

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 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, and two extensions of it: the SELECTABLE MESSAGE QUEUE, which allows to apply some selection mechanisms when retrieving buffered information, and the SMART QUEUE, which exploits the buffering mechanism to deal with mutual relationships among buffered information elements, e.g. for optimization.

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 the following, any information or request to be interchanged between processing elements is called ‘message’.

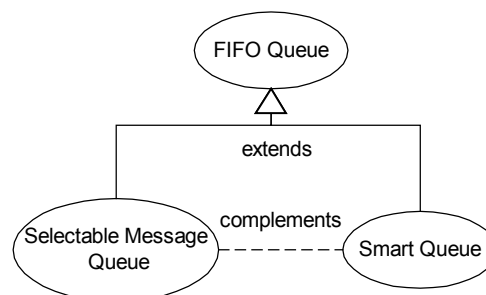


Fig. 1: Relationship between described patterns

The description of each pattern follows a common structure:

Context – describes which kind of applications is presumably addressed.

Examples – are specific applications raising a certain problem.

Problem(s) – lists the problems which the particular patterns should resolve, together with **forces** to be considered.

Solution – gives first a short outline of proposed solution, followed by details in its **details** part.

Discussion – addresses aspects which are not intrinsic part of the solution, but to be considered as well; in general, transparent to clients (i.e. users of the pattern).

Variants – are similar solutions, which are in general not transparent to clients, or alternative implementations. If distinction between ‘discussion’ and ‘variants’ is too weak, they are put together.

Examples resolved – discusses how examples can be resolved with proposed solution, if this is not evident.

Consequences – lists the **upsides** gained when applying the pattern, as well as the **downsides** to be considered.

Related patterns – mentions other patterns used by the described one, patterns solving similar problems, or conflicting patterns.

Known uses – are applications where the pattern has been used already.

Since "context" and "forces" are essentially the same for all described patterns, they are summarized 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)

- *Buffering*: since messages cannot be processed immediately when created, they must be intermediately stored, or 'buffered'.
- *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(s)

- Messages are generated with varying frequency, and it cannot be guaranteed that processing of each message can be completed before the next arrives.
- Or, consumption of messages shall be decoupled in time from their production, e.g. to meet architectural or design requirements, like departing synchronous event processing from their asynchronous logging.
- In any case, the processing order shall be the same as the creation order.

2.3 Solution

Provide a FIFO QUEUE as an intermediate buffering mechanism which

- allows to add messages to the buffer,
- allows to remove messages from the buffer in same order as added,
- takes care for over- and underflow,
- 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.

Details

In an object-oriented environment, a class like `FifoQueue` is usually providing that mechanisms:

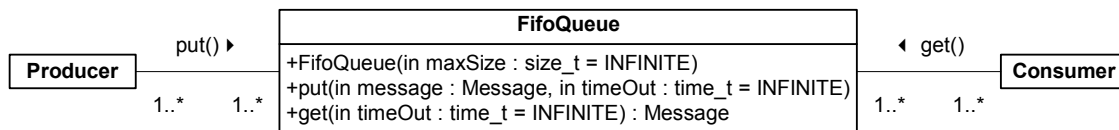


Fig. 2: `FifoQueue` – general class diagram

Figure 2 displays only elements of `FifoQueue` (in UML-notation) 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.

- A `FifoQueue`-instance is called 'empty', if it has no messages buffered.
- A `FifoQueue`-instance is called 'full', if `MaxSize` was not set to `INFINITE`, and the number of currently buffered messages equals `MaxSize`.

`put()`: appends the provided message at the end of the internally held list of buffered messages.

- If the `FifoQueue` 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 no longer than specified by this argument.
- If other `put()`- or `get()`-operations are currently under way, 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.

`get()`: removes the message at the begin of the internally held list of buffered messages, and returns it to the caller.

- If the internal 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 no longer than specified by this argument.

- If other `put ()` - or `get ()` -operations are currently under way, 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.

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 Messages Buffering

An important issue of FIFO QUEUE is the structure of the internal buffer. The simple first-in/first-out rule suggests a single-linked list, as shown in figure 3.

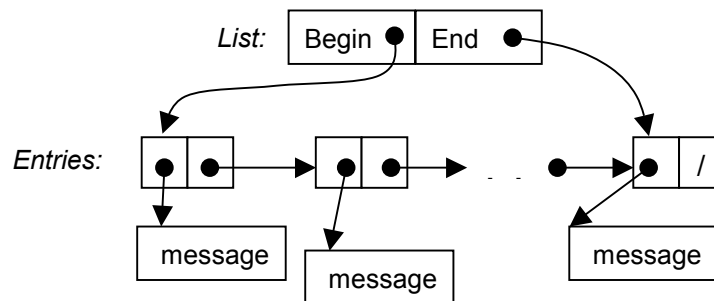


Fig. 3: Single-linked list with externally stored messages

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`, that is a `FifoQueue` with a given maximum number of simultaneously buffered entries, this may work out well, while in an unbounded queue, it may waste memory in applications with rare message bursts.

For bounded `FifoQueues`, a fixed-size array of `MaxSize+1` pointers (or references) to messages may be more efficient, where two indices identify the first used and the first free element, as indicated in figure 4.

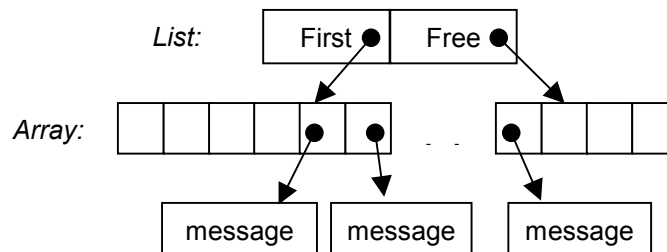


Fig. 4: Pointer (or reference) array with externally stored message

The array is used in round-robin¹ mode. `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 `[]` indicate consideration of the special case where

¹ “Round-robin” means that usage of the array continues with its first element after its last element has been reached.

First points at the last and Free at the first array element. First and Free must never bypass each other. See also variant “Safe Message Passing” for a simpler version of this approach.

Of course, both single-linked list and array are only examples for possible implementations, and should be understood as suggestions only.

Synchronization

If several producers and consumers may access a `FifoQueue` concurrently, synchronization is needed. In general, well known patterns as mentioned in section 2.7 “Related Patterns” could be applied here. However, there are two special cases to be considered (which are symmetrical to each other):

- if a `get()` is waiting on an empty queue, any `put()` must not be locked;
- if a `put()` is waiting on a full queue, any `get()` must not be locked.

A possible solution would be (in C++-like notation):

Private members of `FifoQueue`:

```
Mutex    m_Mx;
Event    m_NotEmpty; // set as long as at least one message buffered
Event    m_NotFull;  // set as long as queue not full
```

The public method `put()` of `FifoQueue`:

```
FifoQueue::put( in message: Message, in timeout: time_t ) {
    time_t tStart = now(); // for considering integral timeout
    time_t remTime = timeout;
    bool    bReady = false;
    bool    bWaitOnNotFullEvent = false;
    do {
        if (bWaitOnNotFullEvent) {
            m_NotFull.wait( remTime ); // wait() returns as soon as
            bWaitOnNotFullEvent = false; // either event set or timeout passed
        }
        else { // MutexLock applies the Scoped Lock Idiom [Schmidt++00]
            MutexLock Lock( m_Mx, remTime ); // try to lock m_Mx within
            // remaining time
            if (Lock.isLocked()) { // locking was successful
                if (m_NotFull.isSet()) { // and queue not full
                    //-- put code which appends message at end of queue here --
                    bReady = true;
                    m_NotEmpty.set(); // indicate that there is something in queue
                    if ("queue full") m_NotFull.reset();
                }
                else
                    bWaitOnNotFullEvent = true;
            }
        }
        if (timeout == INFINITE) remTime = INFINITE;
        else
            remTime = timeout - (now() - tStart);
    } while (!bReady && ((remTime == INFINITE) || (remTime > 0)));
    if (!bReady)
        //-- put code for failure handling here --
}
```

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

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 way would be to simply hand over references. However, if `put()` cannot guarantee that the producer will not use the reference anymore, this approach is dangerous. For `get()`, it is less risky, because it can be assured, at least conceptually, that its `FifoQueue` instance will not use the message references handed-over to consumers anymore, because this is completely under control of the `FifoQueue`-implementor.

A safer approach – though more time and memory consuming – is 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 4 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 loses 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 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 ("weak load balancing").* Since consumers wait on producers if no messages available, and producers wait on consumers if FIFO QUEUE is bounded and full, a weak load balancing is automatically achieved, avoiding that producers and consumers drift too far apart.

Downsides

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

2.7 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.

2.8 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 Mail 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 `FIFOQUEUE` 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.

3.3 Solution

Provide a `SELECTABLE MESSAGE QUEUE` as an extension of a `FIFO QUEUE` that additionally

- allows to remove oldest message with a certain attribute,
- allows to add attributed messages.

Details

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

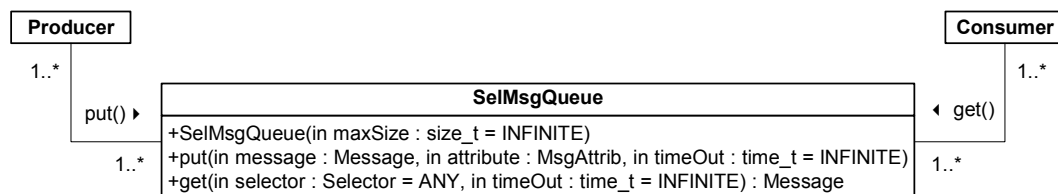


Fig. 5: `SelMsgQueue` – general class diagram

Figure 5 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.

`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 sophisticated 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, messages can be inserted in the right order by `put ()` already. For instance, if the attribute is priority, and `get ()` will always ask for the message with the highest priority, inserting messages in descending order requires $N/2$ comparisons on average, if N is the current number of queued messages, while searching for the message with the highest priority by `get ()` would always require N comparisons, if `put ()` simply appends at end of the internal list. The same applies, if size is the attribute and `get ()` will always request for the smallest message (e.g. from a printer queue).

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.

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

A variant regards the attribute as private part of the messages. In this case, `get ()` must delegate the selection to the messages; e.g. (again, in C++-like notation):

```
SelMsgQueue::get( in selector: MsgSelect, in timeout: time_t ): Message {
    Iterator i = First;
    do {
        if ((*i)->eval( selector ) == true) {
            -- remove (*i) from queue, and return it --
        }
        // progress i to next message
    } while (-- i points at a message --);
    return -- nothing --;
```

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 `ANY`.

However, if this could lead to starvation of low-priority requests, priorities of all queued requests should gradually be increased. In the first approach, this would be achieved by re-queuing messages from one internal queue to that with the next higher priority, in the second approach by simply increasing the priority attribute of all queued messages.

- *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.

Upsides

- *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.

Downsides

- *Limited selector expressiveness.* Consumers can only ask for attribute tests expressible with the provided `Selector`.

3.7 Related Patterns

The `SELECTABLE MESSAGE QUEUE` pattern is an extension of the `FIFO QUEUE` pattern. In contrast to `SMART 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 `MsgAttrib`. 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 whose `can_run()` method returns true.

To some extent, the *Pipes and Filter* pattern [Buschmann++96] may 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 Mail Service JMS, if used with message selectors.

4 Smart Queue

Use SMART 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 could be 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.
- Retrieving a message may require to keep it in the queue, but not to retrieve it again as long as not a specific ‘reset’-operation is applied.

4.3 Solution

Provide a SMART 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.
- `get()` has a new argument `keep` of type `boolean`. If `keep == true`, it will not remove the message from the queue.

- A new method `reset()` ensures that oldest 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.

Details

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

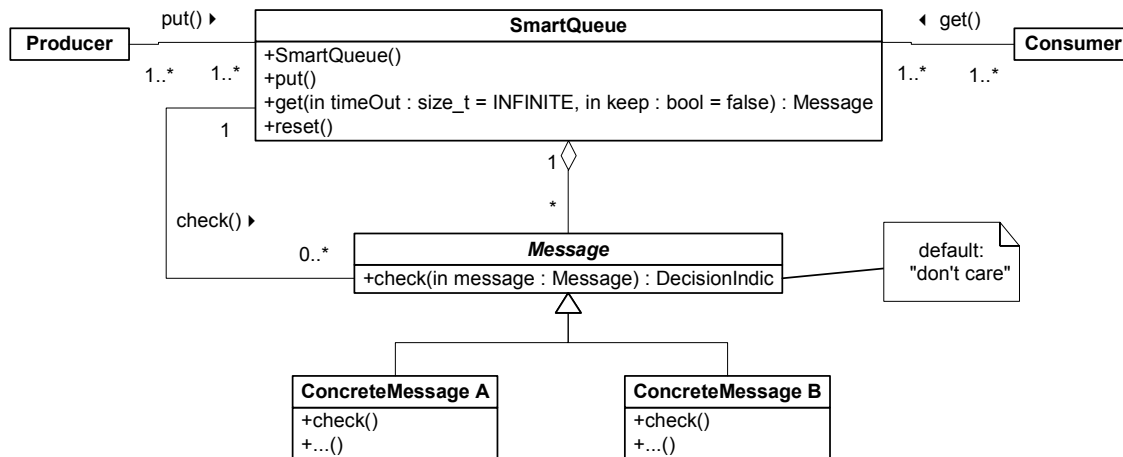


Fig. 6: SmartQueue – class diagram

(Only relevant and new arguments are shown in figure 6.)

`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 returns a decision indicator as listed in the following table, where texts are formulated from point of view of that message object which's `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: SmartQueue – 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 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 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.

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 QUEUE pattern.

Decision Indicators

Also, the SMART 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 Queue

The last discussion about evaluation order raises a problem with the given solution, because the SMART 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 Queue

Another approach to solve this problem is to evaluate inter-message relationships by the SMART QUEUE rather than by the messages. Of course, in this case SMART QUEUE 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 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, this variant gave name to the whole pattern.

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 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:* `get()` sets always `keep = true`. On retrieval of an acknowledge message, it is provided to the SMART QUEUE 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 QUEUE and the `check()`-methods.
- Putting the evaluation code into the message implementations (by means of their `check()`-methods), both avoids that the SMART 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.
- Elimination of obsolete messages before they are processed may improve efficiency of the whole application.
- SMART 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`.
- SMART 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.
- 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 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 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* [Hammer++99] – can efficiently be realized using the SMART 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 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 QUEUES. See also related patterns before.

5 Acknowledgement

An earlier version of the SMART 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., Hammer, 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
- [Giotto++01] Giotto, P., Grant, S., Kovacs, M.; *Professional JMS Programming*; Wrox Press, 2001; ISBN 1-861-00493-1

- [Hammer++99] Hammer, 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
- [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>