Memento [GoF]

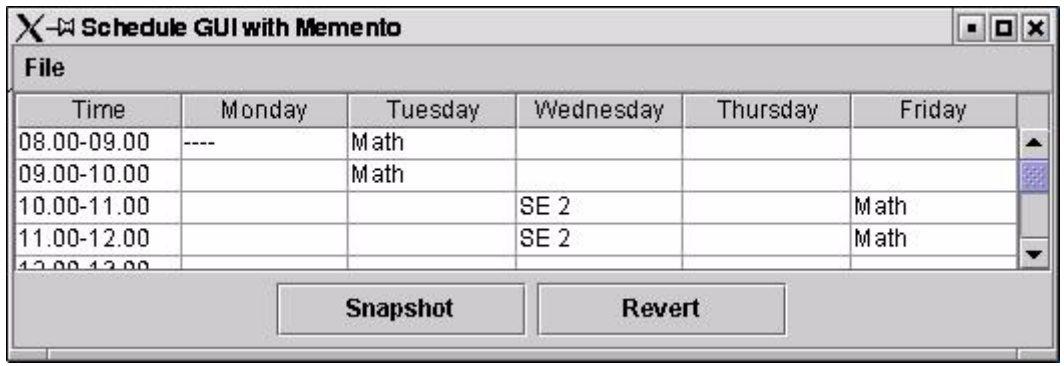
Intent

Capture an object's internal state and externalize it so that it can be restored to that state later.

Motivation

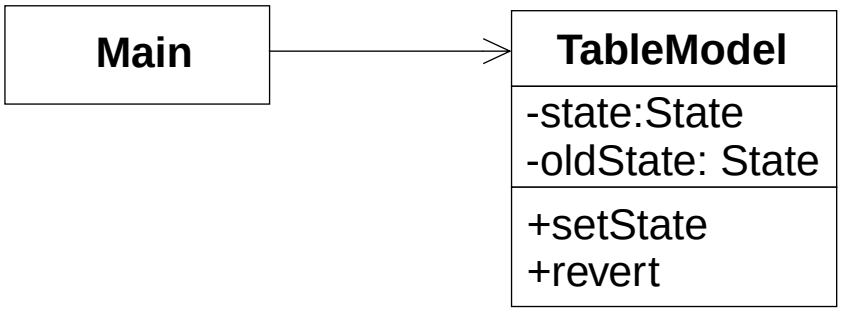
Suppose we have an application where the user can enter data into a table. By doing so the user might well change his/her mind and wanting to restore the data in the table to the values that existed before the changes have been made.

The application's GUI might look like:



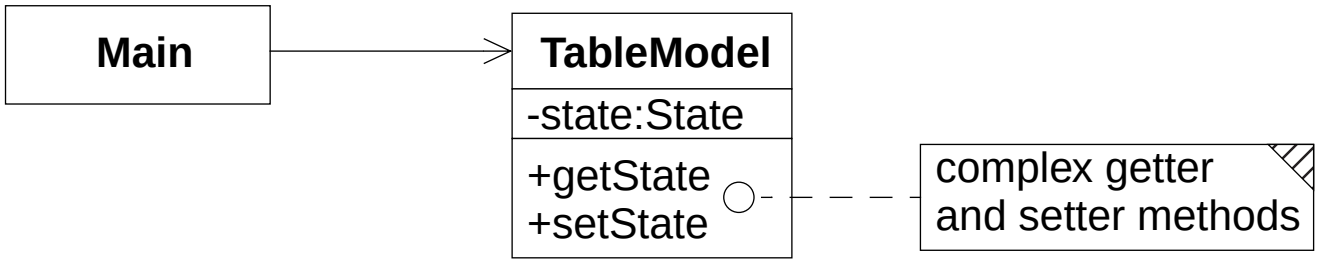
Changing the table's values and then pressing the "Revert" button restores the values of the table it had before making the changes.

One solution to this problem is to enhance the `TableModel` object such that it keeps previous state information, and to provide a kind of `revert` method.



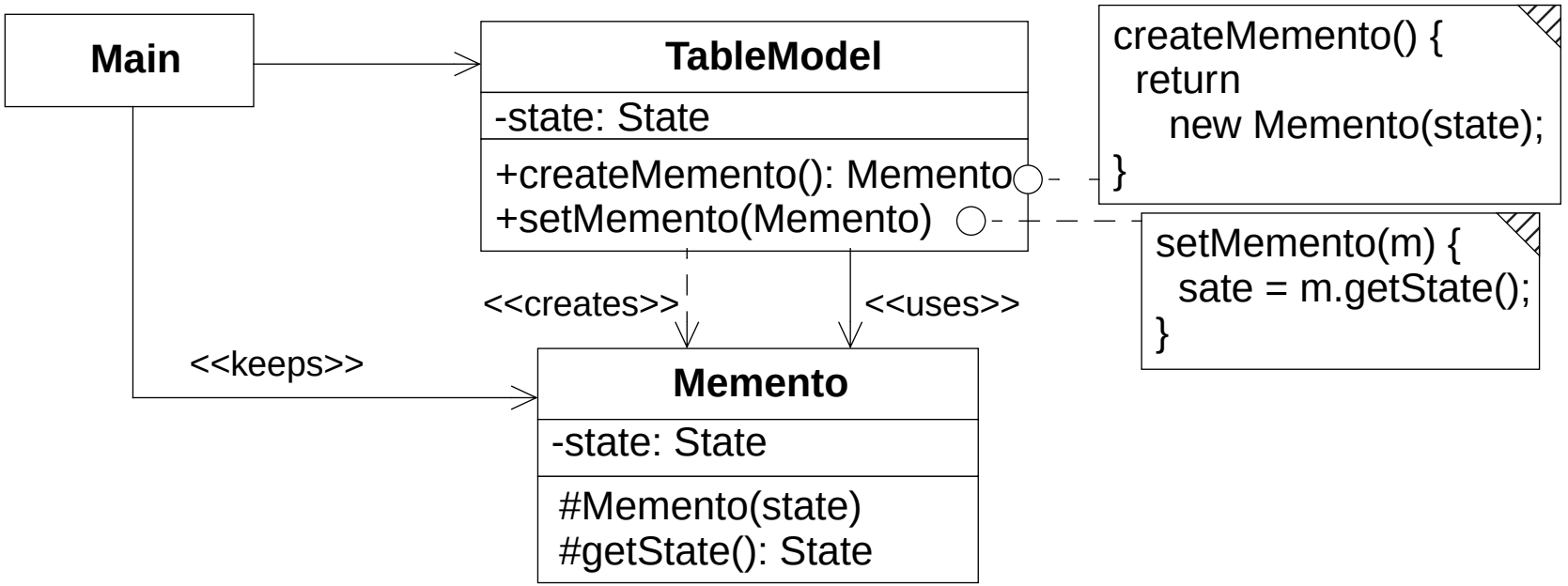
However, this lays the burden on the `TableModel` object. In addition, a memory penalty must always be paid even if the revert facility in not used.

In another variant, a caretaker, e.g. the application's main class, queries the state of the table before making the changes:



However, the set of complex getter and setter methods would break the `TableModel`'s encapsulation.

A wiser approach is *externalize* the object's state into a *memento object* which is a snapshot of the state of the `TableModel`. A caretaker (here the `Main` class) asks the `TableModel` for a memento prior making changes. If the caretaker later wants to restore the `TableModel` later, it returns the memento object to `TableModel`. `TableModel` in turn queries the memento object to restore its previous state.

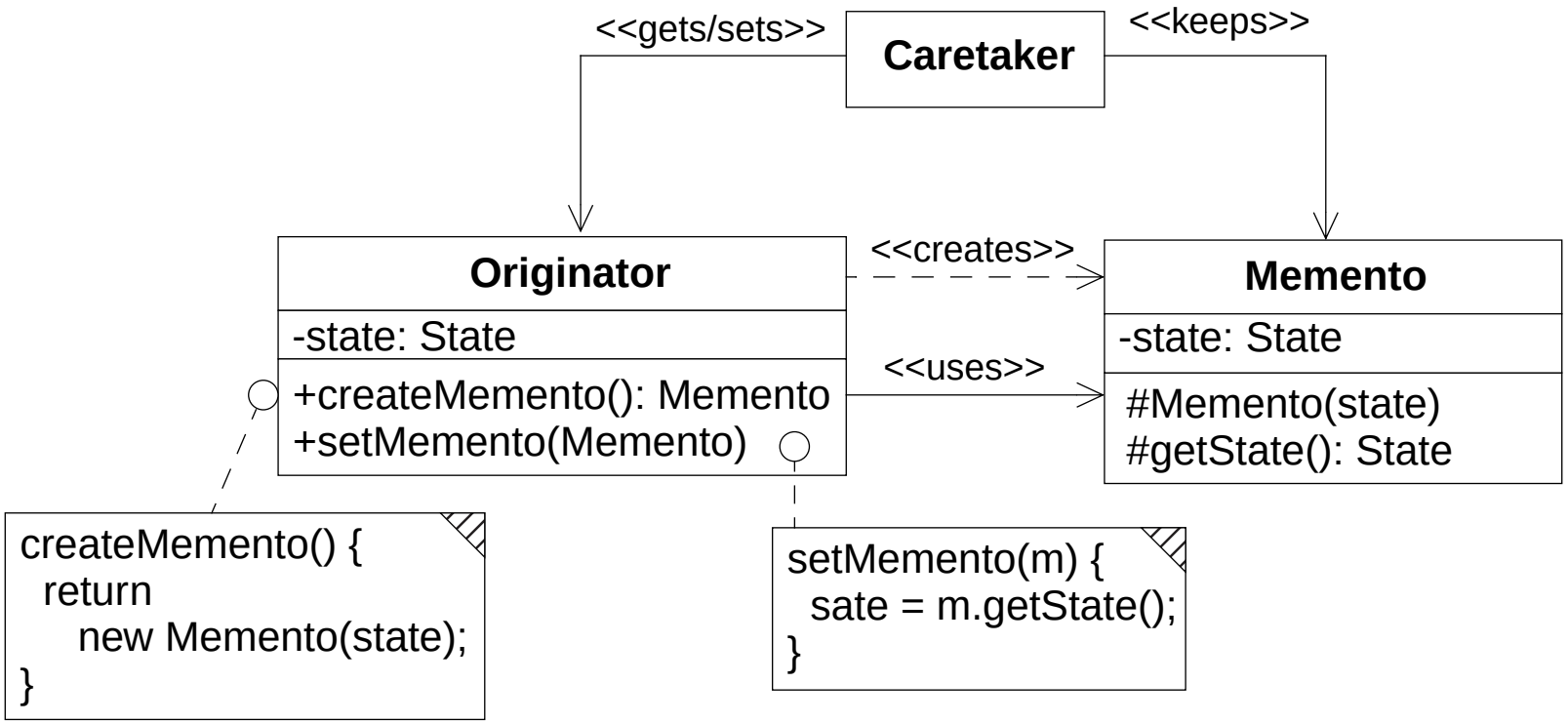


Applicability

Use the Memento pattern when

- a snapshot of an object's state must be saved so that it can be restored to that state later.
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure

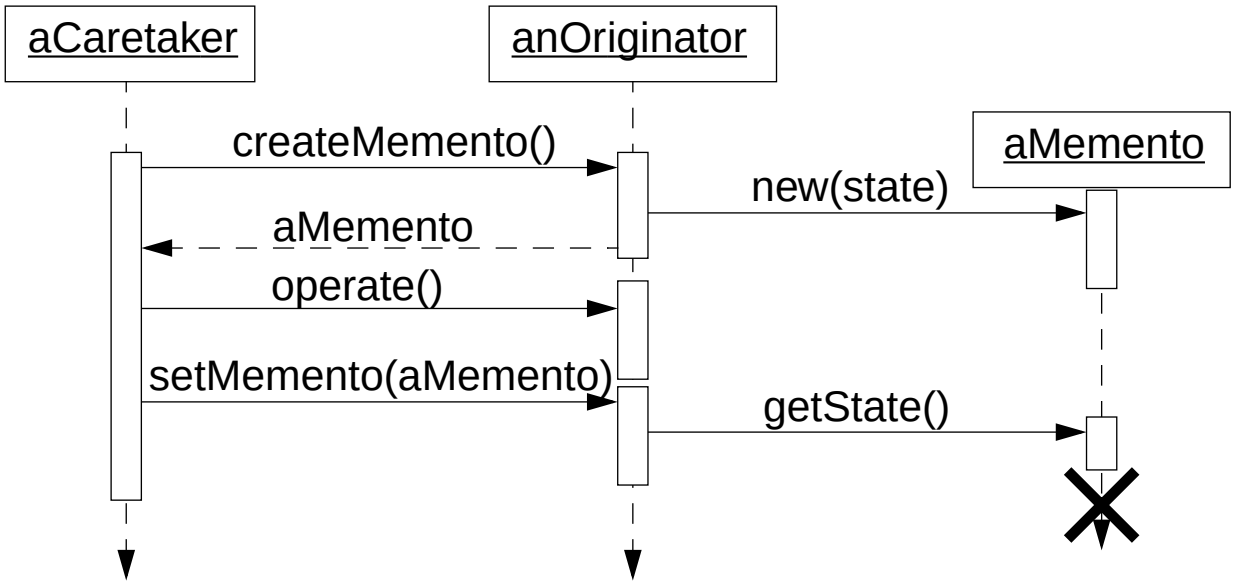

<<gets/sets>>
Caretaker
<<keeps>>
Originator
-state: State
+createMemento(): Memento
+setMemento(Memento)
<<creates>>
<<uses>>
Memento
-state: State
#Memento(state)
#getState(): State
createMemento() {
return
new Memento(state);
}
setMemento(m) {
sate = m.getState();
}

Participants

- **`Memento`** (`Memento`)
  - stores the whole or some portion of the Originator object's internal state.
  - protects against access by objects other than the Originator.
  - has effectively two interfaces: a narrow interface for Caretaker, and a wide interface for Originator.
- **`Originator`** (`TableModel`)
  - creates a Memento containing a snapshot of its current internal state.
  - uses the Memento to restore its internal state.
  - can access all the data in the Memento necessary to restore itself to its previous state.
- **`Caretaker`** (`Main`)
  - is responsible for keeping the Memento safe.
  - never operates on or examines the contents of a Memento.
  - can only pass the Memento to other objects.

Collaborations

- A Caretaker requests a Memento from an Originator, holds it for a time, and passes it back to the Originator when it needs to revert it to an earlier state.
- The state of a Memento is assigned or retrieved only by the Originator.



Consequences

1. Preserving encapsulation boundaries.
   – The Memento pattern shields other objects from potentially complex Originator internals.
2. It simplifies Originator.
   – simplifies Originator by having clients manage the versions of Originator's state they ask for.
   – keeps clients from having to notify Originators when they're done.
3. Using mementos might be expensive if
   – the amount of information to be stored in the Memento is large or
   – mementos are requested from and returned to the Originator often enough.
4. Defining narrow and wide interfaces.
   – It may be difficult in some languages to ensure that only the Originator can access the Memento's state.
5. Hidden costs in caring for mementos.
   – An otherwise lightweight Caretaker might incur large storage costs when it stores mementos.
   – But the caretaker does not know beforehand how much state is in the Memento it cares for.

Implementation

1. *Language support.* C++ lets you define narrow and wide interfaces of Memento with two levels of static protection by declaring:
   – the Originator a *friend* of Memento and Memento's wide interface private and
   – only the narrow interface public.
2. *Language support.* In Java, make the narrow interface *package-visible* at most.
3. *Language support.* In Java, you might consider Memento to be an *inner class* of Originator:

```
public class Originator {
   private T state;
   private class Memento { // value object
      private T mstate;
      private Memento(T state) { mstate = copy_of(state); }
      private T getState() { return mstate; }
   }

   public Memento createMemento() {
      return new Memento(state);
   }
      // continued...
```

```
    public void setMemento(Memento m) {
       state = m.getState();
    }
    //..
}
```

4. *Storing incremental changes.* When mementos get created and passed back to the originator in a predictable sequence, then Memento can save just the delta to the original's state. For example, undoable commands in a history list can use mementos that store just the incremental change a command makes to ensure that commands are restored to their exact state when they're undone.
5. *Serialization*. If the Caretaker wants to store away the Memento objects then the Memento class must implement the `Serializable` interface and protocol.

Sample Code

Let's consider a sketch of the code of the example in the Motivation section. `ScheduleTableModel` (the Originator) is a subclass of `JTable`'s default model, class `DefaultTableModel`. `ScheduleTableModel` provides the support for the `TableMemento`.

```
public class ScheduleTableModel extends DefaultTableModel {
   public ScheduleTableModel(Object[][] data,
                             Object[] columnNames)
   { super(data, columnNames); }

   public TableMemento createMemento()
   { return new TableMemento(columnIdentifiers, dataVector); }
   public void setMemento(TableMemento memento)
   {
      if (memento != null) {
         setDataVector(memento.getDataVector(),
                       memento.getColumnIdentifiers());
      }
   }
}
```

Class `TableMemento` stores the previous state of a `ScheduleTableModel` as instances of a `Vector` class. Getter methods allow to retrieve the data:

```
public class TableMemento implements Serializable { // V.O.
   private List columnIdentifiers;
   private List dataList;

   TableMemento(Vector columnIdentifiers, Vector dataVector) {
      this.columnIdentifiers = copy_of(columnIdentifiers);
      this.dataVector = copy_of(dataVector); // code not shown
   }

   Vector getColumnIdentifiers() {
      List l = Collections.unmodifiableList(columnIdentifiers);
      Vector result = new Vector(l);
      return result;
   }
   Vector getDataVector() {
      Vector result = makeVector(dataList); // code not shown
      return result;
   }
}
```

Notice the package visibility of the constructor and the getter and setter methods.

An example of a Caretaker might look like:

```
// In a Caretaker class:
   ScheduleTableModel tableModel = ...;
   ...
   // Get the old state of the table:
   TableMemento memento = tableModel.createMemento();
   ...
   // Modify the table:
   tableModel.setDataVector(newData, columnNames);
   ...
   // If you don't like the changes:
   tableModel.setMemento(memento);
```

Related Patterns

- Commands can use mementos to maintain state for undoable operations.
- Mementos can be used for iteration.