape *cpy_;
};
```

Class `RectShapeInterface` supports recovery and an optimistic synchronization policy. When an invocation proceeds a `RectRecPoint` object is created and prepared. The method execution continues in the `rect` object returned by operation `prepare`. Afterwards it is verified whether there were concurrent conflicting invocations (operation `postControl`). Operations `preControl` and `postControl` are responsible for synchronization. In the context of an optimistic policy the former registers the invocation while the latter verifies whether there were conflicting invocations.

```
void RectShapeInterface::MoveBy(Point delta)
{
  RectRecPoint *recPoint;
  RectShape *rect;

  if (preControl() == eCONTINUE)
  {
    recPoint = new RectRecPoint();
    rect = recover_->prepare(recPoint);
  }
  else return;

  rect->MoveBy(delta);

  if (postControl() == eERROR)
    recover_->undo(recPoint);
  else
    recover_->redo(recPoint);
}
```

11 Known Uses

The need for recovery is widely recognized in database and distributed systems. Most of the solutions to this problem restrict the number of policies supported and do not decouple recovery from other issues.

[5] describes two different recovery policies: update-in-place and deferred-update.

Distributed systems, e.g. Argus [6], Arjuna [7] and Hermes/ST [8], use the recovery pattern encapsulated in the platform mechanisms. Argus uses a deferred-update policy, invocations are done on a version of the object which is discarded if the invocation is rollbacked. Arjuna uses a copy policy: a class `recoveryRecord` is defined which contains the previous object's state to be used in case of rollback. In Hermes only the persistent objects are recoverable, Hermes uses a copy policy that allows recovery with attribute granularity.

The architecture of the object-oriented model for distributed transaction processing presented in [9] provides recoverability for objects. The recovery pattern is used in the context of distributed failure recovery of transactions.

12 Related Patterns

The ***Customizable Object Synchronization Pattern*** [4] abstracts several object synchronization policies. It decouples object synchronization from object concurrency and object functionality (sequential part). Some of the supported policies require object recovery. In 9 we described the trade-offs when integrating these two patterns.

The ***Memento Pattern*** [1] is used to externalize object state, without violating encapsulation, such that the object can be restored later. In the Object Recovery pattern `Recover Point` objects contain, for copy policies, the `Resource` object state. In this case the Memento pattern can be applied between the `Resource` and the `Recover Point` where the former plays the role of `Originator` and the latter plays the role of `Caretaker`.

The ***Recoverable Distributed Pattern*** [10], as the Object Recovery pattern, emphasizes the separation of concerns. It decouples failure detection from recovery.

Acknowledgments

We would like to thank David Matos for his careful proofreading and Francisco Rosa for his comments.

References

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

[2] Grady Booch. *Object-Oriented Analyis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[3] William Weihl. The Impact of Recovery in Concurrency Control. *Journal of Computer and System Sciences*, 47(1):157–184, August 1993.

[4] António Rito Silva, Joao Pereira, and José Alves Marques. Customizable Object Synchronization Pattern. In *European Conference on Pattern Languages of Programs, EuroPLoP '96*, Kloster Irsee, Germany, July 1996.

[5] William Weihl. Transaction-Processing Techniques. In Sape Mullender, editor, *Distributed Systems - Second Edition*, pages 329–352. Addison-Wesley, 1993.

[6] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[7] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An overview of the arjuna distributed programming system. *IEEE Software*, January 1991.

[8] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Concurrency control for distributed nested transactions in hermes. *International Conference for Concurrent and Distributed Systems*, 1993.

[9] Michael Mock, Reinhold Kroeger, and Vinny Cahill. Iimplementing Atomic Objects with the RelaX Transaction Facility. *Computing Systems*, 5(3):259–304, Summer 1992.

[10] Nayeem Islam and Murty Devarakond. Recoverable Distributor: A Design Pattern for Fault-Tolerant state Sharing in Distributed Computing, October 1995. OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems.