

# Improving Responsiveness in Interactive Applications Using Queues

William C. Wake<sup>1</sup>, B. Douglas Wake<sup>2</sup>, and Edward A. Fox<sup>3</sup>

## Abstract

This paper presents a set of patterns for use in designing the low-level interaction structure of an interactive application.

## Introduction

Programs can be categorized as being batch, real-time, or interactive. In a batch application, output is not synchronized with input. (The input data is determined before the program is run.) In a real-time application, the program reacts to the passage of time (and the “correctness” of a computation depends upon when the result of that computation is delivered, see e.g., [Sha & Goodenough 1990]). In an interactive, or responsive, application, the program presents output in response to input, and the user reaction to that output can affect future input to the program [Ambriola & Notkin 1988].<sup>4</sup>

An interactive program must react to the user’s actions as quickly as possible. In graphical applications in particular, timely reaction to user actions is critical to achieving an adequate interactive feel. For example, if the cursor position doesn’t consistently reflect the mouse motion, the system will feel sluggish.

The interaction style of an application should be reflected in its structure. Modern programs with a graphical user interface often manage small interactions, e.g., reacting to individual keystrokes, and the program design must accommodate that burden.

The following patterns focus on a design that can help one improve the performance and responsiveness of an interactive program.

1. Event Queue
2. Merge Compatible Events
3. Generate Artificial Events
4. Decouple Execution and Output

---

<sup>1</sup> Department of Computer Science, Virginia Tech.

<sup>2</sup> Representing himself.

<sup>3</sup> Department of Computer Science and the Computing Center, Virginia Tech

<sup>4</sup>I’d appreciate a reference to the 3-way split of batch vs. real-time vs. reactive - it’s not in [Ambriola & Notkin 1988], but I believe it is from an article in either *IEEE Computer* or *IEEE Software* within the last five years.

# 1. Event Queue

## Context

- An interactive application, especially one with a graphical user interface
- The application receives input as events, such as mouse movement or keystrokes

## Problem

- The program must systematically deal with a mix of several event types

## Solution

To accommodate low-level interaction as used in GUI programs, we can adopt one of the classical architectures used by simulation programs: an event queue. (For example, see [Pooch & Wall 1993].) For each event in the queue, information about the type of the event (e.g., “keystroke”) is augmented with a timestamp and data specific to that type of event (e.g., “shift-A”). Events in the queue are ordered by time.

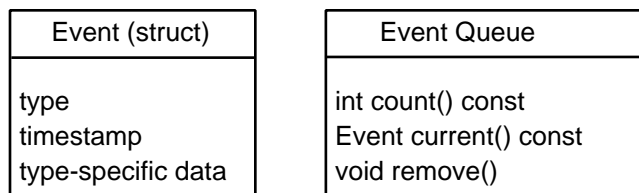


Figure 1: The event struct and the event queue object.

The event queue object can maintain a count of the number of pending events, retrieve the current event, and remove the current event from the queue. Some implementations may have filtering of events, either to restrict which types will be fetched from the queue, or to restrict which events will be inserted in the first place. For example, a non-graphical application might choose to ignore mouse motion.

Event queues are well managed by one of the classical simulation program structures:

```
done = false
while (!done) do
    e = eq->Current()
    eq->Remove()
    switch (e.type) of
    case type-1: ...
    case type-2: ...
    :
    end (switch)
end (while)
```

## Consequences

- Applications that respond to events encourage a less rigid interaction model. For example, a user can resize a window without leaving text insertion mode.
- Event-driven applications often strive to minimize the number of modes. Such applications must be prepared to deal with many types of events in various program contexts.

## Examples

- Simulations. Simulations have long used an event queue structure for overall organization [Pooch & Wall 1993].
- Programming for a graphical user interface. Programs on the Apple Macintosh uses an event-queue-based design for their basic structure [Apple 1985]. Microsoft Windows uses a similar structure [Petzold 1990]. However, many user interface frameworks such as PowerPlant [Metrowerks 1995] hide the event queue by using the *Chain of Responsibility* pattern [Gamma et al. 1995, p. 223].

## Related Patterns

- There is usually only one event queue per application; it might be an instance of the *Singleton* pattern [Gamma et al. 1995, p. 127].
- Events are a low-level structure. You should consider using the *Command* and/or *Chain of Responsibility* patterns from [Gamma et al. 1995, pp. 233 and 223 respectively]. *Command* can encapsulate an event into an object; *Chain of Responsibility* can dispatch the event to the object that should handle it.
- *Merge Compatible Events (2)* and *Generate Artificial Events (3)* can be used with an event queue to improve a program's responsiveness. *Decouple Execution and Output (4)* can be used to apply the event queue structure to output as well as input.

## 2. Merge Compatible Events

### Context

- An interactive program based on an event queue
- Sequences of similar or related events are common
- Performance is important

### Problem

- Events are arriving faster than the program can deal with them

### Solution

Instead of fetching one event at a time, use the event queue's `count()` method to see if there are other events pending. If so, peek into the queue to judge whether the next event is compatible with the current one (in terms of forming a compound event). If the events are compatible, they can be combined into a single compound event. This process can be repeated to merge a sequence of compatible pending events into one.

Consider a sequence of events arriving as shown on the timeline below. Events with the same shape are considered compatible; those immediately adjacent are considered to be in the queue at the same time. Then, this sequence of events:

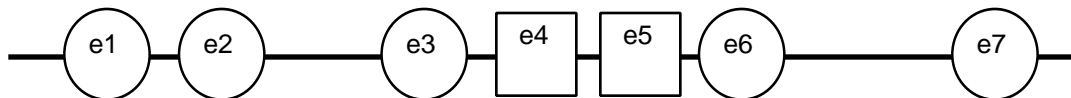


Figure 2: An event sequence.

might be treated as the following sequence instead:

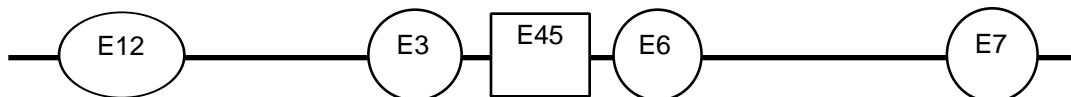


Figure 3: An event sequence with compound events

Event e1 and e2 were both in the queue at the same time, and were compatible, so they were combined. Event e3 was not combined with e1 and e2, as it wasn't yet in the event queue when they were merged. Although e4 was in the queue with e3, it was not compatible, so it wasn't combined. Event e4 was combined with e5, but e6 was not compatible, so it remained a separate event. Finally, e7 wasn't available when e6 was retrieved from the queue.

This pattern borrows an idea from batch systems and applies it to interactive systems: instead of an on-line sequence, where each command must be executed fully and in order, we get a set of commands, which can be optimized together. This improves the performance and responsiveness of the system.

## Counter-Indications

- Which events are compatible is program-dependent. The important thing is that the effect of operating on the two events together should be the same as the effect of operating on them in sequence. For example, on the Macintosh, the effect of a mouse button double-click on an object is usually designed to be an extension of the effect of a single-click. We want a distributive law to hold: if  $f$  is what the program computes, we want

$$\forall e1, e2: f(e1; e2) = f(e1); f(e2)$$

- This pattern should not be applied when the cost of combining the events is more than the cost of simply performing the actions associated with them. If knowing the sequence allows little opportunity for optimization, there will be little benefit in combining events. Ideally, the cost would work like this:

$$\forall e1, e2: \text{Cost}[f(e1; e2)] \leq \text{Cost}[f(e1); f(e2)]$$

$$\wedge \exists e1, e2: \text{Cost}[f(e1; e2)] < \text{Cost}[f(e1); f(e2)]$$

That is, “it never hurts, and sometimes it helps.” If the first inequality doesn’t hold, then it will sometimes take longer to handle the combined event. If the second inequality doesn’t hold, it never pays to combine events.

There are two escape hatches to this guideline:

- (1) The definition of “compatible events” can be used to exclude troublesome event combinations, and
- (2) You might make an explicit cost tradeoff, e.g., “I don’t mind if searching is a little slower, so long as it makes text entry much faster.”

## Consequences

- This pattern reduces the repeatability of a program. Previously, only the **sequence** of events controlled what was done. Using this pattern, the **timing** of events interacts with the sequence of events.
- Memory management can become a little more complicated, as a compound event has more than one event’s information associated with it.

## Examples

- Consider typing text into a WYSIWYG word processor. As each character is typed, the program must calculate line lengths, check word wrapping, and so on. Many of these activities are the same regardless of whether one or several characters are inserted. So, rather than “insert c1” then “insert c2”, we can effectively merge the two events into “insert c1 and c2” to avoid some of the overhead.
- Microsoft Windows has a “repeat count” field in the event record, to indicate that a string of several identical keystrokes was combined into one event. (However, [Petzold 1990, p. 92] indicates that many programs ignore the count.)

### 3. Generate Artificial Events

#### Context

- An interactive program based on an event queue
- During idle time, work could be done that might speed up future interactions
- Responsiveness is important

#### Problem

- The program isn't fast enough

#### Solution

Let the event queue contain artificial events. In particular, the event queue should return a null event when nothing is pending in the queue. (However, do not include null events when providing a count of the number of pending events.) The program should handle null events as it would any other, and use the time for “background” processing that might speed up interaction in the future.

Actions in response to a null event should have a bounded execution time - they shouldn't take so much time to execute that responsiveness suffers.

So, a sequence of events

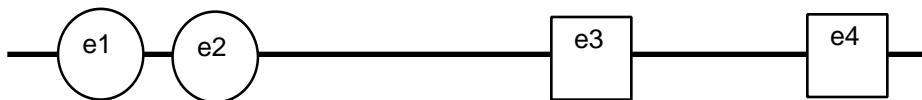


Figure 4: Event sequence with idle time.

has new events inserted, corresponding to times when the system would otherwise be idle:

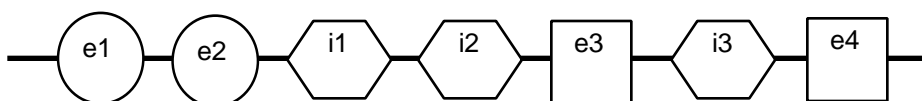


Figure 5: Event sequence augmented with null events.

This pattern borrows an idea from real-time systems: let the passage of time affect the program.

#### Consequences

- Overall responsiveness improves - the program has done some work it would have had to do anyway, but has done it *before* the user is waiting for results.
- The overall system may become slower. Null events steal time that would have gone to

other processes. It is necessary to control the rate at which null events are generated, to limit the load of the program on the system as a whole.

- The program is more complex; it must often be restructured to allow effective incremental work. Consider a data structure that must be reorganized. It is probably easiest to write this so that the rest of the program comes to a halt; then there is no need to worry about the reorganization of the data structure affecting its use. To reorganize things incrementally, we have to keep track of what pieces are done so far, and whether any changes in the data structure affect the partial reorganization that's already been done.

For example, suppose we want to search a list. We might maintain the list in two parts: a sorted part and an unsorted part. The sorted part can be searched via binary search in  $O(\log n)$  time, while the unsorted part must be searched via an  $O(n)$  search. When we get a null event, we could use that opportunity to put one more item in its proper sorted position.

- Synchronization is important: we must ensure that idle-time processing won't interfere with the main task, either by making sure it is done before the main task resumes, or by standard synchronization mechanisms.
- The repeatability of the system is reduced. The system load, rather than simply the event sequence, interacts affects the work the program does. (When the system is heavily loaded, fewer null events may be generated, so the program has less chance to prepare for future interaction.) The speed with which the user interacts with the program affects the actual events processed.

### Notes

- Events other than null events are useful, too. For example, we might have events for timers expiring, for information coming from the network, or for requests from the operating system.
- Some systems allow for threads, as a lightweight form of processes. On such systems, the program could be structured as several interacting threads (using normal synchronization constructs). Unfortunately, many operating systems make this less feasible, as they don't provide for thread-safe interaction with the operating system.

### Examples

- Lisp systems may schedule garbage collection while the system is waiting for user input [Teitelman & Masinter 1981].
- An editor can use idle time to write a file logging the current state (for use in crash and error recovery).
- The Macintosh [Apple 1985] supports null events to let the program do work while no user input is pending.

## 4. Decouple Execution and Output

### Context

- An interactive program
- Performance is an issue

### Problem

- Display is relatively expensive or time-critical

### Solution

Rather than strictly alternating command execution and result display, separate the two by introducing an output queue. We might let null events be the cue to work on clearing the output queue, or we might schedule output by some other mechanism.

### Consequences

- If display is expensive, this lets us improve overall responsiveness, by not requiring the program to wait for output to complete before processing the next input. If display is time-critical, this lets us schedule regular display updates.
- This pattern can introduce timing-related instabilities. Since the output is not (always) an accurate reflection of the current system state, the user may over-compensate for the lack of timely feedback. It may be necessary to simplify output to provide the user with current information. For example, dragging a window on the Macintosh causes an outline of the window to be dragged rather than its full contents. This lets display keep up with mouse motion, and makes the interaction feel smoother (especially on slower machines).

### Examples

- The Macintosh [Apple 1985] uses a “dirty region” event (on the input event queue, however) to tell an application it must update its windows. [Finseth 1991] describes Emacs output as a pair of processes, one creating the text, the other trying to keep up with displaying it.
- The WM CPU decouples input and output queues for better performance ([Wulf 1988]).
- Virtual reality systems can use this decoupling to allow regular updates of the display at a rate independent of the speed at which the “world” is updated (e.g., [UVa UIG 1995]).

### Related Patterns

- The *Merge Compatible Events (2)* and *Generate Artificial Events (3)* patterns can be applied to the output queue as well.
- This is a variation on the *Observer* pattern of [Gamma et al. 1995, p. 293]. In the classical Model-View-Controller architecture, views update synchronously after changes in the model. This pattern splits view updating from the model changes.

## Conclusion

The manipulation of events and commands has the effect of evening out the demands on the CPU: when events are coming quickly, we combine them using *Merge Compatible Events (2)*, and we require fewer actions. When nothing is happening, *Generate Artificial Events (3)* generates null events, which cause new action to occur.

Many user interface frameworks hide the event queue. Instead, they construct a layer above the queue, and make it seem that messages are sent to interaction objects (using *Chain of Responsibility* from [Gamma et al. 1995]). These patterns aren't meant to preclude hiding the events - but rather to suggest ways to change event dispatching to improve responsiveness.

Applications that are not using a framework may be able to apply the patterns directly. Applications that use a framework may find it worthwhile to modify the framework to take these patterns into account.

These patterns are not applicable only to systems with graphical interfaces - even strictly keyboard-based interfaces can make use of them.

The final pattern (*Decouple Execution and Output (4)*) represents an old idea that's coming back around. The Emacs editor has long decoupled the display from command execution [Finseth 1991]. The idea is achieving prominence again due to the requirements of virtual reality systems.

## Acknowledgements

This paper has benefitted from discussions with and criticisms from Steve Wake (STL, Blacksburg, VA) and the conference reviewer. The first and third authors were partially supported by NSF grant IRI-9116991.

## References

[Ambriola & Notkin 1988] Vincenzo Ambriola and David Notkin. "Reasoning About Interactive Systems," *IEEE Transactions on Software Engineering*, 14(2), February, 1988, pp. 272-276.

[Apple 1985] Apple Computer, Inc. *Inside Macintosh, Volume I*, Addison-Wesley, Reading, MA, 1985. QA76.8 M3 R67 1985.

[Finseth 1991] Finseth, Craig A. *The Craft of Text Editing: Emacs for the Modern World*, Springer-Verlag, New York, NY, 1991. QA76.76 T49 F56 1991.

[Gamma et al. 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995. QA76.64 D47 1994.

[Metrowerks 1995] Metrowerks, Inc. *Inside CodeWarrior 6*, Metrowerks, Mooers, NY, 1995.

[Petzold 1990] Charles Petzold. *Programming Windows: The Microsoft Guide to writing applications for Windows 3*, Microsoft Press, Redmond, WA, 1990. QA76.8 I2594 P474 1990.

[Pooch & Wall 1993] Udo W. Pooch and James A. Wall. *Discrete Event Simulation: A Practical Approach*, CRC Press, Boca Raton, FL, 1993. QA76.9 C55 P66 1993.

[Sha & Goodenough 1990] Liu Sha and John B. Goodenough. "Real-Time Scheduling Theory and Ada," *IEEE Computer*, 23(4), April, 1990, pp. 53-62.

[Teitelman & Masinter 1981] Warren Teitelman and Larry Masinter. "The Interlisp Programming Environment," *IEEE Computer*, 14(4), April, 1981, pp. 25-34. Reprinted in Barstow, Shrobe, and Sandewall (editors), *Interactive Programming Environments*, McGraw-Hill, 1984. QA76.6 I525 1984.

[UVa UIG 1995] UVa User Interface Group, "Alice: Rapid Prototyping for Virtual Reality," *IEEE Computer Graphics and Applications*, 15(3), May, 1995, pp. 8-11.

[Wulf 1988] Wm. A. Wulf. "The WM Computer Architecture," *ACM Computer Architecture News*, 16(1), 1988, pp. 70-84.