Kevlin Henney / kevlin@curbralan.com

Collections for States

IN DESIGN, AS with anything in life, we often fall into a groove, repeating the same solutions time and again. Sometimes this is appropriate—hence the emphasis in the patterns community on capturing recurring practice—but at other times force of habit is the real decision maker. Patterns are all about capturing breadth as well as depth of knowledge, and we can find that problems that seemed similar have enough minor differences that the appropriate solutions are in fact poles apart.

Consider implementing a life cycle model for an object. It is tempting to use a flag-and-switch approach, but this is often clumsy for models with significant behavior. Objects for States (commonly known as the State pattern)[1,2] allows state-specific behavior to be represented elegantly through a delegation and inheritance structure. However, it is not the case that Objects for States is *the* state pattern, and that other techniques—including flags—are inappropriate in realizing state models. The Collections for State pattern addresses the implementation of simple state models over collections of objects.

Problem

Consider a collection of similar objects managed by another object. The managed objects have life cycles with only a few distinct states, and the Manager object[3] often operates on them collectively with respect to the state they are in. These objects can be modeled as individual state machines.

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

Kevlin Henney is an independent consultant and trainer based in the UK.

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.[4] The objects manipulated by the user are the workpieces, and these may be saved to some kind of persistent storage, for example, a database. These objects all share a simple life cycle model (see Figure 1) in addition to their type-specific state.

For such a simple state model, the use of the Objects for States pattern[1,2]—where a class hierarchy is introduced to reflect the separate states and the behavior associated with them—would be overkill. Instead, we can use a simpler flag-based approach, representing state as an attribute. In Java this would lead to the code in Listing 1. The saveState method acts as a Template Method.[1]

In an application, which acts as the manager for workpiece objects, saving all the changes since the last save could be implemented as shown in Listing 2.* However, this suffers from a verbose control structure and an excessive amount of checking. We might try to simplify the application class logic with the rearrangement of responsibilities shown in Listing 3. Here we have hidden the condition check to simplify usage. However, both of these approaches fail to address the basic flaw: the brute force of the "linear search, query, then act" model. This is both cumbersome and inefficient, especially where many

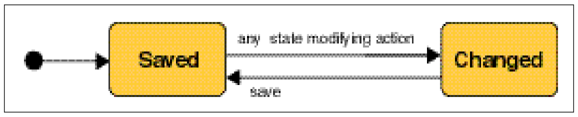

Figure 1. ***Statechart representing the simple life cycle of a workpiece object.***

* Note that for brevity, and to keep a focus on the core problem at hand, handling of threading and exception issues are omitted from the code.

LISTING 1.

Representation of workpiece saved as a flag.

```
public abstract class Workpiece
{
   public void save()
   {
      saveState();
      changed = false;
   }
   public boolean saved()
   {
      return !changed;
   }
   ...
   protected abstract void saveState();
   private boolean changed;
}
```

workpiece objects exist but only a small proportion are modified between saves.

Forces

For an object whose own behavior changes substantially depending on its state in a life cycle, the most direct implementation is for the object to be fully aware of its own state, meaning the object is self-contained and includes all of the mechanisms for its behavior. This awareness strengthens the coupling between the object and its life cycle, 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 Objects for States, selection is expressed through polymorphism and behavior within a state is modeled and implemented cohesively. This allows greater independence between the object and its state model, but can be complex in implementation, and, for large state models, it is very easy to lose sight of the life cycle 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 they may be used independently. In all of these cases the object is tied to a single given life cycle model. If the object's own behavior is largely independent of its state, and such state behavior is largely governed by its application context, an internal representation of the life cycle can increase the coupling of the object to its environment.

If an object's life cycle is managed externally, the object is independent of its context and its representation is simplified. However, this means that significant decisions about the object's behavior are now taken outside the object. One of the benefits of flag variables, Objects for States, or state transition tables is that the state model is made explicit internally. Objects are fully aware of what state they are in and can act appropriately when used outside the collective. Also, anyone examining the class can easily determine its objects' possible life cycles.

Modeling states explicitly and internally means that changes to the life cycle model require changes to the class code. So life cycles cannot easily be modified independently of the class's representation.

With Objects for States, having additional 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 or state transition tables the data must be held elsewhere, typically as potentially redundant data within the stateful object. If

LISTING 2.

Saving all changed workpiece objects by querying then saving them.

```
public class Application
{
   public void saveChanges()
   {
      Iterator workpiece = changed.iterator();
      while(workpiece.hasNext())
      {
         Workpiece current = (Workpiece) workpiece.next();
         if(!current.saved())
            current.save();
      }
   }
   ...
   private Collection workpieces;
}
```

LISTING 3.

The workpiece assumes responsibility for saving or not, depending on its state.

```
public abstract class Workpiece
{
   public void save()
   {
      if(changed)
      {
         saveState();
         changed = false;
      }
   }
   ...
}

public class Application
{
   public void saveChanges()
   {
      Iterator workpiece = changed.iterator();
      while(workpiece.hasNext())
         ((Workpiece) workpiece.next()).save();
   }
   ...
}
```

the life cycle 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, effort is wasted traversing all objects only to perform actions on a subset. External life cycle 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 life cycle 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.

Resolution

Applying this solution to our example leads to the code structure shown in Listing 4. This is both significantly simpler and more efficient, in terms of traversal, than the previous attempts. The saved collection refers to the objects that are unchanged, and when modified via the application these are transferred to the changed collection. As this is a reference move, it is cheap. To save all of the changed workpiece objects, the application object simply needs to run through the changed collection saving each workpiece there, and then merge them back into the saved collection.

LISTING 4.

Applying the proposed solution to the example.

```
public abstract class Workpiece
{
   public abstract void save();
   ...
}

public class Application
{
   public void saveChanges()
   {
      Iterator workpiece = changed.iterator();
      while(workpiece.hasNext())
         ((Workpiece) workpiece.next()).save();
      saved.addAll(changed);
      changed.clear();
   }
   ...
   private Collection saved, changed;
}
```

LISTING 5.

Adding a single collection to hold all of the objects managed.

```
public class Application
{
   ...
   Collection workpieces, saved, changed;
   ...
}
```

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 collection (see Listing 5).

The extra collection is the "boss" collection, holding all of the objects in a definitive order. It is used for state-independent operations that apply to all the objects. The constraint is clearly that objects can only be present in one of the two state collections but must be present in the boss collection, and any object present in the boss collection must be present in one of the state collections.

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 collection (see Listing 6).

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 together. Other than selecting the correct collection for the state, there is no selection or traversal required. Thus, this is both a cleaner expression of the model and a more time efficient one.

An object's collection implicitly determines its state, 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, which can be considered a benefit for resource constrained environ-

LISTING 6.

Ignoring a collection for saved objects.

```
public class Application
{
   public void saveChanges()
   {
      Iterator workpiece = changed.iterator();
      while(workpiece.hasNext())
         ((Workpiece) workpiece.next()).save();
      changed.clear();
   }
   ...
   private Collection workpieces, changed;
}
```

ments but need not be a necessary consequence of applying this pattern. Adding back-references from the object to its collection or manager would lose this space saving.

The manager object now has more responsibility and behavior, 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 life cycle 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 section. 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 Z-ordering, versioning, or some other property, e.g., if they are active objects whether or not they are currently active. When orthogonal life cycles 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,[5] all communication with the objects will be routed through it and therefore the manager will be fully aware of what changes objects might 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,[1] 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, create a dependency on the manager, and increase the object's footprint.

A bidirectional relationship can also arise if the object's own behavior 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 adopt some redundant internal state that acts as a cache.

There are as many collections as there are states of interest in the life cycle. Figure 2 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, as in a display or update. Iterating through many separate collections is not appropriate 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

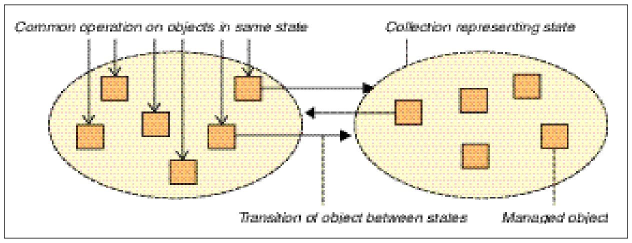
Common operation on objects in same state
Collection representing state
Transition of object between states
Managed object

Figure 2. ***Schematic representing the features of the proposed solution.***

for in the boss collection.

It is not as easy to accommodate additional state-dependent data for each object as to manage the state externally. Either the object must hold redundant state or it must be held along with the object's entry in the collection state. 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: 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 on individually and with frequent state changes.

The Collections for States pattern can be used in conjunction with other life cycle implementations as an optimization. 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 inquiry.

Discussion

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 inside out when compared to Objects for States, with state modeled extrinsically rather than intrinsically, i.e., the life cycle is worn on the outside. It is focused on collections of objects rather than individual objects, and the entity/behavior separation is in the opposite direction, with behavior managed by the collective rather than something within the individual. Such structural inversion illustrates that solutions to apparently similar problems, such as 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.

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. Such a language could be integrated with and extend existing state implementation languages.[2,6] 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.

We can see this pattern in systems programming and operating systems. It is the strategy used in file systems for handling free as opposed to 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 (last-in/first-out) access. An example of this can be seen in the Command Processor pattern.[5]

Collections for States is also a common strategy for denormalizing a relational database. The library loan example would include a table representing overdue loans. 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, optimizing lookup for queries concerning overdue books.

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

Acknowledgments

This article is based on a paper workshopped at the EuroPLoP '99 conference. I would like to thank Paul Dyson, Charles Weir, Brad Appleton, Frank Buschmann, and the rest of the POSA team at Siemens, and the members of the EuroPLoP '99 Writers' Workshop who reviewed the original paper.

References

1. Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.
2. Henney, K., "Matters of State," *Java Report*, Vol. 5, No. 6, June 2000, pp. 126–130.
3. Sommerlad, P., "Manager," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds., Addison–Wesley, 1998.
4. Jackson, M., *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison–Wesley, 1995.
5. Buschmann, F. et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
6. Dyson, P. and B. Anderson, "State Patterns," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds., Addison–Wesley, 1998.

Real World Java *continued from page 30*

"None of the problems we're having with Java are showstoppers," says Brilon. "They are all problems we can work around. And, at the end of the day, when we can tally up the math of what we're doing, we see the advantages far outweigh the disadvantages. So we're happy with our decision." ■

URLs

Apache Software Foundation www.apache.org	**Linux** www.linux.org
Enhydra.org www.enhydra.org	**Blackdown** www.blackdown.org
MySQL download web.mysql.com	**Intel** www.intel.com
Sun Microsystems Inc. www.sun.com	**Lutris Inc.** www.lutris.com
Palm Inc. www.palm.com	**Penguin Systems Inc.** www.penguincomputing.com
Microsoft Corp. www.microsoft.com	**Red Hat Systems Inc.** www.redhat.com

Philip J. Gill is Associate Editor of *Java Report*. He can be contacted at javabuzz@sigs.com.

Scott's Solutions *continued from page 104*

HTML has two types of tags: those which come in pairs (<body> and </body>) and those that do not (<p>). This second type of tag must be handled in the handleSimpleTag() method. In our case, we don't care about those tags; we simply print them out if the dumpOK flag is set to true.

That's basically all there is to it, although to complete the example, we'd really need to do a lot more work. We'd have to keep track of anchors in each document to make sure they are not repeated, and we'd want to change hyperlinks between the original documents to hyperlinks within our new single document. We might want to insert <hr> tags every time a document ends to indicate a visual separation between chapters; we could do that by printing <hr> in the handleEndTag() method when the tag in question is HTML.Tag.HTML. However, you might want to manipulate the HTML, the document parser gives you an easy way to do it. ■

Write to Us

Send comments, suggestions for topics, etc. to Managing Editor Anna M. Kanson at akanson@sigs.com