Collections for States

A Pattern for Implementing Simple State Models over Collections of Objects

Kevlin Henney

kevlin@acm.org

Context

You are managing a collection of similar objects that have lifecycles with only a few distinct states. These objects are often operated on collectively with respect to the state they are in.

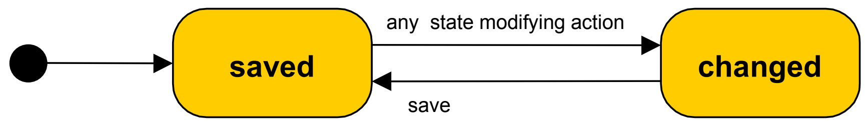
Problem

Objects, whose lifecycle and state is principally of interest to the object that manages them, may be modelled as individual state machines. However, in many ways the objects are independent of this state model, which is a view their manager has of them.

What is a suitable model for the collected objects and their managing objects that emphasises and supports this independence? How is the responsibility for state representation and management divided between the collected objects and their managing objects?

Example

Consider an application that holds a number of graphical objects that may be manipulated by a user, such as a CAD tool or graphical editor. The general shape of this application follows from the WORKPIECES frame [Jackson1995]. The objects manipulated by the user are the workpieces, and these may be saved to some kind of persistent storage, e.g. some kind of database. These objects all share a simple lifecycle model, in addition to their type specific state:



For such a simple state model we can assume that the use of the OBJECTS FOR STATES pattern [Gamma+1995] – where a class hierarchy is introduced to reflect the separate states and the behaviour associated with them – is inappropriate. Instead we can use a simpler flag based approach, representing state as an attribute. In C++ this would lead to the following code (presented inline for brevity):

```
class workpiece
{
public:
    void save()
    {
        save_state();
        changed = false;
    }
    bool saved() const
    {
        return !changed;
    }
    ...
private:
    virtual void save_state() = 0;
    bool changed;
    ...
};
```

The `save_state` member function here acts as a TEMPLATE METHOD [Gamma+1995], and the associated C++ idiom of declaring the pure `virtual` function as `private` reinforces this.

In an application, which acts as the manager for `workpiece` objects, saving all the changes since the last save could be implemented as follows (assuming a non-throwing `save` function):

```
class application
{
public:
    void save_changes()
    {
        for(iterator current = workpieces.begin();
            current != workpieces.end();
            ++current)
        {
            if(!current->saved())
            {
                current->save();
            }
        }
    }
    ...
private:
    typedef std::list<workpiece *>::iterator iterator;
    std::list<workpiece *> workpieces;
    ...
};
```

This suffers from a verbose control structure and an excessive amount of checking. We might try to simplify the application class logic with the following rearrangement of responsibilities:

```
class workpiece
{
public:
    void save()
    {
        if(changed)
        {
            save_state();
            changed = false;
        }
    }
    ...
};

class application
{
public:
    void save_changes()
    {
        for(iterator current = workpieces.begin();
            current != workpieces.end();
            ++current)
        {
            current->save();
        }
    }
    ...
};
```

Here we have hidden the condition check to simplify usage. We could alternatively make better use of the algorithms in the C++ standard library, such as `for_each`, to abstract control flow. However, these refinements still fail to address the basic flaw, which is the brute-force linear search, query and act model. This is both cumbersome and inefficient, especially where many `workpiece` objects exist but only a small proportion are modified between saves.

Forces

For objects whose own behaviour changes substantially depending on their state in a lifecycle, the most direct implementation is for the object to be fully aware of its own state. This means that the object is self contained and includes all of the mechanism for its behaviour. It strengthens the coupling between the object and its lifecycle, and increases the footprint of the individual object.

For simple state models, flag based solutions are attractive because they do not require much code structuring effort, although they may lead to simplistic code with a lot of repetition and explicit, hard wired control flow. With the OBJECTS FOR STATES [Gamma+1995] pattern selection is expressed through polymorphism and behaviour within a state is modelled and implemented cohesively. This supports less dependence between the object and its state model, but it can be complex in implementation and, for large state models, it is very easy to lose sight of the lifecycle model. State transition tables can be used to drive and increase the flexibility of either the flag-based or the OBJECTS FOR STATES approach, or it may be used independently. In each of these cases the object is tied to a single, given lifecycle model. If the object's own behaviour is largely independent of its state in this model, and such state behaviour is largely governed by its application context, an internal representation of the lifecycle can increase the coupling within the object unnecessarily.

If an object's lifecycle is managed externally, this increases the object's independence from the context of its application and simplifies its representation. However, this means that significant decisions about the object's behaviour are now taken outside that object. One of the benefits of flag variables, OBJECTS FOR STATES or state transition tables held within the object is that they make the state model explicit internally. Objects are fully aware of what state they are in and they can act appropriately when used outside the collective. Also, anyone examining the class can easily determine its objects possible lifecycles.

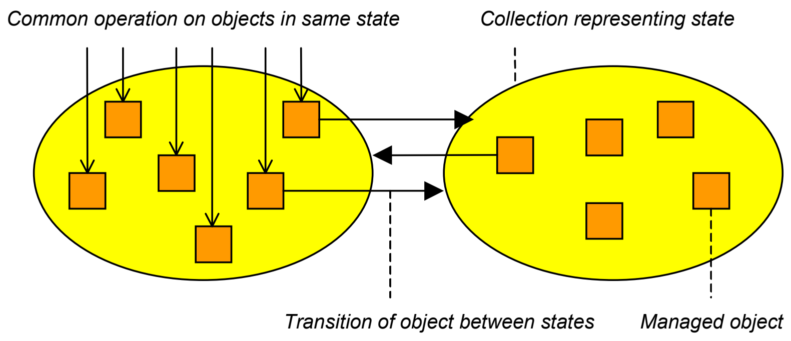
Modelling the states explicitly within objects means that if the lifecycle model is modified, the class code must be modified. Thus lifecycles cannot easily be modified independently of the representation details of the class.

With OBJECTS FOR STATES adding extra state dependent data is relatively simple: the class representing a particular state also defines any data that is relevant to objects in that state. For flag-based modelling or state transition tables the data must be held elsewhere, typically as redundant data within the stateful object. If the lifecycle is managed externally, extra state-dependent data can introduce the same management burden, i.e. the manager must add additional context data to its collection entry, or potentially redundant data must be held within the object.

With collections of objects in which objects in the same state receive uniform treatment, an internal state representation can lead to poor performance. All collected objects are traversed regardless of the state they are in, and a state-based decision is taken for each one: explicitly, in the case of flag variables; implicitly through polymorphic lookup, in the case of OBJECTS FOR STATES; implicitly through table lookup, in the case of state transition tables. In each case there is wasted effort associated with traversing all objects only to perform actions on a subset that depends on object state. External lifecycle management can support a conceptually more direct approach, which also has performance benefits.

Solution

Represent each state of interest by a separate collection that refers to all objects in that state. The manager object holds these state collections and is responsible for managing the lifecycle of the objects: when an object changes state, the manager ensures that it is moved from the collection representing the source state to the collection representing the target state.


Common operation on objects in same state
Collection representing state
Transition of object between states
Managed object

Resolution

Applying this solution to our example leads to the following code structure:

```
class workpiece
{
public:
    virtual void save() = 0;
    ...
};
class application
{
public:
    void save_changes()
    {
        for(iterator current = changed.begin();
            current != changed.end();
            ++current)
        {
            current->save();
        }
        saved.merge(changed);
    }
    ...
private:
    typedef std::list<workpiece *>::iterator iterator;
    std::list<workpiece *> saved, changed;
    ...
};
```

This is both significantly simpler and more efficient – in terms of traversal – than the previous attempts. The `saved` container refers to the objects that are unchanged and, when modified via the application, these are transferred to the `changed` container. As this is a pointer move it is cheap. To save all of the changed `workpiece` objects, the `application` object simply needs to run through the `changed` container saving each `workpiece` there, and then merge them back into the `saved` container.

If we need to treat all of the objects collectively, regardless of state, it would be tedious to set up two loops to run through all of the objects – one for `saved` and one for `changed` – and so we can represent the superstate of saved and changed `workpiece` objects, i.e. all `workpiece` objects, with an additional container:

```
class application
{
    ...
    std::list<workpiece *> workpieces, saved, changed;
    ...
};
```

This acts as the boss container that holds all of the objects in a definitive order. It is used for state-independent operations that apply to all the objects. The constraint governing container membership is that objects can only be present in one of the two state containers, but must be present in the boss container, and any object present in the boss container must be present in one of the state containers.

In the context of this application, if unsaved objects are of interest because we can collectively save them, but saved objects are not operated on as a group, we can refactor to eliminate the use of the `saved` container:

```
class application
{
public:
    void save_changes()
    {
        for(iterator current = changed.begin();
            current != changed.end();
            ++current)
        {
            current->save();
        }
        saved.clear();
    }
private:
    ...
    std::list<workpiece *> workpieces, changed;
    ...
};
```

Consequences

By assigning collections to represent states we can now move all objects in a particular state into the relevant collection and perform actions on them as a group. Other than selecting the correct collection for the state, there is no other selection or traversal required. Thus this is both a cleaner expression of the model, as well as a more time-efficient one.

An object's collection implicitly determines its state, and so there is no need to also represent the state internally. Because the objects are already being held collectively this can lead to a smaller footprint per object. This can be considered a benefit for resource-constrained environments, but need not be a necessary consequence of applying this pattern: the addition of back pointers from the object to its container or manager would counter this space saving.

The manager object now has more responsibility and behaviour, and the managed object less. On the one hand this can lead to a more complex manager, on the other it means that the object lifecycle model can be changed independently of the class of the objects. The state model can be represented within the manager using OBJECTS FOR STATES, state transition tables, or explicit hardwired conditional code, as in the motivating example. The state management can become more complex if, in changing, the state model acquires significantly more states and therefore more collections to manage.

Multiple state models can be used without affecting the class of the objects. For instance, we can introduce orthogonal state models for the objects in the motivating example based on their Zordering, versioning or some other property, e.g. if they are active objects whether or not they are currently active. Where orthogonal lifecycles are involved, allocating separate managers for different state models simplifies the implementation: a single manager would become significantly more complex through acquisition of this extra responsibility.

Where the manager acts as some kind of BROKER [Buschmann+1996] all communication with the objects will be routed through it and therefore the manager will be fully aware of what changes objects undergo. If objects can be acted on independently, state-changing events must not cause the manager to keep the wrong view, and so the manager would have to be notified. The notification collaboration can be implemented as a variation of the OBSERVER pattern [Gamma+1995], with the object either notifying the manager of a change or requesting a specific state transition of the manager. This would result in the object maintaining a back pointer to the manager, which would increase the complexity of the code and create a dependency on the manager, as well as increase the object's footprint. Referential integrity of this bidirectional relationship can be managed with MUTUAL REGISTRATION [Henney1997, Henney1999b] if the object is responsible for managing its own transitions or if the manager is not responsible for the object's lifetime.

A bidirectional relationship can also arise if the object's own behaviour is not completely independent of the manager's view of it, and must therefore have some awareness of its own state. In this case an alternative is to also adopt some redundant internal state that acts as a cache.

There are as many collections as there are states of interest in the lifecycle. This number includes superstates or a boss collection that holds all of the objects (notionally a superstate of all of the states for the objects, anyway). Thus this pattern can be applied recursively, and each superstate collection corresponds to the union of the corresponding substate collections. A boss collection is necessary where all of the objects need to be treated collectively, e.g. display or deletion, and iterating through many separate collections is inappropriate either because of convenience or because of ordering. The presence of a boss collection may obviate the need for one of the state collections, i.e. there are no collective operations performed in that state and the object is already accounted for in the boss collection.

It is not as easy to accommodate additional state-dependent data for each object as the state is managed externally. Either the object must hold redundant state or it must be held along with the object's entry in the state container. This leads to an increase in code complexity and runtime footprint. For instance, in the motivating example we could hold additional data for saved objects indicating when they were last saved, or for changed objects to record when they were changed.

Where the frequency of state change is high this pattern can become impractical, as the mediation of state change from one collection to another via a manager can come to dominate the execution time of the system. Internal representations have the benefit of immediacy when objects are acted upon individually and with frequent state changes.

The COLLECTIONS FOR STATE pattern can be used in conjunction with other lifecycle implementations as an optimisation. For instance, in a library where each loan knows its due date, an explicit collection of overdue loans can be cached for quick access, rather than requiring a search over all loans for each enquiry.

Discussion

This pattern was originally documented as CONTEXT DETERMINES STATE [Henney1997]. COLLECTIONS FOR STATES is felt to be a more accurate name and follows the style of the OBJECTS FOR STATES [Gamma+1995] indicating some commonality of intent. Note that the name OBJECTS FOR STATES is used here in preference to its common name of STATE as this is both a more accurate description of its structure – STATE describes intent, whereas OBJECTS FOR STATES describes structure – and it does not give the reader the impression that this is the only solution style possible for object lifecycles, i.e. it is "*a* state pattern" not "*the* state pattern".

In COLLECTIONS FOR STATES variation in state is expressed through collection objects rather than through polymorphic classes. In this and many other respects it is an inversion of the OBJECTS FOR STATE pattern, with state modelled extrinsically rather than intrinsically, i.e. the lifecycle is worn on the outside. It is focused on collections of objects rather than individual objects, and the entity–behaviour separation is in the opposite direction with behaviour managed by the collective rather than something within the individual. Such structural inversion illustrates that solutions to apparently similar problems, e.g. state management, need not be similar; radically different structures, where roles and responsibilities are almost reversed between the solution elements, can arise from particular differences in the context and problem statement [Henney1999a].

A set of decisions can guide developers to weigh up the use of this pattern, as opposed to or in conjunction with OBJECTS FOR STATES, state transition tables and flag-based approaches, suggesting that all these approaches can be related within a generative framework, i.e. a pattern language. Where independence and collective requirements dominate, COLLECTIONS FOR STATES proves to be a good match, leading to a better representation of the conceptual model and a simplification of object implementation.

The author has used COLLECTIONS FOR STATES consciously and unconsciously in a number of systems and examples, including a server side application very similar in detail to the motivating example. However, there are many other examples of the COLLECTIONS FOR STATES pattern in practice, and from a diversity of domains.

In systems programming and operating systems we find this pattern recurring: it is the strategy used in file systems for handling free versus used file blocks; it is used by many heap managers for holding free lists, especially debugging heap managers that track memory usage, i.e. free, used and freed; schedulers hold separate queues to schedule processes, which may be in various different states of readiness, e.g. running, ready to run or blocked. The theme of processing states can also be seen at a higher level in GUI systems supporting a comprehensive edit command history facility: once initially executed a command can be in one of two states, either done so that it can be undone or undone so it can be redone. In this case the collections used are quite specific, and must support stack-ordered (LIFO) access. An example of this can be seen in the COMMAND PROCESSOR pattern [Buschmann+1996].

COLLECTIONS FOR STATES is also a common strategy for denormalising a relational database. In a loaning library, a table representing overdue loans can be derived from the loans table. The low frequency of state change and the relative sizes of the collections make this external state model an efficient and easy to manage caching strategy, optimising lookup for queries concerning overdue books.

We can also see a similar structure in action in the real world when people are grouped together because of some aspect of state, e.g. queuing at airports with respect to EU and non-EU passports.

References

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

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

[Henney1997] Kevlin Henney, "Beyond the Gang of Four", *Software Design Masterclass: Getting the Best out of Patterns*, Unicom, 1997.

[Henney1999a] Kevlin Henney, "Patterns Inside Out", *OT '99*, 1999.

[Henney1999b] Kevlin Henney, "Mutual Registration: A Pattern for Ensuring Referential Integrity in Bidirectional Object Relationships", *EuroPLoP '99*, 1999.

[Jackson1995] Michael Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, 1995.

Acknowledgements

I would like to thank Paul Dyson for his sound shepherding, careful comments and illuminating insights, Charles Weir for his observations and additional comments, and Frank Buschmann and the rest of the POSA team at Siemens for their valuable pre-EuroPLoP workshopping of this paper.