Stateful Event Selector

(v1.0)

Taras Shymbra

Intent

When serially processing an event sequence decouple event selection from event handling. Stateful Event Selector uses a queue, state and a separate thread to transform incoming event flow and forward it to an event handler which may therefore remain stateless and passive.

Motivation

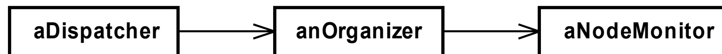
Consider a communication application that monitors a remote network node by periodically collecting status data from it. Data is exchanged in a simple requests-response manner using a packet-based protocol. The collaboration is initiated by a request timer - when it expires the application must send a request packet. After the request packet is transmitted on the network a notification is generated. An acknowledgement should arrive shortly afterwards followed by a response packet. When the request is negatively acknowledged the application repeats the request until response finally arrives.

With this basic setup the events that occur in the system are RequestTimer, Request, RequestSent, Ack, Nack and Response. Most of events arrive from two sources: the network and the timer mechanism. The event Request is an internal event generated within the system. Most common event flow is RequestTimer -> Request -> RequestSent -> Ack -> Response. Having received a Nack the system must be able to handle following flow of events RequestTimer -> Request -> RequestSent -> Nack -> Request.

A straightforward implementation of the application would utilize single thread executing some kind of synchronous event demultiplexer which blocks awaiting an event from either event source. Events including request timer timeouts, arrived packets and send notifications are dispatched to the NodeMonitor object which acts as a core event handler. Its main responsibility is to processes status data extracted from the response packets. The NodeMonitor has to maintain state of the request sending to enforce the protocol rules. Its code runs in the dispatcher thread.

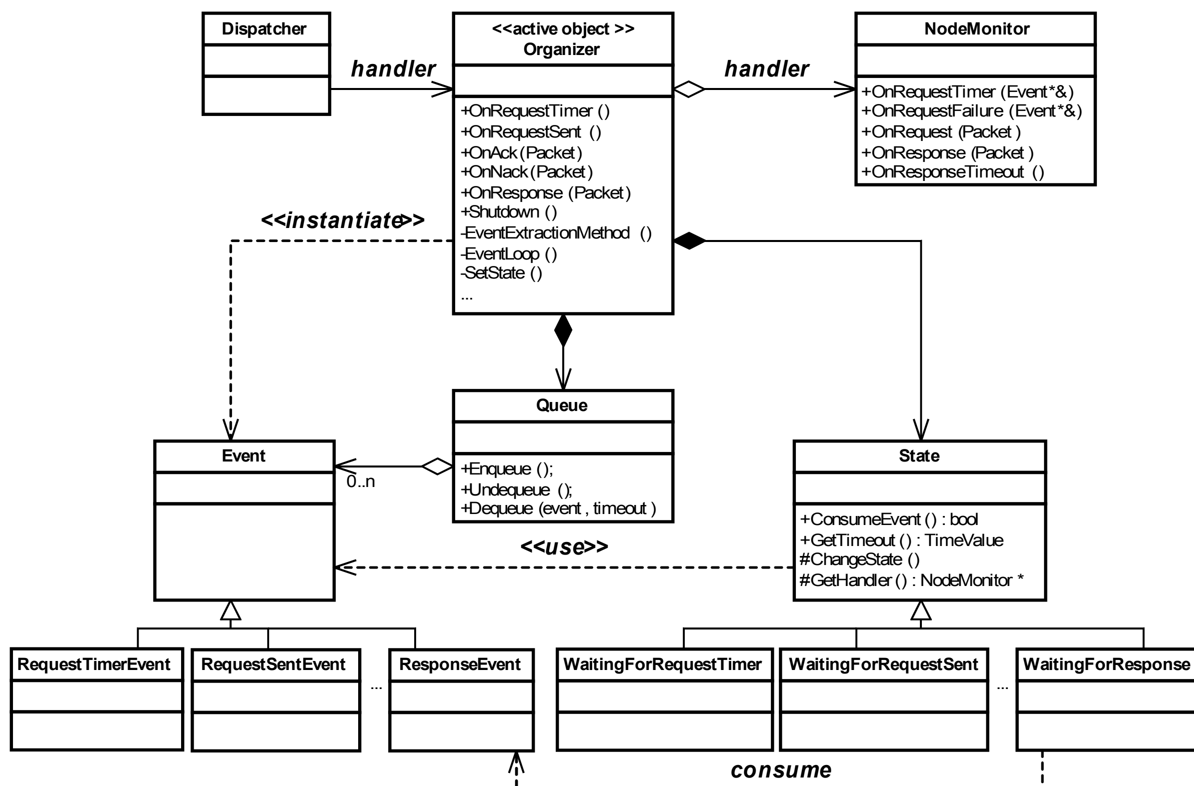
Negative acknowledgement provides a way to signal errors detected by the remote node. Its failure, request preparation delays or problems with the network link must be recognized by the application. Situations like missed acknowledgement or late arrival of a response after the dispatcher has already issued next request event lead to explosion of code in the NodeMonitor as developers have to make sure absence of packets is detected and all possible sequences of events are covered.

We can simplify the implementation by separating the actual response data processing from all low level infrastructure related aspects including request delivery status tracking, detection of communication errors, dealing with unexpected events etc. We therefore put an object between the Dispatcher and the NodeMonitor objects to take care of those aspects as shown in the following object diagram:



This object, called Organizer, intercepts the event flow from the dispatcher and puts events into its queue. A separate thread then consumes them by dequeueing an event at a time and deciding what to do with the event according with the current selector state. From other perspective, such intermediary object can be viewed as a kind of selector which performs selection from incoming event flow to yield the next event which will be sent to NodeMonitor.

A RequestTimer event can be immediately forwarded to the NodeMonitor if the previous request has already seen its response packet. This timer event can as well be postponed for later processing if the selector is in the "waiting for response" state. But when the response does not appear within the defined by the state time interval, an event ResponseTimeout is raised by the organizer thread. Similarly, when request delivery error is detected via acknowledgement mechanism, an event RequestFailure is generated. This means the organizer not only forwards original events but also generates secondary ones to notify the NodeMonitor about detected conditions. The organizer can also discard events. For example, a response that has arrived too late must be discarded - the NodeMonitor object had already received ResponseTimeout and resent the previous request again.



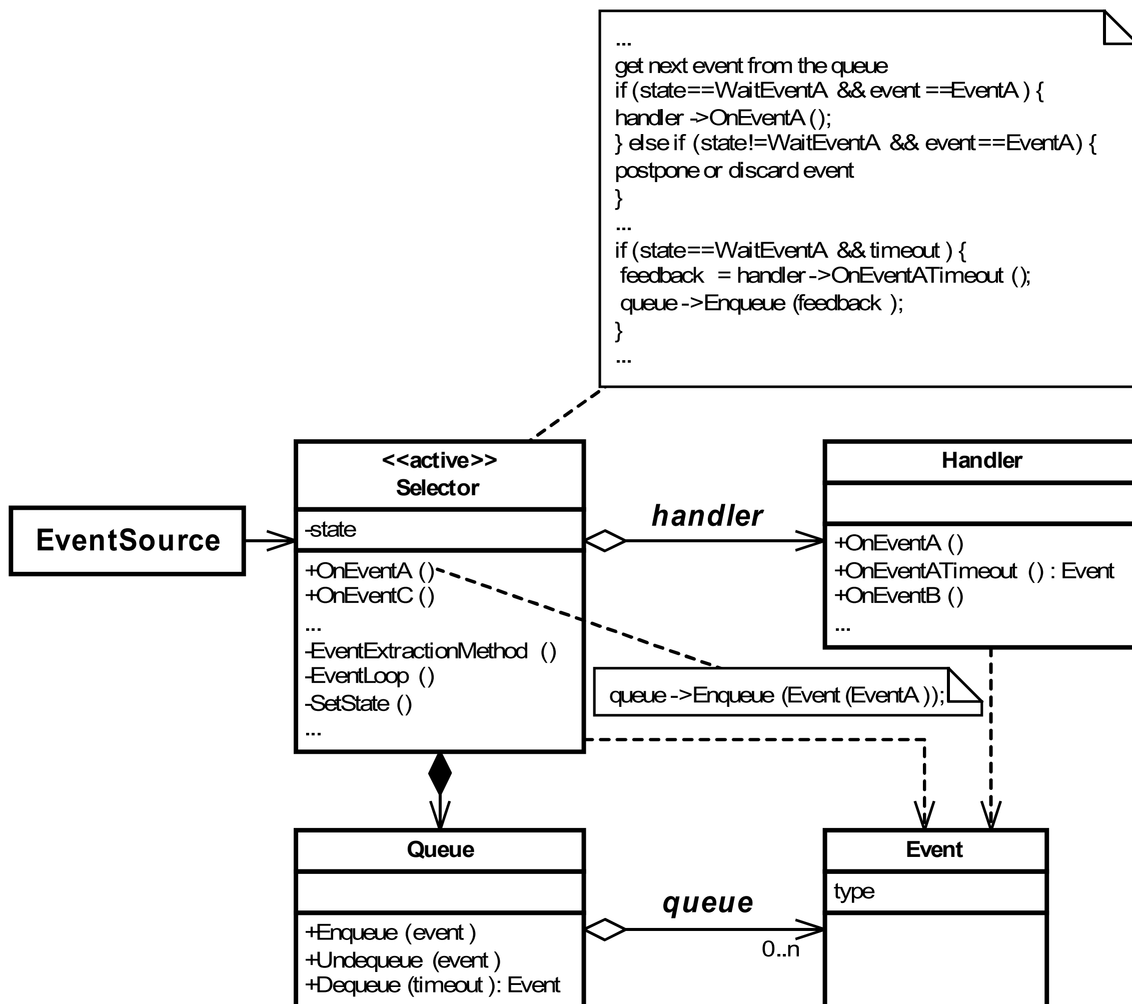
This diagram illustrates the components of the stateful event selector. The state that used to reside in the NodeMonitor class has migrated to the Organizer class. Receiving an enriched and filtered event flow, the NodeMonitor can event better perform its core task which is processing of response packets.

Applicability

Use Stateful Event Selector when:

- You can not control how original event flow is generated (e.g. events arrive from a remote source or out-of-box event dispatcher) but you need additional manipulations on it like events reordering, filtering, delaying, injecting of new events or other types of event sequence altering. You also want the added behaviour to be transparent to event handlers.
- You need to enforce immediate reaction by handlers when certain event does not arrive within a given time interval. Stateful Event Selector actively detects event timeouts and notifies handlers about them.
- Some dispatched events may be unexpected at a particular moment so that an event handler can't handle them immediately. But it is impossible to return it back to the event source. You want to avoid maintaining storage for deferred events within each event handler.

Structure



Participants

- **EventSource** (Dispatcher)

    - generates flow of original events to be handled
    - works directly only with a Selector object
- **Selector** (Organizer)
    - declares event handling interface to accepts the original event flow
    - stores events in the queue
    - runs in its own thread of control selecting events from the queue
    - may generate secondary events which, for example, signalize selection errors (e.g. OnEventATimeout())
    - maintains state to determine which event should be selected next. The state also defines selection constraints
    - forwards events of interest to Handler object whose reference it maintains
- **Event**
    - represents either original or secondary event
    - can be put onto the Queue and reside there until Selector dequeues it
    - may serve as basis class for event subtypes
- **Handler** (NodeMonitor)
    - declares another event handling interface that accepts events from Selector
    - may use the interface to return feedback events back to Selector (e.g. operation OnRequestATimeout() returns an event object)
- **Queue**
    - temporarily stores events processed by the Selector object

Collaborations

- A Selector object receives events from an EventSource and stores them in a Queue object.
- The Selector processes events from the Queue in a separate thread. An event can be consumed in three ways: forwarded, postponed or discarded. Selector’s state machine defines event selection and consuming.
- A Handler receives transformed event flow from the Selector.

Consequences

Stateful Event Selector introduces a level of indirection between event sources and event handlers. It following benefits and liabilities:

1. *It enhances application concurrency.* Having separate thread of control selector adds layer of the asynchronous events processing. The Stateful Event Selector has many of the same consequences that Active Object pattern [ACTOBJ]. It allows event dispatching thread and selector thread to run simultaneously. When events arrive from different threads it allows serialisation of the event flow using a synchronized queue and guarantees that only selector thread executes code which controls states transitions and calls the handler. Hence the handler is not required to be thread-safe.

2. *Value added event queueing*. Stateful Event Selector effectively combines sequential event reading with event timeout detection using a bounded queue. Selector thread may notify handlers about timeouts immediately. Hence, handlers don't need to register within dispatcher for timeout events or involve timer mechanisms from underlying OS. This pattern helps build timed state machines in which an event must occur before timeout.

Waiting for events on a bounded queue enables effective loading of the selector thread. When selector is in a state of waiting for a particular event it may process another events which arrive to the queue in the meantime. Since selector thread immediately responds to any new event in the queue, it can be easily interrupted for example by a shutdown event.

3. *Simplified implementation of complex event driven state machines*. Normally, a state machine should directly transform an incoming event into action or raise an error condition. In such state machine all possible event-to-state combinations must be coded upfront. In Stateful Event Selector an event is first placed into the Queue where it can reside indefinitely long. The selector attempts to present the event to its internal state machine which may decide not handle it at the moment. During a next attempt the state machine may be in appropriate configuration to accept and handle the event. State machine gains control when to respond to an event and must not do it immediately. Therefore, you can start with simple state machine implementation that ignores unsuitable events by default.

4. *It promotes reuse*. Different handler implementations can be used with the same selector and make reuse of its state machine - just configure the selector with different subclasses of Handler class. The selector can be reused across a system for detection of a primitive event sequence that represents an atomic transaction.

5. *Building layers of abstraction*. Handlers may use selector's state machine as a low level service to implement higher level protocols. In our example, the Organizer class manages single request-response transaction. NodeMonitor may keep additional state to realize complex node monitoring sequences which may include many requests. On the other hand, handler may also act as selector for another handler. Such chain of selectors reflects typical layered architecture.

Implementation

The following issues should be considered when applying the Stateful Event Selector pattern:

1. *Transporting events*. There are two general approaches to transporting events from Dispatcher to Handler via Selector. In the case of Organizer each event is transported via a hard-coded operation invocation by the Dispatcher. This approach, described in the Reactor pattern [POSA2] as the multi-method dispatch interface strategy, has an advantage of allowing the demultiplexing of event types to be performed within an EventSource rather than in Selector. When an operation defined by multi-method interface of the Organizer is invoked the creation and queueing of an Event object is triggered. The application defines an Event class for representing an event and new kinds of events are defined by subclassing. Consider, for instance, the event Response. When an Organizer receives this event via the method OnResponse(Packet) it puts the associated response packet onto the queue by wrapping it with a specific subtype of the Event class. On the other side, Organizer having a dequeued Event object figure out its subtype by performing downcast, extract the wrapped Packet object and then call the method OnResponse(Packet) on its NodeMonitor object. So, despite of clear advantages of type safety and flexibility, this approach imposes additional conversion burden on the Selector. Alternative may avoid polymorphism for representation of different event types entirely.

An alternative approach is to make single event handling function that accepts an event object as a parameter. Selector as well as Handler will have only one event handling method. To ensure conforming interface both classes would descend from a common BaseEventHandler class. This approach provides transparency and flexibility but it also requires conditional statements in the handler method. It also addresses the event queueing issue since you can implement the queue to store Event objects directly without any conversions. Having uniform interface with the Handler the Selector can be transparently integrated into existing application. Decorator design pattern [GoF] can also be used for wrapping Handler object with one or more Selector objects. In this case Selector would act as Decorator, Handler as ConcreteComponent and BaseEventHandler as Component in terms of the decorator pattern participants.

After all, a hybrid solution can be pursued. When one of the event receivers has rich interface while other another might have only one event sink method.

2. *Defining interface of the Handler.* Assuming that we use rich event handling interface for events transport rather than single event sink, the next step is to refine the Handler interface. It may not include an operation for each event issued by the EventSource. Basically you should make the Handler interface as narrow as possible. Only a subset of original events will be of interest to the handler. Additional methods will be required to handle Selector generated events. Having hard-coded interface the Handler protects itself against events it does not care about. When you use single event handling method it is harder to guarantee that the Handler receives events it can handle. You may trap unknown events with a default conditional case which fails by default (e.g. raises an exception) to signalise an event leak in the Selector.

3. *Should selector run in its own thread*? Clearly, to avoid overhead, a selector may borrow the caller thread. In this case, it would an instance of a passive Selector. Its state transitions in the selector are entirely EventSource-driven. Developers have to make sure EventSource does not spend too much time executing Selector methods.

   The passiveness of the selector introduces indeterminacy to duration of event processing in any of the Selector methods. For example, during the call to OnAck() the Organizer may need to perform additional bookkeeping depending on the state like checking the queue and processing pending RequestTimer events.

   A separate thread of control makes an active Selector. It ensures that caller thread spends a well determined period of time executing any of the selector methods - it has only to enqueue an event. Moreover, an additional active entity in the system has an advantage of reusing it for asynchronous tasks. NodeMonitor can use the Selector's thread for preparation and sending of request packets.

4. *Connecting selectors and handlers*. There can be either one-to-one or one-to-many relationship between selectors and handlers. Basically, a selector has only one handler object and calls it directly. The simplest implementation initialises the selector object by simply passing a reference to a handler via selector’s constructor. The pattern does not define whether this handler can be registered with other selectors. When handler object is shared among selectors a smart pointer with reference counting can be utilised to keep track of selector objects which are using the handler and delete it automatically.

Reference to an event handler object can be passed to a selector object at a later moment when events already arrive. Since they are enqueued they are not lost. Handlers may be also changed dynamically but extreme care must be taken to ensure proper synchronization during the object switch. In extreme case the selector may work without handler simply discarding all incoming events.

5. *Feedback Events from Handler*. Normally events flow in one direction i.e. from Dispatcher to Selector and then to Handler. The pattern does not specify the event sources for Dispatcher. Handler itself can act as an event source adding events to the flow. For example, an event after having been forwarded to a Hander object can trigger another event which may be added to the original flow and can eventually return back to the Hander. Since, in this way handlers can respond back to selectors, we refer to such event cascades as feedback events.

   Most effective way of adding feedback events to the original event flow is that of accessing the Selector directly. In this case, a handler needs a reference to a selector object. However, to avoid cyclic dependencies, and make handlers completely independent from selectors, handler-generated events can be simply pulled via Handler's event handling methods. For instance, such an event can be assigned to an out-parameter. Selector enqueues the received feedback event and treats it as any other event from the original event flow.

   Feedback events introduce a wide range of interesting two-way event communication scenarios. Along with Dispatcher and Selector, the Handler becomes just another source of events and can contribute to the event flow to drive the selector's state machine. For instance, an event that needs great deal of time to be processed can be deferred by splitting it into many events each representing a single processing step. When Handler receives such an expensive event it processes it partly and generates a feedback event to finish the rest at a more convenient time. Fragmenting of lengthy event processing allows other events to "sneak in" and hence they can be handled in a more timely way.

   As an example, consider the signature of the OnRequestTimer() operation of NodeMonitor. It handles RequestTimer event in two steps: first it constructs the request packet (assume it is a lengthy process) and responds by generating Request event. This feedback event object of type RequestEvent is passed to Organizer and immediately put onto the queue. Later this RequestEvent, will return back to NodeMonitor.

6. *Queue*. The Queue is a Selector's internal data structure used to store events from the EventSource. Either programming language features or off-the-shelf library components can be used for its implementation.

   Basically, it should be a simple container with "first in first out" (FIFO) ordering that allows adding of incoming events to its back and removing of the events from the front. The minimal interface to Queue consists of operations Enqueue and Dequeue. The Dequeue method should provide the bounded-buffer functionality to block the Selector for some timeout amount of time when it tries to remove event objects from an empty Queue. When Selector is active it is also reasonable to charge the Queue with synchronisation issues since to provide thread-safe interface for caller and

Selector threads. In general, the Queue would be very similar to the activation queue described in the Active Object pattern [ACTOBJ].

More complex applications would impose additional requirements on the Queue such as support of various ordering and dequeueing strategies, ability of iteration through its elements or ability to undo the Dequeue operation i.e. return the element back to its previous position.

7. *Event selection*. Active Selector implements an event extraction method that blocks on the Queue waiting for incoming events. As soon as an event or timeout occurs Selector's thread gets notified, unblocks and continues execution of the extraction method. It uses with the current state what do with the occurred event. According to the state, the event can be either expected or unexpected. If the event is expected it is immediately dequeued and consumed and may cause state re-evaluation and transition.

   Basically, we have two cases when the event may be unexpected – either it is some late event and can be discarded or it is usual event (e.g. RequestTimer) that just arrived at the wrong moment and can be postponed. The Selector must accumulate such events until the expected event finally arrives. Afterwards the accumulated events must be processed in the same order as they arrived.

   Here a very important issue must be discussed with regard to events accumulation. Should an unexpected event be dequeued or should it be left in the queue? If we use basic bounded Queue, in which case Dequeue operation unblocks when it contains at least one element, we must extract this element to avoid endless looping of the event extraction method. Once dequeued, such pending event must be stored somewhere. We can simply store the pending event on the stack of the Selector's extraction method and go into recursion. The extraction method calls itself recursively until expected event is extracted from the queue. During unwinding, all pending events return back to the queue, to their previous positions. Finally the events are back in the queue in the same order and the selector re-starts extracting events from the queue's head. One important issue that must be considered here is exception safety. It can be dangerous to leave events on the stack alone because an exception, thrown in the extraction method, will cause "events leak". In C++ this problem may be solved by wrapping an event object with a local guard object allocated on the stack which, when destroyed automatically during stack-unwind, would return the event back to the queue in its destructor.

   Since recursion is rather complicated and error-prone programming technique it may be much easier to fully leverage the Queue. What we need is a Queue implementation that allows blocking read not only from its head but from any other position. The extraction method specifies at which position it is going to wait. Initially, it would block on the queue's head by calling the dequeue method with 0 as the position index (assuming the head is the queue's zero reading position). If an unexpected event is detected, it is left in the queue, but the next event extraction iteration increments the position index. It is reseted to zero after expected event arrives. Alternatively, you can simply build a queue with a notification mechanism. When a new event arrives, extraction method wakes up and iterates through the queue to check if it contains the expected event.

   After going through Selector's machinery an event may be forwarded to Handler. Note, however, that consuming of expected events does not necessarily mean event

forwarding. The state of Selector decides whether an event is expected, whether it should be consumed and whether it must be transported to the Handler expected event vents are processed consumed by Selector and never leave it.

8. *Selector state*. Selector maintains state information between occurring of events. State controls all aspects of event selection from the Queue and is therefore the main director of Selector activities. Specifically, current state defines how an active Selector should respond when it is notified about a new event in the Queue, how long it should block waiting on the Queue. State transitions are entirely driven by events, both original and secondary.

   Most obvious way to maintain state information is to use State design pattern [GoF]. Selector acts as a context which delegates event consuming to current state object state object and consults it to obtain parameters for the next event extraction iteration. The event consuming behaviour including event forwarding to the handler, and state transitions is encapsulated within state subclasses. Such a decentralisation allows easy modification or extension of the state machine be defining new subclasses without changing the Selector which can be implemented generically. It further decouples queue access and event selection logic from actual state-specific event processing.

   Generic implementation of Selector behaviour, which is completely decoupled from state, requires a generic event processing operation in the state class hierarchy. Selector manipulates with two polymorphic objects through pointers or references to their base classes. Upon receiving an event object, the current state object must examine its type to take event consuming decision. Each concrete implementation of event processing operation will have a web of `if-else` or `case` statements. Alternatively, multimethods technique [AAMCD] can be applied to pick up the appropriate event-to-state case handling code. Multimethods is a mechanism that would dispatch the event consuming function call to different concrete functions depending on the dynamic types of event and state objects involved in the call.

Sample Code

The following C++ code implements the network monitoring example from the motivation section.

Let's start with the `NodeMonitor` class. It provides a multi-method interface to have a separate operation for each event. The interface conveys information about events the `NodeMonitor` is interested in:

```
class NodeMonitor
{
public:
      void OnRequestTimer(Event*& fe);
      void OnRequestFailure(Event*& fe);
      void OnRequest(Packet& rp);
      void OnResponse(Packet& rp);
      void OnResponseTimeout();
};
```

The operations `OnRequestTimer` and `OnRequestFailure` will be used by `Organizer` to pull feedback events from a `NodeMonitor` object. For instance, handling of RequestFailure event may result in generation of a `RequestEvent`:

```
void NodeMonitor::OnRequestFailure(Event*& fe)
{
      std::vector<unsigned char> data;
      cout<< "NodeMonitor OnRequestFailure\n";
      //Use the previous data and resend the packet
      fe = new RequestEvent (Packet (data));
}
```

Events are represented by Event class hierarchy:

```
class Event
{
public:
      Event(){}
      virtual ~Event(){}
};
```

Each type of original event has its subclass. So, an instance of the class `RequestEvent` represents a Request event.

Secondary events may not need an event class since they are generated during selection and therefore may not be stored in the Queue. As soon as they occur, the corresponding handler method is called on a `NodeMonitor` object. For simplicity's sake, `Event` subclasses don't have any operations or attributes so we can omit their declarations and definitions. Notice that `NodeMonitor` encapsulates how it responds to RequestTimer and RequestFailure. Different implementations can generate feedback event of different types which will have different impact on state transitions within an `Organizer`.

Given the `NodeMonitor` declaration let's continue with the `Organizer` class:

```
typedef enum
{
      SEL_RSLT_ERROR = -1,
      SEL_RSLT_TIMEOUT = 0,
      SEL_RSLT_OK = 1,
      SEL_RSLT_SHUTDOWN = 2,
} SEL_RSLT;

class Organizer
{
      friend class State;
public:

      Organizer(NodeMonitor* nm);
      virtual ~Organizer();

      void OnRequestTimer();
      void OnRequestSent();
      void OnAck(Packet& ackp);
      void OnNack(Packet& ackp);
      void OnResponse(Packet& ackp);

      void Shutdown();

private:
```

```
        auto_ptr<Queue> queue_;
        auto_ptr<State> state_;
        NodeMonitor* nm_;

        NodeMonitor* GetHandler();
        void SetState(State* newState);

        SEL_RSLT EventExtractionMethod();
        void EventLoop();
        void StartEventLoop();
};
```

The class owns one `Queue` and one `State` object. `Organizer` must also initialize the pointer to the `NodeMonitor` instance and it does so in the constructor.

`State` class is declared friend of `Organizer` to allow it to change the current state via the `SetState` and give it privileged access to `NodeMonitor` object via `GetHandler`.

The `State` hierarchy is an example of the State design pattern [GoF]. The base class provides two public operations required by `Organizer` for event selection. Other two protected operations serve as helpers for subclasses to transit the state and access the `NodeMonitor` object which is a protected attribute in the `Organizer` class. First declare base `State` class:

```
class State
{
public:
        virtual ~State()
        {

        }

        virtual bool ConsumeEvent(Event* ev, Event*& fe,
        Organizer* selector) = 0;

        virtual TimeValue GetTimeout() = 0;
protected:

        void ChangeState(Organizer* selector, State* state);
        NodeMonitor* GetHandler(Organizer* selector);
};
```

The key to any state subclass is the `ConsumeEvent` member function implementation. The concrete implementations are similar: each of them analyses the type of an input event to take corresponding action and then transits the state of the `Organizer` object passed as an argument. The event might be propagated to the `NodeMonitor` object accessed via `GetHandler` method of the `State` class. At the level of `NodeMonitor` a feedback event can be raised and passed back to `Organizer` via the out parameter `fe`. The class `WaitingForAckState` has one of the most interesting implementations of this operation:

```
bool WaitingForAckState::ConsumeEvent(Event* ev, Event*& fe,
Organizer* selector)
{
        if (AckEvent* ackEv = dynamic_cast<AckEvent*>(ev))
        {
                ChangeState(selector, new WaitingForResponseState);
```

```
            return true;
        }
        else if (NackEvent* nackEv = dynamic_cast<NackEvent*>(ev))
        {
            GetHandler(selector)->OnRequestFailure(fe);
            ChangeState(selector, new WaitingForRequestState);
            return true;
        }
        else if (TimeoutEvent* tmtEv = dynamic_cast<TimeoutEvent*>(ev))
        {
            GetHandler(selector)->OnRequestFailure(fe);
            ChangeState(selector, new WaitingForRequestState);
            return true;
        }

        return false;
    }
```

Another important example of event consuming is represented by `WaitingForRequestTimerState`:

```
    bool WaitingForRequestTimerState::ConsumeEvent(Event* ev, Event*& fe,
    Organizer* selector)
    {
        if (RequestTimerEvent* reqTimerEv =
        dynamic_cast<RequestTimerEvent*>(ev))
        {
            GetHandler(selector)->OnRequestTimer(fe);
            ChangeState(selector, new WaitingForRequestState);
            return true;
        }
        else
        {
            //this is the initial state. We have to clear all events
            //from previous request-response transaction.
            //Therefore, return true for all events to signalize they
            //are consumed.
            return true;
        }
    }
```

What's relevant above is the consuming of all events besides RequestTimer. `Organizer` is in this state initially and returns to it every time a single request-response transaction is accomplished. It expects the beginning of a new request to be originated by the request timer. Therefore, all unexpected events including late acknowledgements or response packets, which belong to any of previous transactions, are simply discarded and marked as "consumed" by returning `true`.

The operation `GetTimeout` provides a time value indicating how long the Organizer can be in the given state:

```
    TimeValue WaitingForAckState::GetTimeout()
    {
        return 5; //seconds to wait
    }
```

This is most trivial implementation since it returns a hard-coded number. Real-world application would calculate timeout dynamically. For example, as soon as RequestSent event

arrives the Organizer may start countdown immediately in the OnRequestSent method by adjusting the timeout value in the WaitingForRequestSentState object. Of course, timeout may exceed before the Organizer comes to dequeueing the event which may indicate problems with meeting application constraints - events can't be consumed and processed quickly enough.

The Queue class used by the Organizer exposes very simple interface:

```
class Queue
{
public:
      Queue();
      virtual ~Queue();
      void Enqueue(Event* ev);
      void Undequeue(Event* ev);
      SEL_RSLT Dequeue(Event*& ev, TimeValue& timeout);
};
```

Organizer exposes a multi-method event handling interface which is significantly different form the interface of NodeMonitor since it is interested in another set of events. In each operation of the interface, Organizer must create an event object and put it into the Queue. As an example, the operations OnResponse and OnRequestTimer demonstrate event object's enqueueing common in implementations of selector's event handling methods:

```
void Organizer::OnRequestTimer()
{
      //Perform lazy activation of the thread
      StartEventLoop();
      queue_->Enqueue(new RequestTimerEvent);
}

void Organizer::OnResponse(Packet& ackp)
{
      queue_->Enqueue(new ResponseEvent(ackp));
}
```

The Shutdown operation can be also considered as part of the event handling interface since it transforms request to stop the event selection into an event object exactly like the above two methods:

```
void Organizer::Shutdown()
{
      queue_->Enqueue(new ShutdownEvent);
}
```

Note that in this example the selection thread which runs the event selection loop is started lazily when the first RequestTimer event arrives. The loop calls the event extraction method until either error or shutdown event is detected:

```
void Organizer::EventLoop()
{
      while (1)
      {
            int result = EventExtractionMethod();
            if (SEL_RSLT_ERROR == result
                  || SEL_RSLT_SHUTDOWN == result)
```

```
                {
                        break;
                }
        }
}
```

Below is one of the possible implementations of the event extraction method:

```
SEL_RSLT Organizer::EventExtractionMethod()
{
        Event* ev = NULL;
        Event* fe = NULL;
        SEL_RSLT result = queue_->Dequeue(ev, state_->GetTimeout());

        if (SEL_RSLT_OK == result)
        {
                bool is_expected = state_->ConsumeEvent(ev, fe, this);
                if (fe) queue_->Enqueue(fe);

                if (!is_expected)
                {
                        cout << "unexpected event\n";
                        result = EventExtractionMethod();
                        if(ev) queue_->Undequeue(ev);
                }
                else
                {
                        //release ev
                        delete ev;
                }

        }
        else if (SEL_RSLT_TIMEOUT == result)
        {
                state_->ConsumeEvent(new TimeoutEvent, fe, this);
                if (fe) queue_->Enqueue(fe);
        }
        else if (SEL_RSLT_SHUTDOWN == result)
        {
                return result;
        }

        return result;
}
```

As described in the Implementation section, event selection consists of two steps. First we try to get the next event from the queue. Second the event gets forwarded to current state object which ultimately decides whether the event is unexpected and extraction method must go into recursion.

Detected timeouts are transformed into event objects to get consumed as well. `EventExtractionMethod` also ensures the consumed events get deleted. The method is the best place to release event objects because this frees state subclasses from resource management burden. For convenience, the detection of shutdown event is encapsulated within the `Queue` class. The `Queue` takes an added responsibility – an option unavailable if you opt to use available queue implementation straight out of box

The above implementation is very generic. The `Organizer` does not know a thing about the communication protocol it implements; it's the state subclasses that encapsulate how events

get processed and define state transitions. The `State` class encodes constraints dictated by the communication protocol in terms of its member functions. This `Organizer` can be reused for different protocols that have different set of events and states.

The following code connects the participants and provides one interesting test scenario when a RequestTimer event arrives before the previous request is fully accomplished:

```
NodeMonitor handler;
Organizer selector(&handler);

Packet dummyPacket;

selector.OnRequestTimer();
Sleep(200);
selector.OnRequestSent();
selector.OnAck(dummyPacket);
selector.OnRequestTimer();
selector.OnResponse(dummyPacket);
Sleep(200);
selector.OnRequestSent();
selector.OnAck(dummyPacket);
selector.OnResponse(dummyPacket);

selector.Shutdown();
…
```

Assuming each NodeMonitor method prints corresponding output to identify itself we will have following output:

```
NodeMonitor OnRequestTimer
NodeMonitor OnRequest
unexpected event
NodeMonitor OnResponse
NodeMonitor OnRequestTimer
NodeMonitor OnRequest
NodeMonitor OnResponse
```

Known Uses

The ACE Task framework form the Adaptive Communication Environment [ACE] provides a great basis for applying the Stateful Event Selector pattern in C++. The class `ACE_Task` class includes an `ACE_Message_Queue` in each instance and may be started in a separate thread. Application programmers build an event selector as a subclass of ACE_Task.

The event handling machinery in the Java Swing [JFC] may be considered as a simplified version of Stateful Event Selector. It extracts events such as button actions or mouse clicks from the queue and handles them using single event-dispatching thread. Since every incoming event is expected and immediately handled, the event machinery has one state which is "waiting for any event". The Swing's method `invokeLater()` is part of the event handling interface to the Selector. It enqueues any code that must be executed in the event-dispatching thread. If `invokeLater` is called from the event dispatching thread the code will still be deferred until all pending events have been processed. This represents an example of an event feedback.

Related patterns

The Stateful Event Selector is related to the State design pattern [GoF]. Selector's state is usually implemented using the State pattern. In this case, Selector is a Context for its clients. The difference is that in traditional implementation of the State pattern an event received by Context leads to calling of a virtual method on the current state object. In this pattern an event first goes to the queue and then may or may not get translated into method call executed in another thread.

A Selector is an Active Object [ACTOBJ]. However there are significant differences between both patterns. In contrast to Active Object, which is a pure intermediary between proxy and servant, Selector represents state and implements some specific application logic.

In terms of Pipes and Filters architectural pattern a Selector is a Filter - a processing unit in a stream that enriches, refines, or transforms input data.

References

[AAMCD] Andrei Alexandrescu , Modern C++ Design: Generic Programming and Design Patterns Applied, Addison Wesley, 2001.

[ACE] The ADAPTIVE Communication Environment (ACE(TM)), available at http://www.cs.wustl.edu/~schmidt/ACE.html

[ACTOBJ] R. Lavender, D. Schmidt: Active Object, An Object Behavioral Pattern for Concurrent Programming, available at http://www.cse.wustl.edu/~doc/pspdfs/Act-Obj.pdf

[GoF] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[JFC] Java Foundation Classes (JFC), Swing, available at http://java.sun.com/javase/technologies/desktop

[POSA2] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects, John Wiley and Sons, 2000