

Smart Queue – A Pattern for Message Handling in Distributed Environments

Wolfgang Herzner, Michael Thuswald
ARC Seibersdorf research

Purpose

Smart Queue enriches the traditional FIFO-queue (*First In First Out*) for asynchronous messaging and event handling by supporting fault tolerance in environments with failure prone transmission, handling expiration times, and alike. It enables us to collect functionality concerned with asynchronous messaging at one place (or at least a well-defined number of places) and therefore improves aspect-orientation.

Motivation

With little doubt, FIFO-queues are at the heart of almost every asynchronous information transfer, with writers or senders adding entries (messages, requests) at one end of the queue, while readers/recipients fetching entries from the other. Sometimes, however, more would be needed than this basic functionality. Consider, for instance, a situation where such a queue is used to collect messages from a process A to be transmitted to another process B. With faulty transport media it is possible that a message transfer fails unrecognized by A. A common behavior pattern to cope with this is to wait for some confirmation from B, but this can cause significant performance loss. An alternative would be to transmit messages without wait but to keep them 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.

Or, queued entries could become outdated before being processed, may be because some expiration time has been passed, or a new entry overrides semantically an older one of same type, or a new entry neutralizes an older one. For instance, a low priority display showing a process situation, and some file copying process reports progression state periodically into a queue collecting all messages to that display. On entering such a message, any older message of the same type indicates that the display didn't have yet enough time to process it; hence, the older message is outdated and should not be sent/processed anymore. Conversely, in soft real-time applications the violation of some expiration time (or, better, deadline) would lead to error handling rather than elimination of the message.

Although all these requirements could be solved using conventional FIFO-queues by implementing the necessary functionality somewhere else, both existing implementations as well as further considerations suggest that an enhanced queue which 'smartly' supports solutions to these problems may both improve maintenance and efficiency.

Context

Asynchronous information transfer where simple FIFO-queuing is not sufficient, but time and/or context dependent handling of queue entries needed, be it for fault tolerance, optimization, detection of time constraint violation, or other reasons.

Problem

(Distributed) Applications which need asynchronous information transfer between their components rely on some FIFO-queuing mechanism to realize that transfer mode. However, such simple mechanisms often have to be extended by more complex aspects like keeping entries after their first get until successfully processed, correlate new entries with existing to detect outdated entries, or raise error handling on deadline violations. Resolving these problems effectively requires the resolution of following

Forces

- *Efficiency.* Adding such features should increase additional computing and memory requirements as minimal as possible.
- *Compactness.* Code related with asynchronous information transfer should be collected in as few places as possible, rather than spread over large parts of the application; thus improving the aspect-orientation of the resulting code.

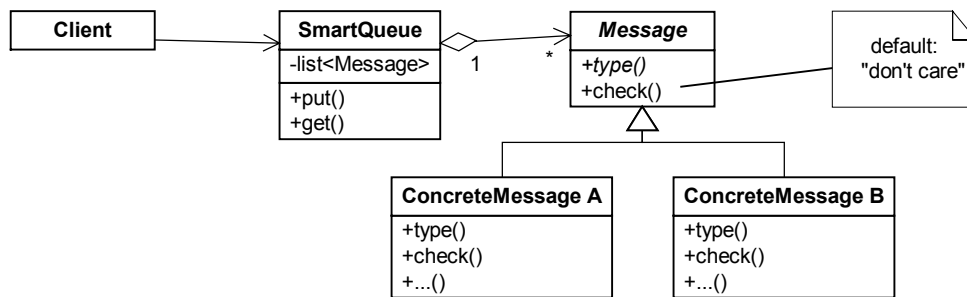
- *Flexibility.* With respect to the addressed features, adaptation of system behavior to new requirements should be as simple as possible.
- *Simplicity.* Compared with possibly more 'direct' solutions, the pattern itself should add as little complexity as possible.

Solution

Assuming that `put ()` appends a message to a FIFO-queue, and `get ()` fetches the oldest entry from it:

1. Design messages such that
 - the 'type' of a message can easily be determined,
 - for each message object, it can (efficiently) be determined if and how it interrelates with another message;
 - time constraint violations can easily be detected, if necessary
2. `put ()` compares the new message object with each entry (in its list of queued messages) in queuing order and treat the result, as indicated in the table further below;
3. `get ()` always removes the oldest entry from the list and returns it, if any.
(Same behavior as for conventional FIFO-queues.)

In an object-oriented environment, it is fairly obvious that message objects themselves have the knowledge how they interrelate with other messages. Consequently, the structure of the Smart Queue pattern can be illustrated by the following class diagram¹:



Details

Due to restricted space, only not evident details are given.

`check ()` takes a (reference to a) message object, determines its relationship to itself by means of `type ()` and possible further information, as indicated by "`...()`" in the concrete message classes, and returns a decision indicator as listed in the following table (texts in table are formulated from point of view of that message object which's `check ()`-method has been activated):

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

¹ This diagram has been drawn with Microsoft Visio 2000™. This tool uses '+'-prefixes to denote public class elements, and '-' for private ones.

A subclass doesn't need to overwrite `check()` if the default behavior (return *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 end of the list is encountered. In the first case, it executes the indicated decision; in the latter case, it appends the new message at the end of its list.

Discussion

The described solution provides a basic functionality for the described problems, but leaves some aspects open or does not treat them like expiration time. These shall be addressed in this and the next clause (Variants).

Message Types

Although not a primary aspect of the Smart Queue pattern – since it could be left up to `check()` to find out what type the provided message object is, `type()` has been included to emphasize that efficient identification of a message object's real class is a key issue of this pattern.

How `type()` is realized best, depends on the chosen implementation environment and language. In environments which provide runtime type information, this feature can be used; in others some different mechanism has to be implemented. Similarly, the possibly necessary access of appropriate subclass elements, once a message's real type has been identified, is left unspecified by Smart Queue.

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.

Queue Size Limits And Other Conventional FIFO-Queue Aspects

Of course, `SmartQueue` could – and should – behave like an ordinary FIFO-queue for clients. For instance, limitation of the number of entries, especially in environments with small memory capacities. This is the more the case if `SmartQueue` is derived from a conventional FIFO-queue class. Then of course, its method signatures have to be adapted to that of the superclass.

Variants

Depending on further requirements, the basic solution described above is to be extended.

Expiration Times and Deadlines

One important issue not yet addressed is the consideration of expiration times and deadlines. Since for a message's expiration it is sufficient to check for it before that message is used, the Smart Queue pattern is extended in two ways. 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 `SmartQueue`-methods `put()` and `get()` on each entry invoke its `isExpired()` before any other method, and remove that entry if `true` has been returned.

For deadlines, in general this approach is inappropriate, because they have to be detected as soon as possible, and missing them has to be treated as error (*deadline violation*). (In the Smart Queue pattern, such an error occurs whenever an entry with an elapsed deadline is still in the queue.) Detection of deadline violations has to be done in an own, concurrent activity. For instance, in a multi-threading environment, a concise solution would be to provide a private method of `SmartQueue`, which executes in an own thread and regularly checks the message-list for missed (or even better: nearly to be missed) deadlines. On detection, it initiates error handling. With large queues, some optimization will be needed there, like keeping pending deadlines in a time-sorted list.

Resend Control and Fault Tolerance

For allowing to resend messages when e.g. transmission failed after fetching them from the queue, first `get()` does not delete the oldest entry anymore, but keeps it and returns a copy. Then, the decision indicator *ignore and delete* can be used to get rid of the oldest fetched but not yet deleted entry. In detail: after the oldest entry has been fetched by `get()` and delivered, the process owning the smart queue waits until the confirmation message arrives. This is then handed over to `SmartQueue.put()`, which causes the oldest entry to be deleted.

However, to avoid the waiting-induced performance loss, i.e. by fetching from the smart queue and deliver them without waiting on confirmation, the following extension to the Smart Queue pattern is helpful: `SmartQueue`

holds a private attribute indicating the last entry accessed by `get()`. The latter propagates this attribute to the next entry before starting its described behavior. Note that deleting the entry identified by that attribute (e.g. due to expiration) also has to update that attribute. Initially, this attribute has a value indicating that it identifies no entry, causing `get()` to access to oldest entry.

This can now conveniently be used to support fault tolerance. If, for instance, at some time it is detected that fetched messages have not been processed successfully by the recipient, that attribute simply has to be reinitialized.

With more than one recipient, though, this pattern grows more complex, because not only for each recipient such a control attribute has to be hold, `get()` also must be able to recognize the intended recipient, implying some registration and identification mechanism.

Not Object-oriented Environments

In not OO-environments or applications, where messages are treated as data structures rather than objects, `SmartQueue` by itself will take over the functionality of the methods of `Message` and its derived subclasses. To some extent, this can lead to an even higher concentration of code related with the aspect of smart queue handling than in the OO-approach.

Consequences

Benefits

- Concentration of code related with advanced queue handling in a small number of well-defined locations (`SmartQueue`-implementation, and the `check()`-methods) supports maintenance.
- Improve of efficiency by eliminating obsolete messages before their processing.
- Support for fault-tolerance.

Liabilities

- Threat of danger due to ill-coded or malicious `check()`-methods and ‘growing’ queues.
- The decision indicator *ignore* appears to violate the FIFO-rule, if the old entry replaces its attributes with that of the new message.
- Performance overhead if `check()`s execute slow and needed to be invoked without terminating result too often.

Known Uses

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. 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.

The recorders (recording units) of DVS exploit the Smart Queue pattern in a different way. Monitoring displays may request video frames from an arbitrary number of cameras with varying frame rates. To adapt the transmission rate to both the current network load and capacity of the monitoring display node, frames are rejected to be queued if the Smart Queue contains already a certain number of frames of the specific camera.

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. For instance, 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).

Related Patterns

Smart Queue would be an extension of (FIFO-)Queue. The Event Queue just mentioned can be regarded as a special case. According to [Rising00], up to 1999 the only further queue pattern described in literature, which addresses general FIFO-queues, can be found in [Beck97] as Smalltalk-idiom. Newer literature like

[Schmidt++00] or proceedings available at internet from Plop [Plop..02], EuroPlop [EuroPlop..02], or [OOPSLA'01] do not describe such pattern either. Nevertheless, an Ordered Collection [Beck97] will be needed to implement the list of queued messages.

Several patterns used for user interfaces of real-time (oriented) systems, like Merge Compatible Events or Five Minutes of no Escalation, can efficiently be implemented with Smart Queues, as already mentioned under *Known Uses*.

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

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
- [EuroPlop..02] <http://www.hillside.net/patterns/EuroPloP/>
- [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
- [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
- [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
- [OOPSLA'01] <http://www.cs.wustl.edu/~mk1/RealTimePatterns/OOPSLA2001>
- [Plop..02] <http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html>
- [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*; Wiley & Sons, 2000/2001; ISBN 0-471-60695-2
- [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