Kevlin Henney / kevlin@curbralan.com

Matters of state

IDENTITY, STATE, AND behavior can typify an object. For example, value objects[1,2]—used for representing simple pieces of information in a system, such as dates, strings, and quantities—have state, simple behavior, and transparent identity.

Objects with significant identity typically represent the load-bearing structures in a system, such as persistent data and graphical objects. They are also typically objects with significant life cycles that respond to method calls quite differently, depending on what general state or mode their content represents. Objects that do not have significant identity tend to have less interesting life cycles. A stateless service object has no state and therefore no life cycle; a value-based object tends to have either a trivial life cycle model or no life cycle model (e.g., Immutable Value[1, 2]).

The life cycle of an object can be modeled in UML using a Harel statechart.[3] Statechart diagrams can tell developers a lot about the behavior of objects over time, and unlike many conventional class models, the move to code is far from trivial. Java offers features that simplify the implementation of object life cycle models.

General Pattern

Objects for States is a general-purpose (as opposed to domain-specific) pattern[4] that can be used to move from a documented state model to a practical implementation. It demonstrates a classic use of delegation into a class hierarchy. 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 broad intent, whereas Objects for States describes structure—and does not give the reader the impression that this is the only solution style possible for object life cycles; i.e., it is "*a* state pattern" not "*the* state pattern".*

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

State Anatomy. Although often presented as a single pattern, many have recognized[5] that there is a great deal more to implementing such a collaboration than can be elegantly captured in a single pattern. A pattern that covers a great deal of complexity or describes a number of implementation alternatives can often be opened up to reveal a more involved decision structure better represented through a pattern language.

Pattern languages combine a number of patterns into a coherent whole, connecting them together through a set of decisions, with one pattern effectively completing another.[1, 2] A pattern that plays a role in a language does not belong exclusively to that language: It may be used to resolve similar problems in other pattern languages and contexts.

The Patterns. A pattern language describes how to build a system or an aspect of a system. This article looks at the Objects for States pattern from a Java perspective. It shows a fragment of a protolanguage (i.e., a work in progress), focusing specifically on how the mode of the object is represented—as opposed to, say, how its transitions are managed. Figure 1 depicts the basic flow, showing that once Objects for States has been selected we have an either/or decision as to how to best represent the objects housing the actual delegated behavior.

Stateful Object and Stateless Object have applicability outside this context, e.g., in considering the statefulness (or otherwise) of a transactional middle tier. They are here woven into the language through common links and a shared example. The patterns are described using a simple *intent-problem-solution-example* form, with a simplified example.

*Unfortunately, I have seen the havoc this misconception can wreak on a system: Every single statechart was converted unquestioningly into an implementation of "*the* State" pattern. Patterns solve particular problems well; if applied inappropriately they reduce rather than increase communication and understanding. A clear name counts for a lot.

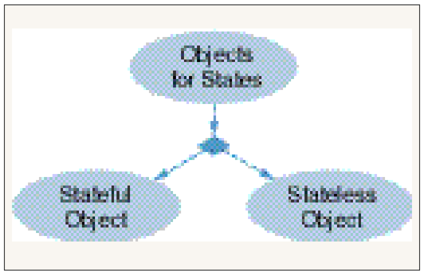

Figure 1. ***The Objects for States pattern and the construction of its representation.***

Objects for States

Allow an object to alter its behavior significantly by delegating state-based behavior to a separate object.

Problem. How can an object significantly change its behavior, depending on its internal state, without hardwired multipart conditional code? The temptation is to use a flag internal to the object and in each method to switch to the correct behavior accordingly. This creates long and complex methods. It becomes difficult to locate the behavior for a particular mode, and harder still to modify the behavioral model correctly.

Solution. Separate the behavior from the main class, which holds the context, into a separate class hierarchy, where each class represents the behavior in a particular state. The context object—the instance of the main class with which the client communicates—aggregates an object that is used to represent the behavior in one of its modes. Method calls on the context are forwarded to the mode object; responsibility for implementing behavior in a particular state is therefore delegated and localized. This behavioral mode object can be implemented as either a Stateful Object or a Stateless Object.

The transition to a different state means replacement of the behavior object by another of a different class. Polymorphism replaces the explicit conditional lookup. Transitions between states can be managed either by the behavioral mode objects themselves or by a centralized mechanism such as a table lookup.

To move with confidence from an existing design that uses a type switch of some kind to one that takes advantage of Objects for States, the Replace Type Code With State/Strategy refactoring[6] can be applied. The separation and increased messaging through forwarding methods means that this design is not suitable for distribution; i.e., mode objects should live in the same process as their context objects.

LISTING 1.

Implementation of the clock's life cycle using a flag and explicit switching.

```
public class Clock
{
  public synchronized void changeMode() ...
  public synchronized void increment()
  {
    switch(mode)
    {
    case displayingTime:
      throw new Exception("Change mode to change time");
      break;
    case settingHours:
      ++hours;
      break;
    case settingMinutes:
      ++minutes;
      break;
    }
  }
  public synchronized void cancel() ...
  ...
  private static final int displayingTime  = 0;
  private static final int settingHours    = 1;
  private static final int settingMinutes  = 2;
  private int hours, minutes;
  private int mode = settingHours;
}
```

Example. Consider the life cycle for a simple clock shown in Figure 2.

Leaving aside the issue of transitions and time updates, a tempting procedural implementation can be seen in Listing 1. Such a control-flow-based approach is error prone and scatters the state-based behavior across many methods.

Using Objects for States turns the basic structure inside out through delegation. All behavior relevant to a particular mode is encapsulated in an object and the entire selection is expressed through polymorphism (see Figure 3).

Stateful Object

Allow a behavioral object that is separate from the data object on which it operates access to that data by providing a field with a reference back to it.

Problem. Given a separation of behavior from the state that the behavior operates on, so that the state is one object and the behavior in another, what is the simplest and most direct design for the behavioral class? The separation means that whereas the behavior and data were previously in the same object—and therefore the behavior could act directly on it—the behavior can no longer directly manipulate the data.

Solution. Implement the behavioral class so that an instance has access to the contents of the object on which it operates. Because of the knowledge of its context, it has state of its own. At any one time, a behavior object is dedicated to a single context object.

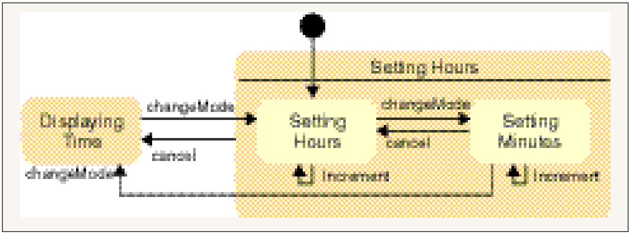

Figure 2. ***Statechart representing the life cycle of a simple clock object.***

The simplest and most direct approach is to use inner classes so that the context object's state is implicitly visible to the behavior classes. Alternatively, an explicit back reference from the behavior object to the context object can be defined, with either raw access to the data—through package or class-level visibility—or via additional query and modifier methods.

This retains the separation between behavior and the data of interest, but means that a behavioral object is now tied to a single data object at a time. This can lead to increased object creation and collection costs, especially if the behavior object is changed for another and the previous object is discarded rather than cached. If this is significant, implementing the behavioral mode object as a Stateless Object represents an alternative.

Example. Implementing the clock class with a Stateful Object for the mode leads to Listing 2. An improvement in the creation/collection performance of the program would be to preinstantiate all the required mode objects for an object and cache them in an array.

Stateless Object

Allow a behavioral object that is separate from the data object on which it operates access to that data by passing a reference back to it as an argument with each method call.

Problem. Given a separation of behavior from the state it operates on, so that the state is in one object and the behavior in another, what design for the behavioral class minimizes object creation and collection costs? A Stateful Object is simple to implement and manage, but can lead to increased object numbers and reduced performance if the objects are created and discarded frequently.

Solution. Implement the behavioral class so that it has no data in it whatsoever. The context object is now passed in as an argument to each of the delegated methods of the behavioral objects. The simplest approach that

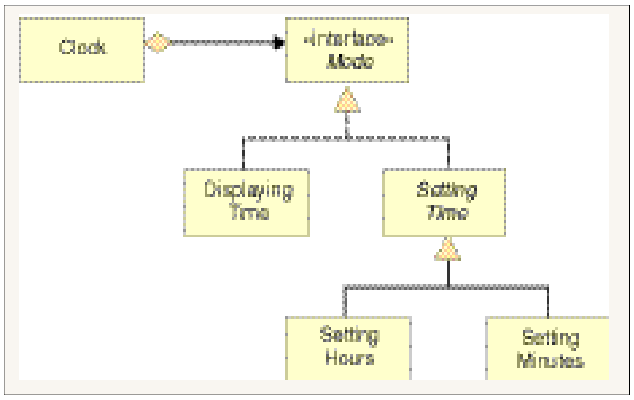

Figure 3. ***Representation of the clock state model using Objects for States.***

LISTING 2.

Implementation of the clock's life cycle using a Stateful Object for the mode.

```
public class Clock
{
  public synchronized void changeMode() { mode.changeMode();
}
  public synchronized void increment()   { mode.increment(); }
  public synchronized void cancel()      { mode.cancel(); }
  private interface Mode
  {
    void changeMode();
    void increment();
    void cancel();
  }
  private class DisplayingTime implements Mode ...
  private abstract class SettingTime implements Mode
  {
    public void cancel() { mode = new DisplayingTime(); }
  }
  private class SettingHours extends SettingTime ...
  private class SettingMinutes extends SettingTime
  {
    public void changeMode() { mode = new DisplayingTime();
}
    public void increment()  { ++minutes; }
  }
  private int hours, minutes;
  private Mode mode = new SettingHours();
}
```

LISTING 3.

Implementation of the clock's life cycle using a Stateless Object for the mode.

```
public class Clock
{
  public synchronized void changeMode()
    { mode.changeMode(this); }
  public synchronized void increment()
    { mode.increment(this); }
  public synchronized void cancel()
    { mode.cancel(this); }
  private interface Mode
  {
    void changeMode(Clock context);
    void increment(Clock context);
    void cancel(Clock context);
  }
  private static class DisplayingTime implements Mode ...
  private static abstract class SettingTime implements Mode
...
  private static class SettingHours extends SettingTime ...
  private static class SettingMinutes extends SettingTime
  {
    public void changeMode(Clock context)
    { context.mode = displayingTime; }
    public void increment(Clock context)
    { ++context.minutes; }
  }
  private int hours, minutes;
  private Mode mode = settingHours;
  private static final Mode
    displayingTime = new DisplayingTime(),
    settingHours   = new SettingHours(),
    settingMinutes = new SettingMinutes();
}
```

allows behavioral objects to see the encapsulated content of the context objects is to declare them as static top-level classes in the context class. Alternatively, package-level visibility or additional query and modifier methods must be provided.

Such behavioral objects are stateless, so they may be treated as Flyweight[4] objects that can be shared transparently among all instances of the context class. The absence of state also means that they are implicitly threadsafe, and therefore need no synchronization. Given that only a single instance of a behavioral class is ever required, Singleton[4] can be used to good effect when the definition of the behavioral class is outside the context class. Otherwise, Singleton is overkill where private static data would suffice.

The entire selection of behavior is now expressed purely through polymorphism and callbacks on a separate object, and the state now resides only in the context object. If the behavioral model is quite stable, the Visitor pattern may be used to put all the behavioral code back into the context object, with the behavioral mode object doing selection only and calling the relevant method on the context object to do all of the work. However, although this is sometimes an option, it can make the implementation unwieldy for large state models: The context class suffers method explosion.

Example. Implementing the clock class with a Stateless Object for the mode leads to Listing 3.

Conclusion

A common pattern for implementing object life cycle models can be expressed in terms of other more granular patterns. They may be linked together in a language; i.e., the main pattern can be said to *contain* or *be completed by* the other patterns. In this article, the focus was on the representation of the Objects for States pattern in Java, as opposed to a more detailed view of the whole pattern.[5]

The Stateful Object and Stateless Object patterns assist in the realization of Objects For States, but are not tied exclusively to it: The issues raised and resolutions given can be found in many object systems whether local or distributed, (e.g., EJB). ■

References

1. Henney, K., "Patterns of value," *Java Report,* Vol. 5, No. 2, Feb. 2000, pp. 64–68.
2. Henney, K., "Value added," *Java Report,* Vol. 5, No. 4, Apr. 2000, pp. 74–82.
3. Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison–Wesley, 1999.
4. Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.
5. Dyson, P. and Anderson, B. "State Patterns," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds., Addison–Wesley, 1998.
6. Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison–Wesley, 1999.