Patterns in C - Part 2: STATE

By Adam Petersen <adampetersen75@yahoo.se>

Every non-trivial program passes through a number of different states during its lifecycle. Describing this lifecycle as a finite state machine is a simple and useful abstraction. In this part of the series, we will investigate different strategies for implementing state machines. The goal is to identify mechanisms that let the code communicate the intent of expressing the problem as a finite state machine.

Traditional Solution with Conditionals

Consider a simple, digital stop-watch. In its most basic version, it has two states: started and stopped. A traditional and direct way to implement this behavior in C is with conditional logic in the shape of switch/case statements and/or if-else chains.

The digital stop-watch in this example is implemented as a First-Class ADT [1].

```
typedef enum
{
   stopped,
   started
} State;

struct DigitalStopWatch
{
   /* Let a variable hold the state of our object. */
   State state;
   TimeSource source;
   Display watchDisplay;
};

void startWatch(DigitalStopWatchPtr instance)
{
   switch(instance->state)
   {
      case started:
         /* Already started -> do nothing. */
         break;
      case stopped:
         instance->state = started;
         break;
      default: error(”Illegal state”); break;
  }
}

void stopWatch(DigitalStopWatchPtr instance)
{
   switch(instance->state)
   {
      case started:
         instance->state = stopped;
         break;
      case stopped:
         /* Already stopped -> do nothing. */
         break;
      default: error(”Illegal state”); break;
  }
}
```

While this approach has the advantage of being simple and easy to understand, it introduces several potential problems:

1. *It doesn’t scale*. In large state machines the code may stretch over page after page of nested conditional logic. Imagine the true maintenance nightmare of changing large, monolithic segments of conditional statements.

2. *Duplication*. The conditional logic tends to be repeated, with small variations, in all functions that access the state variable. As always, duplication leads to error-prone maintenance. For example, simply adding a new state implies changing several functions.

3. *No separation of concerns*. When using conditional logic for implementing state machines, there is no clear separation between the code of the state machine itself and the actions associated with the various events. This

makes the code hide the original intent (abstracting the behaviour as a finite state machine) and thus making the code less readable.

A Table-based Solution

The second traditional approach to implement finite state machines is through transition tables. Using this technique, our original example now reads as follows.

```
typedef enum
{
   stopped,
   started
} State;

typedef enum
{
   stopEvent,
   startEvent
} Event;

#define NO_OF_STATES 2
#define NO_OF_EVENTS 2

static State TransitionTable[NO_OF_STATES][NO_OF_EVENTS] = {
                                { stopped, started },
                                { stopped, started } };

void startWatch(DigitalStopWatchPtr instance)
{
   const State currentState = instance->state;
   instance->state = TransitionTable[currentState][startEvent];
}

void stopWatch(DigitalStopWatchPtr instance)
{
   const State currentState = instance->state;
   instance->state = TransitionTable[currentState][stopEvent];
}
```

The choice of a transition table over conditional logic solved the previous problems:

1. *Scales well*. Independent of the size of the state machine, the code for a state transition is just one, simple table-lookup.

2. *No duplication*. Without the burden of repetitive switch/case statements, modification comes easily. When adding a new state, the change is limited to the transition table; all code for the state handling itself goes unchanged.

3. *Easy to understand*. A well structured transition table serves as a good overview of the complete lifecycle.

Shortcomings of Tables

As appealing as table-based state machines may seem at first, they have a major drawback: it is very hard to add actions to the transitions defined in the table. For example, the watch would typically invoke a function that starts to tick milliseconds upon a transition to state started. As the state transition isn't explicit, conditional logic has to be added in order to ensure that the tick-function is invoked solely as the transition succeeds. In combination with conditional logic, the initial benefits of the table-based solution soon decrease together with the quality of the design.

Other approaches involve replacing the simple enumerations in the table with pointers to functions specifying the entry actions. Unfortunately, the immediate hurdle of trying to map state transitions to actions in a table based solution is that the functions typically need different arguments. This problem is possible to solve, but the resulting design looses, in my opinion, both in readability as well as in cohesion as it typically implies either giving up on type safety or passing around unused parameters. None of these alternatives seem attractive.

Transition tables definitely have their use, but when actions have to be associated with state transitions, the STATE pattern provides a better alternative.

Enter STATE Pattern

In its description of the STATE pattern, *Design Patterns* [2] defines the differences from the table-based approach as "the STATE pattern models state-specific behavior, whereas the table-driven approach focuses on defining state transitions". When applying the STATE pattern to our example, the structure in Figure 1 emerges.

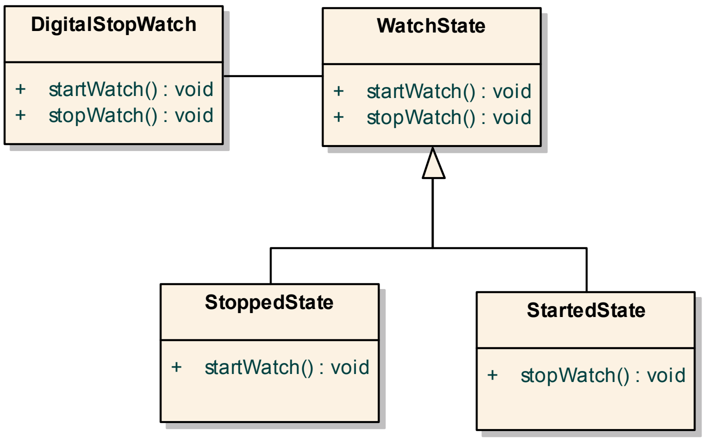

Figure 1: STATE pattern structure

This diagram definitely looks like an object oriented solution. But please don't worry – we will not follow the temptation of the dark side and emulate inheritance in C. However, before developing a concrete implementation, let's explain the involved participants.

- `DigitalStopWatch`: *Design Patterns* [2] defines this as the context. The context has a reference to one of our concrete states, without knowing exactly which one. It is the context that specifies the interface to the clients.
- `WatchState`: Defines the interface of the state machine, specifying all supported events.
- `StoppedState` and `StartedState`: These are concrete states and each one of them encapsulates the behavior associated with the state it represents.

The main idea captured in the STATE pattern is to represent each state as an object of its own. A state transition simply means changing the reference in the context (`DigitalStopWatch`) from one of the concrete states to the other.

Implementation Mechanism

Which mechanism may be suitable for expressing this, clearly object oriented idea, in C? Returning to our example, we see that we basically have to switch functions upon each state transition. Luckily, the C language supplies one powerful feature, pointers to functions, that serves our needs perfectly by letting us change the behaviour of an object at run-time. Using this mechanism, the interface of the states would look as:

Listing 1: The state interface in `WatchState.h`

```
/* An incomplete type for the state representation itself. */
typedef struct WatchState* WatchStatePtr;

/* Simplify the code by using typedefs for the function pointers. */
typedef void (*EventStartFunc)(WatchStatePtr);
typedef void (*EventStopFunc)(WatchStatePtr);

struct WatchState
{
   EventStartFunc start;
   EventStopFunc stop;
};
```

Breaking the Dependency Cycle

After getting used to the scary syntax of pointers to functions, the interface above looks rather pleasant. However, with the interface as it is, a dependency cycle will evolve.

Consider the pointers in the WatchState structure. Every concrete state has to define the functions to be pointed at. This implies that each time an event is added to the interface, all concrete states have to be updated. The resulting code would be error-prone to maintain and not particularly flexible.

The good news is that breaking this dependency cycle is simple and the resulting solution has the nice advantage of providing a potential error-handler. The trick is to provide a default implementation, as illustrated in the listing below.

Listing 2: Extend the interface in `WatchState.h`

```
/* ..previous code as before.. */
void defaultImplementation(WatchStatePtr state);
```

Listing 3: Provide the default implementations in `WatchState.c`

```
static void defaultStop(WatchStatePtr state)
{
  /* We'll get here if the stop event isn't supported in the concrete state. */
}

static void defaultStart(WatchStatePtr state)
{
  /* We'll get here if the start event isn't supported in the concrete state. */
}

void defaultImplementation(WatchStatePtr state)
{
   state->start = defaultStart;
   state->stop = defaultStop;
}
```

Concrete States

The default implementation above completes the interface of the states. The interface of each state itself is minimal; all it has to do is to declare an entry function for the state.

Listing 4: Interface of a concrete state, `StoppedState.h`

```
#include ”WatchState.h”

void transitionToStopped(WatchStatePtr state);
```

Listing 5: Interface of a concrete state, `StartedState.h`

```
#include ”WatchState.h”

void transitionToStarted(WatchStatePtr state);
```

The responsibility of the entry functions is to set the pointers in the passed WatchState structure to point to the functions specifying the behavior of the particular state. As we can utilize the default implementation, the implementation of the concrete states is straightforward; each concrete state only specifies the events of interest in that state.

Listing 6: `StoppedState.c`

```
#include ”StoppedState.h”
/* Possible transition to the following state: */
#include ”StartedState.h”

static void startWatch(WatchStatePtr state)
{
   transitionToStarted(state);
}

void transitionToStopped(WatchStatePtr state)
{
   /* Initialize with the default implementation before specifying
      the events to be handled in the stopped state. */
   defaultImplementation(state);
   state->start = startWatch;
}
```

Listing 7: `StartedState.c`

```
#include ”StartedState.h”
/* Possible transition to the following state: */
#include ”StoppedState.h”

static void stopWatch(WatchStatePtr state)
{
   transitionToStopped(state);
}

void transitionToStarted(WatchStatePtr state)
{
   /* Initialize with the default implementation before specifying
      the events to be handled in the started state. */
   defaultImplementation(state);
   state->stop = stopWatch;
}
```

Client Code

The reward for the struggle so far comes when implementing the context, i.e. the client of the state machine. All the client code has to do, after the initial state has been set, is to delegate the requests to the state.

```
struct DigitalStopWatch
{
   struct WatchState state;
   TimeSource source;
   Display watchDisplay;
};

DigitalStopWatchPtr createWatch(void)
{
   DigitalStopWatchPtr instance = malloc(sizeof *instance);

   if(NULL != instance)
   {
      /* Set the initial state. */
      transitionToStopped(&instance->state);
      /* Initialize the other attributes here. */
   }

   return instance;
}

void destroyWatch(DigitalStopWatchPtr instance)
{
   free(instance);
}

void startWatch(DigitalStopWatchPtr instance)
{
   instance->state.start(&instance->state);
}

void stopWatch(DigitalStopWatchPtr instance)
{
   instance->state.stop(&instance->state);
}
```

A Debug Aid

In order to ease debugging, the state structure may be extended with a string holding the name of the actual state. Example:

```
void transitionToStopped(WatchStatePtr state)
{
   defaultImplementation(state);

   state->name = “Stopped”;
   state->start = startWatch;
}
```

Utilizing this extension, it becomes possible to provide an exact diagnostic in the default implementation. Returning to our implementation of `WatchState.c`, the code now looks like:

```
static void defaultStop(WatchStatePtr state)
{
   /* We’ll get here if the stop event isn’t supported in the concrete state. */
   logUnsupportedEvent(“Stop event”, state->name);
}
```

Extending the State Machine

One of the strengths of the STATE pattern is that it encapsulates all state-specific behavior making the state machine easy to extend.

- *Adding a new event*. Supporting a new event implies extending the **`WatchState`** structure with a declaration of another pointer to a function. Using the mechanism described above, a new default implementation of the event is added to `WatchState.c`. This step protects existing code from changes; the only impact on the concrete states is on the states that intend to support the new event, which have to implement a function, of the correct signature, to handle it.

- *Adding a new state*. The new, concrete state has to implement functions for all events supported in that state. The only existing code that needs to be changed is the state in which we'll have a transition to the new state. Please note that the STATE pattern preserves one of the benefits of the table-based solution: client code, i.e. the context, remains unchanged.

Stateless States

The states in the sample code are stateless, i.e. the **`WatchState`** structure only contains pointers to re-entrant functions. Indeed, this is a special case of the STATE pattern described as *"If State objects have no instance variables […] then contexts can share a State object"* [2]. However, before sharing any states, I would like to point to Joshua Kerievsky's advice that *"it's always best to add state-sharing code* **after** *your users experience system delays and a profiler points you to the state-instantiation code as a prime bottleneck"* [3].

In the C language, states may be shared by declaring a static variable representing a certain state inside each function used as entry point upon a state transition. As the variables now have permanent storage, the signature of the transition functions is changed to return a pointer to the variable representing the particular state.

Listing 8: Stateless entry function, `StartedState.c`

```
WatchStatePtr transitionToStarted(void)
{
   static struct WatchState startedState;
   static int initialized = 0;

   if(0 == initialized)
   {
      defaultImplementation(&startedState);
      startedState.stop = stopWatch;

      initialized = 1;
   }
   return &startedState;
}
```

The client code has to be changed from holding a variable representing the state to holding a pointer to the variable representing the shared state. Further, the context has to define a callback function to be invoked as the concrete states request a state transition.

Listing 9: Client code for changing state

```
void changeState(DigitalStopWatchPtr instance, WatchStatePtr newState)
{
   /* Provides a good place for controls and trace messages (all state
transitions have to go through this function). */
   instance->state = newState;
}
```

The stateless state version comes closer to the State described in *Design Patterns* [2] as a state transition, in contrast with the previous approach, implies changing the object pointed to by the context instead of just swapping its behaviour.

Listing 10: State transition in `StoppedState.c`

```
static void startWatch(DigitalStopWatchPtr context)
{
   changeState(context, transitionToStarted());
}
```

A good quality of the stateless approach is that the point of state transitions is now centralized in the context. One obvious drawback is the need to pass around a reference to the context. This reference functions as a memory allowing the new state to be mapped to the correct context. Another drawback is the care that has to be taken with the initialization of the static variables if the states are going to live in a multithreaded world.

Consequences

The main consequences of applying the STATE pattern are:

1. *Reduces duplication introduced by complex, state-altering conditional logic*. As illustrated in the example above, solutions based upon large segments of conditional logic tends to contain duplicated code. The STATE pattern provides an appealing alternative by removing the duplication and reducing the complexity.

2. *A clear expression of the intent*. The context delegates all state dependent operations to the state interface. Similar to the table-based solution, the STATE pattern lets the code reflect the intent of abstracting the problem as a finite state machine. With complex, conditional logic, that intent is typically less explicit.

3. *Encapsulates the behavior of each state*. Each concrete state provides a good overview of its behavior including all events supported in that very state. This encapsulation makes it easy both to identify as well as updating the relevant code when changes to a certain state are to be done.

4. *Implicit error handling*. The solutions based on conditional logic, as well as the table-based one, requires explicit code to ensure that a given combination of state and event is valid. Using the technique described above of initializing with a default implementation, the controls are built into the solution.

5. *Increases the number of compilation units*. The code typically becomes less compact with the STATE pattern. As *Design Patterns* says "such distribution is actually good if there are many states" [2]. However, for a trivial state machine with few, simple states, the STATE pattern may introduce an unnecessary complexity. In that case, if it isn't known that more complex behavior will be added, it is probably better to rely on conditional logic in case the logic will be easy to follow.

Summary

The STATE pattern lets us express a finite state machine, making the intent of the code clear. The behavior is partitioned on a per-state-basis and all state transitions are explicit.

The STATE pattern may serve as a valuable tool when implementing complex state-dependent behavior. On the other hand, for simple problems with few states, conditional logic is probably just right.

Next Time

We'll continue with *Design Patterns* [2] and investigate the Strategy pattern, which is closely related to STATE. The STRATEGY pattern lets us implement different variation points of an algorithm, interchangeable at run time.

References

1. Adam Petersen , "Patterns in C, part 1", C Vu 17.1
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J, "Design Patterns", Addison-Wesley
3. Joshua Kerievsky , "Refactoring to Patterns", Addison-Wesley

Acknowledgements

Many thanks to Magnus Adamsson, Tord Andersson, and André Saitzkoff for their feedback.