A Pattern Language of Call Processing

Greg Utas
utas@nortelnetworks.com

1. Introduction

The call processing component of a switching system is responsible for setting up basic calls between external interfaces and for providing a broad range of features such as call forwarding, conference calling, and call waiting. This article discusses design concepts that address some of the challenges faced when developing a call processing system. Each design concept is presented in the form of a *pattern*, a term recently adopted by the software design community [1,2,3,4,5]. A group of related patterns constitutes a *pattern language* for its domain, which in this case is the domain of call processing. This article develops a pattern language for call processing by using a number of examples. Each pattern within the language is presented using an outline that has become fairly standard within the design patterns community:

Context: the situation in which the pattern applies
Problem: the challenge to be addressed
Forces: factors that contribute to the problem and/or its solution
Solution: how to address the problem
Rationale: the reasoning behind the solution
Resulting context: any interesting side effects or problems that the solution creates
Examples: examples of how the pattern is applied
Related patterns: any patterns that are closely related to the one just described

Although the patterns focus on call processing systems, a number of them are applicable to transactional, real-time systems in general. Some of the patterns are also instances or refinements of patterns described elsewhere. They are included because of their importance to the overall pattern language and to illustrate how previously published patterns can be applied to the call processing domain.

Because this article deals with software implementation, it discusses base classes that are defined by a call processing framework which uses the article's patterns. This involves the development of some terminology. To assist the reader, Section 8 provides a glossary, and Figure 2 illustrates relationships between the framework classes discussed in the article.

2. Summary of the Pattern Language

The language (see Figure 1) begins by advocating the use of *State Machine* to implement basic calls and features. This pattern is somewhat prescriptive in terms of implementation details because it seeks to avoid disadvantages associated with double-dispatching. *Separation of Call Halves* splits basic calls into originating and terminating state machines in order to enable scalable distributed processing and avoid an $O(n^2)$ explosion when interworking the protocols of

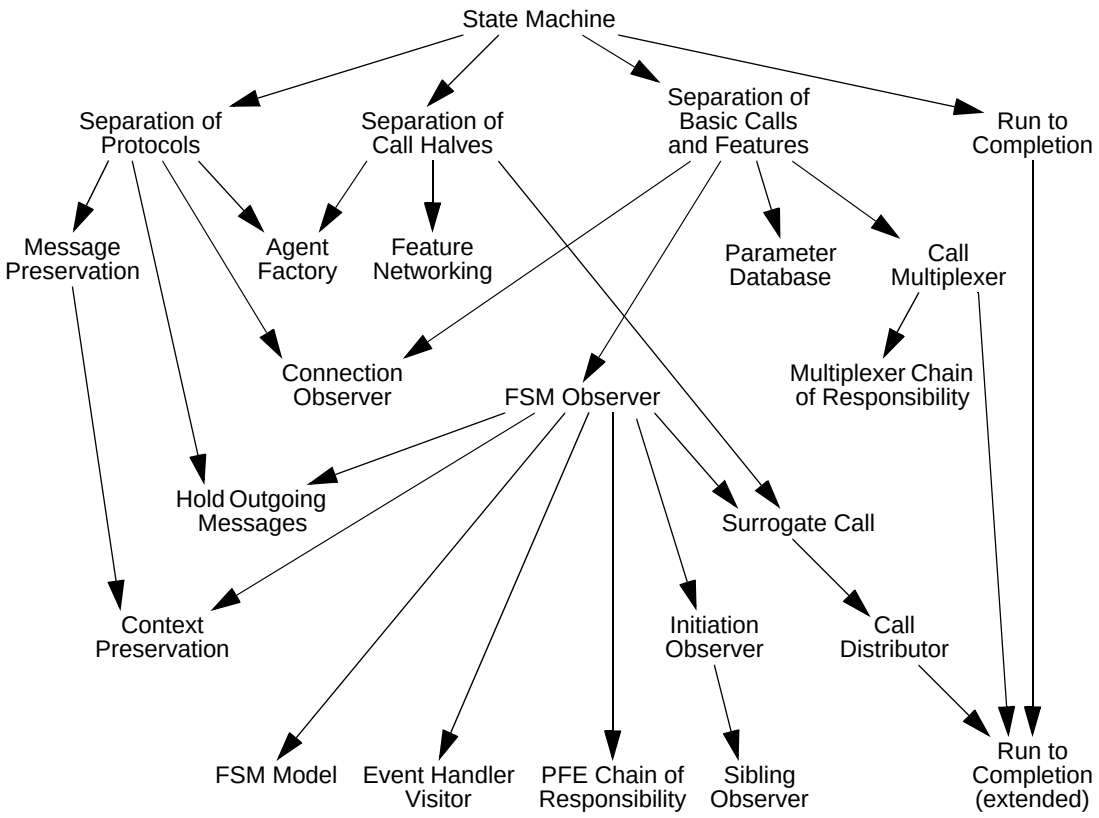

Figure 1. A map of the pattern language. Arrows indicate dependencies. For example, *Separation of Call Halves* and *FSM Observer* are prerequisites for *Surrogate Call.*

originating and terminating interfaces. In each half-call, *Separation of Protocols* provides a separate PSM (Protocol State Machine) for each protocol involved in the call. *Run to Completion* minimizes critical section bugs by making each call processing transaction an indivisible operation. *Agent Factory* describes how to instantiate the appropriate subclasses of the dynamic objects used in a call. Finally, *Message Preservation* describes how saving entire messages, so that they can be referenced during subsequent transactions, makes it easier to handle certain scenarios.

The patterns described thus far are used in basic calls. However, the introduction of features gives rise to additional patterns. *Separation of Basic Calls and Features* is a metapattern that discusses the importance of keeping basic call and feature state machines completely separated. This is largely achieved by *FSM Observer*, which allows features to observe the behavior of basic call state machines and augment or override this behavior as required. Some features, however, need to modify many basic call subclasses, and so *FSM Model* is introduced to provide a common abstract state machine for all basic calls. This frequently allows a single version of a feature to modify all of the necessary basic call subclasses. A feature may also need to modify messages built by basic call software; this is addressed by *Hold Outgoing Messages*, which allows a feature to access an outgoing message and modify it before it is sent.

The next group of patterns addresses situations in which a feature affects more than one half-call. When a feature affects both halves of a call, *Feature Networking* splits it into two state machines; this is just an elaboration of *Separation of Call Halves*. When a feature allows more than one call to be presented to the same interface, *Call Multiplexer* provides it with a separate context that

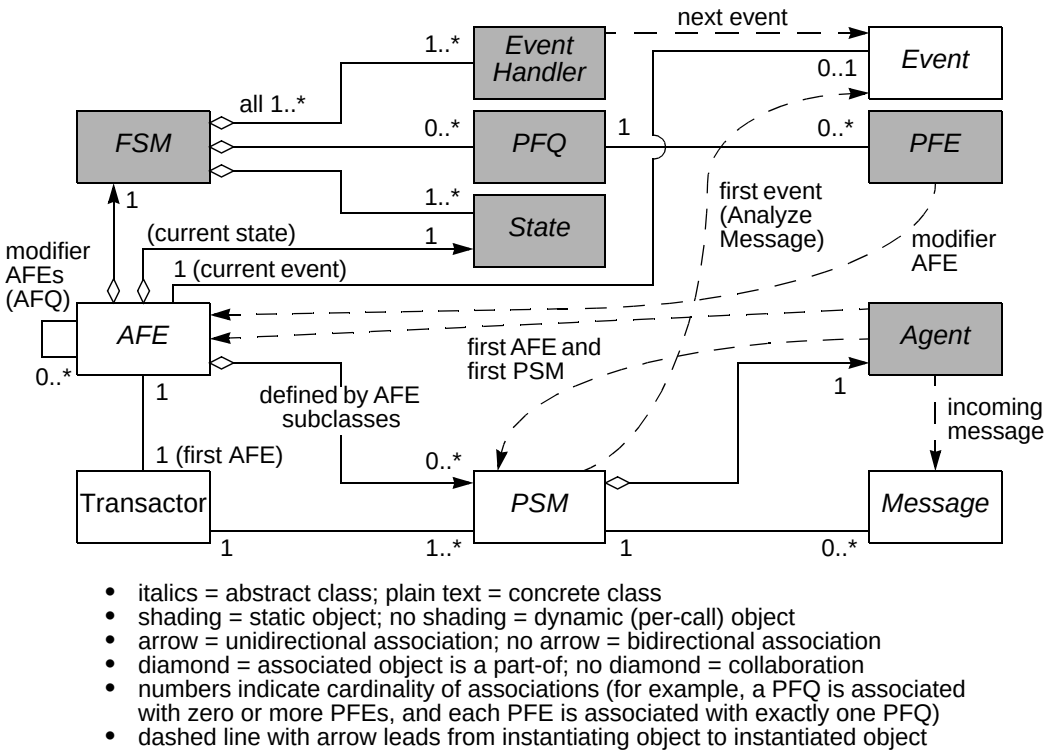

Figure 2. A class diagram showing relationships between the types of objects discussed in this article. The Transactor class is not discussed elsewhere but is shown here because it is involved in scheduling a call process. An incoming message is temporarily assigned to a Transactor, which is then queued to run. When the call runs, its Transactor passes the message to the PSM associated with the I/O address on which the message arrived. This PSM subsequently raises the Analyze Message event to pass the message to the first AFE.

subtends those calls, which makes it easier to coordinate actions that affect more than one call. Other features redirect a call to a new interface. Here *Surrogate Call* allows the redirected leg of the call to be set up on behalf of the redirecting user, who is logically—but not physically—involved in the call. Finally, when a feature simultaneously redirects a call to multiple interfaces, *Call Distributor*—essentially the reverse of *Call Multiplexer*—provides the feature with a separate context for managing the resulting fan-out.

Run to Completion, as described for basic calls, is inadequate for handling certain scenarios involving features. When *Call Multiplexer* or *Call Distributor* is applied, a user's processing context no longer consists of a single half-call. The transaction initiated by an incoming message must now ripple through multiplexers and distributors as well as half-calls; all of these are peer processing contexts that communicate using asynchronous messages. *Run to Completion (extended)* therefore broadens the definition of a transaction so that all processing within a user's context is performed as an extended critical section. This eliminates state-space explosions and race conditions that would occur if pure asynchronous messaging were used between the user's half-calls and multiplexers or distributors. However, some features must interrupt a transaction to interact with other nodes in the network. For various reasons, it is undesirable to use synchronous messaging (blocking sends) in these situations. *Context Preservation* allows a feature to save the current processing context so that it can transparently resume an interrupted transaction when it receives an asynchronous response from another node.

The final set of patterns handles interactions between features. In many of these situations, features must cooperate even though their state machines are completely decoupled from one another. *Parameter Database* allows call setup information to be modified by one feature and subsequently retrieved by another. *Connection Observer* allows a feature to monitor precisely what a user is receiving and transmitting, even as other features modify the connection topology. *Multiplexer Chain of Responsibility* sorts out signalling and connection interactions when more than one multiplexer is running in a user's context. When an observable basic call event might trigger more than one feature, *PFE Chain of Responsibility* arranges the features so that the most desirable one is able to trigger first. If a feature is already running on the call and an incompatible feature is triggered, *Initiation Observer* allows the active feature to deny the initiation of the new feature. When more than one feature is modifying the same half-call, *Event Handler Visitor* allows one feature to effect a basic call state transition without hiding it from the others, so that all features have the chance to react appropriately. And if two features must interact in some peculiar manner, *Sibling Observer* allows them to define a private protocol for this purpose.

3. Patterns for Basic Calls

3.1 State Machine

- **Context**: You are developing a call processing application.
- **Problem**: When the application receives a message, how can it decide what work to perform?
- **Forces**: When a message arrives from an external interface, the work to be performed is determined by the application's current state. As work is performed, the application's state must be updated so that it can correctly react to the next message.
- **Solution**: Provide a framework for implementing applications as state machines. Define the abstract classes **FSM** (finite state machine), **state**, **event**, and **event handler**. Each application has its own FSM subclass, which is a *Flyweight* [1] that is shared by all run-time (per-call) instances of the application. The FSM subclass registers the *Flyweight* state and event handler subclasses that implement the application. The FSM's two registries, for states and event handlers, are implemented as simple arrays by assigning a unique integer identifier to each state and event handler subclass. Registration then simply consists of indexing into the appropriate array and saving a reference to the state or event handler subclass.

  At run time, events are instantiated ("raised") to perform work on behalf of an FSM, which is assigned an **active FSM element** (AFE) to house its per-call data. The AFE contains a reference to its corresponding FSM, its current state, and the event that it is processing. This allows the correct event handler to be invoked. This event handler sets the next state and returns the next event to be processed if another event handler must perform additional work.

  Event subclasses are also assigned unique integer identifiers to index an array belonging to state subclasses. The entries in this array are event handler identifiers. Given an FSM, state, and an event, a two-step procedure invokes the correct event handler:

```
eh_id = current_fsm->states[current_state_id][current_event_id];
next_event = current_fsm->event_handlers[eh_id](current_event);
```

- **Rationale**: Standards often use Specification and Description Language (SDL) to describe the behavior of call processing applications. SDL is based on state machines, so the use of state machines makes the relationship between specifications and implementations clear. And if all call processing applications are implemented using a common framework, it is much easier for one designer to support applications developed by other designers.

Object-oriented designs frequently use double-dispatching to invoke event handlers. The approach described above improves on double-dispatching in many ways. First, indexing by state and event identifiers to find event handlers is more intuitive. Second, the decoupling of states, events, and event handlers allows state machines to be modified with far less recompilation. Third, any new or modified subclasses can easily register in a running system, which is critical because switches must be continuously available. Finally, the framework performs all event handler dispatching, which prevents designers from adding inappropriate "hooks" to such software.

- **Related patterns**: This pattern extends *Three-Level FSM* [6], *State Objects* [7], and *Owner-Driven Transitions* [8] by decoupling states, events, and event handlers. The importance of using a common state machine framework will be further reinforced when we see how the patterns *FSM Observer* and *FSM Model* add features to basic calls.

3.2 Run to Completion

- **Context**: You are using *State Machine* to develop call processing applications.
- **Problem**: How do you prevent state machines from interfering with each other?
- **Forces**: Call processing applications contend for resources among themselves and with other parts of the system, such as hardware maintenance and provisioning applications. These applications share resources such as inter-switch trunks, service circuits, and memory for objects. Under round-robin, preemptive scheduling, critical sections must be used when obtaining and releasing shared resources.
- **Solution**: When a call begins to process a message, let it run until it has completed all of its work. At that point, the call takes a real-time break to wait for its next input. The scheduler is not allowed to preempt one call to run another. This pattern is also described in [9].
- **Rationale**: Running one call at a time eliminates most critical sections in call processing applications, which simplifies their design and makes them less prone to error. It also improves the system's call-handling capacity by reducing the amount of time spent on process swapping.
- **Resulting context**: If a call runs to completion, how does it wait for its next input? Call processing is a transactional, real-time system that must react to asynchronous inputs (messages) that arrive from external interfaces. The solution is *Half-Sync/Half-Async* [10], which decouples the handling of I/O from applications. I/O interrupts are allowed when a call is running, but incoming messages are simply queued against a call. No call processing occurs until the call runs, so resource contention between calls does not increase. Finally, a sanity timeout is required on each transaction so that a call cannot hang the switch by running forever!

3.3 Separation of Call Halves

- **Context**: You are using *State Machine* to develop call processing applications.
- **Problem**: How do you implement state machines for basic calls?
- **Forces**: Many calls are set up between external interfaces that use different protocols. There are hundreds of inter-switch trunk protocols in use around the world. A switch intended for global markets must support calls between many combinations of these protocols.
- **Solution**: Separate each basic call into two halves, an originating basic call (OBC) and a terminating basic call (TBC), which are implemented as separate state machines. Define a call interworking protocol for all communication between OBC and TBC subclasses. This is an example of *Half-Object Plus Protocol* [11].

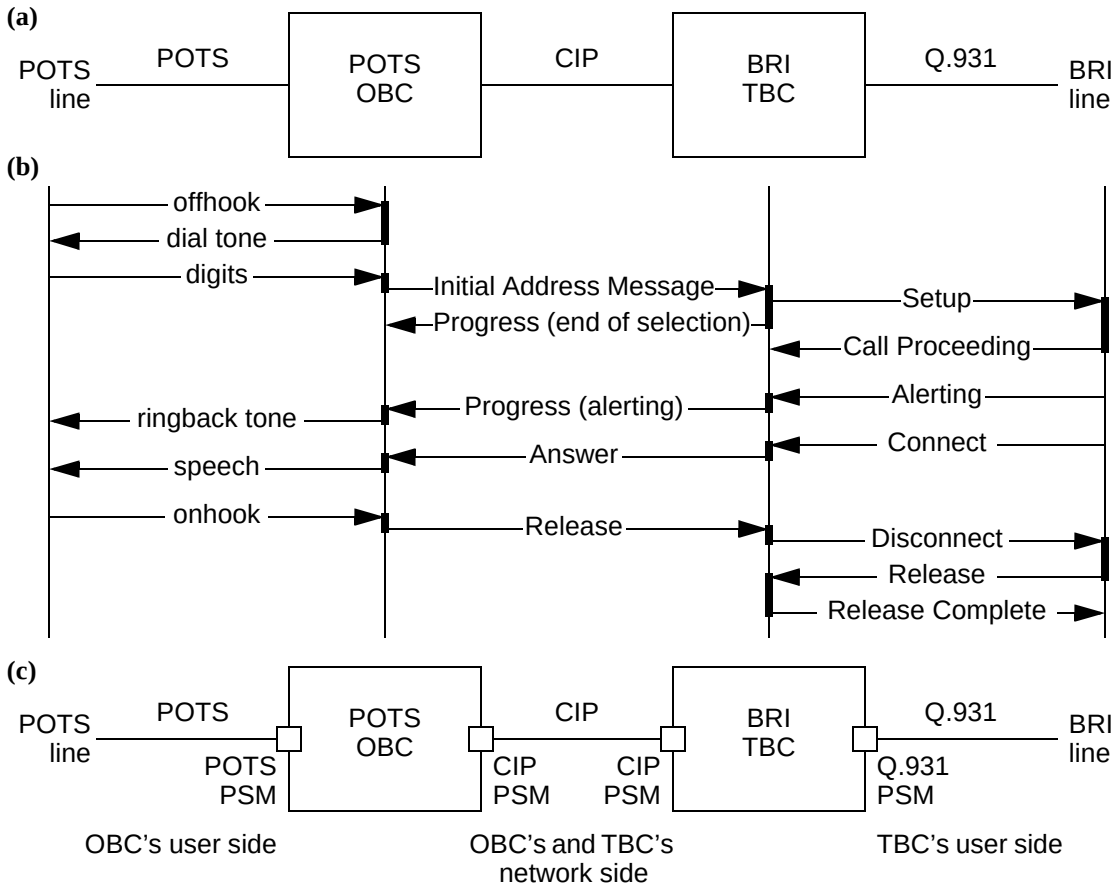

Figure 3. Call interworking between a Plain Ordinary Telephone Service (POTS) line OBC and an ISDN Basic Rate Interface (BRI) line TBC. The diagrams show (a) the basic call AFEs and protocols used in the call, (b) the sequence of messages that set up and take down the call, and (c) basic call PSMs (the small squares) in addition to the AFEs. Each AFE and its PSMs comprise a context that runs to completion.

- **Rationale**: There are many reasons for this pattern. First, software that deals directly with both originating and terminating external protocols becomes $O(n^2)$ in size, where n is the number of protocols. The standard call interworking protocol reduces the amount of software to $O(n)$. Second, Intelligent Network standards [12,13] are based on originating and terminating call halves. A switch supports Intelligent Network capabilities more easily if its internal design is based on a similar model. Third, the pattern allows distribution because OBC and TBC can run on separate processors. A multi-processor system based on such distribution will have a higher call-handling capacity than a single-processor system. In a multi-processor system, it is unclear where a central call object, as opposed to half-call objects, should run. A half-call, however, can run in the processor that owns the external interface where the call originated or terminated. Once call halves are separated, an intermediate call object is not needed.
- **Example**: Figure 3a shows a typical OBC-TBC configuration. Figure 3b shows the messages that establish a call. The call interworking protocol (CIP) is based on ISUP [14], the most comprehensive protocol for setting up calls between switches, but it also adds parameters that support internal switch capabilities such as call recording and connection control.

3.4 Separation of Protocols

- **Context**: You are using *State Machine* to develop call processing applications.
- **Problem**: How do you support protocols?
- **Forces**: Protocols are fundamental to call processing. All external interfaces supported by a switch are specified in terms of protocols. Protocols are also used to communicate with service circuits that are internal to the switch. A call often uses a number of protocols simultaneously, and each protocol dialogue begins and ends independently.
- **Solution**: Define the abstract class **protocol state machine** (PSM) to support protocols. Each protocol will have its own PSM subclass. Implement each per-call usage of a protocol using a separate PSM, and pass all messages sent or received by call processing applications through PSMs. Each PSM is associated with an I/O address where it receives messages, and it knows the I/O address of its conjugate (peer) PSM, with which it communicates using asynchronous messages. PSMs closely resemble the Port objects described in [15].
- **Rationale**: This pattern allows each protocol's PSM to be created (destroyed) when an initial (final) message in the protocol is sent or received. Because it is the first object to process an incoming message, and the last one to process an outgoing message, a PSM can enforce its protocol. And because PSMs are distinct objects, they are easily reused by different applications. Reuse is most common for service circuit PSMs, which AFEs create as required.
- **Resulting context**: FSMs and PSMs are decoupled, so their states are independent. This is important in allowing call phases to be repeated. During call reorigination, for example, a user logs into a calling card account, usually over an ISUP trunk, makes a call, and then makes another call after dialling '#'. This can cause answer to occur more than once. For each call, OBC creates a billing record and records the time of answer and disconnect. However, an ISUP Answer Message cannot be sent when a subsequent answer occurs because the ISUP protocol does not allow this. So although OBC enters its Active (talking) state more than once, it must consult its ISUP PSM before telling it to send an Answer Message, because one might have already been sent during a previous call.
- **Examples**: Both OBC and TBC use two primary PSMs. One PSM appears on the OBC or TBC *network side* and uses the call interworking protocol (CIP) to communicate with its conjugate PSM on the other half-call. The other PSM appears on the OBC or TBC *user side* and uses a standard call control protocol to communicate with an external interface, such as a subscriber line or inter-switch trunk. Figure 3c shows the PSMs in a basic call.

  There are also other types of PSMs. Some communicate with service circuits such as tone generators and receivers, announcements, and conference bridges. Others communicate with Service Control Points to provide Intelligent Network protocols such as CS-2 [12] and AIN 0.2 [13], which allow call processing applications to run outside of the switch.

3.5 Agent Factory

- **Context**: A message arrives to create a call. You are using *Separation of Call Halves* and *Separation of Protocols* to implement calls.
- **Problem**: How do you create the objects that will handle the message?
- **Forces**: When a call's first message arrives, a PSM and AFE must be created. In addition, a message object must be created as a wrapper for each incoming message (byte stream). However, all of these objects are subclassed based on the protocol they support. How do you instantiate the correct subclasses?

- **Solution**: Define the abstract class **agent**, each of whose *Singleton* subclasses acts as an *Abstract Factory* [1] for creating AFEs, PSMs, and incoming messages. When provisioning an external interface, assign it an I/O address on which its messages will be received and sent. The interface will use this address for as long as it remains provisioned. Associate the I/O address with the agent subclass that supports the type of external interface being provisioned. When a message arrives on an I/O address, delegate creation of the message object to the agent that is associated with the address. If the I/O address is not yet associated with a PSM, the message is the first one for a new call, in which case the agent must also "bootstrap" the call by creating a PSM and an AFE. The creation of objects by an agent subclass is shown in Figure 2.
- **Rationale**: I/O addresses are used by different types of interfaces. The types of messages received on an I/O address are determined by the interface associated with the I/O address. The creation of message objects, PSMs, and AFEs should therefore be delegated to an object that is subclassed based on the type of interface that owns the I/O address. The abstract agent superclass allows the I/O interrupt handler to delegate instantiation in a uniform manner.
- **Examples**: In Figure 3c, the POTS agent subclass creates the POTS PSM and the POTS OBC; it also wraps each message arriving from the POTS line. The creation of the BRI TBC is interesting because the first message received by a TBC is always a CIP Initial Address Message. This message arrives on an I/O address associated with the CIP agent subclass, so it creates a CIP PSM and CIP message wrapper. However, AFEs are subclassed according to the user-side protocol that they support. The TBC to be created must be based on the terminating interface, in this case a BRI line. The CIP agent knows this because OBC includes, in the CIP Initial Address Message, a parameter that identifies the type of agent that will receive the call. OBC obtains this parameter from the switch's translation and routing database, which it uses to analyze the dialled number and determine the call's destination.

3.6 Message Preservation

- **Context**: A message arrives at a call, or a call sends a message. Messages are first-class objects, as shown in Figure 2, and are received and sent through PSMs, as described in *Separation of Protocols*.
- **Problem**: How do you reference parameters in the message during subsequent transactions?
- **Forces**: The parameters in an incoming message are used to set up a call, but call setup is not always completed within a single transaction. And when an outgoing message is sent, a PSM often starts a timer. If a reply to the message is not received before the timer expires, the PSM may have to retransmit the message. Specifications may require this behavior even if message delivery is guaranteed, as a way to handle race conditions or errors in terminal equipment.
- **Solution**: Define `save` and `unsave` methods on an abstract **message** class from which all messages are subclassed. The default behavior is for an incoming message to be destroyed when the transaction ends, and for an outgoing message to be destroyed after it is sent. However, invoking the `save` method increments a counter within the message, in which case the message is not destroyed until the call ends or an `unsave` operation drops the count to zero. A saved message is owned by the PSM through which the message was received or sent.
- **Rationale**: It is inefficient to copy parameters to "permanent" locations when they need to be referenced during a subsequent transaction. It is also inefficient, and difficult, to reconstruct an outgoing message when it must be retransmitted. Not only does a message object allow a message to be saved, but it also allows details such as message building and parsing to be hidden from applications. Finally, preserving only parts of a message churns software because the list of parameters to be preserved grows as new applications are added. The easiest solution

is to simply preserve the entire message. The memory cost of doing so is usually outweighed by improved performance and reduced software churn.

- **Examples**: A call saves its first incoming message, such as a Q.931 Setup or a CIP Initial Address Message, until answer occurs; this allows the message's parameters to be referenced during call setup. A PSM saves an outgoing message, such as a Q.931 Release, when it must retransmit it if an expected response (a Release Complete, in this case) is not received.

4. Patterns for Features

This section extends the pattern language for basic call processing with patterns that support features such as call forwarding, call waiting, and conference calls.

4.1 Separation of Basic Calls and Features

- **Context**: You have developed a call processing system whose basic calls use *State Machine*.
- **Problem**: How do you add features to the system?
- **Forces**: The system will support dozens of features. Many features will be developed in each software release, so their designers must be able to work in parallel. Some features are as complex as basic calls, being comprised of many states and dozens of events. Features must also interact with basic call and each other. Features need to override basic call behavior, augment it, or reuse it. There are also cases in which multiple features might run on a call, either sequentially or in parallel. In such cases the features must interact harmoniously or, at the very least, avoid interfering with each other. This problem is known as *feature interaction*.
- **Solution**: Implement each feature as a separate state machine.
- **Rationale**: Separating state machines avoids the drawbacks of *One Big State Machine*, which quickly becomes a morass of convoluted, error-prone software that designers are constantly waiting in line to modify. A call processing system that does not separate state machines will resemble a *Big Ball of Mud* [16].
- **Resulting context**: This pattern is a metapattern because it says nothing about *how* to separate basic calls and features. Achieving such a separation is difficult because features must seamlessly interact with both basic calls and each other. The remaining patterns in this section focus on how features can interact with basic calls even though their state machines are separated. The topic of feature interaction—how multiple features on the same call can interact even when they all run as separate state machines—is discussed in Section 5.

4.2 FSM Observer

- **Context**: You are using *Separation of Basic Calls and Features* and are developing a feature that affects the behavior of basic calls.
- **Problem**: How can you keep feature logic out of basic call but allow the feature to reuse basic call software whenever possible?
- **Forces**: When a basic call event is raised, a feature must be able to override, augment, or reuse basic call behavior, depending on feature requirements.
- **Solution**: Allow features to observe the behavior of the basic call FSM. Before basic call processes an event, allow a feature to override basic call behavior by intercepting the event. After basic call processes an event, inform each feature of the state transition that just occurred,

in case some features need to perform additional actions. Features that react to a state transition can only augment, not override, basic call behavior.

Features register interest in basic call events in two ways. Here it is necessary to distinguish *active* features, which have already been initiated, from *passive* features, which are waiting to be triggered. Just like basic call, each active feature has it own AFE. When a feature is initiated, its AFE is created and placed in an active FSM queue (AFQ) owned by basic call.

In the same way that active features have AFEs, passive features have **passive FSM elements** (PFEs). Each PFE is registered in a passive feature queue (PFQ) owned by a basic call FSM. Basic call defines a PFQ for each of its observable events and state transitions.

Just before a basic call event handler finishes executing, it specifies the PFQ that should be informed of the state transition that just occurred, as well as the PFQ that should be given an opportunity to react to the next event, if any. Features in basic call's AFQ are automatically informed of each observable event and state transition. To put it another way, a PFE registers with a basic call *class* (an FSM) so that it is notified when *any* basic call in that class signals the event or state transition in which it is interested. The PFE may then create an AFE, which registers with a basic call *object* (an AFE) so that it is notified of events and state transitions signalled only by that *specific* basic call instance.

After a feature processes an event, it returns the next event to be processed and an instruction that specifies how this event should be routed. **Continue** indicates that the feature will process the next event as well; this allows work to be split among a series of event handlers. **Consume** indicates that there is no next event, and so the transaction ends. **Pass** returns the same event. If no other feature is interested in the event, it returns to basic call, which processes it. **Reenter** is used after raising a new basic call event, which then overrides any event that basic call had planned to process. This allows a feature to reuse basic call software after it has altered the normal call flow.

- **Rationale**: This pattern allows basic calls and features to be implemented as separate state machines while maximizing reuse of basic call software through the use of the event routing instructions **pass** and **reenter**.
- **Resulting context**: This pattern can be applied recursively, to allow one feature to modify the behavior of another feature. Because the framework provides the abstract classes that support all state machines, it is easy to support such recursion.

  Active features (in the AFQ) observe events before passive features (in the PFQ). The reason for this is that active features are already modifying basic call behavior and therefore have priority when overriding basic call events, so as to ensure that their overrides take effect. The same approach is used during state transitions, although here the order is not important: all features in the AFQ and PFQ are notified because they can only augment basic call behavior.

  The AFQ actually behaves as a stack so that the most recently triggered feature will be in the foreground, where it has the first opportunity to react to events. The ordering of features in the PFQ is discussed later, in *PFE Chain of Responsibility*.
- **Examples**: Call waiting (CWT) has a PFE that reacts to the basic call event that is raised when a user is busy. If the user subscribes to CWT, the PFE asks for CWT to be initiated, in which case a CWT AFE is created and placed in basic call's AFQ. Control then transfers to CWT's AFE, which attempts to present the call to the user instead of allowing basic call to release it because the subscriber is busy.

  Figure 4 shows the Intelligent Network service switching function (SSF) running in the context of an originating basic call and communicating with a Service Control Point (SCP). The SSF

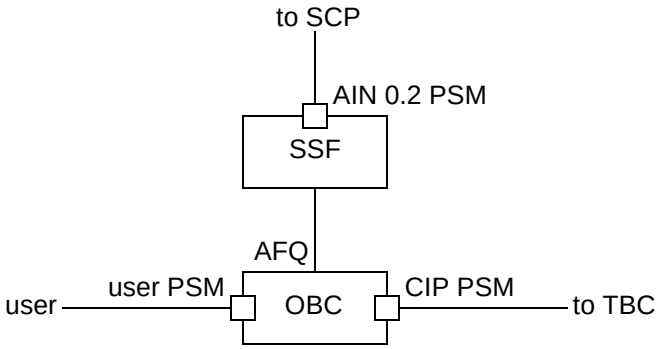

Figure 4. The Intelligent Network service switching function (SSF) running as a feature that modifies the behavior of a basic call OBC. SSF has created a PSM in order to communicate with a Service Control Point (SCP) using the AIN 0.2 protocol. The "protocol" between OBC and SSF is synchronous and is shown using an object association. In the OBC to SSF direction, the protocol consists of basic call events and state transitions. In the SSF to OBC direction, it consists of the event routing instructions (**consume**, **pass**, and **reenter**) associated with basic call events. OBC, SSF, and their three PSMs comprise a context that runs to completion.

feature allows features implemented in the SCP to observe basic call events and state transitions so that they can override and augment basic call's behavior in the same way as switch-based features.

- **Related patterns**: This pattern extends *State Machine* with *Recursive Control* [17] and *Orthogonal Behavior* and *Broadcasting* [18]. The overall behavior is guided by event routing instructions and is *dynamically defined* by whatever features are currently modifying basic call.

4.3 FSM Model

- **Context**: You are developing a feature that is required on many types of basic calls. Each basic call supports *FSM Observer*.
- **Problem**: How can you avoid developing a different version of the feature for each basic call?
- **Forces**: There are many similarities between the basic call state machines that support different external interfaces. Each OBC, for example, must collect digits, analyze the digits to find a route over which the call can be completed, send a call setup request to TBC, wait for an acknowledgment from TBC, then wait for TBC to answer, and finally wait for the user or TBC to release the call. However, the ways in which different OBCs implement these actions depend on the user-side protocol.
- **Solution**: Define a generic call model that is based on all basic call state machines. In fact, Intelligent Network standards already define a basic call model that is split into two parts, one for OBC and one for TBC. This call model is a good starting point for use within a switch.

  A call model specifies a standard set of states, events, and state transitions used by all basic calls. Each basic call maps its actions to the state-event space defined by the call model. This allows a feature to use *FSM Observer* to monitor the behavior of *any* basic call, without regard to the underlying user-side protocol. Differences among user-side protocols are hidden by basic call event handlers, which are subclassed (when necessary) according to the user-side protocol.
- **Rationale**: This pattern allows features to be reused on different types of basic calls. Basing the call model on Intelligent Network standards makes it easier for the Intelligent Network service switching function to support those standards.

- **Resulting context**: This is only a partial solution to reusing features on different basic calls. In the same way that basic calls are subclassed according to the user-side protocol that they support, so too are features. Thus, if a feature must directly interact with a user-side protocol, it must still be subclassed. In many of these cases, however, much of the feature's software will be protocol-independent and thus reusable on all call types. To make the feature totally generic, it would be necessary to create an abstract superclass user-side PSM that hides the differences between all user-side PSMs. The CIP PSM is a start in this direction, but its scope is limited to the setup and takedown of basic calls between different interfaces. Extending the CIP PSM to act as a superset of all line (access) protocols is very difficult. A detailed discussion of this topic is beyond the scope of this article. Suffice it to say that CIP exists to reduce the amount of call software from $O(n^2)$ to $O(n)$, as described in *Separation of Call Halves*. An abstract user-side PSM is then an attempt to reduce the amount of feature software from $O(n)$ to $O(k)$, where $k=1$. This attempt will be futile, however, because each feature's protocol-specific components will simply move from its FSM's event handlers to user-side PSMs. This will also increase system complexity by significantly increasing the collaboration required between event handlers and PSMs.
- **Examples**: Features that need to be used by many different types of basic calls include toll-free (800 numbers in North America), local number portability, and the Intelligent Network service switching function.
- **Related patterns**: The use of a *call model* is central to Intelligent Network standards. The name *FSM Model* reflects the fact that the pattern can be applied to any feature, not just basic calls. However, this usage is rare because most features observe basic calls, not other features.

4.4 Hold Outgoing Messages

- **Context**: A feature needs to alter a message that is built by basic call, which uses *Separation of Protocols* and supports *FSM Observer*.
- **Problem**: How can the feature alter the message?
- **Forces**: The feature may need to add a parameter to the message or modify a parameter that was already added to the message by basic call.
- **Solution**: Queue each outgoing message on the appropriate PSM and send it only when the transaction ends.
- **Rationale**: Basic call remains free of feature-specific messaging logic. A feature can locate a queued message and alter it, usually when it is informed of a basic call state transition. A feature can also be given the option of being notified when the transaction ends, so that it can modify any message before it is sent.
- **Example**: If a user is forwarding incoming calls, GSM's call forward notification feature reminds the user that call forwarding is activated whenever the user originates a call. The feature does this by including a parameter in the Q.931 Call Proceeding message sent to the user. The feature triggers after basic call has processed the Send Call event, finds the Call Proceeding message built by OBC's Send Call event handler, and modifies it by adding the call forward notification parameter.

4.5 Feature Networking

- **Context**: You are developing a feature that must perform actions on both users in a call. Basic calls use *Separation of Call Halves*, so the feature's actions affect more than one processing context.
- **Problem**: How can the feature coordinate its actions across both user contexts?
- **Forces**: Each user could reside on a different switch. Even if the users reside on the same switch, their features run in separate contexts: OBC and TBC. Because OBC and TBC may be running on separate processors, they can only communicate by exchanging CIP messages.
- **Solution**: Split the feature in half, just like basic call. Run one half on OBC and the conjugate half on TBC. To support communication between the two halves, define a generic feature parameter that can be added to any message in the call interworking protocol. Each feature parameter has an identifier to specify the feature associated with the parameter. The rest of the parameter's contents are feature-specific. The two halves of the feature can then communicate by exchanging feature parameters that are appended to standard CIP messages.
- **Rationale**: This pattern extends *Separation of Call Halves* to apply to features as well as basic calls. Reasons for this separation are given in *Separation of Call Halves*. Moreover, the only way that a feature can affect a user *on another switch* is to add a parameter to an inter-switch trunk protocol such as ISUP. Because the feature must be designed in this way when it operates *between* switches, it is reasonable to use the same design *within* a switch. Otherwise the feature will be implemented in two ways, which will increase development costs.
- **Resulting context**: A networked feature usually triggers on OBC, after which *Hold Outgoing Messages* allows it to add a feature parameter to a CIP message. This parameter can cause the feature's conjugate AFE to be created on the remote half-call, in this case TBC.
- **Example**: Call waiting originating (CWO) allows an originating user to impose call waiting on a busy user, even if the busy user does not subscribe to call waiting. After OBC builds the CIP Initial Address Message (IAM), CWO adds its feature parameter to this message. If the called user is busy, CWO's conjugate (on TBC) triggers if it finds CWO's parameter in the IAM. If TBC is a trunk, the parameter has no effect. However, let us say that ISUP was enhanced to support a CWO parameter. In this case, CWO's conjugate would wait for ISUP TBC to build the outgoing ISUP Initial Address Message. It would then include the CWO parameter in the ISUP IAM if that parameter was also present in the CIP IAM.

4.6 Call Multiplexer

- **Context**: You are using *Separation of Basic Calls and Features* and are developing a feature that affects more than one of a user's calls. A conference call is an example of such a feature.
- **Problem**: How can the feature coordinate its actions across more than one of the user's calls?
- **Forces**: If a feature affects only one of a user's calls, it runs in the context of that call. But if a feature affects many calls, it cannot run within a single call because that call can disappear while other calls remain active. This occurs, for example, if one member of a conference drops out. Moreover, calls only communicate using asynchronous messages sent and received by PSMs, and so the feature would have to run on all of the calls to coordinate its actions between them. This would be unwieldy because each instance of the feature would need a separate PSM to communicate with every other instance. Only in this way, for example, could all conference members be notified when one member dropped out.

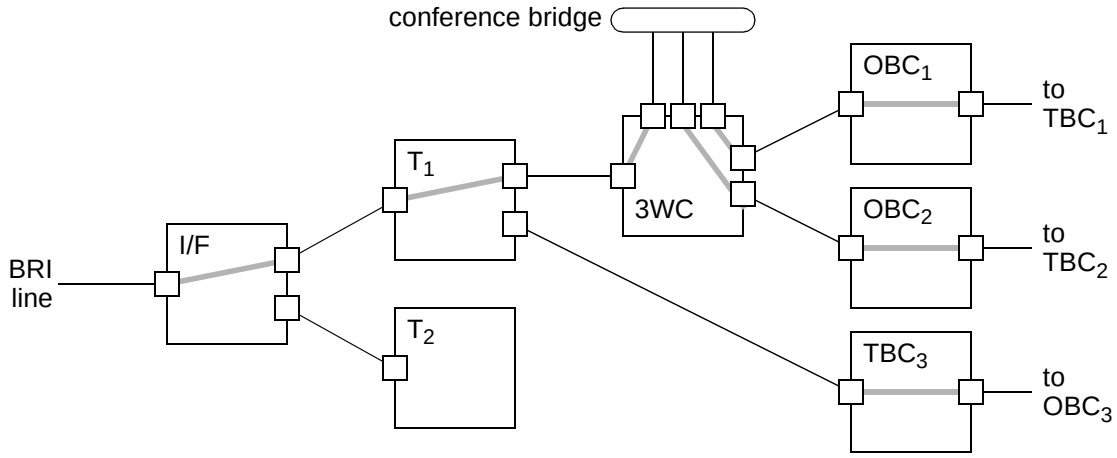

Figure 5. A BRI line, showing its interface multiplexer, I/F, which is serving two terminal multiplexers, T_1 and T_2. T_2 is idle, whereas T_1 has three calls, OBC_1, OBC_2, and TBC_3. OBC_1 and OBC_2 have been conferenced by the three-way calling (3WC) multiplexer. Speech connections are shown as internal lines that connect PSMs belonging to the same multiplexer or basic call. TBC_3 is on hold, which is illustrated by the absence of a speech connection within T_1 that leads from I/F to TBC_3. The diagram also shows, for the first time, the use of PSMs to interact with a service circuit that is internal to the switch, in this case a conference bridge.

- **Solution**: Allow a feature to run as a *peer* of the calls that it affects. Insert the feature between the user interface and those calls. The feature will have one user-side PSM to communicate with the user interface, and also a number of network-side PSMs, each of which will communicate with one of the user's calls.
- **Rationale**: This simplifies coordination of multiple calls by allowing them to be associated. The association is provided by the feature that requires it. The feature runs in a separate context that subtends precisely those calls that the feature affects.
- **Resulting context**: Asynchronous messaging between a multiplexer and the calls that it subtends introduces race conditions that complicate the design of multiplexers. This issue is addressed in *Run to Completion (extended)*.
- **Examples**: Three-way calling and call waiting use this pattern to coordinate the actions of two calls. Thus, when three-way calling receives a hookflash from a POTS line, it can easily coordinate the actions required by both of its calls, either by conferencing them or by releasing the second call and returning to the simple configuration where only the original call remains.

  A terminal that supports independent calls uses a *terminal multiplexer* to manage those calls. This is the case for BRI lines and GSM mobiles. BRI lines also use an *interface multiplexer* to manage the terminals on a BRI interface. When a BRI interface multiplexer receives a message on its user side, it relays the message to the correct terminal multiplexer by looking at the message's terminal endpoint identifier. Similarly, a BRI terminal multiplexer relays incoming user-side messages to the correct call (OBC, TBC, or a three-way calling multiplexer) by looking at the message's call reference. Figure 5 shows various multiplexers that are serving a BRI interface.

4.7 Surrogate Call

- **Context**: Basic calls use *Separation of Call Halves* and support *FSM Observer*. You are developing a feature that redirects a call to a new destination.

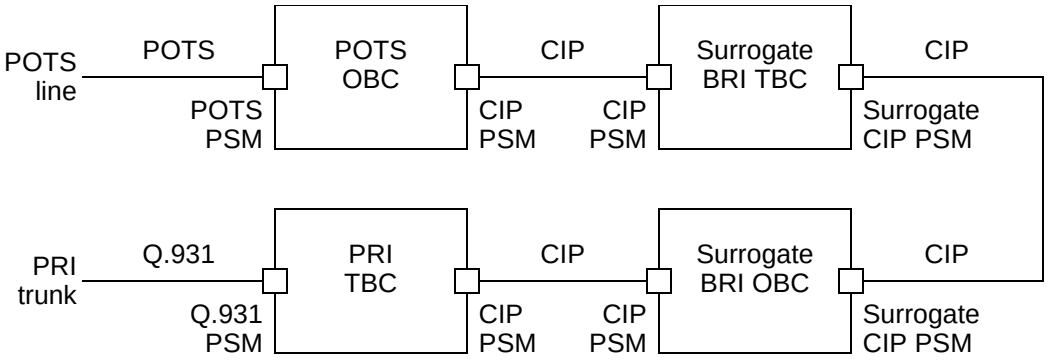

Figure 6. The configuration after a BRI line's call forwarding on no reply (CFNRy) feature has forwarded a call from a POTS line to a PRI trunk. The surrogate BRI TBC was originally a standard BRI TBC whose user-side PSM communicated with a terminal multiplexer. However, the BRI line was idled when the call was forwarded, so this multiplexer has disappeared. The forwarded leg was set up by the surrogate BRI OBC, and the two surrogate half-calls are now relaying CIP messages between the POTS line and PRI trunk. The initial message from the surrogate TBC to the surrogate OBC was a CIP Initial Address Message created by the CFNRy feature. It contained the information necessary to set up the forwarded leg, as well as an encapsulated Q.931 Setup message. The Setup message allows the surrogate OBC to reuse standard BRI OBC event handlers, which expect to see a Q.931 Setup message on the user-side PSM.

- **Problem**: How do you implement the feature?
- **Forces**: When a call from user *A* to user *B* is redirected to user *C*, the redirection feature runs on behalf of user *B*. The redirected part of the call, from *B* to *C*, is set up using *B*'s dial plan, is subject to *B*'s restrictions, such as not being allowed to make toll calls, and is charged to *B*.
- **Solution**: Allow a half-call to be set up on a user's behalf even if the user is not connected to the call. To support this, each half-call must be able to run in surrogate mode, in which it only performs logical actions on behalf of a user. It also relays CIP messages between the calls that have been joined (redirected) together instead of interworking those messages to the user's PSM, which does not exist in the call. The relaying of CIP messages occurs between two PSMs that replace the user-side PSMs that exist when the user is connected to the call.
- **Rationale**: This pattern directly follows from requirements that user *B*'s features remain part of the call. When *B* redirects a call from *A* to *C*, the result is *not* a direct call from *A* to *C*. The *connection* is from *A* to *C*, but there are two *calls*: *A* to *B*, and *B* to *C*. This is reflected in the fact that *B* pays for the *B* to *C* call, so *B*'s billing software must run on that call.
- **Resulting context**: Although a redirection feature usually relays an entire CIP message, this is not true for the Initial Address Message (IAM). Some IAM parameters describe end-to-end call characteristics, such as the need for echo suppression, whereas others describe leg-by-leg characteristics, such as the billing number and long distance service provider for a call leg. All CIP IAM parameters must be partitioned into those relayed during call redirection (end-to-end parameters) and those deleted and provided afresh (leg-by-leg parameters).
- **Examples**: Features such as call forwarding, call transfer, and call pickup use this pattern. Figure 6 shows the configuration that results after call forwarding on no reply redirects a call.

4.8 Call Distributor

- **Context**: You are developing a feature that presents a call to multiple destinations. *Surrogate Call* is available for redirecting a call to a single destination.
- **Problem**: How can the feature set up a separate call to each destination?

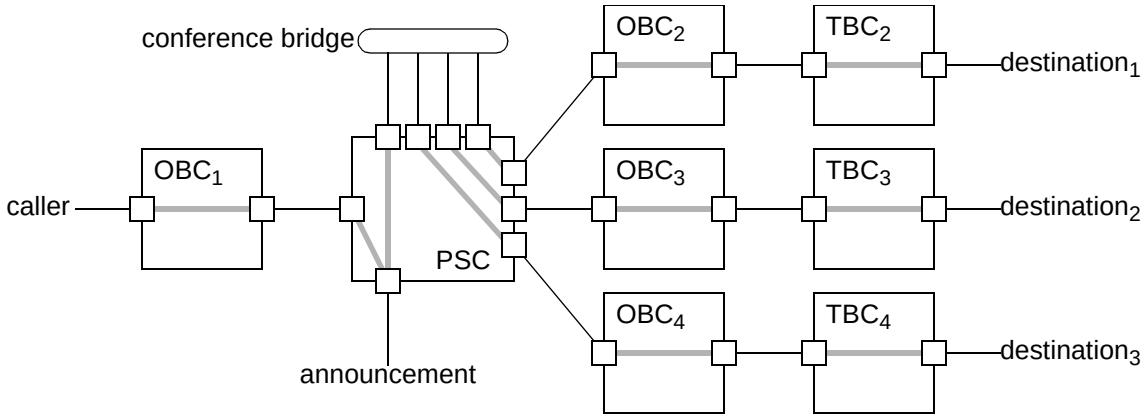

Figure 7. The use of *Call Distributor* by the preset conference (PSC) feature to present a call to three locations. PSC runs on a surrogate TBC and uses three surrogate PSMs to originate calls (OBC_2, OBC_3, and OBC_4) for the purpose of reaching the target destinations (TBC_2, TBC_3, and TBC_4). This is equivalent to simultaneous call forwarding: OBC_2, OBC_3, and OBC_4 are all surrogate OBCs associated with the user who subscribes to the PSC feature. Some destinations have not yet answered, and so PSC is playing an announcement to both the originator of OBC_1 and the conference, informing them that the conference will be established after all destinations have been given time to answer.

- **Forces**: Each destination has its own phone number but may be served by a different switch.
- **Solution**: Use a variant of *Call Multiplexer*. A multiplexer maps multiple calls onto a single *user-side* PSM. Here, however, it is necessary to map multiple calls, one to each destination, onto a single *network-side* PSM.
- **Rationale**: This pattern reuses OBC, which is capable of setting up a call to any destination. Because there are multiple destinations, a separate OBC instance is required to reach each one. The distributor runs as a surrogate TBC that hides the existence of these multiple OBCs from the original OBC that originated the call.
- **Resulting context**: Asynchronous messaging between a distributor and the calls that it subtends introduces race conditions that complicate the design of distributors. This issue is addressed in *Run to Completion (extended)*.
- **Examples**: The flexible alerting feature simultaneously forwards a call to multiple locations. The call is awarded to the first destination that answers, and other destinations are released. The preset conference feature is similar, except that any destination that answers is conferenced into the call. Figure 7 shows how preset conference presents a call to multiple destinations. The Session Initiation Protocol (SIP) defined for Internet telephony uses the term *forking proxy* to describe simultaneous forwarding.

4.9 Run to Completion (extended)

- **Context**: You are using *Run to Completion* and developing a feature that uses *Call Multiplexer* or *Call Distributor*.
- **Problem**: Race conditions occur between the feature and the calls that it subtends.
- **Forces**: Peer features (namely OBCs, TBCs, multiplexers, and distributors) communicate with asynchronous messages.
- **Solution**: Extend *Run to Completion* so that all features running *on behalf of the same user* run to completion. When one feature sends another feature a message, and both features are

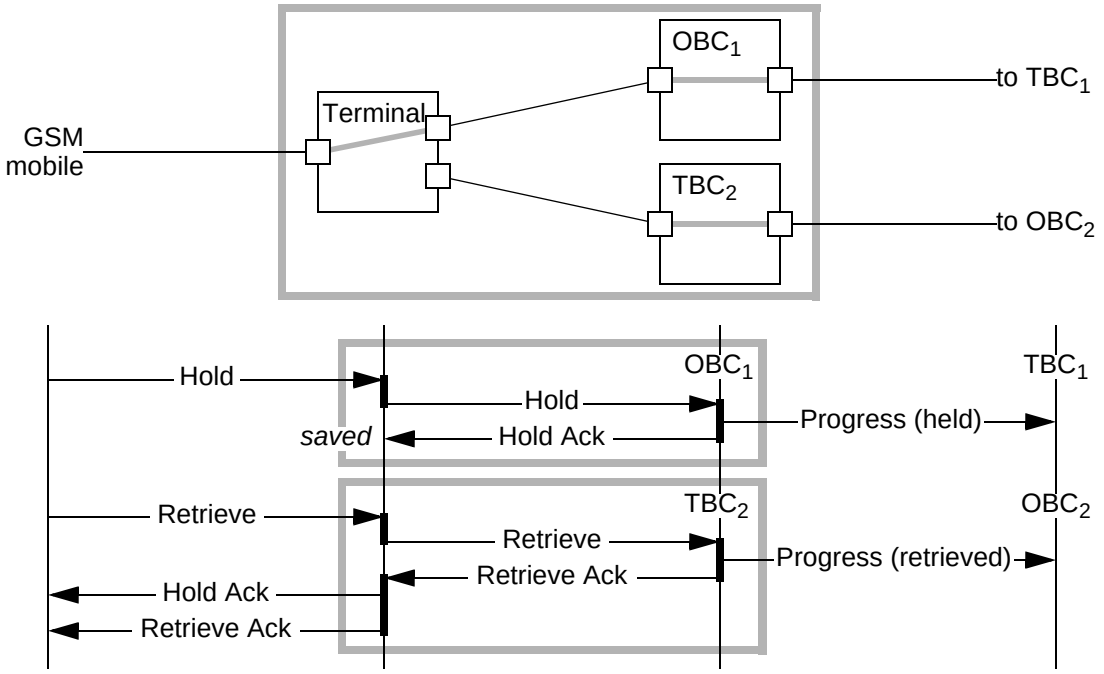

Figure 8. Run-to-completion with a single user context. The GSM terminal multiplexer and basic calls are running on behalf of the same user, so all messages between them are processed to completion before any external messages (from the user or the two remote call halves) are accepted. Only outgoing messages can cross the boundary of a shaded box while any message inside the box remains unprocessed.

running on behalf of the same user, place the message in a priority queue that is emptied prior to accepting any messages from outside the user's processing context.

- **Rationale**: This eliminates race conditions that make multiplexers and distributors that much more difficult to implement.
- **Examples**: All of the AFEs in Figure 5 run as an unpreemptable cluster. And in Figure 7, preset conference, OBC_2, OBC_3, and OBC_4 run as an unpreemptable cluster.

  GSM standards specify that when a mobile has more than one call, its active call may not be held until a message to retrieve another call arrives. Three peer features therefore cooperate to hold a call: a GSM terminal multiplexer and two basic calls, as shown in Figure 8. When the terminal multiplexer receives a Hold message, it relays this message to the call to be held. If the reply is a Hold Ack, it saves this message and waits for a Retrieve from the user. When the Retrieve arrives, the terminal multiplexer relays it to the call to be retrieved. If the reply is a Retrieve Ack, the terminal multiplexer sends the saved Hold Ack to the user and then relays the Retrieve Ack to the user as well. The message sequence that begins with receiving the Hold and that ends with saving the Hold Ack, as well as the one that begins with receiving the Retrieve and that ends with sending the final Retrieve Ack, occur within a single user context and therefore run unpreemptably. Messages from outside this context remain queued. This allows the three features to focus on the task at hand, without worrying about race conditions that would occur if messages from the user or the two remote parties were processed during this time.

4.10 Context Preservation

- **Context**: You are using *FSM Observer* and *Message Preservation* and are developing a feature that needs to interrupt a basic call transaction by sending a message to another processor and waiting for a response. The response will either cause the feature to change the call's default behavior or cause it to resume the interrupted transaction.
- **Problem**: How do you implement this requirement?
- **Forces**: When a feature interrupts a transaction, the current basic call event goes unprocessed. When the feature receives its response, this event must either be processed or discarded.

  An interruption always occurs before processing an event, but there are two scenarios to consider. If the feature is already active, the interruption occurs while traversing the basic call AFQ. But if the feature was just triggered by the event, the interruption occurs while traversing a PFQ. The interruption leaves the current message and event unprocessed.
- **Solution**: Extend *Message Preservation* so that the feature can save the current context. This consists of not only the context message, but also the current basic call event and the position in the AFQ or PFQ. When the response arrives, the feature can either discard or restore this context. In the former case, the feature usually reenters basic call with a new event. In the latter case, the feature uses a new event routing instruction, **resume**, which transparently resumes processing of the interrupted transaction. The feature's query-response operation is therefore performed asynchronously.
- **Rationale**: Asynchronous messaging is used because hardened real-time systems should avoid synchronous messaging. Synchronous messaging can cause deadlocks, as well as unacceptable latencies when timeouts occur. It is also inefficient from a scheduling perspective.

  The feature cannot use the **reenter** instruction to resume processing of the saved event because the AFQ or PFQ would then be traversed again, which could cause other features to retrigger. The feature cannot use the **pass** instruction because the saved message and event are no longer in context when the feature is processing the response to its query.
- **Examples**: The Intelligent Network service switching function often uses this pattern. It may, for example, query a Service Control Point (SCP) during call setup to determine if the call should be blocked. If the SCP responds with a Continue instruction, the pending event and message are restored so that call setup can resume. But if the SCP indicates that the call should be barred, the pending event and message are discarded prior to releasing the call.

  When a GSM mobile answers a waiting data call on which data interworking is required, the AFQ already contains the call waiting (CWT) and data interworking (IWF) features. CWT must react to the Local Answer event by ensuring that the radio channel is encoded for data, and IWF must prepare its rate adaption circuit. Before basic call can process the Local Answer event, CWT must receive confirmation that the radio channel is properly encoded, and IWF must receive confirmation that its rate adaption circuit is ready. Both CWT and IWF use *Context Preservation* when they interrupt processing of the Local Answer event. After CWT receives its confirmation, it returns the **resume** instruction, and IWF runs next. After IWF receives its confirmation, it also returns **resume**, and basic call finally processes the Local Answer event by enabling the connection and sending a CIP Answer Message to the originator.

5. Patterns for Feature Interactions

The patterns presented thus far allow basic calls and features to run as separate state machines. *FSM Observer*, *FSM Model*, and *Hold Outgoing Messages* allow features to modify basic call

without changing its software. *Call Multiplexer*, *Call Distributor*, and *Feature Networking* allow features to coordinate their actions across more than one call. But how can features interact when more than one is running? To keep features partitioned, generic techniques are required so that features do not constantly have to check for the presence of other features before deciding what to do. This is the topic of the patterns in this section.

5.1 Parameter Database

- **Context**: You are implementing two features. The first feature changes a basic call parameter that must be subsequently accessed by the second feature. The two features and basic call conform to *Separation of Basic Calls and Features*.
- **Problem**: How do you implement this interaction while keeping basic call and the two features unaware of each other?
- **Forces**: Features must sometimes read and/or write basic call parameters, and some parameters can be read/written by multiple features running on the same call. If a feature changes a parameter, other features may later need to use the new value. A feature cannot always use *Hold Outgoing Messages* because another one may need to access a modified parameter *before* the feature publishes it by adding it to a message.
- **Solution**: Store the parameter in a database that is associated with basic call.
- **Resulting context**: Basic call also uses the database, typically when it builds the CIP Initial Address Message. The database also allows a feature to place a parameter in the database and remove itself from the call if it has nothing to do except wait to modify a message that is not built until much later.
- **Rationale**: Associating the parameter with basic call ensures that it is accessible to basic call and all of its features.
- **Examples**: Calling number delivery blocking uses *Parameter Database* when it sets the presentation indicator in the calling address, which is one of the parameters in the database. If the Intelligent Network service switching function is subsequently triggered, it will include the correct presentation indicator when it sends the calling address to the Service Control Point. The database also contains a stack of all analyzed called addresses, such as those that triggered the local number portability or toll-free features. This allows OBC to include them as generic addresses when it builds the CIP Initial Address Message. In most cases, however, parameter history need not be retained: usually only the most recent version of a parameter is required.

5.2 Connection Observer

- **Context**: You are implementing a feature that must monitor what a user's bearer channel is transmitting and receiving. Other features, however, can modify the user's connection. The system uses both *Separation of Basic Calls and Features* and *Separation of Protocols*.
- **Problem**: How can the feature monitor the user's connection without being aware of other features that affect the connection?
- **Forces**: Connections can be modelled as associations between PSMs, in the manner of Figure 5. When a PSM's signalling channel is also associated with a bearer channel, the PSM acts as an endpoint or relay point for connections to and from its underlying bearer channel. Basic call creates a connection association between its user-side and network-side PSMs. Other features, however, may cause either of these PSMs to receive from another PSM instead.

- **Solution**: To monitor what the user is transmitting, the feature listens to the user-side PSM. This PSM must also allow the feature to register as a connection observer, which means that the feature will be informed whenever the PSM's incoming connection disappears or changes to another PSM. The feature can then listen to the conjugate PSM on the new incoming connection, if any, which allows the feature to monitor precisely what the user is receiving.
- **Rationale**: Connection events should not be added to the basic call model because event handlers become unnecessarily complicated if they are split up to factor out connection work. Moreover, connection events are orthogonal to basic call events: in GSM, for example, bearer channel allocation can occur in one of three different OBC states. It is therefore necessary to provide an observable connection model in addition to the basic call model.
- **Example**: GSM's call intercept feature provides wiretaps of GSM calls, and its data interworking feature inserts a rate adaption circuit in the basic call connection. When the circuit is inserted, the user-side PSM receives from the circuit, not from the network-side PSM. *Connection Observer* allows call intercept to monitor precisely what the user is receiving, even when the data interworking feature is running on the call.
- **Related patterns**: The *Observer* pattern is described in [1].

5.3 Event Handler Visitor

- **Context**: You are using *FSM Observer* and are implementing a feature that must perform some work that cannot be accomplished by raising a basic call event and reentering basic call to reuse one of its event handlers. Logically, however, the work should change the basic call state.
- **Problem**: How can the feature change the basic call state?
- **Forces**: A feature is not allowed to change the basic call state directly because this would prevent other features from observing the basic call state transition. When the basic call state changes, other features may need to perform additional work associated with the state transition.
- **Solution**: Have the feature raise a generic "Force Transition" event and reenter basic call. The parameter to this event is an event handler that is supplied by the feature but executed in basic call's processing context. If the event handler updates the basic call state, the state machine framework informs features of the state transition, just as if basic call had updated the state itself.
- **Rationale**: Basic call cannot anticipate all of the state transitions that future features may require. The work associated with a new state transition is feature-specific, so it should be performed by the feature that requires it. However, other features must be able to observe the work as if it had been performed by basic call.
- **Example**: POTS TBC presents a call by applying power ringing to the subscriber's line. Call waiting, however, presents a call by injecting a tone into the speech path, so it cannot use TBC's Present Call event handler to present the waiting call. Call waiting must present the call using one of its own event handlers. However, call waiting should use *Event Handler Visitor* to do this so that the Call Presented state transition is processed after the call is presented. This state transition must be observable because it triggers the no-answer timer for call forwarding on no reply. If the state transition is hidden, this timer will not be started, and the waiting call will not be forwarded if the subscriber ignores it.
- **Related patterns**: The *Visitor* pattern is described in [1].

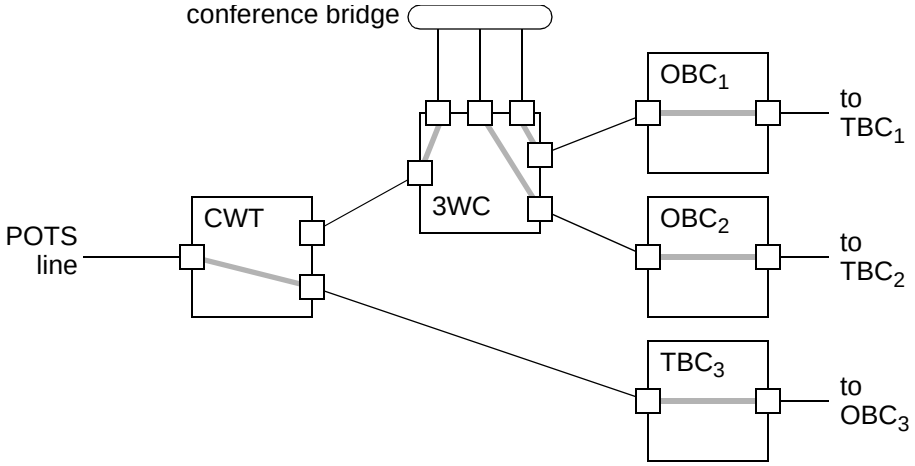

Figure 9. Resolving the hookflash contention between call waiting (CWT) and three-way calling (3WC) by giving CWT a higher priority than 3WC. Once CWT's multiplexer is inserted, it intercepts hookflashes, which no longer control 3WC. If the user goes onhook, CWT re-alerts the user, and so 3WC (if it has already conferenced its calls, as shown here) must be prepared for an offhook (reanswer), an event that it would not normally see.

5.4 Multiplexer Chain of Responsibility

- **Context**: You are using *Call Multiplexer* to design multiplexer features that can simultaneously run on behalf of the same user.
- **Problem**: How do you resolve interactions between the multiplexers?
- **Forces**: The order in which multiplexers are arranged determines their signalling and connection interactions. If multiplexer *A* is closer to the user interface than multiplexer *B*, then *A* reacts to user-to-network messages before *B*, and *B* reacts to network-to-user messages before *A*. Since *A* subtends *B*, *A* can hold all of the calls subtended by *B*, whereas *B* can only hold a subset of the calls subtended by *A*.
- **Solution**: Assign a priority to each peer AFE (multiplexers and basic call) that can run within a user's processing context. A higher priority means that the AFE is placed closer to the user interface. An interface multiplexer has the highest priority when processing user-to-network messages, and OBC and TBC have the lowest priority.
- **Rationale**: Priorities cause peer AFEs to be configured in a consistent order, regardless of the order in which they are created. This is important because it produces consistent behavior. Ideally the behavior is what is required by the specifications!
- **Examples**: *Multiplexer Chain of Responsibility* resolves the interaction—prohibited by most specifications—that arises when a waiting call arrives after a user has initiated three-way calling. If the call waiting multiplexer is given priority over the three-way calling multiplexer, as shown in Figure 9, call waiting will process user messages first. A hookflash will switch between the three-way call and the waiting call, and an onhook will release the active call and cause the remaining call (even if it is a three-way call) to re-alert the user. *Multiplexer Chain of Responsibility* also trivially resolves the situation in which call waiting occurs at both ends of a call. Here, the two call waiting features run in separate user contexts and therefore do not interfere with one another. Although this can lead to interesting interactions, such as two lines re-alerting each other, this is a natural outcome when features are distributed, whether between switches or within a switch.

- **Related patterns**: The *Chain of Responsibility* pattern is described in [1].

5.5 PFE Chain of Responsibility

- **Context**: You are using *FSM Observer*. More than one feature can trigger on the same event.
- **Problem**: How do you ensure that the proper feature triggers in reaction to the event?
- **Forces**: If more than one feature can trigger, the specifications will state which feature is more desirable. In most cases, the features will also be mutually incompatible.
- **Solution**: Assign a priority to each PFE that registers in the same PFQ. Arrange the PFEs in order of their priority. After a PFE triggers its feature, it can prevent further traversal of the PFQ by returning the **consume** event routing instruction, which ends the current transaction.
- **Rationale**: Priorities allow features to trigger in the order required by specifications. When a feature overrides basic call behavior, it should be able to prevent other features from triggering if the call is no longer in the state that those features would expect.
- **Examples**: *PFE Chain of Responsibility* allows call waiting to have priority over call forwarding on busy at the Local Busy event. If call waiting is possible, call forwarding on busy does not occur. *PFE Chain of Responsibility* also allows terminating call screening to have priority over the component of automatic recall that remembers the last calling address. This prevents automatic recall from returning a call to an address from which the user rejects calls. However, terminating call screening should probably react to the Authorize Termination event, and automatic recall to the Termination Authorized state transition. They would then appear in separate PFQs, and automatic recall would never have the opportunity to remember the address of a rejected caller.

5.6 Initiation Observer

- **Context**: You are using *FSM Observer*. A feature is active when an incompatible feature is triggered. Both features are associated with the same user.
- **Problem**: How do you prevent the incompatible features from running together?
- **Forces**: Specifications often state that two features are incompatible, so that one cannot be used in the presence of the other. There are various reasons for this. Perhaps the features would react to an event in ways that would interfere with each other, as in the infamous hookflash contention between call waiting and three-way calling. Or perhaps one feature would violate a policy associated with the other, such as call waiting holding an emergency call to answer the waiting call.
- **Solution**: When feature initiation is requested, notify any active features and allow them to deny the initiation request.
- **Rationale**: This maximizes the simultaneous use of compatible features while preventing the simultaneous use of incompatible features. Knowledge of feature incompatibilities must exist *somewhere* in the software. *Initiation Observer* distributes it among the features, in much the same way that it is distributed among many feature specification documents. Each feature knows which features it must deny.
- **Examples**: *Initiation Observer* is used frequently. It allows emergency calling to deny call hold. More broadly, it allows the active feature to follow up in any appropriate manner. For example, call forward programming can reenter OBC to collect and analyze the forward-to address. It can then reject the programming attempt if it observes the initiation of originating call screening, abbreviated dialling, or emergency calling. Similarly, automatic callback can

determine if the target user is idle by reentering TBC to perform a trial termination. Automatic callback monitors TBC's progress, hoping to eventually observe the Facility Selected state transition. If automatic callback instead observes the initiation of call forwarding unconditional, it can deny the automatic callback request. If it observes the initiation of call waiting, it can treat the user as busy or idle, depending on what is required by the specifications.

Initiation Observer is also used when inserting a multiplexer. The multiplexers and half-calls running on behalf of user *A* form a tree whose root is *A* and whose leaves are *A*'s half-calls. To insert a new multiplexer, *M*, permission is required from (a) the half-calls and multiplexers that appear in the subtree below *M*, and (b) the multiplexers that appear on the path between *M* and *A*. This allows call waiting to be denied by emergency calling and three-way calling.

5.7 Sibling Observer

- **Context**: You are implementing two features. Specifications call for a certain behavior when both features are active. The first feature can use *Initiation Observer* to detect the creation of the second feature.
- **Problem**: How can the features interact to satisfy the specifications?
- **Forces**: Compatible features should run independently. However, specifications sometimes violate this principle. If possible, such specifications should be ignored. But this is not always feasible.
- **Solution**: Support event-based communication between features when it is needed. Implement the interaction between the features with a protocol that is specific to the features involved. If feature *A* must observe feature *B*, then *B* raises an event for *A* whenever *B* does something that *A* needs to observe. The event contains any information required by *A*, and the framework delivers the event to *A* if *A* is active. After *A* processes the event, it has the option of returning a new event as a response to *B*.
- **Rationale**: The time required to define an inter-feature protocol discourages designers from creating arbitrary couplings to handle interactions. However, the mechanism is generic enough to handle situations in which features truly need to communicate.
- **Example**: GSM's call intercept (CI) feature uses *FSM Model* and *Connection Observer* to monitor basic call and the user's connection. However, specifications also require CI to monitor other features, such as the Intelligent Network service switching function (SSF). CI and SSF use *Sibling Observer* to define a protocol that allows SSF to inform CI of each trigger that results in a query to an Intelligent Network Service Control Point (SCP), as well as the SCP's response.

5.8 Feature Networking (reused)

- **Context**: You are implementing two features that need to interact when they are running on behalf of different users. *Feature Networking* is available when one feature needs to perform actions on both users in a call.
- **Problem**: How can the interaction be supported across the two user contexts?
- **Solution**: Use the previously described *Feature Networking* pattern. The first feature places a feature-specific parameter in a CIP message and the second feature reacts to this parameter.
- **Examples**: CIP defines a presentation indicator in its calling address parameter so that a terminator's calling number delivery feature can avoid displaying a number that was marked private by the originator's calling number delivery blocking feature. CIP also defines a

redirection counter that limits the number of times that a call can be forwarded, to prevent infinite call forwarding loops or excessively long call forwarding chains. Each call forwarding feature also sends a CIP Progress Message to the originator, to indicate that forwarding has occurred and to provide the address to which the call was forwarded. If the originating call screening feature reacts to the Remote Progress event, it can look for this parameter to prevent the originator from being forwarded to an address that would have been blocked by his originating call screening feature if he had dialled it directly.

Feature Networking can also resolve a very annoying interaction. If a conferee subscribes to music on hold, music disrupts the conference when the conferee puts it on hold. A conference feature should therefore use a CIP Progress Message to inform each conferee that he is part of a conference, and music on hold should not trigger when the user is a conferee.

6. Experiences with the Pattern Language

All the patterns in this article are currently used in Nortel Networks' GSM Mobile Switching Center (MSC). A framework that supports these patterns was used to reengineer this MSC's call processing software over a period of 18 months. During that time, 550K lines of software were developed at a cost of 55 designer-years, a significant improvement in productivity. The MSC's features are completely decoupled from basic calls and from each other. Other Nortel design groups have also adopted the pattern language described in this article.

Some of the patterns in this article are also used in other products. Nortel's DMS-100 uses *Run to Completion, Separation of Basic Calls and Features*, and *Call Multiplexer*. Lucent's 5ESS uses *Separation of Call Halves* [19], which is also important in Intelligent Network standards, as are *FSM Observer* and *FSM Model*. There are undoubtedly many other cases where these and other patterns described in this article are used, but they are difficult to uncover because of the lack of substantive literature on call processing software architectures.

7. Acknowledgments

Much of this article is taken from *An Overview of Selected Call Processing Patterns* [20] and *A Pattern Language of Feature Interaction* [21]. Tony Blake, Sam Christie, Dennis DeBruler, Jim Doble, Bob Hanmer, Neil Harrison, Allen Hopley, Gerard Meszaros, Martin Nair, Linda Rising, and the anonymous reviewers from the 1998 Feature Interaction Workshop all offered comments that helped to improve the original articles on which this one was based.

8. Glossary

AFE	Active FSM Element: an object that contains an FSM's per-call data
AFQ	Active FSM Queue: the queue containing the AFEs that are modifying basic call behavior
BRI	Basic Rate Interface: an ISDN line that uses the Q.931 call control protocol
CIP	Call Interworking Protocol: the call control protocol used between OBC and TBC
FSM	Finite State Machine
GSM	Global System for Mobile Communications
ISUP	ISDN User Part: a call control protocol used by CCS7 trunks
network side	the direction towards the network
OBC	Originating Basic Call: an AFE used to originate a call
PFE	Passive FSM Element: an object that observes a basic call event or state transition to trigger the creation of its feature's AFE

PFQ	Passive FSM Queue: a queue containing the PFEs that are interested in the same event or state transition
POTS	Plain Old Telephone Service
PRI	Primary Rate Interface: an ISDN trunk that uses the Q.931 call control protocol
protocol	a set of signals (message types), parameters, and rules that specify the order in which signals may be sent and received and the parameters that are mandatory or optional for each signal
PSM	Protocol State Machine: an object that implements an instance of a protocol dialogue
Q.931	the ISDN call control protocol
TBC	Terminating Basic Call: an AFE used to receive a call
user side	the direction towards the user

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-63361-2.

[2] J. Coplien and D. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1995. ISBN 0-201-60734-4.

[3] J. Vlissides, J. Coplien, and N. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, Massachusetts, 1996. ISBN 0-201-89527-7.

[4] R. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, Massachusetts, 1998. ISBN 0-201-31011-2.

[5] N. Harrison, B. Foote, and H. Rohnert, editors. *Pattern Languages of Program Design 4*. Addison-Wesley, Reading, Massachusetts, 1999. ISBN 0-201-43304-4.

[6] R. Martin. “Discovering Patterns in Existing Applications.” In [2], pages 383-389.

[7] A. Ran. “MOODS: Models for Object-Oriented Design of State.” In [3], pages 129-131.

[8] P. Dyson and B. Anderson. “State Patterns.” In [4], pages 138-140.

[9] D. DeBruler. “A Generative Pattern Language for Distributed Computing.” In [2], pages 69-89.

[10] D. Schmidt and C. Cranor. “Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-Structured Concurrent I/O.” In [3], pages 437-459.

[11] G. Meszaros. “Pattern: Half-object + Protocol (HOPP).” In [2], pages 129-132.

[12] *Distributed Functional Plane for IN CS-2*. Recommendation Q.1224, ITU-T, September 1997.

[13] *AIN 0.2 Switching Systems Generic Requirements*. GR-1298-CORE, Bellcore, September 1997.

[14] *Specifications of Signalling System No. 7*. Recommendations Q.761 and Q.763, ITU-T, September 1997.

[15] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, New York, 1994. ISBN 0-471-59917-4.

[16] B. Foote and J. Yoder. “Big Ball of Mud.” In [5], pages 653-692.

[17] B. Selic. “Recursive Control.” In [4], pages 147-161.

[18] S. Yacoub and H. Ammar. “A Pattern Language of Statecharts.” Presented at the 1998 Pattern Languages of Programs Conference, Monticello, Illinois, August 1998. http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P22.pdf.

[19] L.G. Anderson, C.H. Bowers, D.L. Carney, J.J. Kulzer, and W.W. Parker. “Distributed Systems Tradeoffs.” In *Proceedings of the 12th International Switching Symposium*, pages 26-33. IEEE Service Center, Piscataway, New Jersey, 1987.

[20] G. Utas. “An Overview of Selected Call Processing Patterns.” In *IEEE Communications*, pages 64-69, April 1999.

[21] G. Utas. “A Pattern Language of Feature Interaction.” In K. Kimbler and W. Bouma, editors, *Feature Interactions in Telecommunications and Software Systems V*, pages 98-114. IOS Press, Amsterdam, September 1998. ISBN 90-5199-431-1.