The Selex Design Pattern: Decomposing State Machines Cluttered by Message Multiplexing

Frank Roessler
Avaya Labs
Basking Ridge, NJ
USA
+1-908-696-5127
roessler@avaya.com

Birgit Geppert
Avaya Labs
Basking Ridge, NJ
USA
+1-908-696-5116
bgeppert@avaya.com

ABSTRACT
State machine specifications and their implementations are often complex because they have many responsibilities mixed together. A potential cause for responsibility clutter is *message multiplexing*, which means that one or more incoming and/or outgoing messages of the state machine contain data that belongs to different concerns. The Selex pattern untangles responsibility clutter due to message multiplexing without changing the external behavior of the state machine.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Patterns, Information hiding, Domain-specific architectures; D.2.2 [**Design Tools and Techniques**]: State diagrams, Modules and interfaces

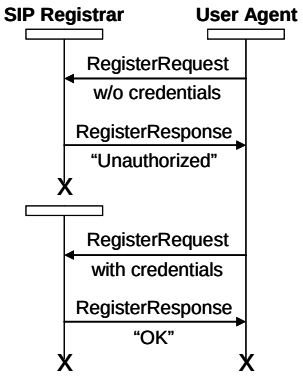
General Terms
Design.

Keywords
Design pattern, Communication protocol, State machine, Composition, Message multiplexing.

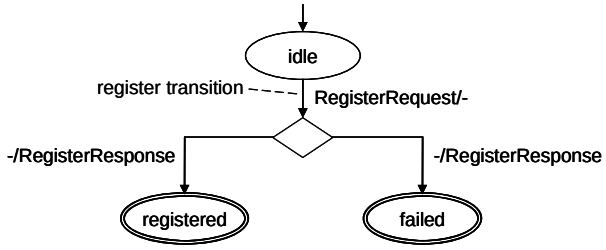
1. INTENT
Decompose a state machine cluttered by message multiplexing into a set of component state machines with clearly separated responsibilities. The externally visible behavior of the composite is kept unchanged.

What is message multiplexing? In the field of telecommunications, message multiplexing means transmitting messages from multiple sources over a single channel. Here, we use *message multiplexing* as a software engineering term, meaning that a single message contains data from separate concerns.

2. MOTIVATION
Consider the SIP [5] registration protocol. It has two responsibilities: a) authenticating the registering user agent (authentication) and b) submitting user locations to a location service (address binding). Figure 1(a) illustrates the four-way handshake for successfully registering a user agent. You could model and eventually implement the SIP registrar as shown in Figure 1(b). The state machine looks fairly simple. After all it has just one transition. The problem is that SIP puts data for authentication as well as address binding into the same incoming message `RegisterRequest` and therefore the single transition must handle both responsibilities. If you wanted to add many more responsibilities (by extending `RegisterRequest`) this approach would not scale very well.

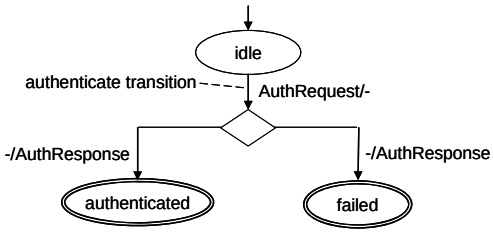


(a) Message flow (success path)

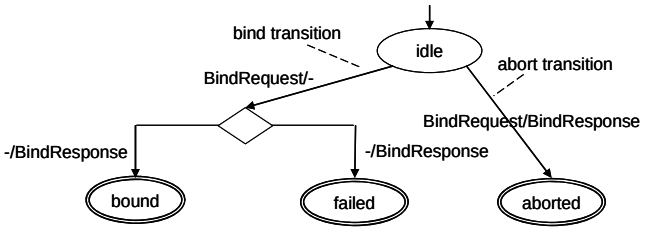


(b) SIP registrar - authentication mixed with address binding

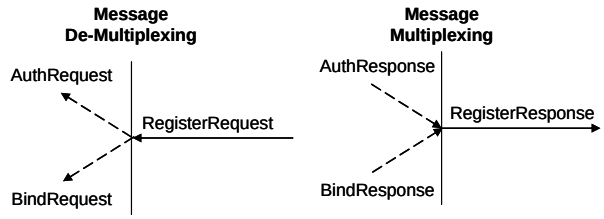
Figure 1. SIP registration.



(a) Component state machine `Authentication`



(b) Component state machine `AddressBinding`



(c) Part of glue code

Figure 2. SIP registrar - authentication separated from address binding.

Rather than following the approach of Figure 1(b), you could first de-multiplex `RegisterRequest` into an authentication and address binding part (Figure 2(c)) and then feed these micro messages into separate component state machines - one responsible for authentication (Figure 2(a)) and another one responsible for address binding (Figure 2(b)). While processing their micro messages, the component state machines create new outbound micro messages. Once they are all ready, you can multiplex them to the outgoing `RegisterResponse` and send the composite message over the network (Figure 2(c)).

The component state machines are simple, so a straightforward flag-and-switch approach will suffice for their implementation. You need one class for authentication, another one for address binding, and implement each transition as a separate method (Figure 3).

To de-multiplex a `RegisterRequest` you can define a separate interface for each of the two responsibilities. `Authentication` and `AddressBinding` get a reference to the composite message, but will only see the part relevant to them (Figure 3). You have to make a decision on how to sequence transition invocations. For instance, you can only submit an address binding to the location service, if the registering user agent has been successfully authenticated. You can express such causal dependencies as conditional statements on the control state of component state machines (cf. `register()` method in Figure 3).

For decoupling of responsibilities the component state machines should not call each other and should not know about each other. Still, they have to contribute jointly to the external `RegisterResponse` message. You therefore need another object (`MessageCreator`) that coordinates creation of outgoing messages (cf. Section 9).

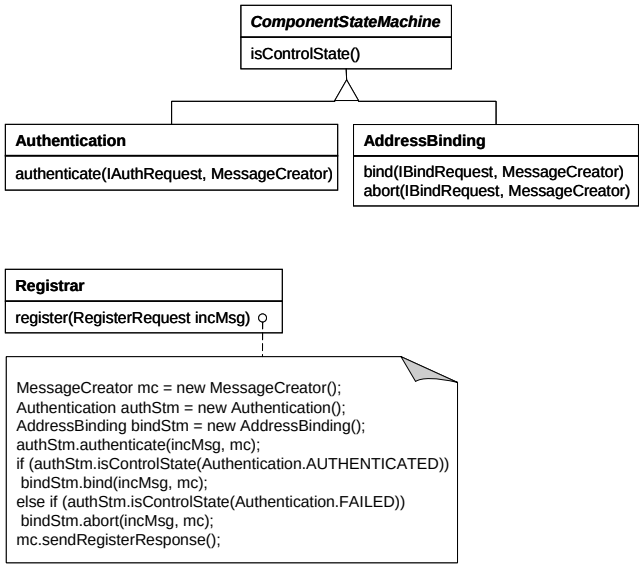


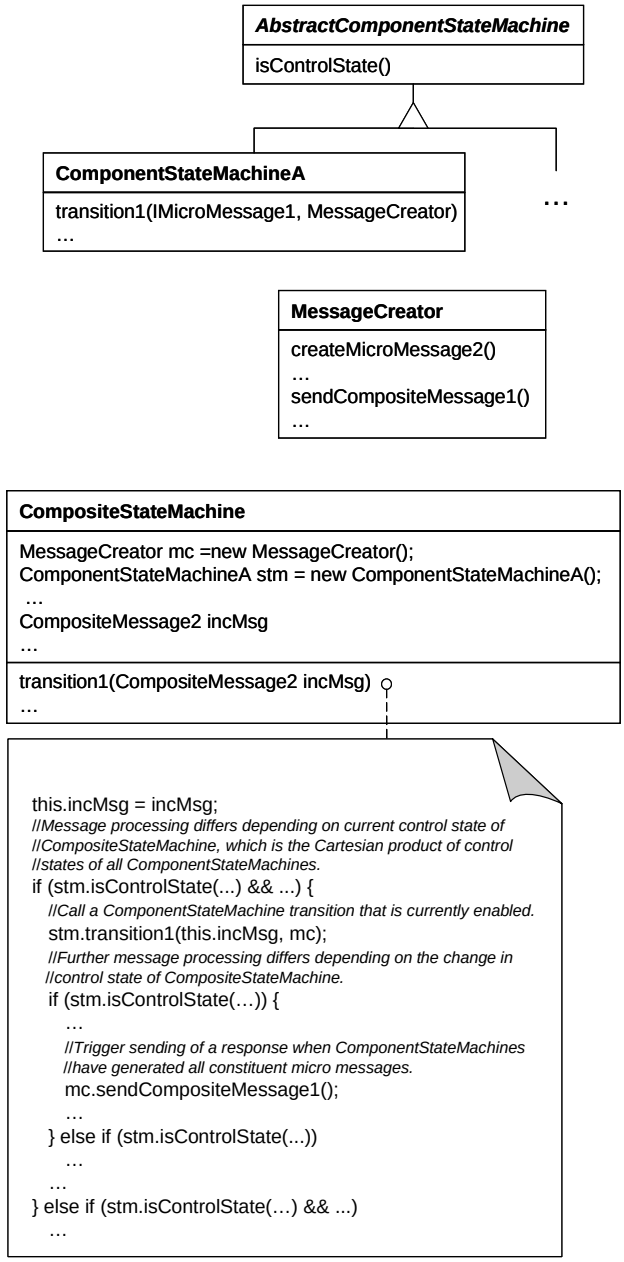
Figure 3. SIP registrar - Selex design.

3. APPLICABILITY

Use the Selex pattern when:

1. An incoming and/or outgoing message of a state machine carries data that belongs to different concerns and you want to decompose the state machine into single responsibilities.
2. Changing externally visible behavior of the state machine is not possible.

4. STRUCTURE



5. PARTICIPANTS

- `AbstractComponentStateMachine` (`ComponentStateMachine`)
  - Implements common behavior of `ComponentStateMachines`. By calling `isControlState()`, transitions of `CompositeStateMachine` can determine in what control state a `ComponentStateMachine` currently is.
- `ComponentStateMachine` (`Authentication`, `AddressBinding`)
  - `ComponentStateMachine` is a state machine implementing a well separated responsibility. For each concern that is multiplexed on incoming and/or outgoing messages of `CompositeStateMachine` there is one `ComponentStateMachine`.
  - An incoming/outgoing message of `ComponentStateMachine` is called *inbound/outbound micro message*. Micro messages are not individually transmitted over the network. Micro messages are bundled into composite messages and then sent over the network.
  - A micro message can be represented by a micro message interface on composite messages, restricting access to only the relevant fields of a composite message.
  - `ComponentStateMachine` creates outbound micro messages only through the `MessageCreator`.
- `CompositeStateMachine` (`Registrar`)
  - `CompositeStateMachine` is a state machine responsible for composing `ComponentStateMachines`.
  - An incoming/outgoing message of `CompositeStateMachine` is called *incoming/outgoing composite message*. Composite messages are sent over the network and consist of one or more micro messages generated/processed by `ComponentStateMachines`.
  - `CompositeStateMachine` has one transition for each incoming composite message, which handles *de-multiplexing* and *sequencing* of the message as well as triggering the transmission of outgoing composite messages over the network.
  - *Message sequencing* means calling transitions of `ComponentStateMachines` in the right order, so that causal dependencies among `ComponentStateMachines` are met.
  - *Message de-multiplexing* means splitting an incoming composite message into its constituent micro messages and passing the micro messages to the right transitions of the right `ComponentStateMachines`. Splitting composite messages into micro messages can be implemented by micro message interfaces on composite messages, so that a `ComponentStateMachine` receives a reference to a composite message, but has only limited access to its fields.
- `MessageCreator`
  - Creates and destroys outgoing composite messages.
  - Provides a point of access to outgoing composite messages for `ComponentStateMachines`.

- Makes sure that `ComponentStateMachines` contribute to the right instances of outgoing composite messages, i.e., it implements multiplexing of outbound micro messages to outgoing composite messages.
- Reduces dependencies among `ComponentStateMachines`. Each `ComponentStateMachine` operates on micro messages (represented by interfaces on composite messages) instead of composite messages. You can change multiplexing of outbound micro messages to composite messages without affecting `ComponentStateMachines` as long as they implement the same micro message interfaces.

6. COLLABORATIONS

- After an incoming composite message of type `CompositeMessage` is received and decoded, processing of the message starts with calling the transition on `CompositeStateMachine` that processes messages of type `CompositeMessage`.
- This transition then calls transitions on `ComponentStateMachines` that are responsible for the concerns multiplexed on the incoming message (de-multiplexing). It must call transitions in the right order, so that causal dependencies are met (sequencing), and it must trigger the sending of outgoing composite messages as soon as `ComponentStateMachines` have generated all constituent outbound micro messages.
- Transitions of `ComponentStateMachines` use the `MessageCreator` to gain access to the right outgoing composite messages for which they have data ready (message multiplexing). Through proper micro message interfaces they should only have access to the fields of composite messages that are relevant to them.

7. CONSEQUENCES

The Selex pattern has the following benefits and liabilities:

1. *`ComponentStateMachines` separate responsibilities*. In distributed systems design, we often try to get by with a minimum set of messages, meaning that a few messages cover many responsibilities. De-multiplexing and sequencing of incoming messages as well as multiplexing to outgoing messages allow reversing the cluttering effects of such optimizations on the internal design. `ComponentStateMachines` can be developed and tested independently since they do not call each other and micro message interfaces can shield them from protocol composition.
2. *`ComponentStateMachines` are potential units of reuse*. SIP uses the authentication protocol from Section 2 not just for `RegisterRequests`. Every SIP request could be authenticated this way. If you apply the Selex pattern, you can reuse the `Authentication ComponentStateMachine` for a completely different `CompositeStateMachine` that is handling other SIP requests. You must make sure, though, that the other SIP request and response implement the expected micro message interfaces. The transition for the other SIP request on the new `CompositeStateMachine` will handle the differences in sequencing and de-multiplexing the request. [3] and [4] report on an industrial project where legacy code was refactored towards a Selex design resulting in 20 `ComponentStateMachines`, which were reused in more than 10 different protocol variants.
3. *Collaborations encapsulate behavior across agent boundaries*. If there are multiple state machines communicating with each other, such as the registrar and the registering user agent from the earlier SIP example, you can apply the Selex pattern to all communicating state machines. In that case, `ComponentStateMachines` can be viewed as various roles that a `CompositeStateMachine` (call it agent in this context) plays during execution of the communication protocol. For SIP registration, we have two agents, the registrar and the registering user agent, where both contain an authentication role as well as a role for address binding. Together, the roles of one responsibility form a collaboration that provides a distinct service to its environment. The collaboration-based view makes it much easier to understand end-to-end behavior of distributed systems, since it promotes separate reasoning about collaborations, collaboration composition, and eventually the entire system behavior. Note, however, that you do not have to apply Selex to all of the agents, if you are not interested in end-to-end behavior.
4. *`CompositeStateMachine` can be automatically generated*. Most of the inherent complexity of a Selex design resides in `CompositeStateMachine`. However, there are tools that allow specifying component state machines, causal dependencies between them, as well as multiplexing of messages. These specifications are amenable to validation and code generation, so that `CompositeStateMachine` can be automatically generated from a declarative specification of message sequencing and multiplexing plus a model of each component state machine.
5. *Supporting new `ComponentStateMachines` is difficult*. When it is necessary to add a new `ComponentStateMachine` to the composite, many transitions of `CompositeStateMachine` might be due for a complete review. The encoded causal dependencies can be arranged in subtle ways and they need to be rechecked, so that new causal dependencies do not break old ones. This is why a code generation approach in combination with validation tools is so powerful in this context.

8. IMPLEMENTATION

Consider the following issues when implementing the Selex pattern:

1. *Alternative implementation of `ComponentStateMachines`*. Previous sections made the assumption that component state machines are simple enough so that you can implement them with a straightforward flag-and-switch approach. You can also use another design, in particular, if a component state machine becomes more complex. Possible alternatives to flag-and-switch are the many other state machine design patterns ([1] gives an overview). However, make sure that complexity of the component state machine does not arise from message multiplexing. In that case, first try to further decompose the state machine with the Selex pattern.
2. *Multiple instances of `CompositeStateMachine`*. So far, we have not considered that multiple instances of `CompositeStateMachine` may run concurrently. In that case, you need

a facility for message correlation, i.e., a way to map an incoming composite message to the right instance of `CompositeStateMachine`. Message correlation will be determined based on certain fields of an incoming message, where different messages might use different fields as input to the mapping function. For some messages the `CompositeStateMachine` must be initially created.

3. *Non-multiplexed incoming and outgoing messages.* Many incoming/outgoing messages of a `CompositeStateMachine` might actually belong to just one concern. You can treat those messages as multiplexed messages containing just one micro message and they will fit right into the Selex design.

4. *Standard implementation of `CompositeStateMachine`.* Code for message sequencing represents most of the complexity of `CompositeStateMachine`, but you can always follow a similar code structure for `CompositeStateMachine` transitions (cf. sample code in Section 4): at the top level, use an if/else-statement differentiating behavior according to the current state of the `CompositeStateMachine`, where composite state is the Cartesian product of control states of all `ComponentStateMachines`. Depending on the current composite state you might want to process an incoming composite message differently. If/else-clauses for the different composite states contain nested conditional statements (if/else or while). At the beginning of each level you call a `ComponentStateMachine` transition that is enabled in the current composite state. The transition call is followed by an if/else statement differentiating behavior according to the possible target control states resulting from the transition execution. The new target control state changes the composite state and may then enable transitions from other `ComponentStateMachines` which might then be called and further change composite state, etc. Note that a `CompositeStateMachine` transition contains only the message sequencing logic, calls to `ComponentStateMachine` transitions, and calls to send methods of `MessageCreator`. `MessageCreator` is called when an outgoing composite message is ready to be sent, i.e., when `ComponentStateMachines` have generated all constituent micro messages.

5. *Alternative implementation of `CompositeStateMachine`.* The logic for composing `ComponentStateMachines` is hidden within `CompositeStateMachine`, which is itself another state machine. The problem with the standard implementation of `CompositeStateMachine` transitions is potential duplication of code. It is possible that the outer conditional logic that differentiates behavior according to composite state at message arrival is the same for most of the incoming messages. It is also possible that code segments within the outer if/else-clauses are duplicated. You can resolve code duplication by refactoring `CompositeStateMachine` towards the State pattern [2] or one of its derivatives.

9. SAMPLE CODE

We illustrate the missing pieces for the SIP registrar example from Section 2. SIP registration is a stateless protocol, meaning that the registrar does not keep any protocol state once a `RegisterRequest` has been processed. As a consequence, we do not bother with multiple instances of `CompositeStateMachine` (Section 8).

For this example a straightforward flag-and-switch approach for implementing `ComponentStateMachines` will suffice. Symbolic constants denote different control states, and the current control state is held in a protected field of `ComponentStateMachine`.

```
public abstract class ComponentStateMachine {
   protected static final int AUTH_OFFSET = 0;
   protected static final int BIND_OFFSET = 10;
   protected int controlState;

   public boolean isControlState(int controlState) {
      return (this.controlState == controlState);
   }
}


public class Authentication extends ComponentStateMachine
{
   public static final int IDLE = AUTH_OFFSET + 0;
   public static final int AUTHENTICATED=AUTH_OFFSET + 1;
   public static final int FAILED=AUTH_OFFSET + 2;

   public Authentication(){
      controlState = IDLE;
   }

   public void authenticate(IAuthRequest inboundMicroMsg,
                            MessageCreator msgCreator) {
         IAuthResponse response =
                        msgCreator.createAuthResponse();
        //1. Use data from inboundMicroMsg to do
        //   authentication.
        //2. Set proper values in response.
        //3. Depending on authentication result
        //   switch to new target control state.
   }
}
```

SIP registration uses only one outgoing composite message namely `RegisterResponse`. It is guaranteed that there exists at most one instance of `RegisterResponse` at all times, so that `MessageCreator` can manage it much like a Singleton [2]. The constructor of `RegisterResponse` has package scope to prevent `ComponentStateMachines` from creating instances directly. An instance will be abandoned after it has been sent. Note that the `createX()` methods encapsulate the downcast for `ComponentStateMachines`.

```
public class MessageCreator {
   private RegisterResponse _instance = null;

   private RegisterResponse instance() {
      if (_instance == null)
         _instance = new RegisterResponse();
      return _instance;
   }

   public IAuthResponse createAuthResponse() {
      return instance();
   }
```

```
   public IBindResponse createBindResponse() {
      return instance();
   }

   public void sendRegisterResponse(){
      assert (_instance.isComplete());
      //encode and send message
      _instance = null;
   }
}
```

10. KNOWN USES

Multiple projects at Avaya Labs have used the Selex pattern. Among them are registration, gateway, and conferencing applications. Experience reports on one project are publicly available [3], [4].

11. RELATED PATTERNS

`ComponentStateMachines` as well as `CompositeStateMachine` can be implemented according to the State pattern [2] or other state machine patterns ([1] gives an overview).

12. ACKNOWLEDGMENTS

The authors would like to thank their PLoP shepherd, Paul Adamczyk, for his suggestions during the revision of this paper and for helping to improve the presentation.

13. REFERENCES

[1] Adamczyk, P. 2003 Anthology of Finite State Machine Design Patterns. PLoP

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995 Design Patterns. Addison-Wesley

[3] Geppert, B. and Roessler, F. 2004 Effects of Refactoring Legacy Protocol Implementations: A Case Study. Metrics Symposium

[4] Geppert, B., Mockus, A., and Roessler, F. 2005 Refactoring for Changeability: A way to go? Metrics Symposium

[5] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. 2002 SIP: Session Initiation Protocol. RFC 3261