Message Queues
Three Patterns for Asynchronous Information Exchange*

Wolfgang Herzner, ARC Seibersdorf research
2444 Seibersdorf, Austria
wolfgang.herzner@arcs.ac.at

Abstract

Information exchange between concurrent processing elements like threads or tasks is one of the fundamental issues in information processing systems. In many cases, this information transfer needs to occur asynchronously, i.e. the 'consumer' must be enabled to receive the information at some later point in time than the 'producer' provides it. Of course, this may apply to data transfer within a sequential processing element as well. Messages and their intermediate storage in queues are one of the most common solutions to this problem. This paper describes three patterns, addressing different combinations of requirements: the simple FIFO QUEUE, the SELECTABLE-MESSAGE QUEUE, and the SMART-MESSAGE QUEUE.

Keywords:

Asynchronous information transfer, messaging, message queue, FIFO

1 Introduction

Communication between concurrent processing elements like threads or tasks is one of the fundamental issues in information processing systems. At a closer look, communication is typically realized as a series of unidirectional information transfer steps. For instance, a client sends a request for a document to a server, which sends the requested document back – or an error message in the case the document is not available. Or consider a simple procedure call, where the caller may hand over arguments to the procedure, which again may return results at completion.

Whenever information or data may be or has to be interchanged asynchronously between different elements of a software system, this is usually done with some form of buffering mechanism. Actually, there is such a rich variety of corresponding implementations due to different internal structures and provided features, that it is not easy to lay bare the underlying patterns.

This paper is an attempt to isolate three of those patterns: the presumably most simple one, that is the FIFO QUEUE, which simply allows to retrieve information in the same order as added to the queue, and two extensions of it. The SELECTABLE-MESSAGE QUEUE allows to select messages when retrieving them, while the SMART-MESSAGE QUEUE allows to exploit relationships of new information with already buffered information elements when queueing it. A further distinction between both is that in general SELECTABLE-MESSAGE QUEUE places the major responsibility into the queue, while SMART-MESSAGE QUEUE places the major responsi-

bility into the messages. So, both patterns can be considered as complementary. Figure 1 shows the relationships between all three patterns.

It should be noted, that all described patterns are not reliable by themselves, i.e. if the elements of a system realizing them fail, the information currently controlled by these elements will probably get lost.

In this paper, any information or request to be interchanged between processing elements is called 'message'.

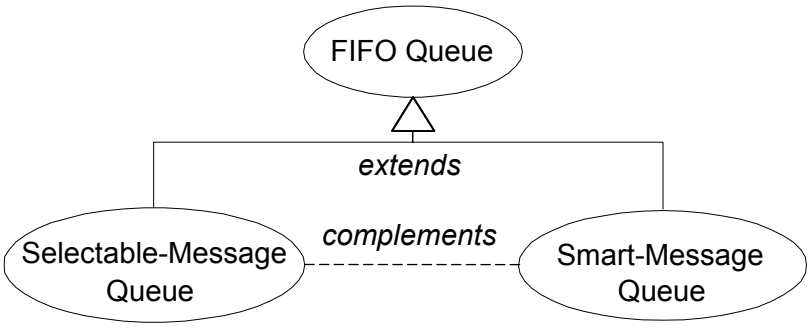

Fig. 1: Relationship among described patterns

The description of each pattern follows a common structure. However, because "context" and "forces" are essentially the same for all described patterns, they are summarized immediately below.

1.1 Context (for all Described Patterns)

Applications, where several modules need to exchange information with each other, and where 'receiving' modules may or need to process the information at a later point in time than 'sending' modules are providing it.

1.2 Forces (for all Described Patterns)

Number of Producers and Consumers: there may be several producers and consumers of messages.

Over/Underflow: production of messages may be too high or too slow for the consumers.

Concurrency: if production and consumption of messages may happen concurrently, they must not disturb each other, and correctness of queued messages must be maintained.

2 FIFO Queue

Use FIFO QUEUE whenever an arbitrary number of pieces of information has to be asynchronously passed from some processing element(s) to another or several others, and processing order has to be maintained.

(‘FIFO’ stands for “first in, first out”.)

2.1 Examples

Drawing commands for a graphical display are usually generated in bursts, while the display handler can process them only at a fairly regular rate.

Similarly, interactive applications may not be able to process *user input events* at the time they are generated.

In *image compression* (e.g. for converting pure pixel data into JPEG format) several compression steps (e.g. DCT conversion and Huffmann encoding [ISO/IEC94]) have usually to be applied in sequence to regular image tiles, but parallelization shall be exploited to increase throughput.

A document shall be transferred over a *packet switching network*, but is larger than the allowed packet size. It has therefore to be sent in pieces, from which the recipient must be able to reconstruct the document. Here, packets represent the messages.

2.2 Problem

The generation of information elements like requests or events must be decoupled from their processing, but the order in which they are processed must be the same in which they have been generated.

2.3 Solution

Basically, provide a buffering mechanism which allows to add messages to the buffer, which allows to remove messages from the buffer in same order as added, and which takes care for over- and underflow, as well as it takes care for concurrency, if needed, i.e. which realizes the "first in / first out (FIFO)" concept and solves the forces listed under 1.2. How this can be achieved is described in the following.

Details

In an object-oriented environment, a class like `FifoQueue` could provide this functionality:

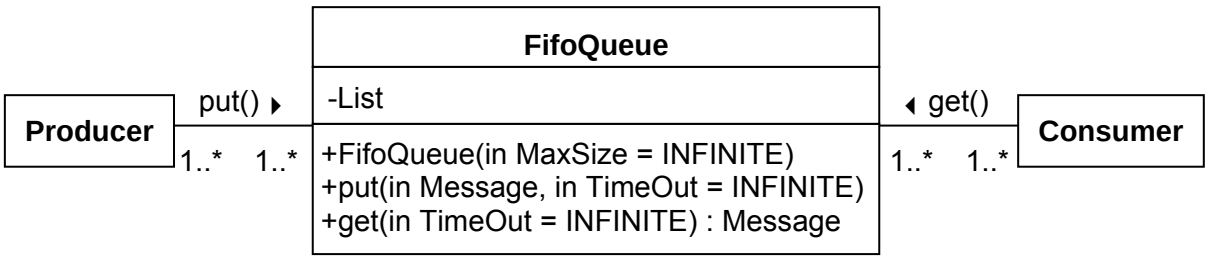

Fig. 2: `FifoQueue` – general class diagram

Figure 2 displays only elements of `FifoQueue` (in UML-notation, without types) relevant for that pattern. Please note that producers and consumers are not necessarily distinct; here, roles are depicted rather than specific objects.

`FifoQueue()`: when creating an instance of that class, the maximum number of messages buffered by it at any time can be defined by `MaxSize`. If omitted, an unbounded empty FIFO-queue is created.

`List` serves for buffering queued messages, and is intialized to contain no messages.

A `FifoQueue` is called 'empty', if it has no messages buffered.

A `FifoQueue` is called 'full', if `MaxSize` was not set to `INFINITE`, and the number of currently buffered messages equals `MaxSize`.

A `FifoQueue` is called ‚bounded', if `MaxSize` was not set to `INFINITE`.

`put()`: appends the provided message at the end of `List`.

If `List` is full, `put()` waits (thus, blocking the caller) until at least one message has been removed from the internal list. However, if `TimeOut` is not `INFINITE`, it will wait not longer than specified by this argument.

If other `put()`- or `get()`-operations are currently performed or pending, it will block until all of them have been completed, with one exception: if the internal list is empty, it will complete disregarding any pending `get()`-operations. This keeps the generation order of messages, because no `put()` can pass another. See clause „Synchronization“ below for a possible implementation.

`get()`: removes the message at the begin of `List`, and returns it to the caller.

If `List` is empty, it waits (thus, blocking the caller) until at least one message has been added to the internal list. However, if `TimeOut` is not `INFINITE`, it will wait not longer than specified by this argument.

If other `put()`- or `get()`-operations are currently performed or pending, it will block until all of them have been completed, with one exception: if the internal list is full, it will complete disregarding any pending `put()`-operations. See clause „Synchronization“ below for a possible implementation.

2.4 Discussion

Error Handling

Unsuccessful activations of `put()` and `get()`, due to full or empty internal buffer and encountered timeout, must be reported to the callers, as well as further internal problems like insufficient memory. How this is achieved, is not prescribed by this pattern. The given method signatures suggest some bypassing mechanism like exceptions; however, it is also possible to add an error indicator to these methods, or indicating a `get()`-timeout by returning no message.

Internal Message Buffering

An important issue of FIFO QUEUE is the structure of the internal buffer (`List` of class `FifoQueue`). The simple first-in/first-out rule suggests a single-linked list. If update rates are high, though, this solution may become inefficient, if it requires creation and destruction of a list entry per `put()`- and `get()`-call, respectively. To avoid this, unused entries could be stored in another list, where `put()` takes entries from as long as available and allocates new

ones only if this list is empty. In a bounded `FifoQueue`, this may work out well, while in an unbounded queue, it may waste memory in applications with rare message bursts.

For bounded `FifoQueues`, another solution could be more efficient, namely a fixed-size array of `MaxSize+1` elements (e.g. pointers or references to messages), where two indices identify the first used (`First`) and the first free (`Free`) element, as illustrated in figure 3.

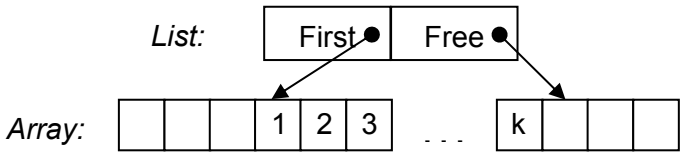

Fig. 3: Bounded (message) queue as array

The array is used in round-robin mode. ("Round-robin" means that usage of the array continues with its first element after its last element has been reached). `put()` enters the new message at `Free` and progresses it by one, `get()` removes from `First` and progresses it. The array is empty when `First=Free`, it is full when `First=[Free+1]`, where `[]` denotes the round-robin behavior in cases when `First` points at the last and `Free` at the first array element. `First` and `Free` must never pass each other.

Almost the same can be achieved with a closed single-linked list, where the last element is connected to the first. This frees from the mentioned special case, but on the cost of more memory consumption and execution overhead due to address resolution instead of index incrementation.

Of course, both (closed) single-linked list and array are just examples for possible implementations, and should be understood as suggestions only. See variant "Safe Message Passing" for alternatives of storing the message data themselves.

Synchronization

If several producers and consumers shall be allowed to access a `FifoQueue` concurrently, synchronization is needed. Following requirements have to be considered:

a) Accesses to the queue, i.e. all `put()`- and `get()`-calls, have to be executed such that the queue's consistency is maintained.

b) If the internal list is empty, `get()`-calls must wait, but `put()`-calls must be executed; analogously, if the internal list is full, `put()`-calls must wait, but `get()`-calls must be executed.

c) The order of `put()`-calls and `get()`-calls, respectively, has to be maintained; that means that no `put()` must pass any pending `put()`, as no `get()` must pass any pending `get()`.

d) No call shall wait longer than defined by its argument `TimeOut`.

The following code example meets these requeirements. We first assume that there exists a Mutex class with following behaviour:

`lock( timeOut ): Bool`

Waits until any other pending `lock()`-call has been satisfied, or the time-out arrived. If

the latter occurred, it returns FALSE, otherwise it lock the mutex for the caller and returns TRUE.

`unlock()`
Sets the mutex to unlocked. If any `lock()`-call is pending, it informs the longest waiting call that the mutex is free.

Then, we need an Event class with following behaviour:

`set()` and `reset()`
set and reset the the event, respectively.

`wait( timeOut = 0 ): Bool`
returns TRUE if the event is set within time-out, otherwise FALSE. Of course, it returns TRUE immediately if the event is already set.

`FifoQueue` gets further private members:

```
Mutex m_MtxPut;    // to queue put()-calls
Mutex m_MtxGet;    // to queue get()-calls
Mutex m_MtxList;   // to synchronize access to message list
Event m_NotFull;   // set as long as list is not full
Event m_NotEmpty; // set as long as list is not empty
```

FifoQueue.**put(Message, TimeOut)** would then work as follows:

```
Time Now, StartTime = now(); // to consider TimeOut correctly
// ensure that any pending put()-calls are completed:
if (m_MtxPut.lock( TimeOut ) == TRUE)
{ // now we are the oldest pending put()-call
  // reduce timeOut by already consumed time if not infinite:
  if (TimeOut != INFINITE)
  { Now = now();  TimeOut -= Now-StartTime;  StartTime = Now; }
  // ensure that queue is not full:
  if (m_NotFull.wait( TimeOut ) == TRUE)
  { // again, reduce timeOut by already consumed time:
    if (TimeOut != INFINITE)
    { Now = now();  TimeOut -= Now-StartTime;  StartTime = Now;
    }
    // ensure that we have exclusive access to internal data:
    if (m_MtxList.lock( TimeOut ) == TRUE)
    { // append message to m_List
      // reset notfull-event, if m_List became full:
      if (m_List is full)
        m_NotFull.reset();
      // in any case, we have to set the notempty-event:
      m_NotEmpty.set();
      // free the list-mutex:
      m_MtxList.unlock();
    }
    ... // else-branches and m_MtxPut.unlock()
```

`FifoQueue.get()` would be symmetrically coded.

In programming languages providing destructors like C++, the *Scoped Lock Idiom* [Schmidt++00] could be applied, which would free from the responsibility to explicitely unlock mutexes and simplify coding of else-branches.

Rendezvous Mechanism

If `MaxSize` is set to `0` in the constructor, a synchronous rendezvous mechanism between a consumer and a producer can be realized, when the synchronization mechanism described above is adapted in the sense that it allows `put()` to wait until `get()` is called or vice versa, and then hands over the message directly from producer to consumer.

Non-Object-oriented Environments

In non-object-oriented environments, message queues are created as data structures. Functions like `FifoQueue_put( FifoQueue, Message, TimeOut )` would provide the corresponding functionality, taking the addressed `FifoQueue` as argument, but work otherwise like the methods described before.

2.5 Variants

Safe Message Passing

An issue to be decided is how messages are passed over between `FifoQueue` and its clients. An efficient and safe approach is to simply hand over references by simultaneously transferring ownership, for assuring that the reference provider cannot use it any more. If the used programming language does not provide such a feature directly, it can be realized by reference objects which store essentially store both the pointer to the referenced object and to the owner. For instance, the standard C++ library provides `auto_ptr` for such functionality.

However, if ownership cannot be assured, simply copy references is dangerous, because `put()` cannot guarantee that the producer will not use the reference anymore. In this case it is safer – though more time and memory consuming – to create a copy of the message provided with `put()`, and return this copy by `get()`. If all messages are of equal or an acceptable maximum size, the array in figure 3 could even be used to directly contain the messages, which avoids the repeated memory allocation for the copies. Then, however, it is recommended that `get()` copies messages into a buffer provided by the consumer; thus, this approach trades memory allocation for copying.

See "Request/Release" below for a further alternative.

Request/Release

In situations where messages may be rather large but of a known maximum size (e.g. images or video frames), and unnecessary copying should be avoided, it can be helpful to provide prepared buffers by the `FifoQueue` to the producers, into which they place their messages directly. These buffers are provided to the consumers as well, which then, however, have to release them for reuse. Therefore, the `FifoQueue`-interface is modified to ("buffer" stands for "message buffer"):

```
request( in timeOut: time_t = INFINITE ): Buffer
```

Returns a (reference to a) free buffer to the caller or nothing, if timeout encountered, and no buffer free.

`put( in message: Buffer, in timeOut: time_t = INFINITE )`

Just takes over buffer and appends it at end of internal list

`get( in timeOut: time_t = INFINITE ): Buffer`

Removes (reference to) buffer from begin of internal list and returns it or nothing, if internal list remains empty during timeOut

`release( in message: Buffer )`

Takes (reference to) provided message buffer, and adds it to free buffer list.

Producers perform the following sequence of operations:

- receive a free message buffer with request(),
- copy/place the message into the received buffer,
- queue the buffer with `put().`

Consumers perform the following sequence of operations:

- receive a queued message with get(),
- process it,
- free the buffer with release().

A downside of this variant is that the `FifoQueue` relies on proper operation of its clients. If clients fail or forget to call `put()` or `release()`, it looses its buffers. Or, they could reuse buffers after they returned them by `put()` or `release()`. Even worse, clients could destroy the message buffers, if the chosen implementation doesn't prohibit them from doing this.

2.6 Examples Resolved

Drawing commands are put by the application into a FIFO QUEUE of appropriately dimensioned size, from which the display handler fetches and processes them.

U*ser input events* are put into a FIFO QUEUE by the input devices in the order of generation, from which the application fetches and processes them..

In *image compression*, a pipeline [Shaw96] can be established by realizing the data flow between the processing elements (computation filters) by means of FIFO QUEUEs, which will enhance throughput in most cases, because it helps to compensate varying processing speeds on different image parts (tiles).

Not only the nodes of a *packet switching network* can exploit FIFO QUEUES for interim packet buffering, but also the gates between applications and network can profit from communication via FIFOQUEUEs, both for maintaining packet ordering and for compensating varying transmission times (jitter).

2.7 Consequences

Upsides

Decoupling of producers and consumers. Clients producing messages don't need to know their consumers and vice versa; they all only need to know the FIFO QUEUE.

Asynchronous processing. Producers don't need to wait until consumers are ready to process their messages. Instead, they can continue without being constrained by the processing speed of consumers.

Automatic wait on extreme loads. Since consumers wait on producers if no messages are available, and producers wait on consumers if FIFO QUEUE is bounded and full, producers and consumers cannot drift more apart than the maximum size of the queue.

Downsides

Complexity. Compared with direct method invocation, FIFO QUEUE are relatively complex, especially if concurrency and under/overflow has to be treated.

2.8 Related Patterns

An earlier description of a *FIFO-queue* pattern can be found in [Beck97] as a Smalltalk-idiom. An *Ordered Collection* [Beck97] can be used to implement the list of queued messages.

Woolf and Brown describe in [Woolf++02] a comprehensive patterns system for messaging in the EAI (Enterprise Application Integration) context. Various forms of messaging mechanisms are contained, including queues, although on a more abstract and domain specific level. An updated version can be found under [Woolf++03].

For controlling access to `put()` and `get()` in multi-threaded environments, use synchronization patterns like *Scoped Locking Idiom, Thread-Safe Interface* or *Monitor Object* [Schmidt++00], or *Hierarchical Locking* [McKenney96] if locking the whole FIFO QUEUE per call causes a too significant performance loss.

With FIFO QUEUE, each message can be retrieved exactly once, i.e. one consumer per message. If the same message shall be retrieved by several consumers, patterns like *Publisher/Subscriber* [Buschmann++96] could be used.

One way for `put()` to learn to know whether it is the only owner of a reference to the handed-over message, is the usage of counted reference patterns like the *Public Countability* (in [Henney01]), which allow clients to get the number of current references to an object.

2.9 Known Uses

FIFO QUEUES are at the heart of many applications and their components. Here, only a few are identified explicitly.

The *Mailbox service* of OpenVMS™ is a many-to-one FIFO QUEUE. Each process may open a mailbox, where it receives messages from several senders.

The point-to-point form of the Java Message Service JMS (e.g. [Giotta++01]), if used without message selectors.

The Digital Video System DVS [Herzner++97] is a distributed surveillance system which has been developed for recording, displaying, archiving, and retrieval of video images from

up to thousand cameras. DVS uses FIFO QUEUES for messages like commands and events, both bounded and unbounded, and the request/release-variant for video frames and sequences.

3 Selectable-Message Queue

Use SELECTABLE-MESSAGE QUEUE whenever information has to be asynchronously passed from some processing element(s) to another or others, and processing order has to be controlled by the receiving processing element(s).

3.1 Examples

Processing requests with different priorities. Requests with higher priorities shall be considered earlier, but all queued requests have to be processed.

Printer queues. Smaller print jobs may be printed before larger ones.

I/O-requests for a disk may be processed in an order which minimizes head moves.

A management computer for a *sheet-metal cutting machine* collecting punch orders for various sheet widths, but can only process those which match the sheet width currently prepared.

3.2 Problem

The order in which messages are added to a FIFO QUEUE may not be optimal for their later processing. That means that consumers want to receive the oldest message fitting their actual demands best, rather than always the oldest message in the queue. However, all queued messages have finally to be processsed.

3.3 Solution

Provide a SELECTABLE-MESSAGE QUEUE as an extension of a FIFO QUEUE that allows to add attributed messages, and which allows to retrieve the oldest message with a certain attribute. How this can be achieved is described in the following.

Details

The structure of SELECTABLE-MESSAGE QUEUE is similar to that of FIFO QUEUE, with somewhat different signatures of `put()` and `get()`:

Figure 4 displays only elements of the `SelMsgQueue` class relevant for that pattern. As with FIFO QUEUE, producers and consumers are not necessarily distinct; here, roles are depicted rather than specific objects.

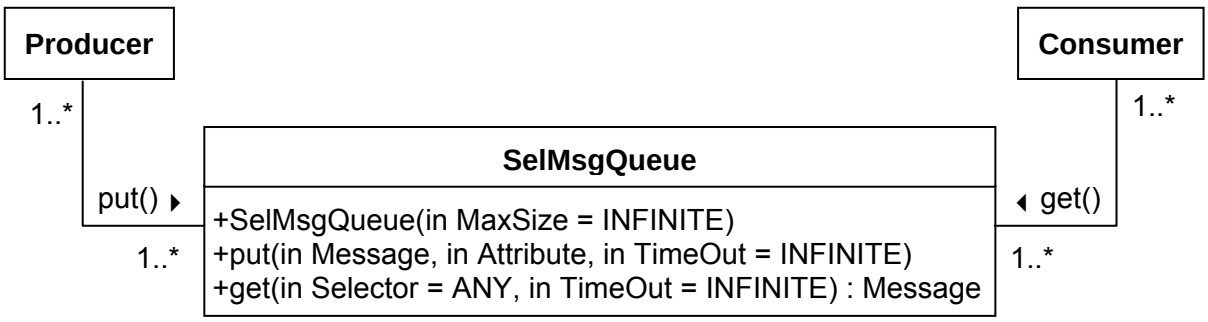

Fig. 4: `SelMsgQueue` – general class diagram

`put()`: appends the provided message at the end of the internal list of buffered messages. In addition, it keeps its attribute so that it can be exploited by later `get()`-calls.

`get()`: beginning with the oldest entry, it checks if the selector meets the attribute of the message. If so, it removes that message from the list and returns it; otherwise it repeats the check with the next entry in the list. If no attribute fits the selector, no message is returned. The selector value `ANY` selects the oldest message without regarding the attribute.
Please note that the structure of `Selector` is not defined; see the discussion below for several variants and aspects.

3.4 Discussion, Variants

'Categorisable' Attributes

If the attribute can assume only a (small) number of distinguishable values, it may be more efficient to store messages in a more efficient way than in one list, for example, in an own list per category. Then, getting the oldest message of a certain category needs only one access to the appropriate list.

Ordering at Insertion Time

If the attribute is well ordered, and `get()` will always request for a message with an extreme attribute value (i.e. only selector values `MIN`, `MAX`, or `ANY` are possible), messages can be inserted in the right order by `put()` already. For instance, the attribute is priority, and `get()` will always ask for the message with the highest priority, or size is the attribute and `get()` will always request for the smallest message (e.g. from a printer queue). If simple sequential structures are used for internal queue representation, as described in FIFO QUEUE, and N is the current number of queued messages, then inserting messages requires *N/2* comparisons on average, while `get()` would always require *N* comparisons.

Of course, more sophisticated storing structures like trees can reduce the number of comparisons significantly, usually at some cost of maintenance overhead.

Composed Attributes

Sometimes, single-valued attributes are not sufficient. In this case, attributes may be composed of more elementary ones. For instance, print jobs may be selected by increasing size,

but should not be delayed too long, so size and queuing time together could build the composed attribute for such messages.

Hints for Selector Implementation

If consumers may be interested in both extreme values of a certain attribute, `Selector` should allow for presenting this, for instance by including `MIN` and `MAX` in its domain range. (Note that in this case "Ordering at Insertion Time" may still be very helpful.)

Or, consumers could ask for attribute value within a certain range. Then, `Selector` should allow to specify such ranges.

Similarly, in the case of composed attributes, `Selector` should allow to specify which attributes the consumer wants to address, and in what combination.

In some applications, it could be helpful to provide automatic fallback to ANY if otherwise no message would be returned, potentially reducing the number of `get()`-calls and improving atomicity.

Arbitrary Removal

If category based storing of queued messages is not possible, as with ordering at insertion time, implementation alternatives discussed with FIFO QUEUE are not feasible, because these do not support removal at arbitrary positions well. Instead, a double-linked list is more appropriate.

Delegation of Selection to the Messages

Possibly, attributes are realized as private to messages. In this case, `get()` must be enabled to delegate the selection to the messages; that means the messages must provide some method

```
eval( in Selector ): Bool
```

which tests whether the owner's attributes fits to the given selector.

Forgotten Messages

There is a risk that individual messages will never become selected. A simple way to reduce this risk is to use `ANY` occasionally as `Selector`. However, if responsibility for avoiding message starvation should be assigned to the message queue (rather than left over to consumers), then it could keep some priority measure which grows with message age, and gradually overrules any message attribute.

This is risky in some other sense, though, because it may cause to return messages on `get( Selector )` which do not fit the selector's value. An exception is when priority is already the message attribute, and the message with highest priority is selected for retrieval; then a common solution is to increase the priority of queued messages at regular intervals.

3.5 Examples Resolved

Priority-driven requests processing. If not too many priorities have to be distinguished, the implementation as described under "'Categorisable' Attributes" appears feasible. Whenever a new request can be processed, the oldest entry from the non-empty queue of highest priority is taken. If usage of internal priority-specific queues is unfavorable, the "Ordering

at Insertion Time" variant can be used. In both cases, `get()` would always be called with `MAX`.

Printer queues. In general, the "Ordering at Insertion Time" variant would be used here. However. this could result in never printing large jobs. To avoid this, a composed attribute, consisting of e.g. size and queuing time, could be used. Then, at some times the consumer task should ask for the oldest queued job, i.e. calling `get()` with a `selector` like "MIN(QUEUEING_TIME)".

I/O-requests on disks. If the message attribute is physical disk location (sector etc.), the disc controller may call `get()` with a set of ranges for the individual attribute elements which matches the current head position best.

Sheet-metal cutting machine. `get()` needs simply to be called asking for equality with the current sheet width. SELECTABLE-MESSAGE QUEUE will probably be realized with applying "categorizable attribute".
This may be a simplification, though, because a matching message still could request for a shape not fitting into the sheet left over from previous cuts. Rather, requests for the same sheet width would be collected, and together placed optimally on the available sheet to minimize material losses. Still, SELECTABLE-MESSAGE QUEUE could be used to collect all requests.

3.6 Consequences

Since SELECTABLE-MESSAGE QUEUE is an extension of FIFO QUEUE, its consequences apply to SELECTABLE-MESSAGE QUEUE as well. In addition, two more consequences should be considered.

Upside

Atomic message retrieval. Getting the (oldest) message fitting certain requirements needs only one `get()`-call, compared to solutions where consumers browse through the queued messages and test them by themselves. Either, a more complex interface is needed (first, consumers have to retrieve messages (for testing) without their removal, then they must indicate which message to remove); or they remove messages and re-queue those which are not needed at the moment, which causes at least overhead, if not more serious problems due to the new queuing order. Even more problems may arise if several concurrent consumers exist.

Downside

Starvation risk. The pattern itself does not guarantee that a queued message is not ignored permanently by `get()`. See discussion about “Forgotten Messages” for possible solutions.

3.7 Related Patterns

The SELECTABLE-MESSAGE QUEUE pattern is an extension of the FIFO QUEUE pattern. In contrast to SMART-MESSAGE QUEUE, SELECTABLE-MESSAGE QUEUE provides this extension as more flexibility on the retrieval side (rather than on the production side), and still forces to fetch all queued messages.

The *Message Selector* of [Woolf++02/03] can easily be implemented with SELECTABLE-MESSAGE QUEUE, if message type is denoted by the `Attribute`. Senders then simply set this argument to the message's type, and receivers set `Selector` in `get()` to the message type they are interested in.

The `ActivationList` of the *Half-sync/Half-async* pattern [Schmidt++00] delegates selection to the queued method requests by selecting the oldest entry of which the `can_run()` method returns `TRUE`.

To some extent, the *Pipes and Filter* pattern [Buschmann++96] could use SELECTABLE-MESSAGE QUEUEs, because pipes may need it for buffering results of a processing step, and filter will distribute them to several following processing steps operating in parallel.

3.8 Known Uses

System V [Sobell94] provides C-functions `msgsnd( …, long msgType, … )` and `msgrcv( …, long msgType, … )`, where `msgrcv()` will return the oldest message with given `msgType`. `msgType=0` will return the oldest entry without any filtering.

Under Windows™ (by Microsoft), `GetMessage()` allows to specify a range of message types (i.e. unsigned integers) to be received from the ***windows message queue*.**

The point-to-point form of the Java Message Service JMS, if used with message selectors.

4 Smart-Message Queue

Use SMART-MESSAGE QUEUE whenever information has to be asynchronously passed from some processing element to another, and new information units may be related to already passed but not yet processed ones.

4.1 Examples

Consider some control system, for instance for network operating, where a *low priority task displays the current situation* on some monitor, and some file transfer process reports progression state periodically into a queue which serves for collecting all messages to that display. On entering such a message, any older message of the same type still in the queue is outdated, and should not be displayed anymore.

In the same control system, several *messages of same kind* could be collected into a single summary message if applications semantics allows for it; e.g., N messages "event X occurred" could be replaced by a message "event X occurred N times (within L time units)".

Similarly, if an *informative message about some transient effect* is still in the queue when the corresponding 'off'-message arrives, both could be removed from the queue without any further replacement.

A FIFO QUEUE is used to collect messages from a node A to be transmitted to another node B. With faulty transport media it is possible that a *message transfer fails* unrecognized by A. Rather than waiting for some confirmation from B after each transmission, which can cause significant performance loss, messages could be sent without wait but kept by A until confirmation arrives. On failure response, transmission is restarted with the oldest not yet deleted message. Similarly, B could request for retransmission due to internal reasons.

4.2 Problems

Appending a new message to the queue may result in removal or modification of already buffered messages, according to interdependencies between the new and buffered messages.

4.3 Solution

Provide a SMART-MESSAGE QUEUE as an extension of a FIFO QUEUE with following additional features:

Design messages so that for each message object, it can (efficiently) be determined if and how it interrelates with other messages, which could require that the 'type' of a message can easily be determined.

`put()` tests if the new message has some relationship with one of the queued messages and treats both, if such a relationship is found, according to that relationship; otherwise, it appends the new message to the internal list.

How this can be achieved is described in the following.

Details

The structure of the SMART-MESSAGE QUEUE pattern differs from that of FIFO QUEUE mainly in including message objects as well:

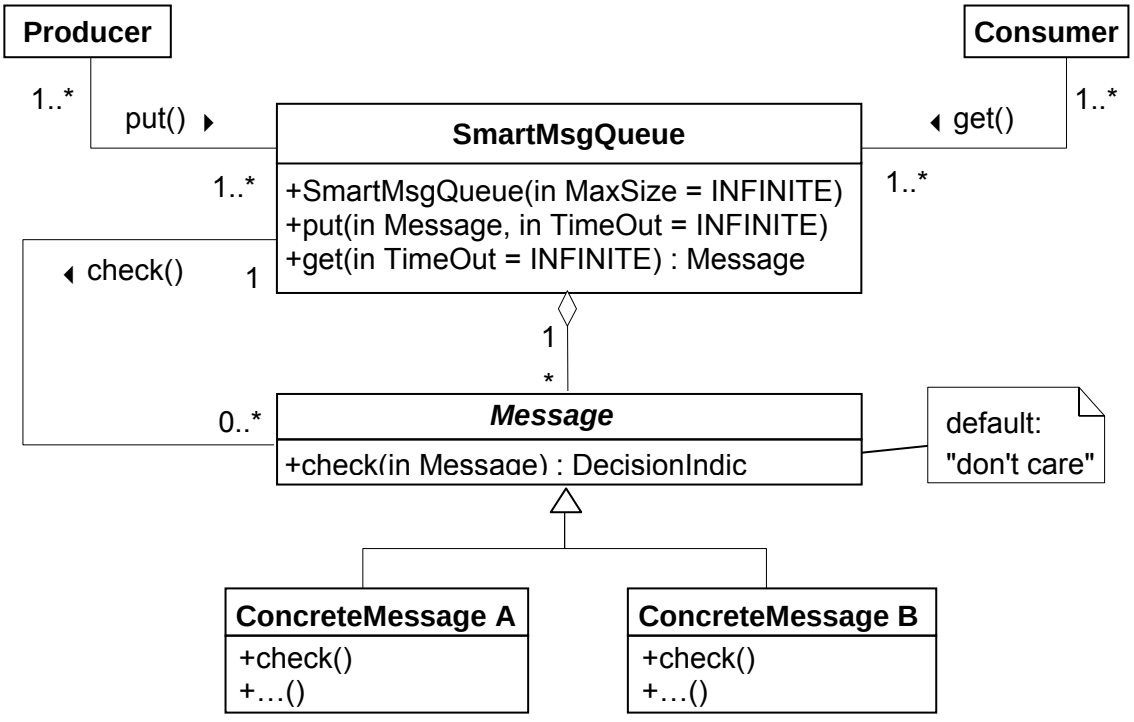

Fig. 5: `SmartMsgQueue` – class diagram

`check()` takes a (reference to a) message object, determines its relationship to itself, possibly using further information, as indicated by "`...()`" in the concrete message classes, and re-

turns a decision indicator as listed in the following table, where texts are formulated from point of view of that message object of which the `check()`-method has been activated.

Decision Indicators	Meaning	Possible Reasons
don't care	no interrelation detected	- the type of the new message is unrelated to mine
ignore	ignore the new message	- the new message is of my type, and repetition is not necessary - of my type, and I have taken over its values
queue	append new message to queue	- the new message is of my type, but both are needed - and the new message is not of my type, but I know that it is needed
ignore and delete	ignore the new message, and delete myself	- the new message neutralizes myself (e.g. it confirms my successful processing by the recipient)
queue and delete	append new message to queue, and delete myself	- the new message overrides myself, but it shall not pass other entries

Tab. 1: SmartMsgQueue – decision indicators

Please note that the decision indicators described in table 1 are only quite typical in our experience, but they are not necessarily the only ones to be used. Implementers of the SMART-MESSAGE QUEUE pattern may use their own set of decision indicators, if necessary.

A subclass doesn't need to overwrite `check()` if the default behavior (returning don't care) is sufficient.

`put()` performs a loop over its entries list, until either another decision indicator than don't care is returned, or the whole list has been traversed. In the first case, it executes the indicated decision; in the latter case, it appends the new message at the end of its list.

4.4 Discussion

Location of Inter-message Relationship Evaluation

In the described solution, the responsibility to evaluate inter-message relationships is placed on the messages themselves for two reasons: first, they 'know' best their relationships (or should at least), and second, when adding new concrete message classes, in general only their `check()`-methods have to address the new relationships, leaving older concrete message classes, and the SMART-MESSAGE QUEUE in particular, unchanged. For instance, if a message pair "set signal X" and "reset signal X" are added, only the `check()`-method of the former has to be coded to return ignore and delete when it is called with a "reset signal X" message. However, see the variants-clause below for alternatives.

Note that `check()` represents the *Strategy* pattern [Gamma++95], directly embedded into the message class-hierarchy.

Message Types and Identifiers

How a message evaluates its relationship with another message, is not specified by this pattern. In environments which provide runtime type information, this feature can be used; in others some different mechanism has to be implemented. For instance, some method like `type()` could be provided by all classes, returning a unique class identifier, presumably of type String. Or, if the *Reflection* pattern [Buschmann++96] has been applied, it can be exploited to get the type of a message object. Sometimes, however, type information will not be sufficient, but also access to certain subclass elements needed, e.g. to distinguish between messages of same type, but for different objects. Again, how this is achieved, is left unspecified by the SMART-MESSAGE QUEUE pattern.

Decision Indicators

Also, the SMART-MESSAGE QUEUE pattern leaves open how the decision indicators are represented. Again, this is left open to choose the best way according to the implementation environment, be it as enumeration or something else.

Arbitrary Removal

As with the SELECTABLE-MESSAGE QUEUE pattern, the implementations proposed in the FIFO QUEUE pattern will not be feasible here. Instead, a double-linked list is more appropriate.

Queue Size Limits and Other fifo queue Aspects

If the queue size is bounded, its overflow may occur more likely if `get()`-calls have the `keep`-flag set. But besides that, its behavior is the same as described for FIFO QUEUE.

Evaluation Order

In which order the `check()`-methods of already queued messages are called by `put()`, depends on the application and, possibly, on the new message. If, for instance, messages can be combined or neutralized, but certain requests shall not be bypassed by others, then from-newest-to-oldest-entry order appears to be preferable, because this does not violate creation time order.

For instance, if display messages containing statistical messages should only be combined if queued in sequence, i.e. without any other message between them, then a new message needs only to be checked by the most recent buffered message.

4.5 Variants

Evaluation Support by Smart-Message Queue

The last discussion about evaluation order raises a problem with the given solution, because the SMART-MESSAGE QUEUE will ask all buffered messages rather than only the most recent one, and the others don't know their relative distance to the new message.

One approach to solve this problem is to provide `check()` with the necessary information, either through additional arguments, or by allowing it to ask for appropriate information

on demand. Since the majority of `check()`-calls will not need the additional information, the second alternative appears to be more efficient.

Evaluation by Smart-Message Queue

Another approach to solve this problem is to evaluate inter-message relationships by the SMART-MESSAGE QUEUE itself rather than by the messages. Of course, in this case the SMART-MESSAGE QUEUE object must be able to get information about message types and possibly further message elements. If the set of concrete message classes is rather stable, this appears to be a feasible approach, because SMART-MESSAGE QUEUE's implementation needs not to be modified too frequently. It also could support maintainability, because the evaluation code will then likely to be concentrated at one place.

Actually, this was the first form of the pattern encountered by the author, and it has already been named "Smart Queue" by the software engineers using it. So, initially this variant gave name to the whole pattern. However, to both reflect the primary solution of this pattern better and to emphasize its relationship to the SELECTABLE-MESSAGE QUEUE pattern, it has finally been renamed to SMART-MESSAGE QUEUE.

Resend Control and Fault Tolerance

For allowing to resend messages when e.g. transmission failed after fetching them from the queue, `SmartMsgQueue` is extended by two public methods:

`getAndKeep( in TimeOut = INFINITE )` behaves essentially like `get()`, but only marks the returned message as received, rather than removing it from the queue.

`reset()` ensures that oldest (marked as) received but not yet deleted message becomes the oldest not yet received message; i.e. the next `get()`-call will return the oldest message in the list, and it will continue from this message.

The decision indicator ignore and delete can then be used to get rid of a fetched but not yet deleted message, namely by providing `put()` with a message that 'tells' such a kept message that it becomes obsolete now.

Some care has to be taken, if a message's `check()`-method needs to consider whether the message has alrady been retrieved. For instance, informational "on"-messages of a certain status (e.g. "light on") should not removed by corresponding "off"-messages if already retrieved, because otherwise the consumer would not be informed that the status has changed ("light off"). See variant "Evaluation Support by Smart-Message Queue" for a possible solution.

Expiration Time

Sometimes, messages may have a limited time span of validity or relevance. If expiration of this time simply means that the message is not needed anymore (rather than a deadline has been missed, which usually implies that some exception handling is to be triggered), then SMART-MESSAGE QUEUE could be modified as follows (because in this case it is sufficient to check for a message's expiration as soon at it is used). First, a virtual public method `isExpired()` is added to `Message`, which returns `TRUE` if it has an expiration time and this has been passed, otherwise `FALSE`. Second, both `put()` and `get()` invoke `isExpired()` on each queued message before `check()` or returning it, respectively, and remove that entry if `TRUE` has been returned. Of course, `getAndKeep()` and `reset()` work accordingly.

4.6 Examples Resolved

File copying process reports progression rate faster than low priority task can display them: any already queued progression state message returns the decision indicator queue and delete.

Repeated "event X occurred" messages: if the "Evaluation Support by Smart-Message Queue" variant has been realized, and the most recently queued message of same type is also the most recently queued one at all: it increments its own occurrence counter, and returns the decision indicator ignore. Otherwise, it must return `queue`.

"Off"-message arrives while corresponding "on"-message still queued: the latter returns ignore and delete.

Provision for safe transmission over faulty medium: using the "Resend Control and Fault Tolerance" variant, where consumers always call `getAndKeep()`. On retrieval of an acknowledge message, it is provided to the `SmartMsgQueue` by `put()`, which will cause the corresponding, still queued message (which should be the oldest in the queue) to return ignore and delete. However, on retrieval of a "transmission error" message, `reset()` is called, causing to repeat transmission of the not yet removed messages.

4.7 Consequences

Upsides

Administration tasks, which are based on mutual inter-message relationships, can be performed without the need for further mechanisms besides the SMART-MESSAGE QUEUE and the `check()`-methods.

Maintainability. Putting the evaluation code into the message implementations (by means of their `check()`-methods), both avoids that the SMART-MESSAGE QUEUE implementation has to be updated, whenever new concrete message classes are implemented, and makes forgetting to implement the new evaluation code more unlikely, because it is placed in the same source module as the other implementation of the new classes.

Efficiency. Elimination of obsolete messages before they are processed may improve efficiency of the whole application.

Fault-tolerance support. SMART-MESSAGE QUEUE offers some support for fault-tolerance (see faulty transmission example).

Downsides

Some *performance overhead* may result if `check()`s execute slow or return (almost) always don't care.

Only mutual relationships supported. SMART-MESSAGE QUEUE does not support consideration of relationships among more than two messages. It is not possible, for instance, that a new message (e.g. "remove all drawing commands") removes more than one already stored message. In such cases, more complex patterns than SMART-QUEUE are needed.

Abuse risk. There is some potential danger due to ill-coded or malicious `check()`-methods, which could be considered as a risk introduced by this pattern. For instance, some "bad-

minded" producer could add a message, which returns ignore on all messages that producer wants to suppress.

4.8 Related Patterns

SMART-MESSAGE QUEUE is an extension of FIFO QUEUE, and it may be combined with SELECTABLE-MESSAGE QUEUE.

The *Merge Compatible Events* pattern [Wake++96] is a SMART-MESSAGE QUEUE application based on the *Event Queue* pattern described in the same contribution. Similarly, the *Five Minutes of no Escalation* pattern of a set of fault-tolerant telecommunication system patterns [Adams++96] – which has later been adapted in the *Input and Output Pattern Language* [Hanmer++99] – can efficiently be realized using the SMART-MESSAGE QUEUE pattern, like several other patterns described there (e.g. *George Washington is Still Dead*, *Bottom Line*).

4.9 Known Uses

Microsoft's *windows message queue* optimizes certain messages. For instance, `WM_PAINT` messages are queued only once, even if several calls to `UpdateWindow()` or `RedrawWindow()` would generate several `WM_PAINT` messages.

In DVS, for each recording unit a central SMART-MESSAGE QUEUE is held, containing recording and other control commands. Since it is mainly used for compensating network problems (e.g. due to transient overload), the 'fault-tolerant' variant is applied.

In event-driven, fault-tolerant systems, as e.g. for telecommunication, patterns and pattern languages have been identified and already described in literature, which can and have actually been implemented by means of SMART-MESSAGE QUEUEs. See also related patterns before.

5 Acknowledgements

I owe a lot to my EuroPLoP'03 shepherd, Uwe Zdun. His comments helped not only to improve several technical aspects of the paper, but especially helped to elaborate its structure.

An earlier version of the SMART-MESSAGE QUEUE pattern has been presented at the OOPSLA'02 pattern writers workshop „Patterns in Distributed Real-Time Embedded (DRE) Systems" [Herzner++02], where the author received invaluable inputs, especially from Chris Gill (Washington Univ.) and Lonnie Welch (Ohio Univ.). I want to express me deep gratitude to them.

6 References

[Adams++96] Adams, M., Coplien, J., Gamoke, R., Hanmer, R., Keeve, F., Nicodemus, K.; "Fault-Tolerant Telecommunication System Patterns"; in *Pattern Languages of Program Design 2 (PLOPD2)*, pp.549-562; Addison-Wesley, 1996; ISBN 0-201-89527-7

[Beck97] Beck, K.; *Smalltalk Best Practice Patterns;* Prentice Hall, 1997

[Buschmann++96] Buschmann, F., Meunier, R., Rohnert, H.; *A System of Patterns – Pattern-Oriented Software Architecture (POSA1)*; Wiley & Sons, 1996; ISBN 0-471-95869-7

[Gamma++95] Gamma, E., Helm, R., Johnsohn, R., Vlissides, J.; *Design Patterns: Elements of Reusable Object-Oriented Systems*; Addison Wesley, 1995; ISBN 0-201-63361-2

[Giotta++01] Giotta, P., Grant, S., Kovacs, M.; *Professional JMS Programming*; Wrox Press, 2001; ISBN 1-861-00493-1

[Hanmer++99] Hanmer, R., Stymfal, G.; „An Input and Output Pattern Language: Lessons From Telecommunications"; in *Pattern Languages of Program Design 4 (PLOPD4)*, pp.503-538; Addison-Wesley, 1999; ISBN 0-201-43304-4

[Henney01] Henney, K.; "C++ Patterns – Reference Accounting"; EuroPlop'01, http://www.hillside.net/patterns/EuroPLoP2001/papers/Henney.zip

[Herzner++97] Herzner W., Kummer M., Thuswald M.; "DVS – A System for Recording, Archiving and Retrieval of Digital Video in Security Environments"; in *Hypertext – Information Retrieval – Multimedia '97*, pp.67-80; UVK Schriften zur Informationswissenschaft 30; Konstanz, 1997

[Herzner++02] Herzner, W., Thuswald, M.; "Smart Queue – A Pattern for Message Handling in Distributed Environments"; presented at *pattern writers workshop „Patterns in Distributed Real-Time Embedded (DRE) Systems"*; Seattle, Nov. 5, 2002

[ISO/IEC94] ISO/IEC 10918-1:1994 (JPEG) "Digital Compression and Coding of Continuous-tone Still Images"; International Organization for Standardization (ISO) Central Secretariat; 1, rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland

[McKenney96] McKenney, P.E., "Selecting Locking Designs for Parallel Programs"; in *Pattern Languages of Program Design 2 (PLOPD2)*, pp.501-531; Addison-Wesley, 1996; ISBN 0-201-89527-7

[Rising00] Rising, L.; *The Pattern Almanac 2000*; Addison-Wesley, 2000, Software Pattern Series; ISBN 0-201-61567-3

[Schmidt++00] Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.; *Pattern-Oriented Software Architecture 2 – Patterns for Concurrent and Networked Objects (POSA2)*; Wiley & Sons, 2000/2001; ISBN 0-471-60695-2

[Shaw96] Shaw, M.; "Some Patterns for Software Architectures"; in *Pattern Languages of Program Design 2 (PLOPD2)*, pp.255-269; Addison-Wesley, 1996; ISBN 0-201-89527-7

[Sobell94] Sobell, M.G.; *UNIX System V: A Practical Guide*; Addison Wesley Higher Education, 1994; ISBN: 0-8053-7566-X

[Wake++96] Wake, W.C., Wake, B.D., Fox, E.A.; "Improving Responsiveness in Interactive Applications Using Queues"; in *Pattern Languages of Program Design 2 (PLOPD2)*, pp.563-573; Addison-Wesley, 1996; ISBN 0-201-89527-7

[Woolf++02] Woolf, B., Brown, K.; "Patterns for System Integration with Enterprise Messaging"; at *PLOP'02 conference*; Monticello/Illinois, Sep. 8-12, 2002; see also http://jerry.cs.uiuc.edu/~plop/plop2002/final/woolfbrown2.pdf

[Woolf++03] http://www.enterpriseintegrationpatterns.com/index.html