ON THE IMPLEMENTATION OF STATE PATTERN

Iwo Herka
Design Patterns, COMP40070
MSc in Advanced Software Engineering

This paper describes Automaton pattern, object-oriented state machine, which extends capabilities of the classic State design pattern. The intent of these patterns is to allow alteration of object's behaviour, when its internal state changes. Presented variation of the standard solution makes it possible to define state relations at run-time, clearly separates state-transition logic from functional logic, minimizes necessary coupling and favors greater code reusability.

Introduction

It has been noted that the perception of the system being modeled determines design strategies used to represent it in a computer program [1]. The following paper aims to examine one of such strategies, as well as its by-products, in the context of *object-oriented* paradigm. OOP represents computational processes as collections of interacting objects, each possessing identity defined by its behavior and internal state [1]. This is a very powerful abstraction, which greatly favors modularity. Breaking up complex systems into computational components allows to develop and maintain them separately, vastly reducing programming effort.

For such behavior to be achieved, however, some issues must be addressed first. With OOP, programmer must be concerned how relation between object's state and behavior is modeled. In some cases, it is beneficial to treat objects as simple containers for variables and procedures, which, similarly to mathematical functions, always return same output for the same input. In other cases, the best solution is to render object immutable (thus disconnecting behavior from state) and profit from memory sharing or parallel execution. On the other hand, if we would decide to make object's behavior function of its state, then functionality of its methods must change at run-time, along with the state.

When behavior is defined in terms of class interface, *State* design pattern addresses issue of varying it at run-time. It works by extracting functional logic into separate state classes, linked with client object by composition, which serve as delegatees for state-dependent methods [2]. Because of this, a programmer does not have to depend on long conditional statements, logic is neatly separated and code is more readable and easy to understand.

Despite many advantages, this approach has certain trade-offs. Most fundamental consequences of the pattern are dictated by the approach of dealing with the problem itself. Switching states requires memory allocation, which increases memory requirements. In some embedded systems with limited resources, this overhead may be intolerable. Because of the layer of indirection between client and actual server (state), compiler cannot inline code as frequently. This translates into performance loss.

Luckily, there have been recognised at least three common problems with *State* implementations that *can* be addressed [4]:

- *Maintenance (evolution) of State/FSM implementations.* *State* pattern is a classic example of the open/closed software, i.e. states are easy do add but hard to modify. However, in real-world applications structure of the system tends to change often, therefore the modifications should be reasonably easy to make. This is not the case with traditional implementations of *State* pattern.
- *Duplication of objects.* Where *State* pattern is initialized multiple times, techniques can be applied to minimize memory overhead (e.g. by combining *State* and *Flyweight* patterns).
- *Data management.* Extracting logic from its context poses a problem of managing data it operates on and increases risks of unwanted side-effects. Programmer must be concerned with the trade-off between state managing its own records versus state object polling its owner.

The pattern description is far from being faultless. It depends, to a large extent, on the context it is being used in, and details of the problem being addressed. In this paper, the focus is on the maintenance problem and methods that can be used to prevent it. By maintenance, it is meant changes to the system which may involve adding, modifying or removing states or transitions between states.

It is not stated by the pattern how and where state transitions should take place, so the problem of modeling them must be resolved during design stage. Most straight-forward solution would place transition logic in the context class. This, however, would

only work if relations between states were fixed and not subject to change [2]. As original authors suggest, letting state objects themselves handle transition process is generally more flexible and appropriate solution. Distribution of transition logic makes it easier to modify and extend by addition of new state classes [2]. On the other hand, this requires special interface for context (to allow changing the state by second party) and introduces high coupling between states (which would require knowledge of each other).

Both approaches pose a serious threat to maintainability. Because relations between states are fixed at run-time, any modifications to transition rules would require changes directly in the source code. This view is supported by van Gurp and Bosh, who suggest that: "The *State* pattern is not black-box. Building a protocol requires developers to extend classes rather than to configure them. To do so, code needs to be edited and extended rather then composed from existing components" [4]. Secondly, high coupling between states makes it hard to edit already existing relations. Adding new states would require changes in multiple places, especially when transitional logic is distributed vertically across class hierarchy via inheritance [4].

This paper illustrates variant of *State* pattern, which stores transitional logic in tabular format and uses abstraction of events to perform context switching. For convenience, it will be refereed to as *Automaton*. Representation of state transition rules as hash table entries makes it possible to define them at run-time, favoring black-box configuration. Introduction of events minimizes interdependencies of states, increasing reusability.

Pattern Description

I. Intent

Similarly to the original variant of the pattern, *Automaton*'s intent is to make it possible to modify monolithic object's behavior at run-time, so that implementation of the interface changes along with its time-varying state.

II. Motivation

Consider class `Strip` which takes a string as an argument and returns its copy with leading and trailing whitespace characters (tokens) removed. `Strip` is a simple FSM with five possible states: `Empty`, `Leading`, `Core`, `Trailing` and `Done`. Parsing states indicate which part of the string is currently being read: leading, core (which remains unchanged) or trailing, consecutively. `Strip` can parse a character (`parseChar`), parse a token (`parseToken`) or send end-of-file signal (`EOF`):

```
public interface IParser {
    public void parseChar();
    public void parseToken();
    public String EOF();
}
```

`Empty` is default state, which transitions to `Leading` when first token is encountered or to `Core` when there are none to be removed. In `Leading` state FSM loops until first character is read and moves to `Core`. Relation between `Core` and `Trailing` is bi-directional to compensate for strings which contain whitespace characters in their core part. Each time a token is parsed outside string's leading part, FSM assumes trailing state. Once allowed character is read again, `Strip` concludes previous token to be false positive and returns back to `Core` state. Finally, when EOF characters is found, FSM transitions to `Done`, independently of the state it is currently in. *State* pattern describes how `Strip` object can exhibit different behavior in each of the states.

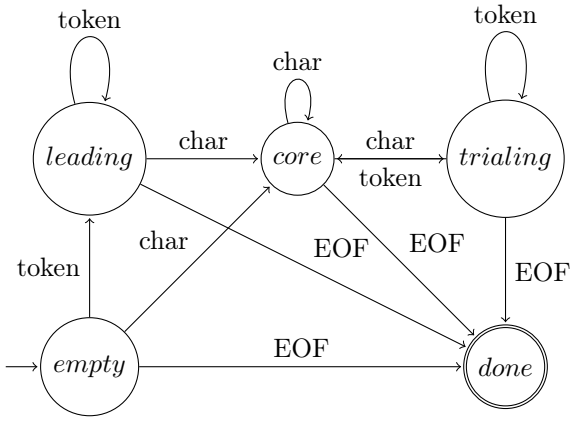

Figure 1 – State diagram of `Strip` FSM.

The key idea behind *Automaton* pattern (similarly to *State* pattern) is to represent generic state as interface (`IParser`) or abstract class, and let concrete states implement state-specific logic. Then, behavior can be bound to context object by composition and changed at run-time. The class `Strip` keeps reference to `IParser` state and delegates any non-trivial logic to it. Difference between two implementations is embodied by strategies used to determine how and when next state is chosen. In traditional approaches, those relations are fixed in code, where current state explicitly calls its successor. Because states know of each other, the smallest modification may involve changes in multiple places.

Automaton variant uses abstraction of *events* to carry out transitions between states. This process can be demonstrated by walking through an example. When current state `Core` reads token from the buffer, hash-table look-up for key `TOKEN` is issued. Value of this table entry is the class of the next state. Then, state can notify context object what transition to carry out. Each state maintains its own table of events. This way, there is no need to introduce additional interface to context object for catching events and checking validity of the transition.

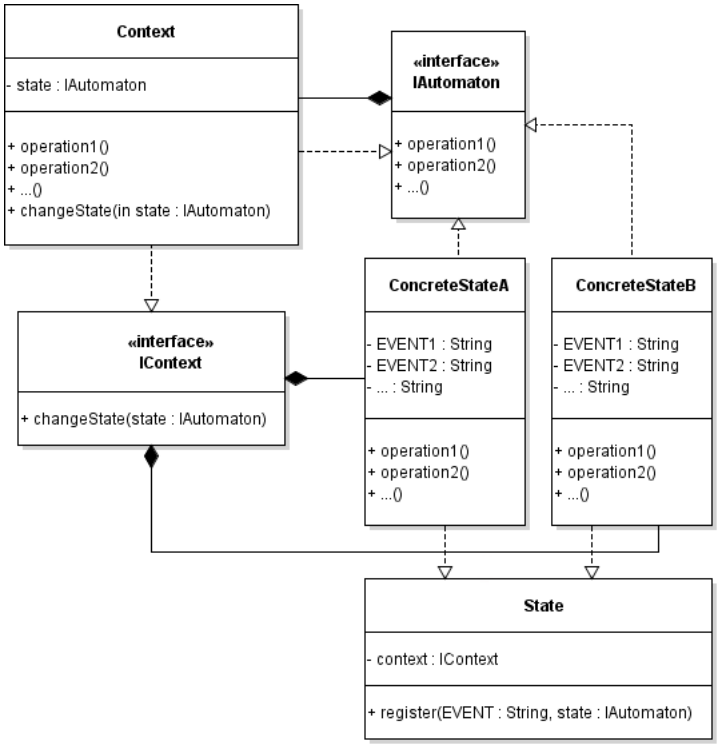

Figure 2 – State diagram of the Automaton

III. Applicability

This variant of the *State* can be used in situations where:

- Object's behavior must change at run-time, depending on its internal state,
- Program has long conditional statements that depend on the state of the object,
- Transitions between states must be configurable, i.e. act as a black-box,
- Transitions between states must change at runtime,
- When states are reused across the system, so that coupling between them should be minimal,
- When low data dependencies between context and states allow to further decrease coupling.

IV. Structure

`IAutomaton` is the interface of the object that clients uses. Both context object and states have to implement this interface. `Context` is used as an abstraction of the context object used by state classes. It is an abstract class with default implementation of `changeState`, which is a simple setter. Hash-maps `transitions` are local storages of state transitions in key-value format. From them, states can be initialized. Strings `EVENT1`, `EVENT2`, ... serve as keys to hash-tables and symbolic representation of events occurring in the system. Because each state maintains separate repository of local transitions, events can be both global and local, i.e. one string can be bound to numerous transitions across multiple states or be unique. For example, in `Strip` FSM, `CHAR` event will result in different transition if system is currently in the state `Leading`, rather than in `Trailing`. Structure of the pattern is presented in Figure 2.

V. Participants

- *Automaton interface* (`«IAutomaton»`). Implemented by the context object and states. It serves as primary interface for interaction with the client.
- *Abstract class Context* (`Context`). Subclassed by the context object. It is used to issue state transitions.
- *Context object* (`Context`). Main object in the system. It binds `«IAutomaton»` interface with its dynamically changing implementation.
- *Abstract class State* (`Context`). Subclassed by concrete states. It is used to register state transitions.
- *Concrete state classes* (`ConcreteStateA`, `ConcreteStateB`). Concrete implementor of the logic; provides functionality for the context object.

VI. Collaborations

- During its lifetime, context object delegates all state-specific behavior to the current `IAutomaton` instance.
- States are initialized with reference to their owner passed as an argument.
- Current state responds to requests, and issues state transitions locally, with help of events.
- Context object serves as main interface provider for clients.

VII. Consequences

Because of its nature, Automaton shares many consequences with the original *State* pattern. Gamma, Helm, Johnson and Vlissides point out that: "[*State*] localizes state-specific behavior and partitions behavior for different states.", "makes state transitions explicit", and "[because of it,] state objects can be shared" [2]. These statements are also true for this variant. Implementation specific consequences are following:

- Relations between states are configurable at run-time. Because of this, Automaton can be used as blackbox framework, i.e. users of the framework can use it by configuring usable components, rather then modifying or subclassing them [4].
- Introduction of events helps to retain more direct relationship between original *finite state machine* concepts and programming pattern.
- Automaton pattern favors maintainability and low coupling by keeping state classes unaware of each other, leaving responsibility of defining inter-state relations to client.
- Context objects are completely separated from concrete states, as long as states does not have to poll context for data.
- Extra of level of indirection between client and actual server (state object) may slow down execution of the program.
- Because state relations are not fixed in code, they have to be kept in memory. This requires more memory overhead.

VIII. Implementation issues

Automaton pattern, as described in this paper, does not solve data management problem, i.e. how flow of data between context and states should be organized. Common trade-off involves state polling its owner for information, versus sending them all at once ahead of time. Former solution is more suitable when data dependency is substantial. Also, where objects are initialized multiple times, memory usage may be heavy. *Automaton* does not describe techniques that can be applied to minimize memory overhead. Oftentimes, this can be reduced by object pooling or incorporating *Flyweight* pattern [4].

IX. Related patterns

As Adamczyk points out: "no finite state machine pattern is an island". Almost every aspect of its design is subject to the problem being addressed [3]. It depends on resource requirements, as well as programing and system environment. For over thirty years, many approaches have been tried. Adamczyk [3] describes twenty four possible implementations. From this anthology, variant described in this paper is most closely akin to *FSM Framework*, published by van Gurp and Bosch [4]. In this pattern, every component of the FSM is modeled separately, i.e.: *FSM object*, *FSM context*, *action*, *transition* and *state* are all represented by distinct classes. Additionally, pattern incorporates elements of the *Command* pattern, via `FSMAction`. Similarly to *Automaton*, in *FSM Framework* state contains a set of transition-event pairs. `Transition` itself, however, is played out differently: "`FSMContext` object forwards the request to the `State` object that looks up the transition corresponding to the incoming event and executes it. The `Transition` object knows which `FSMAction` to execute and resets the current state on the `FSMContext object`" [4]. This strategy allows for even greater flexibility, at the cost of slower performance.

Shalyto, Shamgunov and Korneev [5] describe another similar *State* implementation. In *State Machine* pattern, context object maintains global dictionary of state transitions. In order to allow communication between states and context, context object implements `IEventSink` interface with `castEvent` method. To notify context of incoming transition, states cast instances of the `Event` class. Then, dictionary is checked for the next state. This implementation introduces redundant interfaces and involves close communication between states and context. However, maintaining one repository is easier then multiple local ones.

X. Conclusions

Automaton improves on *State* pattern, by incorporating black-box design. It allows to configure its components at run-time, rather then modifying or extending existing class hierarchy. *Automaton* introduces abstraction of events to carry out transitions between states. Because of this, coupling between state classes is eliminated, favoring greater code reusability.

XII. References

1. Abelson H., Sussman G. J., Sussman J. Structure and Interpretation of Computer Programs, ISBN 0-262-51087-1. MIT Press, 1996.
2. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0-201-63361-2. Addison-Wesley, 1995.
3. Adamczyk P. The Anthology of the Finite State Machine Design Patterns. http://hillside.net/plop/plop2003/Papers/Adamczyk-State-Machine.pdf, 2003.
4. van Gurp J., Bosch J. On the Implementation of Finite State Machines. In Proc. of the IASTED International Conference, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.369&rep=rep1&type=pdf, 1999.
5. Shalyto, A., Shamgunov, N., Korneev, G. State machine design pattern. .NET Technologies, 2006.

XI. Code sample

```
public interface IAutomaton {
    public void connect();
    public void disconnect();
    public void sendMsg(String msg) throws IllegalActionException;
}

public abstract class Context {
    protected IAutomaton mState;

    public void setState(IAutomaton state) { mState = state; }
}

public abstract class State {
    private Map<String, IAutomaton> mTable;
    private Context mContext;

    public State(Context context) {
        mContext = context;
        mTable = new HashMap<String, IAutomaton>();
    }

    public void register(final String EVENT, IAutomaton state) {
        mTable.put(EVENT, state);
    }
}

public class TCPConnection extends Context implements IAutomaton {
    public void connect() { mState.connect(); }

    public void disconnect() { mState.disconnect(); }

    public void sendMsg(String msg) throws IllegalActionException {
        mState.sendMsg(msg);
    }
}

public class Connected extends State implements IAutomaton {
    public final static String DISCONNECT = "DISCONNECT";

    public Connected(Context context) { super(context); }

    public void connect() {}

    public void disconnect() {
        mContext.setState(mTable.get(DISCONNECT));
    }

    public void sendMsg(String msg) {
        System.out.println(msg);
    }
}

public class Disconnected extends State implements IAutomaton {
    public final static String CONNECT = "CONNECT";

    public Disconnected(Context context) { super(context); }
```

```
    public void connect() {
        mContext.setState(mTable.get(CONNECT));
    }

    public void disconnect() {}

    public void sendMsg(String msg) throws IllegalActionException {
        final String errMsg = "Cannot send messages in disconnected state!";
        throw new IllegalActionException(errMsg);
    }
}
```