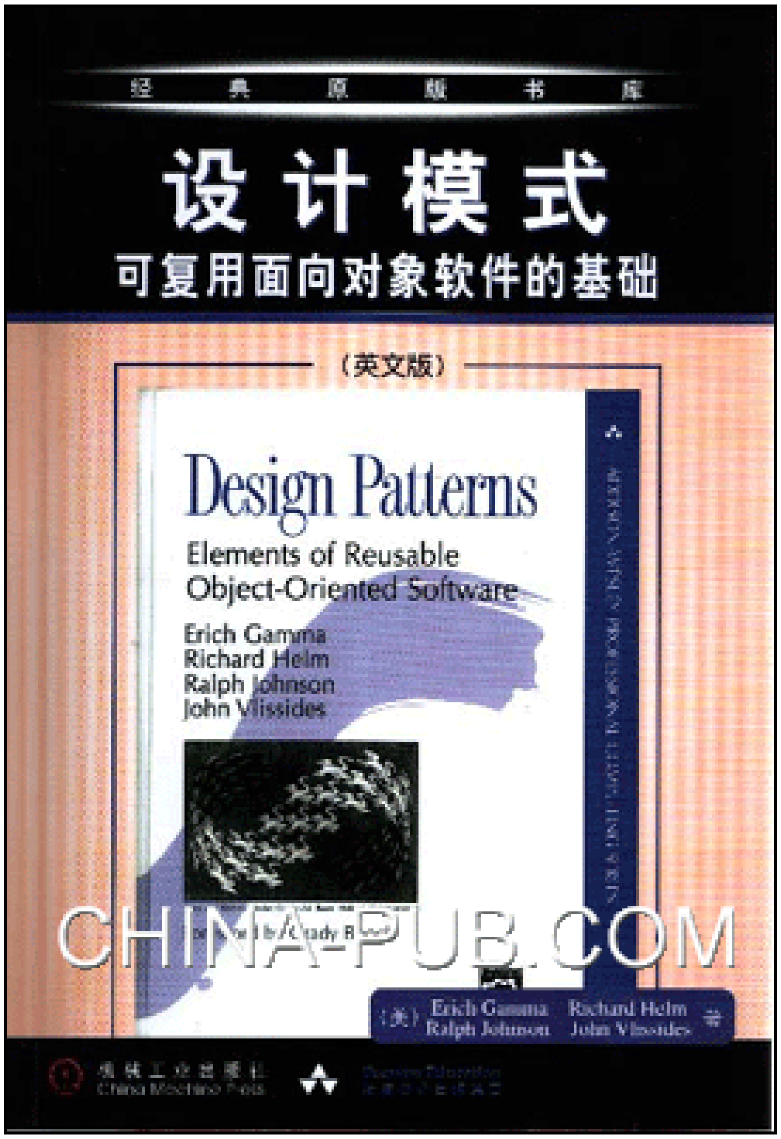


Design Patterns

Elements of Reusable Object-Oriented Software

Produced by KevinZhang

Memento

Intent

Without violating encapsulation, capture and externalize an object'sinternal state so that the object can be restored to this state later.

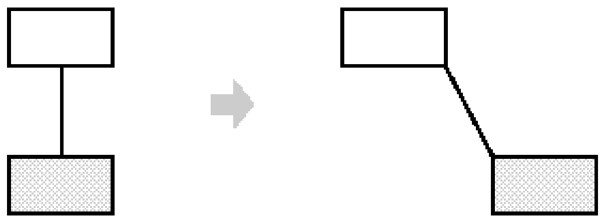
Also Known As

Token

Motivation

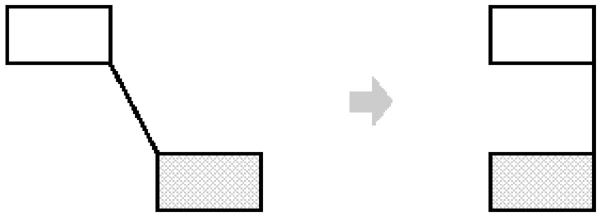
Sometimes it's necessary to record the internal state of an object.This is required when implementing checkpoints and undo mechanismsthat let users back out of tentative operations or recover fromerrors. You must save state information somewhere so that you canrestore objects to their previous states. But objects normallyencapsulate some or all of their state, making it inaccessible toother objects and impossible to save externally. Exposing this statewould violate encapsulation, which can compromise the application'sreliability and extensibility.

Consider for example a graphical editor that supports connectivitybetween objects. A user can connect two rectangles with a line, andthe rectangles stay connected when the user moves either of them. Theeditor ensures that the line stretches to maintain the connection.



A well-known way to maintain connectivity relationships betweenobjects is with a constraint-solving system. We can encapsulate thisfunctionality in a **ConstraintSolver** object.ConstraintSolver records connections as they are made and generatesmathematical equations that describe them. It solves these equationswhenever the user makes a connection or otherwise modifies thediagram. ConstraintSolver uses the results of its calculations torearrange the graphics so that they maintain the proper connections.

Supporting undo in this application isn't as easy as it may seem. Anobvious way to undo a move operation is to store the original distancemoved and move the object back an equivalent distance. However, thisdoes not guarantee all objects will appear where they did before.Suppose there is some slack in the connection. In that case, simplymoving the rectangle back to its original location won't necessarilyachieve the desired effect.



In general, the ConstraintSolver's public interface might beinsufficient to allow precise reversal of its effects on otherobjects. The undo mechanism must work more closely withConstraintSolver to reestablish previous state, but we should alsoavoid exposing the ConstraintSolver's internals to the undo mechanism.

We can solve this problem with the Memento pattern. A **memento** is an object that stores a snapshot of theinternal state of another object — the memento's **originator**. The undo mechanism will request a mementofrom the originator when it needs to checkpoint the originator'sstate. The originator initializes the memento with information thatcharacterizes its current state. Only the originator can store andretrieve information from the memento—the memento is "opaque" toother objects.

In the graphical editor example just discussed, the ConstraintSolver can actas an originator. The following sequence of events characterizes theundo process:

1. The editor requests a memento from the ConstraintSolver as aside-effect of the move operation.
2. The ConstraintSolver creates and returns a memento, an instance of aclass SolverState in this case. A SolverState memento contains datastructures that describe the current state of the ConstraintSolver'sinternal equations and variables.
3. Later when the user undoes the move operation, the editor gives theSolverState back to the ConstraintSolver.
4. Based on the information in the SolverState, the ConstraintSolverchanges its internal structures to return its equations and variablesto their exact previous state.

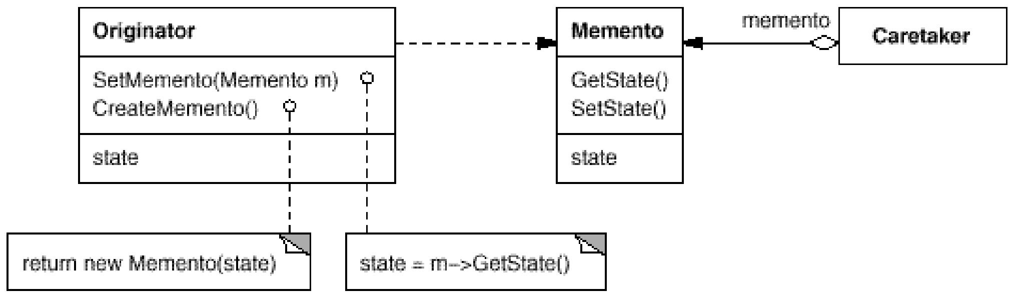
This arrangement lets the ConstraintSolver entrust other objects withthe information it needs to revert to a previous state withoutexposing its internal structure and representations.

Applicability

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved sothat it can be restored to that state later, *and*
- a direct interface to obtaining the state would exposeimplementation details and break the object's encapsulation.

Structure

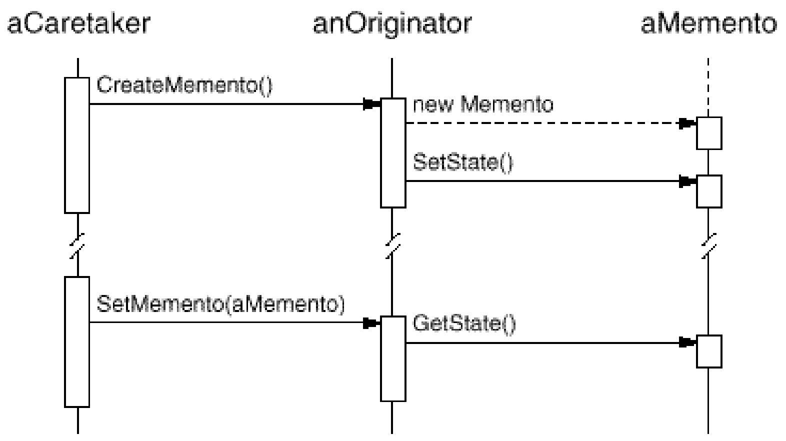


Participants

- **Memento** (SolverState)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a *narrow* interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.
- **Originator** (ConstraintSolver)
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state.
- **Caretaker** (undo mechanism)
  - is responsible for the memento's safekeeping.
  - never operates on or examines the contents of a memento.

Collaborations

- A caretaker requests a memento from an originator, holds it for atime, and passes it back to the originator, as the followinginteraction diagram illustrates:



  Sometimes the caretaker won't pass the memento back to the originator,because the originator might never need to revert to an earlier state.

- Mementos are passive. Only the originator that created a memento willassign or retrieve its state.

Consequences

The Memento pattern has several consequences:

1. *Preserving encapsulation boundaries.*Memento avoids exposing information that only an originator shouldmanage but that must be stored nevertheless outside the originator.The pattern shields other objects from potentially complex Originatorinternals, thereby preserving encapsulation boundaries.
2. *It simplifies Originator.*In other encapsulation-preserving designs, Originator keeps theversions of internal state that clients have requested. That puts allthe storage management burden on Originator. Having clientsmanage the state they ask for simplifies Originator and keepsclients from having to notify originators when they're done.
3. *Using mementos might be expensive.*Mementos might incur considerable overhead if Originator must copylarge amounts of information to store in the memento or if clientscreate and return mementos to the originator often

enough. Unlessencapsulating and restoring Originator state is cheap, the patternmight not be appropriate. See the discussion of incrementality in theImplementation section.

4. *Defining narrow and wide interfaces.* It may be difficult in some languages to ensure that only theoriginator can access the memento's state.
5. *Hidden costs in caring for mementos.* A caretaker is responsible for deleting the mementos it cares for.However, the caretaker has no idea how much state is in the memento.Hence an otherwise lightweight caretaker might incur large storagecosts when it stores mementos.

Implementation

Here are two issues to consider when implementing the Memento pattern:

1. *Language support.* Mementos have two interfaces: a wide one for originators and a narrowone for other objects. Ideally the implementation language willsupport two levels of static protection. C++ lets you do this bymaking the Originator a friend of Memento and making Memento's wideinterface private. Only the narrow interface should be declaredpublic. For example:

```
class State;

class Originator {
public:
Memento* CreateMemento();
void SetMemento(const Memento*);
// ...
private:
State* _state;
// internal data structures
// ...
};

class Memento {
public:
// narrow public interface
virtual ~Memento();
private:
// private members accessible only to Originator
friend class Originator;
Memento();
void SetState(State*);
State* GetState();
// ...
private:
State* _state;
// ...
};
```

2. *Storing incremental changes.*When mementos get created and passed back to their originator in apredictable sequence, then Memento can save just the *incrementalchange* to the originator's internal state.

   For example, undoable commands in a history list can use mementos toensure that commands are restored to their exact state when they'reundone (see Command (263)). The history list defines aspecific order in which commands can be undone and redone. That meansmementos can store just the incremental change that a command makesrather than the full state of every object they affect. In theMotivation example given earlier, the constraint solver can store only thoseinternal structures that change to keep the line connecting therectangles, as opposed to storing the absolute positions of theseobjects.

Sample Code

The C++ code given here illustrates the ConstraintSolver example discussed earlier. Weuse MoveCommand objects (see Command (263)) to (un)dothe translation of a graphical object from one position to another.The graphical editor calls the command's Execute operationto move a graphical object and Unexecute to undo the move.The command stores its target, the distance moved, and an instance ofConstraintSolverMemento, a memento containing state from theconstraint solver.

```
class Graphic;
// base class for graphical objects in the graphical editor

class MoveCommand {
public:
MoveCommand(Graphic* target, const Point& delta);
void Execute();
void Unexecute();
private:
ConstraintSolverMemento* _state;
Point _delta;
Graphic* _target;
};
```

The connection constraints are established by the classConstraintSolver. Its key member function isSolve, which solves the constraints registered withthe AddConstraint operation. To support undo,ConstraintSolver's state can be externalized withCreateMemento into a ConstraintSolverMementoinstance. The constraint solver can be returned to a previousstate by calling SetMemento. ConstraintSolveris a Singleton (144).

```
class ConstraintSolver {
public:
static ConstraintSolver* Instance();
void Solve();
void AddConstraint(
Graphic* startConnection, Graphic* endConnection
);
void RemoveConstraint(
Graphic* startConnection, Graphic* endConnection
);
ConstraintSolverMemento* CreateMemento();
void SetMemento(ConstraintSolverMemento*);
private:
// nontrivial state and operations for enforcing
// connectivity semantics   };

class ConstraintSolverMemento {
public:
virtual ~ConstraintSolverMemento();
private:
friend class ConstraintSolver;
ConstraintSolverMemento();
// private constraint solver state
};
```

Given these interfaces, we can implement MoveCommand membersExecute and Unexecute as follows:

```
void MoveCommand::Execute () {
ConstraintSolver* solver = ConstraintSolver::Instance();
_state = solver->CreateMemento();
// create a memento
_target->Move(_delta);
solver->Solve();
}

void MoveCommand::Unexecute () {
```

```
ConstraintSolver* solver = ConstraintSolver::Instance();
_target->Move(-_delta);
solver->SetMemento(_state);
// restore solver state
solver->Solve();
}
```

Execute acquires a ConstraintSolverMemento mementobefore it moves the graphic. Unexecute moves the graphicback, sets the constraint solver's state to the previous state, andfinally tells the constraint solver to solve the constraints.

Known Uses

The preceding sample code is based on Unidraw's support for connectivitythrough its CSolver class [VL90].

Collections in Dylan [App92] provide an iteration interface thatreflects the Memento pattern. Dylan's collections have the notion of a"state" object, which is a memento that represents the state of theiteration. Each collection can represent the current state of theiteration in any way it chooses; the representation is completelyhidden from clients. The Dylan iteration approach might be translatedto C++ as follows:

```
template <class Item>
class Collection {
public:
Collection();
IterationState* CreateInitialState();
void Next(IterationState*);
bool IsDone(const IterationState*) const;
Item CurrentItem(const IterationState*) const;
IterationState* Copy(const IterationState*) const;
void Append(const Item&);
void Remove(const Item&);
// ...
};
```

CreateInitialState returns an initializedIterationState object for the collection. Next advancesthe state object to the next position in the iteration; it effectivelyincrements the iteration index. IsDone returnstrue if Next has advanced beyond the last elementin the collection. CurrentItem dereferences the stateobject and returns the element in the collection to which it refers.Copy returns a copy of the given state object. This isuseful for marking a point in an iteration.

Given a class ItemType, we can iterate over a collection ofits instances as follows[7]:

```
class ItemType {
public:
void Process();
// ...
};

Collection<ItemType*> aCollection;
IterationState* state;
state = aCollection.CreateInitialState();
while (!aCollection.IsDone(state)) {
aCollection.CurrentItem(state)->Process();
aCollection.Next(state);
}
delete state;
```

The memento-based iteration interface has two interesting benefits:

1. More than one state can work on the same collection. (The sameis true of the Iterator (289) pattern.)
2. It doesn't require breaking a collection's encapsulationto support iteration. The memento is only interpreted by thecollection itself; no one else has access to it. Other approaches toiteration require breaking encapsulation by making iterator classesfriends of their collection classes (see Iterator (289)). The situation is reversed in thememento-based implementation: Collection is a friend of theIteratorState.

The QOCA constraint-solving toolkit stores incremental information inmementos [HHMV92]. Clients can obtain a memento that characterizesthe current solution to a system of constraints. The memento containsonly those constraint variables that have changed since the lastsolution. Usually only a small subset of the solver's variableschanges for each new solution. This subset is enough to return thesolver to the preceding solution; reverting to earlier solutionsrequires restoring mementos from the intervening solutions. Hence youcan't set mementos in any order; QOCA relies on a history mechanism torevert to earlier solutions.

Related Patterns

Command (263): Commands can use mementos to maintainstate for undoable operations.

Iterator (289): Mementoscan be used for iteration as described earlier.

[7]Note that our example deletes the state object at the end of the iteration. But delete won't get called if ProcessItem throws an exception, thus creating garbage. This is a problem in C++ but not in Dylan, which has garbage collection. We discuss a solution to this problem on page 299.