

Patternuri comportamentale

Observer, State si Memento

1. Behavioral Pattern

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioral class patterns use inheritance to distribute behavior between classes. This chapter includes two such patterns. Template Method is the simpler and more common of the two. A template method is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. The other behavioral class pattern is Interpreter, which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.

Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other. The Mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

Chain of Responsibility provides even looser coupling. It lets you send requests to an object implicitly through a chain of candidate objects. Any candidate may fulfill the request depending on run-time conditions. The number of candidates is open-ended, and you can select which candidates participate in the chain at run-time.

The Observer pattern defines and maintains a dependency between objects. The classic example of Observer is in Smalltalk Model/View/Controller, where all views of the model are notified whenever the model's state changes.

Other behavioral object patterns are concerned with encapsulating behavior in an object and delegating requests to it. The Strategy pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses. The Command pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways. The State pattern encapsulates the states of an object so that the object can change its behavior when its state object changes. Visitor encapsulates behavior that would otherwise be distributed across classes, and Iterator abstracts the way you access and traverse objects in an aggregate.

2. Observer

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

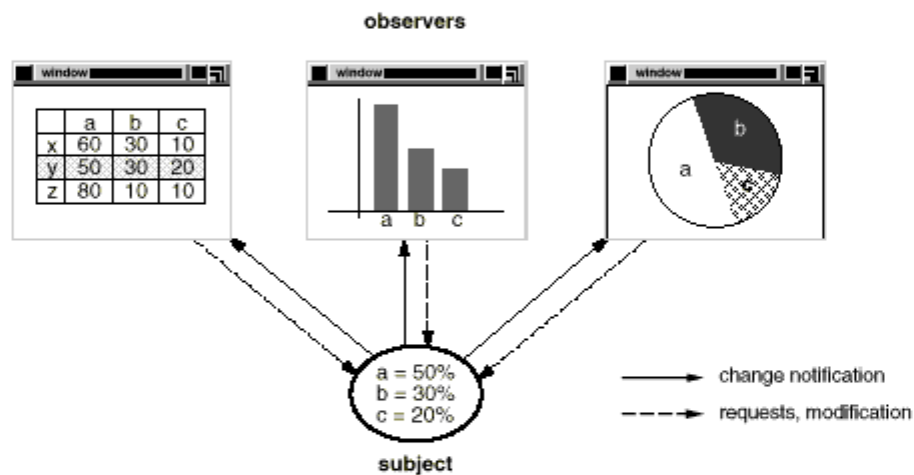
Also Known As

Dependents, Publish-Subscribe

Motivation

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data [KP88, LVC89, P+88, WGM88]. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.

The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

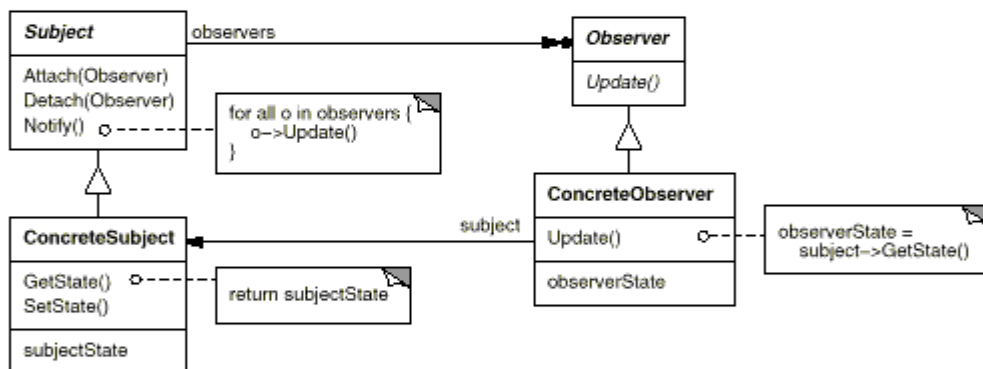
This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure



Participants

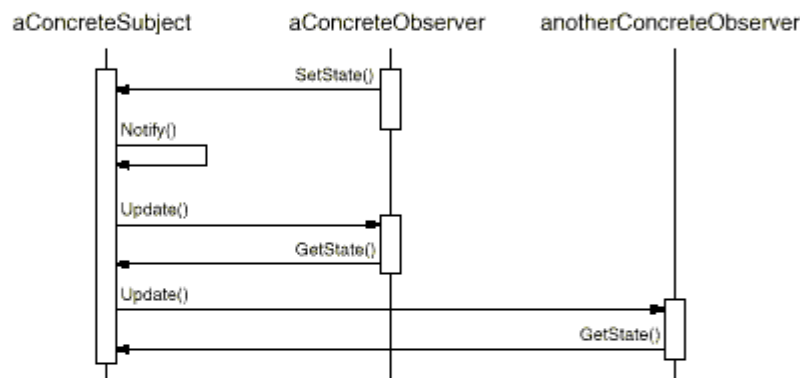
- **Subject**
 - knows its observers. Any number of **Observer** objects may observe a subject.
 - provides an interface for attaching and detaching **Observer** objects.
- **Observer**
 - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
 - stores state of interest to **ConcreteObserver** objects.
 - sends a notification to its observers when its state changes.
- **ConcreteObserver**

- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

The following interaction diagram illustrates the collaborations between a subject and two observers:



Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject. Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely. The Implementation section discusses some common variations.

Consequences

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

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

1. *Abstract coupling between Subject and Observer:* All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level

observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

2. *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.
3. *Unexpected updates.* Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Implementation

Several issues related to the implementation of the dependency mechanism are discussed in this section.

1. *Mapping subjects to their observers.* The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. However, such storage may be too expensive when there are many subjects and few observers. One solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping. Thus a subject with no observers does not incur storage overhead. On the other hand, this approach increases the cost of accessing the observers.
2. *Observing more than one subject.* It might make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may depend on more than one data source. It's necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
3. *Who triggers the update?* The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls Notify to trigger the update? Here are two options:
 - a. Have state-setting operations on Subject call Notify after they change the subject's state. The advantage of this approach is that clients don't have to remember to call Notify on the subject. The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient.
 - b. Make clients responsible for calling Notify at the right time. The advantage here is that the client can wait to trigger the update until after a series of state changes has been made, thereby avoiding needless intermediate updates. The disadvantage is that clients have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call Notify.

4. *Dangling references to deleted subjects.* Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it. In general, simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.
5. *Making sure Subject state is self-consistent before notification.* It's important to make sure Subject state is self-consistent before calling Notify, because observers query the subject for its current state in the course of updating their own state.

This self-consistency rule is easy to violate unintentionally when Subject subclass operations call inherited operations. For example, the notification in the following code sequence is triggered when the subject is in an inconsistent state:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
    // trigger notification

    _myInstVar += newValue;
    // update subclass state (too late!)
}
```

You can avoid this pitfall by sending notifications from template methods (Template Method) in abstract Subject classes. Define a primitive operation for subclasses to override, and make Notify the last operation in the template method, which will ensure that the object is self-consistent when subclasses override Subject operations.

```
void Text::Cut (TextRange r) {
    ReplaceRange(r);          // redefined in subclasses
    Notify();
}
```

By the way, it's always a good idea to document which Subject operations trigger notifications.

6. *Avoiding observer-specific update protocols: the push and pull models.* Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely.

At one extreme, which we call the **push model**, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the **pull model**; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

7. *Specifying modifications of interest explicitly.* You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. One way to support this uses the notion of **aspects** for Subject objects. To register interest in particular events, observers are attached to their subjects using

```
void Subject::Attach(Observer*, Aspect& interest);
```

where `interest` specifies the event of interest. At notification time, the subject supplies the changed aspect to its observers as a parameter to the Update operation. For example:

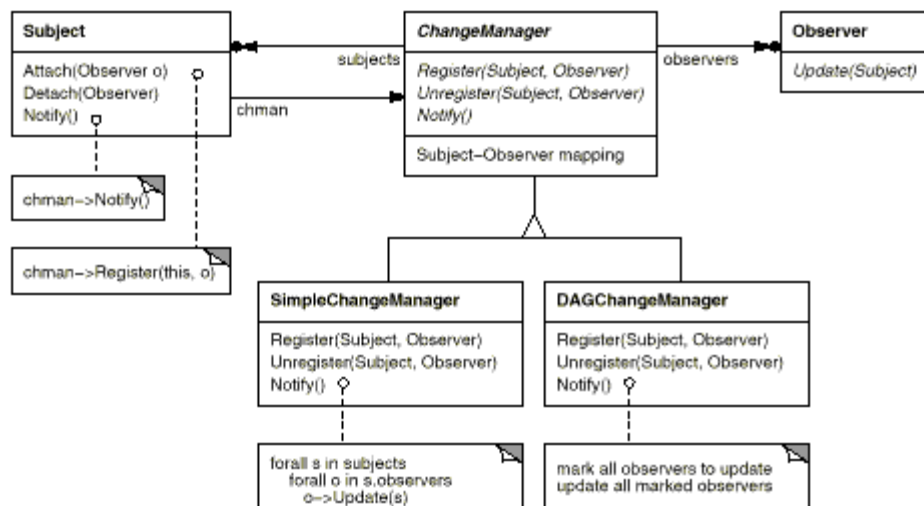
```
void Observer::Update(Subject*, Aspect& interest);
```

8. *Encapsulating complex update semantics.* When the dependency relationship between subjects and observers is particularly complex, an object that maintains these relationships might be required. We call such an object a **ChangeManager**. Its purpose is to minimize the work required to make observers reflect a change in their subject. For example, if an operation involves changes to several interdependent subjects, you might have to ensure that their observers are notified only after *all* the subjects have been modified to avoid notifying observers more than once.

ChangeManager has three responsibilities:

- a. It maps a subject to its observers and provides an interface to maintain this mapping. This eliminates the need for subjects to maintain references to their observers and vice versa.
- b. It defines a particular update strategy.
- c. It updates all dependent observers at the request of a subject.

The following diagram depicts a simple ChangeManager-based implementation of the Observer pattern. There are two specialized ChangeManagers. SimpleChangeManager is naive in that it always updates all observers of each subject. In contrast, DAGChangeManager handles directed-acyclic graphs of dependencies between subjects and their observers. A DAGChangeManager is preferable to a SimpleChangeManager when an observer observes more than one subject. In that case, a change in two or more subjects might cause redundant updates. The DAGChangeManager ensures the observer receives just one update. SimpleChangeManager is fine when multiple updates aren't an issue.



ChangeManager is an instance of the Mediator pattern. In general there is only one ChangeManager, and it is known globally. The Singleton pattern would be useful here.

9. *Combining the Subject and Observer classes.* Class libraries written in languages that lack multiple inheritance (like Smalltalk) generally don't define separate Subject and Observer classes but combine their interfaces in one class. That lets you define an object that acts as both a subject and an observer without multiple inheritance. In Smalltalk, for example, the Subject and Observer interfaces are defined in the root class Object, making them available to all classes.

Sample Code

An abstract class defines the Observer interface:

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

This implementation supports multiple subjects for each observer. The subject passed to the Update operation lets the observer determine which subject changed when it observes more than one.

Similarly, an abstract class defines the Subject interface:

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

ClockTimer is a concrete subject for storing and maintaining the time of day. It notifies its observers every second. ClockTimer provides the interface for retrieving individual time units

such as the hour, minute, and second.

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};
```

The `Tick` operation gets called by an internal timer at regular intervals to provide an accurate time base. `Tick` updates the `ClockTimer`'s internal state and calls `Notify` to inform observers of the change:

```
void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```

Now we can define a class `DigitalClock` that displays the time. It inherits its graphical functionality from a `Widget` class provided by a user interface toolkit. The `Observer` interface is mixed into the `DigitalClock` interface by inheriting from `Observer`.

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
    _subject->Detach(this);
}
```

Before the `Update` operation draws the clock face, it checks to make sure the notifying subject is the clock's subject:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}
```

```

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // draw the digital clock
}

```

An AnalogClock class can be defined in the same way.

```

class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};

```

The following code creates an AnalogClock and a DigitalClock that always show the same time:

```

ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);

```

Whenever the timer ticks, the two clocks will be updated and will redisplay themselves appropriately.

Known Uses

The first and perhaps best-known example of the Observer pattern appears in Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalk environment [KP88]. MVC's Model class plays the role of Subject, while View is the base class for observers. Smalltalk, ET++ [WGM88], and the THINK class library [Sym93b] provide a general dependency mechanism by putting Subject and Observer interfaces in the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern are InterViews [LVC89], the Andrew Toolkit [P+88], and Unidraw [VL90]. InterViews defines Observer and Observable (for subjects) classes explicitly. Andrew calls them "view" and "data object," respectively. Unidraw splits graphical editor objects into View (for observers) and Subject parts.

Related Patterns

Mediator : By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.

Singleton : The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

3. State

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

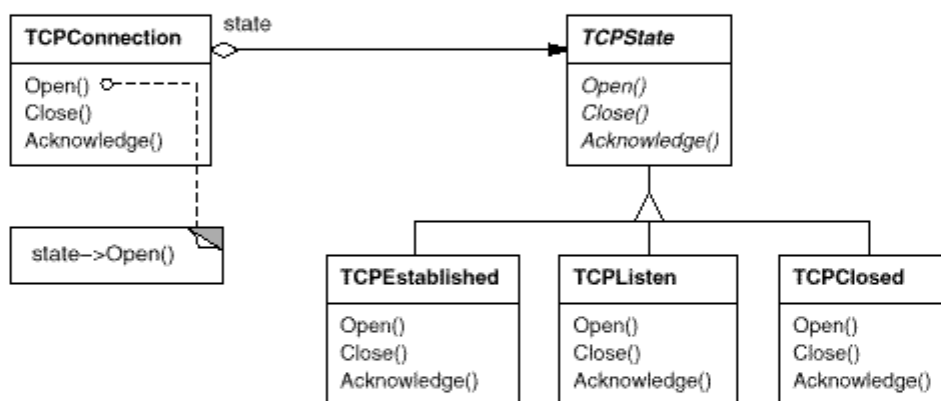
Also Known As

Objects for States

Motivation

Consider a class `TCPConnection` that represents a network connection. A `TCPConnection` object can be in one of several different states: Established, Listening, Closed. When a `TCPConnection` object receives requests from other objects, it responds differently depending on its current state. For example, the effect of an `Open` request depends on whether the connection is in its Closed state or its Established state. The State pattern describes how `TCPConnection` can exhibit different behavior in each state.

The key idea in this pattern is to introduce an abstract class called `TCPState` to represent the states of the network connection. The `TCPState` class declares an interface common to all classes that represent different operational states. Subclasses of `TCPState` implement state-specific behavior. For example, the classes `TCPEstablished` and `TCPClosed` implement behavior particular to the Established and Closed states of `TCPConnection`.



The class `TCPConnection` maintains a state object (an instance of a subclass of `TCPState`)

that represents the current state of the TCP connection. The class TCPConnection delegates all state-specific requests to this state object. TCPConnection uses its TCPState subclass instance to perform operations particular to the state of the connection.

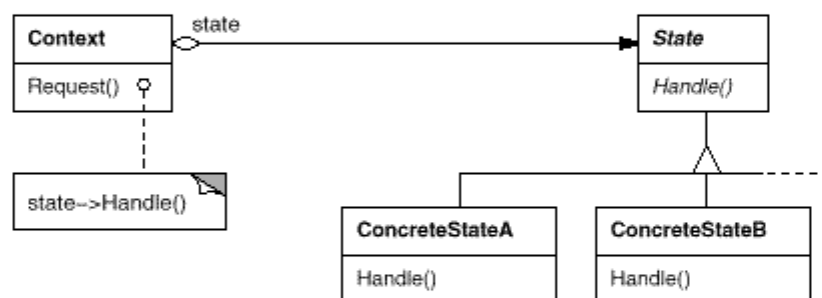
Whenever the connection changes state, the TCPConnection object changes the state object it uses. When the connection goes from established to closed, for example, TCPConnection will replace its TCPEstablished instance with a TCPClosed instance.

Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Structure



Participants

- **Context** (TCPConnection)
 - defines the interface of interest to clients.
 - maintains an instance of a ConcreteState subclass that defines the current state.
- **State** (TCPState)
 - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)

- each subclass implements a behavior associated with a state of the Context.

Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences

The State pattern has the following consequences:

1. *It localizes state-specific behavior and partitions behavior for different states.* The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.

An alternative is to use data values to define internal states and have Context operations check the data explicitly. But then we'd have look-alike conditional or case statements scattered throughout Context's implementation. Adding a new state could require changing several operations, which complicates maintenance.

The State pattern avoids this problem but might introduce another, because the pattern distributes behavior for different states across several State subclasses. This increases the number of classes and is less compact than a single class. But such distribution is actually good if there are many states, which would otherwise necessitate large conditional statements.

Like long procedures, large conditional statements are undesirable. They're monolithic and tend to make the code less explicit, which in turn makes them difficult to modify and extend. The State pattern offers a better way to structure state-specific code. The logic that determines the state transitions doesn't reside in monolithic `if` or `switch` statements but instead is partitioned between the State subclasses. Encapsulating each state transition and action in a class elevates the idea of an execution state to full object status. That imposes structure on the code and makes its intent clearer.

2. *It makes state transitions explicit.* When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal states, because state transitions are atomic from the Context's perspective—they happen by rebinding *one* variable (the Context's State object variable), not several [dCLF93].
3. *State objects can be shared.* If State objects have no instance variables—that is, the state they represent is encoded entirely in their type—then contexts can share a State object.

When states are shared in this way, they are essentially flyweights with no intrinsic state, only behavior.

Implementation

The State pattern raises a variety of implementation issues:

1. *Who defines the state transitions?* The State pattern does not specify which participant defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the Context. It is generally more flexible and appropriate, however, to let the State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.

Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses. A disadvantage of decentralization is that one State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.

2. *A table-based alternative.* In *C++ Programming Style* [Car92], Cargill describes another way to impose structure on state-driven code: He uses tables to map inputs to state transitions. For each state, a table maps every possible input to a succeeding state. In effect, this approach converts conditional code (and virtual functions, in the case of the State pattern) into a table look-up.

The main advantage of tables is their regularity: You can change the transition criteria by modifying data instead of changing program code. There are some disadvantages, however:

- A table look-up is often less efficient than a (virtual) function call.
- Putting transition logic into a uniform, tabular format makes the transition criteria less explicit and therefore harder to understand.
- It's usually difficult to add actions to accompany the state transitions. The table-driven approach captures the states and their transitions, but it must be augmented to perform arbitrary computation on each transition.

The key difference between table-driven state machines and the State pattern can be summed up like this: The State pattern models state-specific behavior, whereas the table-driven approach focuses on defining state transitions.

3. *Creating and destroying State objects.* A common implementation trade-off worth considering is whether (1) to create State objects only when they are needed and destroy them thereafter versus (2) creating them ahead of time and never destroying them.

The first choice is preferable when the states that will be entered aren't known at run-time, *and* contexts change state infrequently. This approach avoids creating objects that won't be used, which is important if the State objects store a lot of information. The second approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly. Instantiation costs are paid once up-front, and there are no destruction costs at all. This approach might be inconvenient, though, because the Context must keep references to all states that might be entered.

4. *Using dynamic inheritance.* Changing the behavior for a particular request could be accomplished by changing the object's class at run-time, but this is not possible in most object-oriented programming languages. Exceptions include Self [US87] and other

delegation-based languages that provide such a mechanism and hence support the State pattern directly. Objects in Self can delegate operations to other objects to achieve a form of dynamic inheritance. Changing the delegation target at run-time effectively changes the inheritance structure. This mechanism lets objects change their behavior and amounts to changing their class.

Sample Code

The following example gives the C++ code for the TCP connection example described in the Motivation section. This example is a simplified version of the TCP protocol; it doesn't describe the complete protocol or all the states of TCP connections.

First, we define the class `TCPConnection`, which provides an interface for transmitting data and handles requests to change state.

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();

    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

`TCPConnection` keeps an instance of the `TCPState` class in the `_state` member variable. The class `TCPState` duplicates the state-changing interface of `TCPConnection`. Each `TCPState` operation takes a `TCPConnection` instance as a parameter, letting `TCPState` access data from `TCPConnection` and change the connection's state.

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

`TCPConnection` delegates all state-specific requests to its `TCPState` instance

`_state`. `TCPConnection` also provides an operation for changing this variable to a new `TCPState`. The constructor for `TCPConnection` initializes the object to the `TCPClosed` state (defined later).

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

`TCPState` implements default behavior for all requests delegated to it. It can also change the state of a `TCPConnection` with the `ChangeState` operation. `TCPState` is declared a friend of `TCPConnection` to give it privileged access to this operation.

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

Subclasses of `TCPState` implement state-specific behavior. A TCP connection can be in many states: Established, Listening, Closed, etc., and there's a subclass of `TCPState` for each state. We'll discuss three subclasses in detail: `TCPEstablished`, `TCPListen`, and `TCPClosed`.

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};
```



```

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};

```

TCPState subclasses maintain no local state, so they can be shared, and only one instance of each is required. The unique instance of each TCPState subclass is obtained by the static Instance operation.⁹

Each TCPState subclass implements state-specific behavior for valid requests in the state:

```

void TCPClosed::ActiveOpen (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

```

After performing state-specific work, these operations call the ChangeState operation to change the state of the TCPConnection. TCPConnection itself doesn't know a thing about the TCP connection protocol; it's the TCPState subclasses that define each state transition and action in TCP.

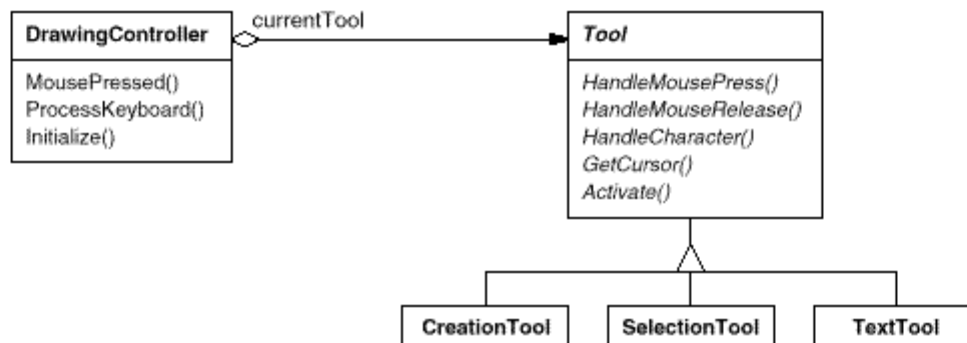
Known Uses

Johnson and Zweig characterize the State pattern and its application to TCP connection protocols.

Most popular interactive drawing programs provide "tools" for performing operations by direct manipulation. For example, a line-drawing tool lets a user click and drag to create a new line. A selection tool lets the user select shapes. There's usually a palette of such tools to choose from. The user thinks of this activity as picking up a tool and wielding it, but in reality the editor's behavior changes with the current tool: When a drawing tool is active we create shapes; when the selection tool is active we select shapes; and so forth. We can use the State pattern to change the editor's behavior depending on the current tool.

We can define an abstract Tool class from which to define subclasses that implement tool-specific behavior. The drawing editor maintains a current Tool object and delegates requests to it. It replaces this object when the user chooses a new tool, causing the behavior of the drawing editor to change accordingly.

This technique is used in both the HotDraw and Unidraw drawing editor frameworks. It allows clients to define new kinds of tools easily. In HotDraw, the DrawingController class forwards the requests to the current Tool object. In Unidraw, the corresponding classes are Viewer and Tool. The following class diagram sketches the Tool and DrawingController interfaces:



Coplien's Envelope-Letter idiom [Cop92] is related to State. Envelope-Letter is a technique for changing an object's class at run-time. The State pattern is more specific, focusing on how to deal with an object whose behavior depends on its state.

Related Patterns

The Flyweight pattern explains when and how State objects can be shared.

State objects are often Singletons.

4. Memento

Intent

Without violating encapsulation, capture and externalize an object's internal 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 mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere so that you can restore objects to their previous states. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.



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

A well-known way to maintain connectivity relationships between objects is with a constraint-solving system. We can encapsulate this functionality in a **ConstraintSolver** object. ConstraintSolver records connections as they are made and generates mathematical equations that describe them. It solves these equations whenever the user makes a connection or otherwise modifies the diagram. ConstraintSolver uses the results of its calculations to rearrange the graphics so that they maintain the proper connections.

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



In general, the ConstraintSolver's public interface might be insufficient to allow precise reversal of its effects on other objects. The undo mechanism must work more closely with ConstraintSolver to reestablish previous state, but we should also avoid 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 the internal state of another object—the memento's **originator**. The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state. The originator initializes the memento with information that characterizes its current state. Only the originator can store and retrieve information from the memento—the memento is "opaque" to other objects.

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

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

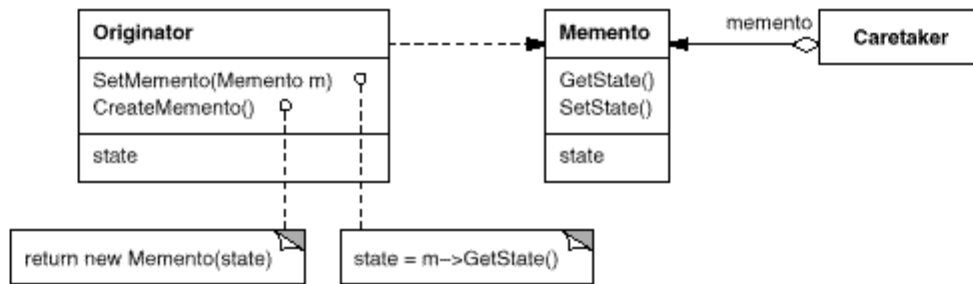
This arrangement lets the ConstraintSolver entrust other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

Applicability

Use the Memento pattern when

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

Structure

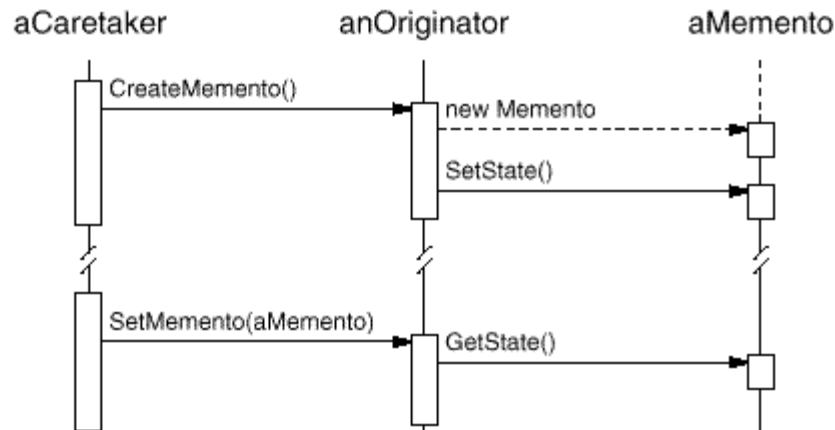


Participants

- **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 a time, and passes it back to the originator, as the following interaction 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 will assign or retrieve its state.

Consequences

The Memento pattern has several consequences:

1. *Preserving encapsulation boundaries.* Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator. The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.
2. *It simplifies Originator.* In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested. That puts all the storage management burden on Originator. Having clients manage the state they ask for simplifies Originator and keeps clients from having to notify originators when they're done.
3. *Using mementos might be expensive.* Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough. Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate. See the discussion of incrementality in the Implementation section.
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.* 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 storage costs 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 narrow one for other objects. Ideally the implementation language will support two levels of static

protection. C++ lets you do this by making the Originator a friend of Memento and making Memento's wide interface private. Only the narrow interface should be declared public. 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 a predictable sequence, then Memento can save just the *incremental change* to the originator's internal state.

For example, undoable commands in a history list can use mementos to ensure that commands are restored to their exact state when they're undone (s. The history list defines a specific order in which commands can be undone and redone. That means mementos can store just the incremental change that a command makes rather than the full state of every object they affect. In the Motivation example given earlier, the constraint solver can store only those internal structures that change to keep the line connecting the rectangles, as opposed to storing the absolute positions of these objects.

Sample Code

The C++ code given here illustrates the ConstraintSolver example discussed earlier. We use MoveCommand objects to (un)do the translation of a graphical object from one position to another. The graphical editor calls the command's Execute operation to move a graphical object and Unexecute to undo the move. The command stores its target, the distance moved, and an instance of ConstraintSolverMemento, a memento containing state from the constraint 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 class `ConstraintSolver`. Its key member function is `Solve`, which solves the constraints registered with the `AddConstraint` operation. To support undo, `ConstraintSolver`'s state can be externalized with `CreateMemento` into a `ConstraintSolverMemento` instance. The constraint solver can be returned to a previous state by calling `SetMemento`. `ConstraintSolver` is a Singleton.

```

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` members `Execute` 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` memento before it moves the graphic. Unexecute moves the graphic back, sets the constraint solver's state to the previous state, and finally tells the constraint solver to solve the constraints.

Known Uses

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

Collections in Dylan [App92] provide an iteration interface that reflects the Memento pattern. Dylan's collections have the notion of a "state" object, which is a memento that represents the state of the iteration. Each collection can represent the current state of the iteration in any way it chooses; the representation is completely hidden from clients. The Dylan iteration approach might be translated to 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 initialized `IterationState` object for the collection. `Next` advances the state object to the next position in the iteration; it effectively increments the iteration index. `IsDone` returns true if `Next` has advanced beyond the last element in the collection. `CurrentItem` dereferences the state object and returns the element in the collection to which it refers. `Copy` returns a copy of the given state object. This is useful for marking a point in an iteration.

Given a class `ItemType`, we can iterate over a collection of its instances as follows⁷:

```

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 same is true of the Iterator (pattern.)
2. It doesn't require breaking a collection's encapsulation to support iteration. The memento is only interpreted by the collection itself; no one else has access to it. Other approaches to iteration require breaking encapsulation by making iterator classes friends of their collection classes. The situation is reversed in the memento-based implementation: `Collection` is a friend of the `IteratorState`.

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

Related Patterns

Command: Commands can use mementos to maintain state for undoable operations.

Iterator: Mementos can be used for iteration as described earlier.

Bibliography

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides.

<http://e-stud.vgtu.lt/users/files/dest/1542/designpatterns.pdf>