Override Current Processing

Philip Eskelin
Credit Suisse First Boston Corp.
Eleven Madison Avenue, 11th Floor
New York, NY 10010
+1 212-325-7955
philip.eskelin@csfb.com

Abstract

Sometimes applications that execute commands with unpredictable response times need to process those commands asynchronously so the user can continue to perform functions in other parts of the application. In a multi-threaded system, worker threads or asynchronous calls can be used to mitigate delays in response times to commands operating over the network, printing, faxing, or other types.

However, if done improperly, this can introduce race conditions, deadlocks, data corruption, floods, hurricanes, or misuse of finite resources. Under systems without threading, long commands are sometimes executed in a manner such that a small fraction of processing is performed, followed by the retrieval and dispatching of all queued messages, before processing the next fraction.

Also, when these commands are executing and their completion is pending, the user may request the invocation of additional commands that are intended to interrupt the processing of the current command in favor of executing the new command, in effect overriding the execution of a currently processing command.

Override Current Processing is a design pattern that can be used to delegate and interrupt the asynchronous execution of commands under a LIFE (Last In, First Execute) idiom in languages such as Java and C++.

Name: Override Current Processing Pattern

Intent

To provide a mechanism for efficiently interrupting a currently executing command in favor of the execution of a new command.

Also Known As

Interruptible Command

Motivation

In a web browser, the user can click on a hyperlink to request the download of an HTML page from a web server. While the browser is downloading it, the user can continue using the browser, and even perform additional requests if desired. For example, clicking on another link, reloading the current page, going back, or going home will cause the browser to cancel the current download in favor of carrying out the new request. When requests occurring in rapid succession, all requests except the last one are interrupted and the new request is executed.

The most obvious solution in a multi-threaded system is to create a thread that handles request so the browser can download the page independent of the thread dispatching events for the user interface, and notify the user interface upon completion. If new requests come in while the download is pending completion, the thread is terminated and a new one is created in its place to execute the new request. A major drawback is that thread termination can leave the system in a corrupt or ambiguous state, and result in abnormal termination of the program.

Another alternative is to poll for incoming data and cancel retrieval when a new request occurs. This works in the case of a browser using sockets that has the ability to poll for the status of read and/or write sockets. Close the currently open socket when the new request occurs will cancel retrieval and a new socket can be created to download the page associated with the new request.

If an asynchronous paradigm is being used for downloading the page from the web server, a third way of solving this problem is to store the very last request and compare it against arriving responses. However, the number of pending requests accumulates quickly and consumes limited network resources and bandwidth.

The situation for the above three solutions is complicated further by the requirement to retrieve web content from multiple forms of media. Many browsers implement both in-memory and file-based caching mechanisms. A general design solution is desired that brings the following forces into equilibrium:

- Commands of the same type that take an inordinate amount of time should be executed in parallel to allow the user to continue using the program.
- Only the most recently invoked command should be permitted to execute in the event of multiple invoked commands.

- The solution should be generic enough to support retrieval on multiple forms of media because caching techniques are typically implemented to improve response time.
- System integrity must be maintained while concurrently overriding current command processing with new commands.
- Conservation of valuable thread and network resources can be critical.

Solution

The Override Current Processing pattern and our web browser example in a multi-threaded system containing an event-driven graphical user interface that is capable of exception handling will be used to demonstrate how these forces can be resolved. A `BrowserWindow` class represents the graphical display of a web page and its associated content elements. It is implemented with member functions responsible for handling dispatched events from the main thread's message loop. Since web page requests can take an inordinate amount of time, it delegates the invocation of requests to a `RequestController` class. A `HistoryList` is used to track history and allow the user to navigation through requests.

While `BrowserWindow` is initializing, it calls the `Execute` method in `RequestController` to start a worker thread that processes requests and fires events back to `BrowserWindow` upon completion. It manages a queue of requests that accumulate while requests are incoming and before the currently processing request can be interrupted.

A `HttpRequest` derived from the `Command` class implements the execution of the web page request. While processing, it frequently checks with the associated `RequestController` to determine whether it has been interrupted by additional requests. If so, the `RequestController` throws an exception that results in firing an event to the `BrowserWindow`, notifying it that the request has been interrupted. If not, the request is successfully completed an event is fired to the `BrowserWindow` notifying it that the request has been completed. If many requests are sitting in the `requestQueue` when the currently processing request checks for interrupts, the most recently invoked request is made the current request, and all remaining requests are destroyed.

When the web browser is closed, `BrowserWindow` destroys its `RequestController`, instructing it to inform the thread to interrupt the currently processing request, expire all currently pending requests, and exit the thread gracefully. The following class diagram depicts the static relationships between each of the classes involved:

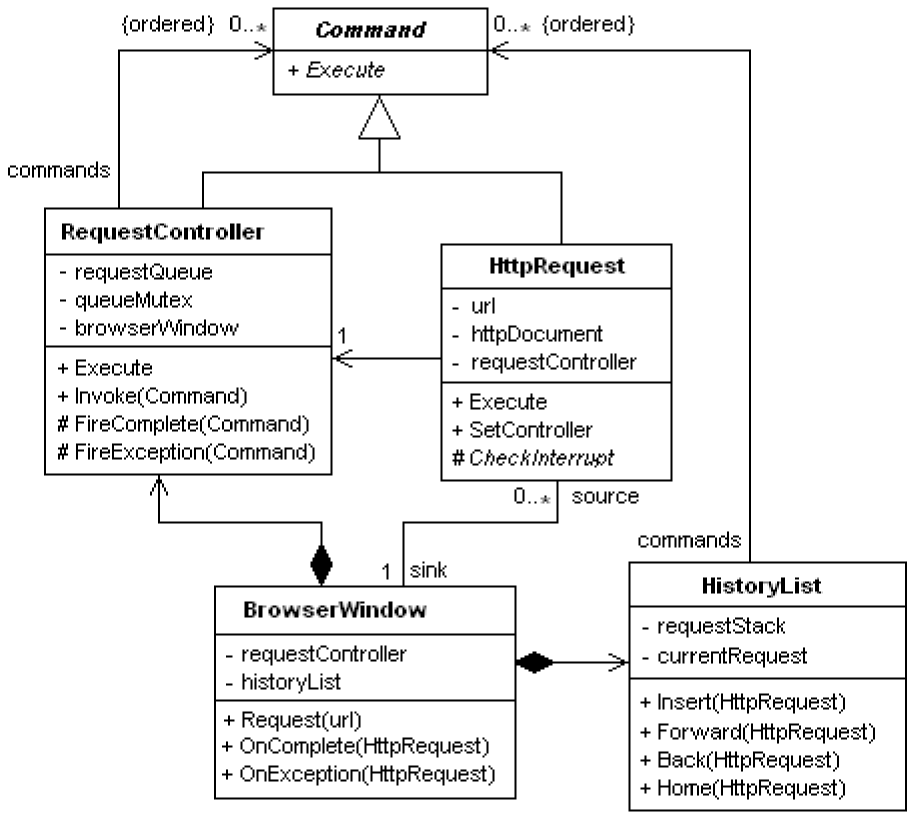

Figure 1 *Participants in a web browser example*

The diagram fails to illustrate the existence of multiple threads because `RequestController` privately encapsulates its implementation of thread creation, request retrieval and processing, and thread termination. The benefit of this is that the safety and responsibility of request processing and interruption is managed internally. Synchronization objects are used to lock `requestQueue` in critical sections of code, and an event object is used after a request has been processed or interrupted to allow the thread to block until the main thread signals that more messages have arrived. Another benefit is that only the internal implementation of `Execute` changes for systems that do not support threads.

The `requestQueue` is used to record all `BrowserWindow`-invoked requests, and only the most recently invoked request is processed in the event of more than one invoked request.

`HttpRequest` is free to use blocking or non-blocking network-, file-, or memory-based I/O at its discretion to retrieve the web content associated with the request. Its only responsibility is to frequently allow the `RequestController` to throw an exception if the request should be interrupted in favor of a more recent request. Therefore, the only responsibility placed upon `HttpRequest` is that it perform all necessary clean-up and post-processing when the exception unwinds itself out of `CheckInterrupt` and into `Execute`.

Resource conservation is enforced by `RequestController` through thread-safe management of a queue in conjunction with a single worker thread which processes requests that are not interrupted until the request itself has allowed it to do so.

Applicability

Use the Override Current Processing pattern when:

- ***An invoker in one thread executes commands in another.*** Even if your system doesn't include threading, you may want to use this pattern to simulate the concurrent execution of a command. Cooperative multitasking is an alternative implementation that can be used in the `RequestController` class.
- ***Currently executing commands should be interrupted in favor of new commands.*** If you don't need this behavior, then this pattern is overkill. Use the `MacroCommand` object to execute `Command` objects in batch if interruption is not required.
- ***A command is executed often.*** You may not need this pattern if the command isn't executed often and merely creating an additional thread that handles the request is sufficient.
- ***A command takes an inordinate amount of time to execute.*** There may not be a need for this pattern if your commands are instantaneous or have predicable execution times that are acceptable to the user when synchronously executed.

Structure & Participants

Figure 2 illustrates a class model of the participants involved. The following is a list of participants and their responsibilities for the Override Current Processing pattern:

- **Command:**
  - The Command class is an interface class that defines an operation to decouple execution from invocation.
- **InterruptibleCommand** (`HttpRequest`):
  - Provides a default implementation of the `Execute` method that is overridden by subclasses.
  - Allows `CommandController` to set a pointer to the controller responsible for processing the `InterruptibleCommand` in question before it is executed.
  - Declares a `CheckInterrupt` method that should be frequently called. Delegates to controller's `CheckInterrupt` method.
- **CommandController** (`RequestController`):
  - Defines `Execute` method to allow clients to start command-processing loop. If multi-threaded, it begins the worker thread.
  - Defines `Invoke` used by invoker to delegate responsibility of executing `Command` object. Internally, it places it on a queue and signals the processing

loop to proceed.

– Events are fired back to invoker for both completion and exception cases. Expired and interrupted `Command` objects are both classified as exceptional cases.

– In multi-threaded systems, `CommandController` is responsible for guarding its `commandQueue` member with `queueMutex` so it can synchronize state changes between each thread involved in invoking and processing commands.

- **InvokingWindow** (`BrowserWindow`):

  – Composed of a `CommandController` object used to delegate invocation of `Command` objects.

  – Should only delete `Command` objects once the completion or exception event for that object has been received.

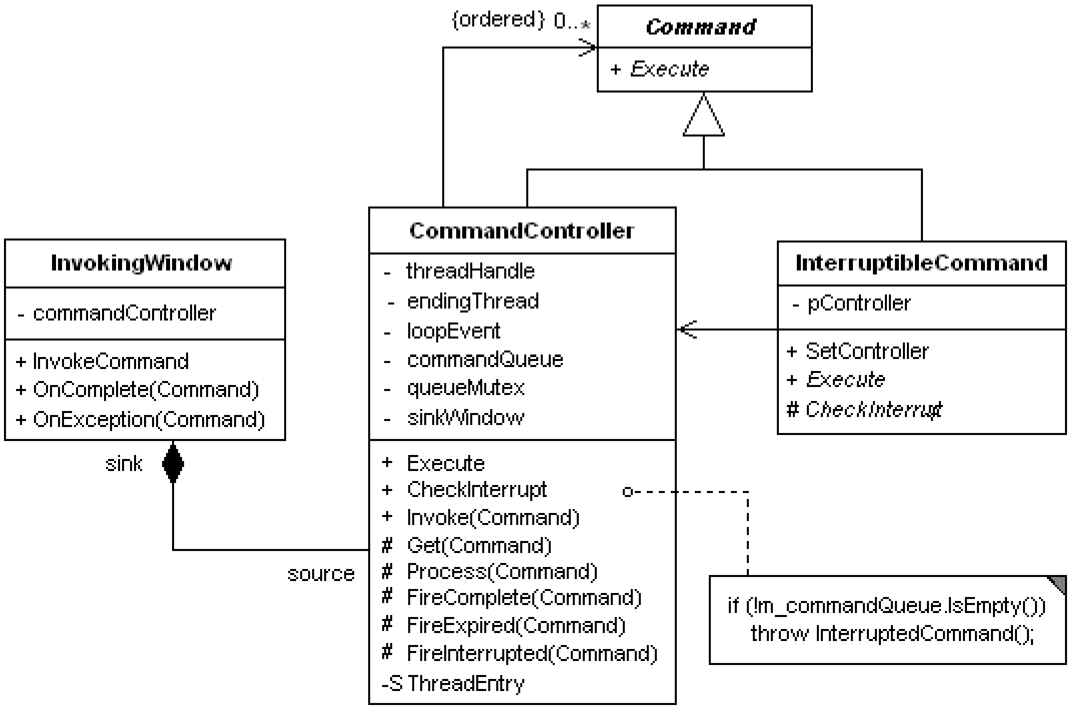

Figure 2 Structure of participants in the Override Current Processing pattern

Collaborations

Coming! And a cool sequence diagram will be done soon too!!!

Consequences

The following are the benefits and liabilities regarding usage of the Override Current Processing pattern:

- ***Termination of active thread.*** While the worker thread is processing the command, it is not possible to terminate the thread on most platforms without the risk of placing the system in an abnormal state. Therefore, when the command is blocking on a synchronous call (e.g., lengthy RPC or database query), it is designed to only interrupt the `InterruptibleCommand` object currently being processed once it calls the `CheckInterrupt` method implemented in `CommandController`.

- ***CommandController Flexibility.*** The `CommandController` is intentionally designed to be associated with the `Command` interface class as opposed to the `InterruptibleCommand` class. This allows `CommandController` the freedom to process both interruptible and non-interruptible commands.

- ***Conservation of resources.*** Multiple requests in a short amount of time will only result in the processing of one `InterruptibleCommand`. Undesirable data transmission for an interrupted command is alleviated.

Implementation

1. ***Waiting for commands.*** In multi-threaded implementations under Win32, the worker thread will block until an event synchronization object is set to signaled (for POSIX implementations, conditional variables can be used for equivalent behavior). The worker thread calls the controller's Get method to retrieve the next command for processing. The first thing should do is wait for the event to enter a signaled state.

   There are two places in `CommandController` where the event can be set to signaled in the main thread that allow Get to proceed. The first and most place is when an invoker delegates invocation of a `Command` object to `CCommandController`, it places it on the end of its commandQueue and sets the event to signaled. The second place is in the destructor for `CommandController`. Here, the `boolean` value `endingThread` is set to true and the event is signaled to instruct Get to empty the queue, exit the loop, and end the worker thread. The destructor waits for the thread to exit before continuing.

2. ***Creating and deleting commands.*** The `Command` object, if dynamically created, whether implemented as interruptible or not, should be created and destroyed in the context of only one thread. Many times caching strategies, history lists, or other features require that the `Command` object stay in memory for the duration of the application. Therefore, if the main thread creates it and delegates invocation to a `CommandController` object, it should remove them if and only if a completion or exception event has been received for that `Command`. If this is not acceptable, the Memento and/or State [Gamma+95] pattern can be used to externalize the state of a given Command. Another alternative is the Asynchronous Completion Token [Pyarali+98].

3. ***Multiple threads in CommandController.*** Implementing `CommandController` with multiple worker threads for command processing and retrieval can be accomplished with a few modifications. Rather than creating multiple instances of `CommandController`, a single instance utilizes multiple worker threads that compete for `commandQueue` and process commands independently. This is a good alternative if high invocation

throughput is expected or if response time when overriding the processing of commands can be lengthy or not granular enough to achieve the responsiveness the user desires.

Known Uses

- ***Web browsers.*** Most popular web browsers (e.g., Internet Explorer and Netscape Navigator) operate in this fashion. If the browser is retrieving the contents of a web page, and the user hyperlinks several times, the browser will cancel its command and only retrieve the contents of the last hyperlink clicked.

- ***Global investment bank contact management application.***

Related Patterns

Command, Thread Per Request (Douglas Schmidt)

Sample Code

TO DO: Add commentary around sample code and add client code.

```
////////////////////////////////////////////////////////////////////
// CInterruptibleCommand

class CCommandController;

class CInterruptibleCommand : public CCommand
{
    DECLARE_DYNAMIC(CInterruptibleCommand)

public:
    virtual ~CInterruptibleCommand();

    virtual void Execute();
    void SetController(CCommandController * pController)

protected:
    CInterruptibleCommand();
    virtual void CheckInterrupt();

private:
    CCommandController * m_pController;
};


IMPLEMENT_DYNAMIC(CInterruptibleCommand, CCommand)

void CInterruptibleCommand::SetController( CCommandController * pController)
{
    m_pController = pController;
}

void CInterruptibleCommand::CheckInterrupt()
{
    if (m_pController != 0)
    {
        m_pController-CheckInterrupt();
    }
}
```

Add more commentary here...

```
///////////////////////////////////////////////////////////////////////
// CCommandController

typedef CList<CCommand *, CCommand *&> CCommandQueue;

class CCommandController : public CCommand
{
    DECLARE_DYNAMIC(CCommandController)

private:
    class InterruptedCommand {};

public:
    CCommandController(HWND hwndSink);
    virtual ~CCommandController();

    void Execute();
    void Invoke(CCommand * pCommand);
    void CheckInterrupt();

protected:

    bool Get(CCommand *& rpCommand);
    void Process(CCommand * pCommand);

    void FireComplete(CCommand * pCommand);
    void FireExpired(CCommand * pCommand);
    void FireAborted(CCommand * pCommand);

private:
    static void ThreadEntry(void * pvData);
    unsigned long m_threadHandle;
    bool m_endingThread;
    CEvent m_loopEvent;
    CCriticalSection m_queueMutex;
    CCommandQueue m_commandQueue;
    HWND m_sinkWindow;
};

IMPLEMENT_DYNAMIC(CCommandController, CCommand)

const UINT WM_CMD_COMPLETE  = ::RegisterWindowMessage(_T("CMD_COMPLETE"));
const UINT WM_CMD_EXCEPTION = ::RegisterWindowMessage(_T("CMD_EXCEPTION"));

CCommandController::CCommandController(HWND hwndSink) :
    m_endingThread(FALSE), m_sinkWindow(hwndSink), m_threadHandle(0)
{
    m_sinkWindow = hwndSink;
    ASSERT(IsWindow(m_sinkWindow));
}

CCommandController::~CCommandController()
{
    if (m_threadHandle != 0)
    {
        m_endingThread = TRUE;
        m_loopEvent.SetEvent();
        WaitForSingleObject(HANDLE(m_threadHandle), INFINITE);
    }
}

///////////////////////////////////////////////////////////////////////
// Member functions called in the context of the main thread.

void CCommandController::Execute()
{
    if (m_threadHandle == 0)
    {
        m_threadHandle = _beginthread(ThreadEntry, 0, (void *)this);
    }
}

void CCommandController::Invoke(CCommand * pCommand)
{
    CSingleLock queueGuard(&m_queueMutex, TRUE);

    if (pCommand-IsKindOf(RUNTIME_CLASS(CInterruptibleCommand)))
    {
        STATIC_DOWNCAST(CInterruptibleCommand, pCommand)-SetController(this);
    }

    m_commandQueue.AddTail(pCommand);
    m_loopEvent.SetEvent();
}

//////////////////////////////////////////////////////////////////////
// Member functions called in the context of the worker thread. UINT

void CCommandController::ThreadEntry(void * pvData)
{
    CCommandController * pController = (CCommandController*)pvData;
    ASSERT_KINDOF(CCommandController, pController);
    CCommand * pCommand;

    while (pController-Get(pCommand))
    {
        pController-Process(pCommand);
    }

    _endthread();
}

bool CCommandController::Get(CCommand *& rpCommand)
{
    CSingleLock loopGuard(&m_loopEvent, TRUE);
    CSingleLock queueGuard(&m_queueMutex, TRUE);

    if (!m_endingThread)
    {
        if (!m_commandQueue.IsEmpty())
        {
            rpCommand = m_commandQueue.RemoveTail();
        }
    }

    POSITION pos = m_commandQueue.GetHeadPosition();

    while (pos != 0)
    {
        FireExpired(m_commandQueue.GetNext(pos));
    }

    m_commandQueue.RemoveAll();

    return !m_endingThread;
}

void CCommandController::Process(CCommand * pCommand)
{
    try
    {
        pCommand-Execute();
        FireComplete(pCommand);
    }
    catch(InterruptedCommand&)
    {
        FireAborted(pCommand);
    }
}

void CCommandController::CheckInterrupt()
```

```
    {
        CSingleLock queueGuard(&m_queueMutex, TRUE);

        if (!m_commandQueue.IsEmpty())
        {
            throw InterruptedCommand();
        }
    }

    void CCommandController::FireComplete( CCommand * pCommand)
    {
        PostMessage(m_sinkWindow, WM_CMD_COMPLETE, WPARAM( pCommand), 0L);
    }

    void CCommandController::FireAborted( CCommand * pCommand)
    {
        PostMessage(m_sinkWindow, WM_CMD_EXCEPTION, WPARAM( pCommand), 0L);
    }

    void CCommandController::FireExpired( CCommand * pCommand)
    {
        PostMessage(m_sinkWindow, WM_CMD_EXCEPTION, WPARAM( pCommand), 0L);
    }
```

And add more commentary here with some above in between key methods.

Conclusion

We'll find one!

Acknowledgements

Special Thanks to shepherd Aamod Sane for being the bad cop when I needed strongest feedback the most. Thanks to Steve Berczuk and Frank Wallingford for offering accidental feedback somewhere in between the first and second drafts. And a very solid thanks to Program Committeeman Jens Coldewey for bailing me out of the Valley of Despair at the perfect moment.

References

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addision-Wesley, 1995.

[Pyarali+98] I. Pyarali, T. Harrison, and D.C. Schmidt. "Asynchronous Completion Token," from *Pattern Languages of Program Design 3,* eds. R. Martin, D. Riehle, and F. Buschmann. Reading, MA: Addison-Wesley, 1998.