

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

Frank Roessler, Birgit Geppert  
Avaya Labs, Basking Ridge, NJ, USA  
{roessler, 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.

## 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.

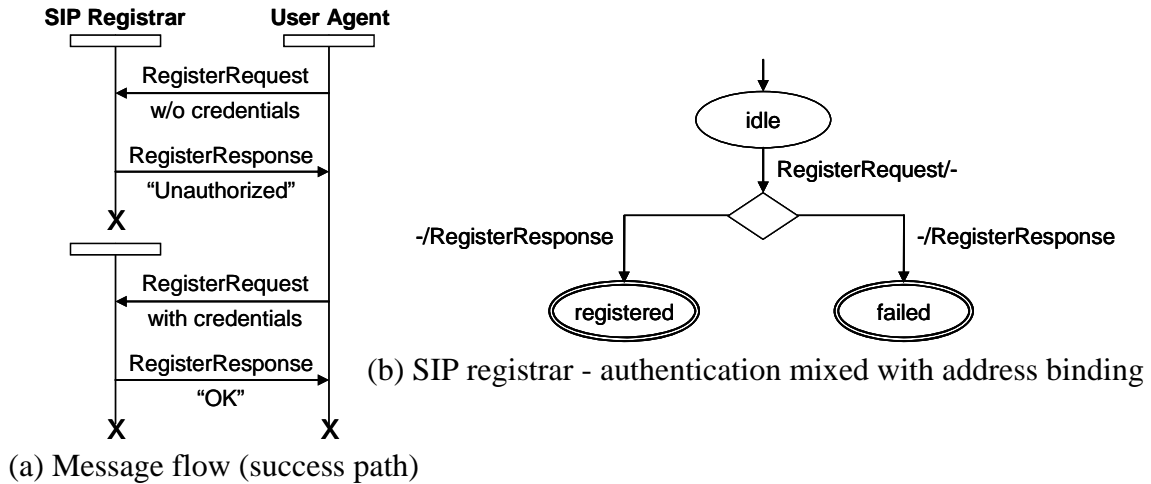
### What is message multiplexing?

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 more responsibilities (by extending RegisterRequest) this approach would not scale very well.

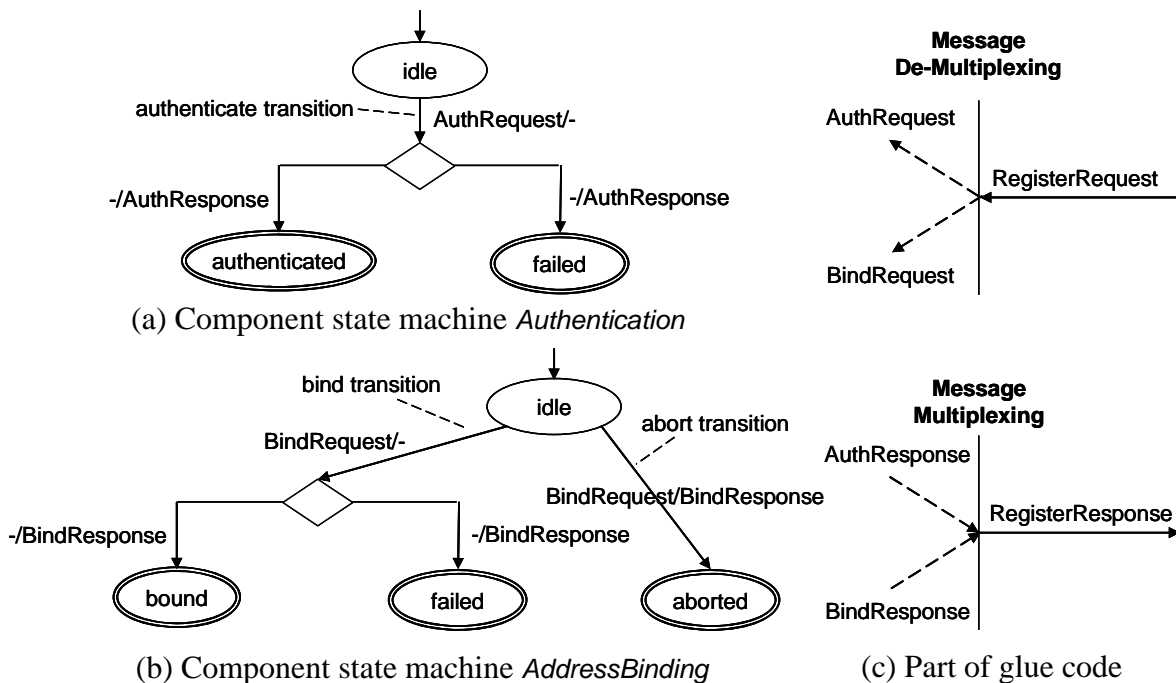
Rather than following the approach of Figure 1(b), you could first de-multiplex RegisterRequest into an authentication and address binding part (Figure 2(c)) and then feed these micro messages into separate state machines - one responsible for authentication (Figure 2(a)) and one responsible for address 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)).



**Figure 1: SIP registration**

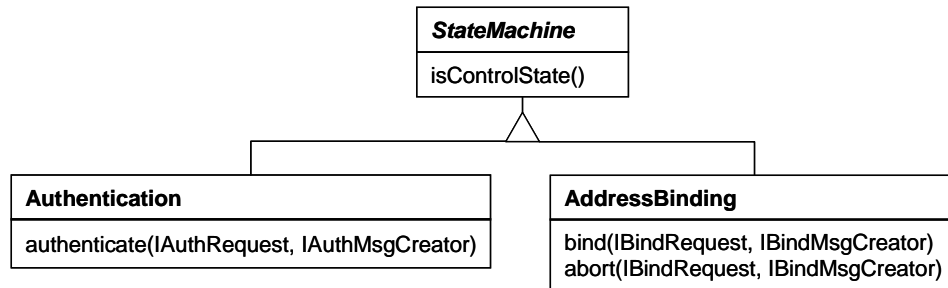
The component state machines are simple, so a straightforward flag-and-switch approach will suffice for their implementation. You need one class for authentication, another one for address binding, and implement each transition as a separate method (Figure 3(a)).

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(b)). 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. demultiplex() method in Figure 3(b)).

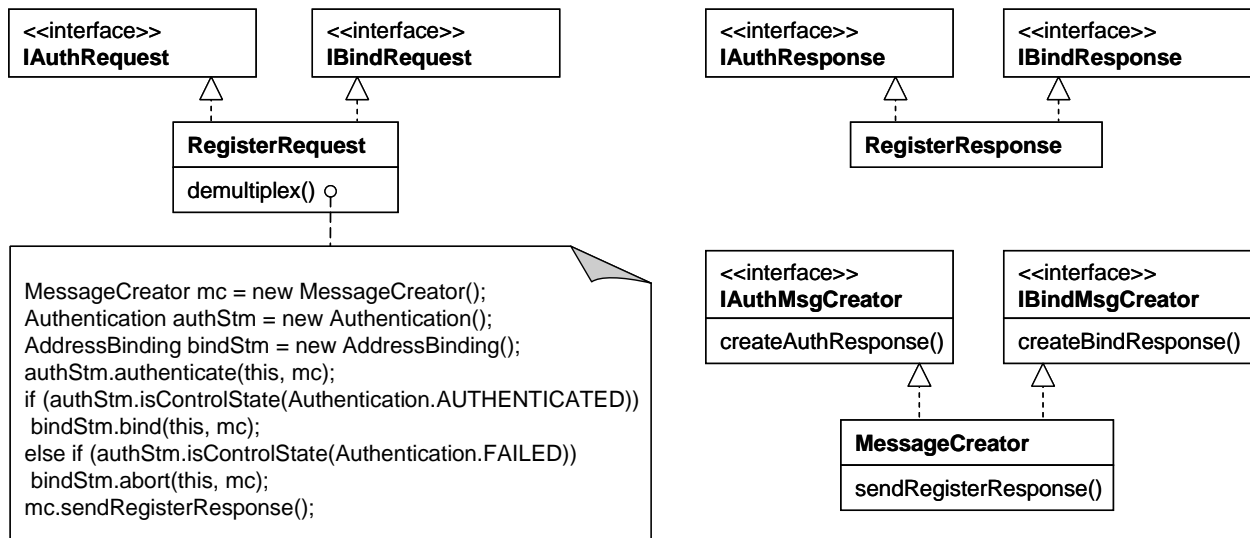


**Figure 2: SIP registrar - authentication separated from address binding**

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 that coordinates creation of outgoing messages (MessageCreator, Figure 3(c)). The MessageCreator returns references to composite messages, even though component state machines do only see the part relevant to them (Figure 3(c)).



(a) Component state machines



(b) Incoming message (de-multiplexing and sequencing)      (c) Outgoing message (multiplexing)

**Figure 3: SIP registrar – Selex design**

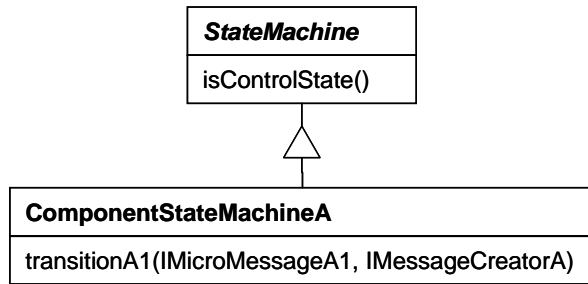
### 3. Applicability

Use the Selex pattern when:

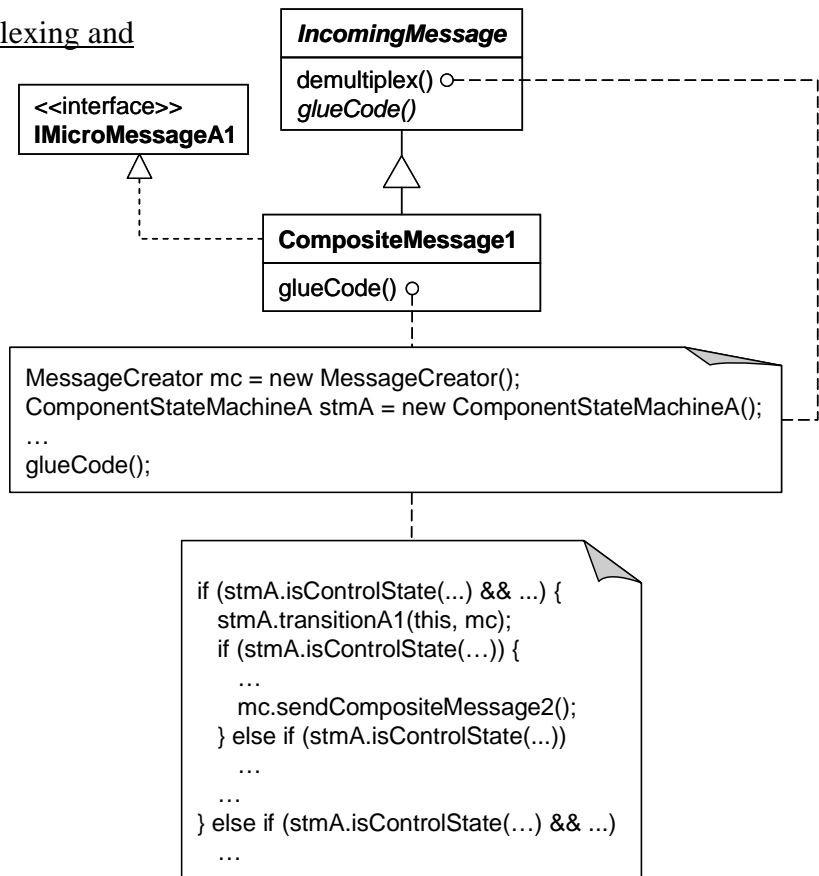
1. An incoming and/or outgoing message of a state machine carries data that belongs to different concerns and you want to decompose the state machine into single responsibilities.
2. Changing externally visible behavior of the state machine is not possible.

## 4. Structure

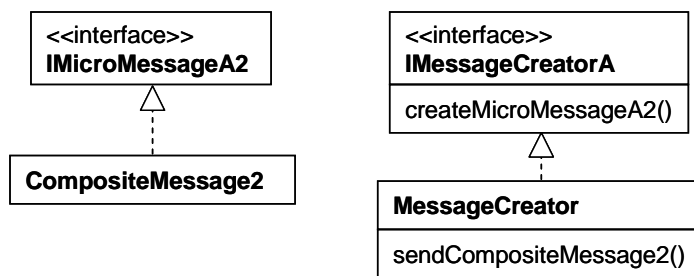
Component state machines:



Incoming messages (de-multiplexing and sequencing):



Outgoing messages (multiplexing):



## 5. Participants

### Component state machines:

- StateMachine (StateMachine)
  - Implements common behavior of ComponentStateMachines. By calling `isControlState()` the `glueCode()` methods of incoming CompositeMessages can determine in what control state a ComponentStateMachine currently is.
- ComponentStateMachine (Authentication, AddressBinding)
  - Implements a well separated responsibility. For each concern that is multiplexed on incoming and outgoing messages of the composite state machine there should be one ComponentStateMachine.
  - Defines its own transitions, inbound micro messages, and outbound micro messages. Micro messages are not externally visible.
  - Each micro message is represented as an interface `IMicroMessage` on an externally visible message.
  - Implements each transition by its own method.
  - Creates outbound micro messages only through the `MessageCreator`.
  - Prepares content of outbound micro messages, but does not send them since micro messages are not externally visible. `GlueCode()` methods do send outgoing CompositeMessages - which are externally visible - once all constituent micro messages are complete.

### Incoming messages (de-multiplexing and sequencing):

- IncomingMessage
  - Implements common behavior of incoming CompositeMessages. The Template Method [2] `demultiplex()` instantiates a `MessageCreator` and the ComponentStateMachines. It defers glue-code details (de-multiplexing, sequencing, and sending of multiplexed outgoing messages) to subclasses.
- CompositeMessage (RegisterRequest)
  - Represents incoming (or outgoing) messages of the composite state machine, i.e., externally visible messages.
  - For each concern that is multiplexed on the CompositeMessage, it implements a different interface `IMicroMessage` that allows access to the data relevant to the concern.
  - CompositeMessages of type `IncomingMessage` are responsible for de-multiplexing and sequencing the incoming message as well as triggering the sending of multiplexed outgoing messages that are created in response to the incoming message. All of that is implemented by a `glueCode()` method.

Message sequencing means calling transitions of ComponentStateMachines in the right order, so that causal dependencies among ComponentStateMachines are met. Code for message sequencing represents most of the inherent complexity of the composite state machine, but you can always follow the same structure: at the top level, use an if/else-statement differentiating behavior according to the current state of the composite state machine (composite state is the Cartesian product of

control states of the ComponentStateMachines). Depending on the current composite state you might want to process an incoming message differently. If- and 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 of the transition. The new target control state changes the composite state and may then enable transitions from other ComponentStateMachines which can be called and further change composite state, etc.

You implement message de-multiplexing by adequate interfaces IMicroMessage on an IncomingMessage, so that your glue code passes a reference to the entire composite message to a transition, but it has only access to the parts relevant to it.

When all ComponentStateMachines that contribute micro messages to an outgoing CompositeMessage are in a control state where they have contributed their part, your glueCode() method must call a send method on the MessageCreator. Calls to send methods fit right into the code structure for message sequencing.

- IMicroMessage (IAuthRequest, IBindRequest)
  - Declares an interface on a CompositeMessage that is specific to one ComponentStateMachine.
  - Allows access to only those message fields that are relevant to the ComponentStateMachine.
  - Transitions of ComponentStateMachines only take IMicroMessages as input parameters (not CompositeMessages).
  - Reduces dependencies among ComponentStateMachines. Each ComponentStateMachine operates on micro messages (represented by IMicroMessages) instead of externally visible messages. You can change external messages without affecting ComponentStateMachines as long as they implement the same IMicroMessage interfaces.

#### Outgoing messages (multiplexing):

- CompositeMessage (RegisterResponse)
  - cf. previous subsection.
- IMicroMessage (IAuthResponse, IBindResponse)
  - cf. previous subsection.
- MessageCreator (MessageCreator)
  - Creates and destroys outgoing CompositeMessages.
  - Provides a point of access to outgoing CompositeMessages for ComponentStateMachines.
  - Makes sure that ComponentStateMachines contribute to the correct instances of outgoing CompositeMessages, i.e., it implements multiplexing of outbound micro messages to outgoing CompositeMessages.
  - Reduces dependencies among ComponentStateMachines. Each ComponentStateMachine operates on micro messages (represented by IMicroMessages) instead of externally visible messages. You can change

multiplexing of outbound micro messages to external messages without affecting `ComponentStateMachines` as long as they implement the same `IMicroMessage` interfaces.

- `IMessageCreator` (`IAuthMsgCreator`, `IBindMsgCreator`)
  - Declares an interface on the `MessageCreator` that is specific to one `ComponentStateMachine`.
  - Allows creating only those outbound `IMicroMessages` that are relevant to the `ComponentStateMachine`.

## 6. Collaborations

- After an incoming `CompositeMessage` is received and decoded, processing of the message starts with calling its `demultiplex()` and `glueCode()` method.
- `demultiplex()` instantiates `ComponentStateMachines` and a `MessageCreator` and calls `glueCode()`.
- `GlueCode()` orchestrates `ComponentStateMachines` by calling the transitions that participate in processing the incoming `CompositeMessage` (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 `CompositeMessages` as soon as `ComponentStateMachines` have provided all constituent parts.
- Transitions of `ComponentStateMachines` use the `MessageCreator` to gain access to the right outgoing `CompositeMessages` for which they have data ready (message multiplexing). Through `IMicroMessage` interfaces they do only see the part of `CompositeMessages` 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 when it comes to the internal structure of components. `ComponentStateMachines` can be developed and tested independently since they do not call each other and `IMicroMessage` interfaces shield them from composition mechanisms.
2. *ComponentStateMachines are potential units of reuse.* SIP uses the authentication protocol from the Motivation section 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 composite state machine that is handling other SIP requests. You must make sure, though, that the other SIP request and corresponding response also implement the `IAuthRequest` and `IAuthResponse` interfaces, respectively. The `glueCode()` method of the other SIP request will handle the differences in sequencing and de-multiplexing the request. [3, 4] report on an industrial project where legacy code was refactored towards a Selex design resulting in 20 `ComponentStateMachines`, which were reused in >10 different protocol variants. Interestingly enough, that case study also showed that the code ran 10% faster after refactoring.

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 composite state machine (we 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 don't have to apply Selex to all of the agents, if you are not interested in end-to-end behavior.
4. *GlueCode() methods can be generated from automata-based protocol specifications.* Most of the inherent complexity of composite state machines resides in the `glueCode()` methods. However, there are tools that allow specifying component state machines, causal dependencies between them, as well as multiplexed messages. These specifications are amenable to validation and code generation, so that `glueCode()` methods do not need to be implemented manually.
5. *Supporting new ComponentStateMachines is difficult.* When it is necessary to add a new `ComponentStateMachine` to the composite, many `glueCode()` methods 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 component state machines.* 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 the composite state machine.* So far, we have not considered that multiple instances of the composite state machine may run concurrently. In that case, you need an additional class `Context` and must adapt `demultiplex()` and `glueCode()` (Figure 4). Instead of instantiating a `MessageCreator` and the `ComponentStateMachines`, `demultiplex()` implements behavior for context switching.

New participant: Context class

- Holds all context information for one instance of the composite state machine. Context information is a) incoming `CompositeMessages` that have not been

completely processed yet, b) ComponentStateMachines, and c) the MessageCreator, which may hold partially assembled outgoing CompositeMessages.

- Implements a static method find() for correlating an incoming CompositeMessage with an existing context. For certain incoming messages, find() must return a new context.
- Implements a static method registerId() to enable dynamic correlation of incoming CompositeMessages (using different context identifiers) with the same context.
- Implements a static method remove() to remove context objects.

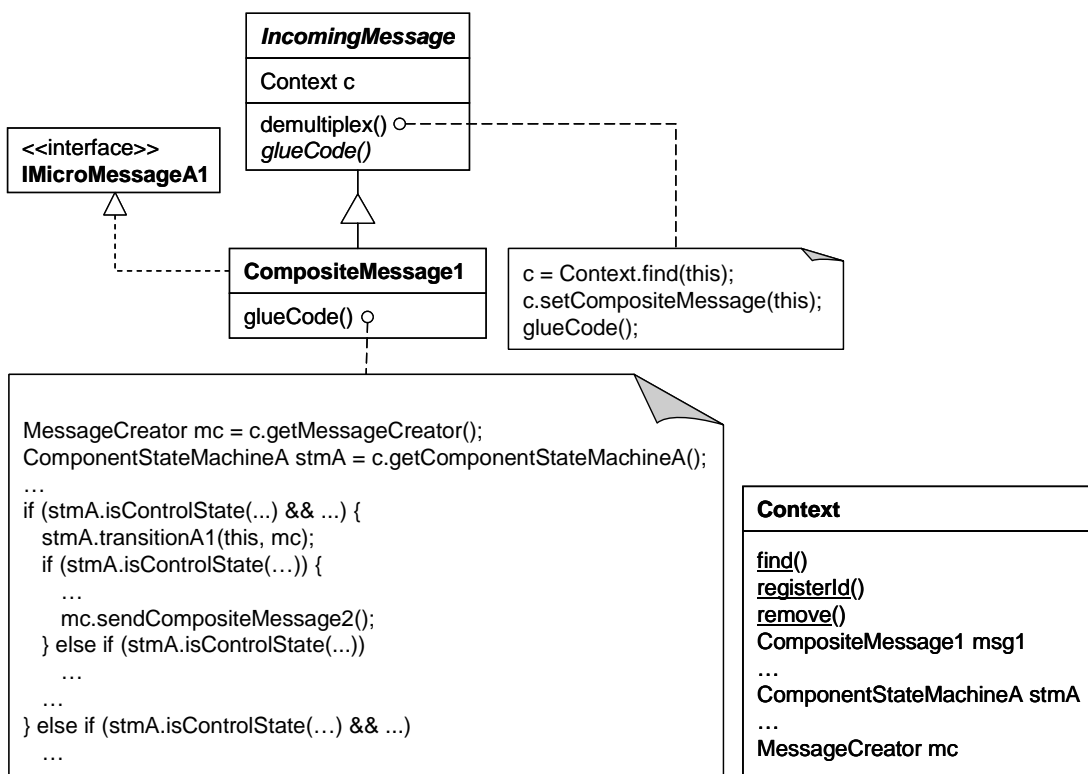


Figure 4: Selex pattern with context switching (extensions)

3. *Non-multiplexed incoming and outgoing messages.* Many externally visible messages of a composite state machine 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. *Alternative implementation of demultiplex() and glueCode() methods.* The logic for composing ComponentStateMachines is hidden within the demultiplex() and glueCode() methods of incoming CompositeMessages. At a closer look, it turns out that the compositional logic is nothing else but another state machine (in addition to ComponentStateMachines). Its control states are the Cartesian product of the control states of ComponentStateMachines and its transitions are responsible for demultiplexing and sequencing incoming CompositeMessages and deciding when to

send outgoing CompositeMessages. This is consistent with the code structure of glueCode() methods as described in Section 5. The problem with glueCode() methods is potential duplication of code. It is possible that the outer conditional logic that differentiates behavior according to composite state is the same for most of the incoming CompositeMessages. It is also possible that code segments within the outmost if/else-clauses are duplicated. You can resolve code duplication by refactoring compositional logic towards the State pattern [2] or one of its derivatives. [1] gives an overview of other state machine patterns.

## 9. Sample Code

We illustrate an implementation of the Selex pattern for the SIP registrar (cf. Motivation section). 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 the Context class and context switching (cf. Implementation section).

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 StateMachine.

```
public abstract class StateMachine {
    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 StateMachine {
    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,
        IAuthMsgCreator 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 CompositeMessage 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 implements IAuthMsgCreator, IBindMsgCreator
{
    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;
    }
}
```

Since there is only one incoming CompositeMessage there is no need for the abstract superclass IncomingMessage. We therefore push down demultiplex() to RegisterRequest and inline glueCode(). We can remove IncomingMessage then.

```
public class RegisterRequest implements IAuthRequest, IBindRequest {

    public void demultiplex() {
        MessageCreator msgCreator = new MessageCreator();
        Authentication authStm = new Authentication();
        AddressBinding bindStm = new AddressBinding();
        //start inline: glueCode()
        authStm.authenticate(this, msgCreator);
        if (authStm.isControlState(Authentication.AUTHENTICATED))
            bindStm.bind(this, msgCreator);
        else if (authStm.isControlState(Authentication.FAILED))
            bindStm.abort(this, msgCreator);
        msgCreator.sendRegisterResponse();
        //end inline: glueCode()
    }
}
```

## 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 glue code for their composition can be implemented according to the State pattern [2] or other state machine patterns ([1] gives an overview). Demultiplex() methods are Template Methods [2].

## 12. Acknowledgements

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.

## References

1. P. Adamczyk, *Anthology of the Finite State Machine Design Patterns*, PLoP 2003
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, 1995
3. B. Geppert, F. Roessler, *Effects of Refactoring Legacy Protocol Implementations: A Case Study*, Metrics 2004
4. B. Geppert, A. Mockus, F. Roessler, *Refactoring for Changeability: A way to go?*, Metrics 2005
5. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session Initiation Protocol*, RFC 3261, June 2002