Microthread

An Object Behavioral Pattern for Managing Object Execution

Joe Hoffert and Kenneth Goldman
{joeh,kjg}@cs.wustl.edu
Distributed Programing Environments Group
Department of Computer Science,
Washington University, St. Louis, MO. 63130, U.S.A.

Abstract

There are times when the execution of an object needs to be suspended and later resumed either due to the object responding to external events or processing requirements internal to the object. The Microthread pattern simplifies the management of an object's execution when the object needs to start, stop, and resume its execution. This pattern has shown itself to be useful in both stand alone and distributed applications. An example usage is shown along with the benefits and liabilities of using the pattern. An implementation outline is also provided along with some sample code. Finally, patterns related to the Microthread pattern are listed.

1.0 Intent

Allow an object to start, stop, and resume its execution. This allows an object to be contextually sensitive to events during its execution or to quiesce while waiting for events or resources needed to continue execution.

2.0 Also Known As

Control Message

3.0 Classification

Object Behavioral

4.0 Motivation/Example

A controller for playing audio or video files should be well encapsulated to shield a user from having to know intricate details about how an audio or video file is played. It should also present a fairly simple and intuitive interface for manipulating the audio or video stream. The user should be able to specify a file and have the controller respond appropriately to relatively simple commands such as start, stop, and resume. In order to be intuitive these controllers should mimic hardware devices such as VCRs and cassette players. (See Figure 1.)

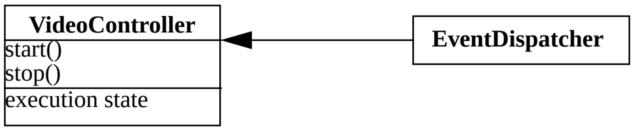

FIGURE 1. Video Controller Example

When a video controller is first told to start it begins to play the specified video file. When the controller is told to stop it must keep track of where the current playback is so that when it resumes it will pick up where it left off. Keeping track of state information allows for greater flexibility. If the controller is playing a video and is told to play, this event may be ignored or may be used as a signal to fast-forward. Likewise, if a controller is stopped and is told to stop, this may cause the controller to ignore the event, reposition the playback to the beginning of the file, or unload the currently viewed file.

To give the controller more flexibility and performance, it may register or unregister for certain events given its current execution state. For example, one way to have the video controller ignore a stop event when it is already stopped is simply to unregister for stop events. It can tell an *EventDispatcher* that it is no longer interested in stop events. This will increase performance since the controller won't have to take the time of processing a stop event simply to ignore it.

The above discussion assumes that the *EventDispatcher* knows specifically about the *VideoController* and its different methods. To decouple this relationship, an interface is defined that an *EventDispatcher* uses and to which the *VideoController* conforms. Now different controllers can register or unregister themselves with *EventDispatcher*s and the *EventDispatcher*s don't need to know any details about *Controller*s other than they support the common *Controller* interface. An *EventDispatcher* simply dispatches the current event to a *Controller* by invoking the *Controller*'s *handleEvent* method. Initially, a *Controller* is told to run itself. The *Controller* can then take whatever steps it needs to prepare for events or to initiate any other functionality.

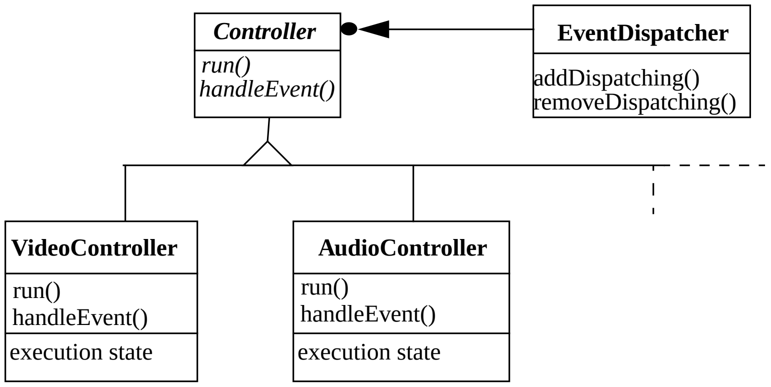

FIGURE 2. Decoupled Controller Example. Controller is able to register and unregister with the EventDispatcher for events of interest.

This structure can be generalized further to support other types of interactions. A component may start execution and then later suspend its execution either because of external events (e.g., a stop event for a video controller) or because of internal processing logic (e.g., the object can't continue execution until it has received certain resources or certain responses from collaborating components). For instance, there may be components in a distributed system that need to negotiate functionality between themselves. Initially, when the first component is run it proposes certain functionality between itself and another component and sends this proposal to the second component. The first component cannot complete the functionality on its own and must wait until it has received a reply from the second component. This reply may also specify what negotiated functionality is actually to be implemented.

The dispatching of events can also be generalized to provide greater flexibility. Sometimes it is appropriate to broadcast events. That is, the event is provided to every potentially interested consumer. There are other times, however, when an event should only be consumed by a single consumer. That is, when an event comes in one and only one consumer of the event is allowed. To provide for this functionality, a potential consumer of an event can be queried as to whether or not it is taking over the event. This indicates whether or not the event is to be passed on to other potential consumers.

The *Controller*s that start, stop, and resume execution can be thought of as very lightweight threads which we call *Microthread*s. In multitasking operating systems, a process or thread runs for a time and then stops (typically due to timeouts or I/O) and then starts up again (because time or other resources are available). These *Microthread*s mimic this behavior. The *Microthread* pattern allows an object to provide this sort of functionality. It facilitates an object's ability to run for a time, stop, and then continue execution while providing a simple and clean interface along with flexibility in processing events.

It is often helpful to have internal (e.g., protected or private in C++) *Microthread* methods to simplify execution processing. Particularly, *resume()* is helpful in encapsulating how an object will resume its execution based on its state and *suspend()/unsuspend()* are useful in encapsulating how a *Microthread* interacts with the *EventDispatcher*(s) when stopping or resuming execution.

5.0 Applicability:

Use the *Microthread* pattern when you want to:

- allow an object to start, stop, and resume execution based on external events or its own internal logic; *or*
- enable functionality where objects (possibly distributed) need to interact with and wait for each other; *or*
- use threads but the operating system does not support them; *or*
- have fine grained control of scheduling the executions of objects; *or*
- have objects that compete for scarce and/or non-shareable resources during their executions. One object starts with the resource. When it is done it relinquishes the resource. This could be noted as an event by an *EventDispatcher* and the *EventDispatcher* can pass this event to any suspended *Microthread*s waiting for that resource; *or*
- process multiple negotiations simultaneously without blocking. Using the *Microthread* pattern, an application can process multiple negotiations at the same time without one of the negotiations blocking all the others. A *Microthread* does not need to run to completion before any other *Microthread* is also allowed to run. Several negotiations, represented by the different interactions of several *Microthread*s, may be ongoing concurrently.

Do *not* use the *Microthread* pattern if you:

- only need simple messages passed between distributed components with no replies, negotiations, or coordination of events; *or*
- want to suspend and resume the execution of an object at arbitrary points to support multi-tasking or load balancing, for instance. Operating system threads are well-suited for this type of use. The *Microthread* pattern deals with autonomous objects and their interactions. Operating system threads deal with groups of objects or processes that are managed as a group for some global purpose.

6.0 Structure

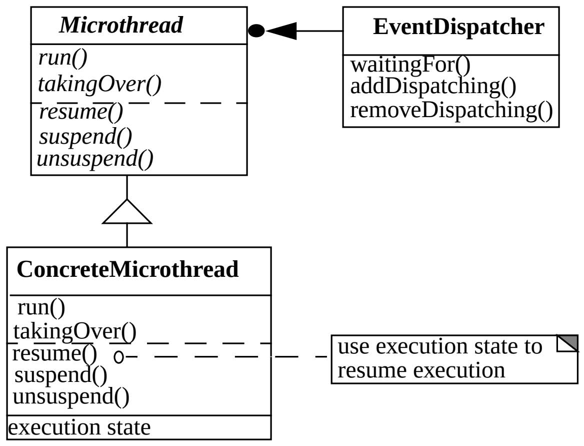

FIGURE 3. Structure of the Microthread Pattern

7.0 Participants

- Microthread (Controller)

  - defines the interface for all the concrete *Microthread*s.
- ConcreteMicrothread (ConcreteController)

  - implements the *Microthread* interface.
- EventDispatcher (EventDispatcher)

  - passes events to *Microthread*s so that they can process the events.

8.0 Collaborations

- *ConcreteMicrothread*s add themselves to *EventDispatcher*s (via the *addDispatching* method) when they are interested in incoming events. They remove themselves from the *EventDispatcher*s (via the *removeDispatching* method) when they are no longer interested in incoming events.

- A *ConcreteMicrothread* indicates to the *EventDispatcher* whether or not it is consuming an event via its return value from the *takingOver()* method.

9.0 Consequences

9.1 Benefits

The Microthread pattern offers the following benefits:

Separation of Concerns: The *Microthread* pattern decouples events from their receivers. This not only alleviates the overhead of binding events to specific receivers but also accommodates multiple receivers for any one event.

It decouples the object that dispatches events (i.e., *EventDispatcher*) from the object that determines interest in events (i.e., *Microthread*). *Microthread*s can change the type of events in which they are interested simply by unregistering themselves with one type of *EventDispatcher* and registering with another type of *EventDispatcher*. Additionally, *Microthread*s can register themselves with multiple *EventDispatcher*s if they are interested in several types of events.

Moreover, the policy of deciding how *Microthread*s are notified of events is separated from the processing of that event. The *EventDispatcher* decides how the *Microthread*s are notified of an event but the *Microthread* decides how to proceed with execution once the event has been delivered. It is easy to change the policy of how messages are notified without affecting how the event is processed.

Flexibility: The *Microthread* pattern allows *Microthread*s to suspend and resume execution any number of times before completing their executions.

Localization of Functionality: The *Microthread* pattern allows *Microthread*s to have a life of their own. They need not be managed by any other object and need not have any long-term dependencies to any objects. All the information needed to begin, suspend, and resume execution (including selection of pertinent events) is encapsulated within the *Microthread*s.

9.2 Liabilities

The Microthread pattern has the following liabilities:

Potential Interface Bloat: The *Microthread* pattern increases the size of the interface for objects due to the extra methods of *run, takingOver, resume, suspend,* and *unsuspend*.

Dangling Suspended Microthreads/Microthread “Leaks”: Suspended *Microthread*s may never resume execution if the events for which they are interested never arrive. These *Microthread*s are still queried when new events arrive which takes up processing time. Timeouts can be used to remove “outdated” suspended *Microthread*s but there is always the issue of how long the timeout should be.

Heavyweight Messages: *Microthread*s may be too heavyweight for some applications. For some distributed applications, protocols only need simple messages passed between components. They may not require replies or negotiations.

10.0 Implementation

This section describes how the implementation of the Microthread pattern might look in C++. The implementation described below is influenced by Playground [1], a distributed programming environment, which uses the Microthread pattern to negotiate functionality between components.

- **Determine events of interest:** Typically, for any application there are several different types of events that occur. Determine which events will be of interest for the *Microthread* subclasses. Specifically, determine about which events *Microthread* subclasses will want to be notified and how the subclasses will react to these events.
- **Implement the *Microthread* interface for the *ConcreteMicrothread* subclasses:** Each applicable *Microthread* subclass will need to implement the methods as declared by the *Microthread* interface. Determine how each concrete *Microthread* will suspend its execution - namely to which *EventDispatcher*(s) will it suspend itself. Determine what applicable state information is needed so that *Microthread*s can resume execution appropriately.

  The state information is used to determine how to resume execution once it has stopped. Each *Microthread* must essentially encode its "program counter" in its state. It then saves this program counter when it suspends itself and branches to it when its execution is resumed.
- **Determine which EventDispatchers will handle which events:** Once the relevant events have been determined, the programmer needs to decide how the events will be handled. Specifically, determine which *EventDispatcher*s will handle which events. There are several different approaches.

  One approach is to have a single *EventDispatcher* handling all events. This may be appropriate for applications that do not anticipate the queue of interested *Microthread*s to be very large. If typically there are only a few *Microthread*s that are waiting for events and there are few events being generated, this strategy probably makes the most sense.

  However, if there will be several *Microthread*s waiting for events and many events being generated then having a single *EventDispatcher* may create a processing bottleneck. An undesirable amount of time may be spent in the *EventDispatcher* iterating through all the suspended *Microthread*s for each incoming event. Additionally, this time is compounded with many events coming in.

  A second approach is to have one *EventDispatcher* for each type of event. If there will be many types of events that will be dispatched and many events being generated, it may make sense to have a separate *EventDispatcher* for each type of event. This will speed up event dispatching since events will only be passed to *Microthread*s that are interested in that type of event.

  This approach does add some complexity since the incoming events will need to be demultiplexed to their appropriate *EventDispatcher*s. Additionally, information will be needed for the incoming events to determine where they should be sent which may increase coupling and reduce information hiding. The *Reactor* pattern [2] can be helpful in demultiplexing events for distributed applications. Each *EventDispatcher* would be a *ConcreteEventHandler* in this pattern.
- **Determine default processing for events:** It may be appropriate for a component to receive events where no *Microthread* is waiting for it. For this case it makes sense to define default processing for these events. For example, incoming events may not be meant for any suspended *Microthread* but instead may be stand alone executable events. In this case, it may be appropriate to run these events. *EventDispatcher*s could have the default behavior of telling events to

run themselves if no *Microthread*s are waiting on them. *EventDispatcher*s could also simply return whether or not the event was consumed. Some other object could then handle default processing.

For other components, it may never make sense to have incoming events that do not have *Microthread*s waiting for them. In this case, it may be appropriate to ignore the event and optionally report a warning or error.

Variations:

Lists for Event Types: *EventDispatcher*s can have lists of different types of suspended *Microthread*s that are only applicable for a specific type of event. This assumes certain types of events are only applicable to certain types of *Microthread* subclasses. This can decrease processing time at the cost of coupling the *EventDispatcher*s with concrete *Microthread* classes since now *EventDispatcher*s must know about specific *Microthread* subclasses.

Event Notification Without Consumption: Some *Microthread*s may want to be made aware of an incoming event but are not interested in consuming the event. This may occur for monitoring purposes, for example. These *Microthread*s would be notified of the event but would leave it for some other *Microthread* to consume. This can easily be facilitated by adding monitoring functionality to a *Microthread*'s *takingOver* method. The *Microthread* could do whatever bookkeeping it wanted to do with the event and the *takingOver*'s return value would indicate that the event was not consumed.

Pipelined Event Transformation: Some *Microthread*s may want to transform the event and return it to the *EventDispatcher* so that it instead passes the transformed event on to subsequent waiting *Microthread*s. For example, this may be desirable in the case where events are encrypted and need to be decrypted for further processing. The decrypting *Microthread* checks if the event is decrypted. If it is, it will decrypt it and return it to the *EventDispatcher* to pass along in place of the original event. This can be facilitated by changing the return value of the takingOver method from a boolean to a pointer to an event. If the return value is a NULL pointer the event has been consumed. Otherwise, the returned event would be passed to the remaining waiting *Microthread*s.

With this approach there may need to be an ordering placed on suspended *Microthread*s in the pipeline. Clearly in the case of a *Microthread* waiting to decrypt applicable events it should be passed any incoming events before other *Microthread*s that are expecting decrypted events. When *Microthread*s suspend themselves on *EventDispatcher*s they can pass a priority. The *EventDispatcher* will then know which suspended *Microthread*s should be queried first when new events arrive.

EventDispatcher as Broadcaster: Some applications may only need support for events being broadcast to all interested *Microthread*s. The *EventDispatcher*(s) can be configured to send all incoming events to all interested *Microthread*s ignoring consumption of the event by any one *Microthread*. This is how the Java event model works. Events are broadcast to all *EventListener*s that are interested in that type of event.

11.0 Sample Code

When an *EventDispatcher* receives an incoming event it queries its interested *Microthread*s. The *EventDispatcher* class header and the implementation of *waitingFor* might look something like this in C++:

```
class EventDispatcher {
public:
  EventDispatcher();
  virtual ~EventDispatcher();

  // Check if the dispatcher is waiting for this event
  bool waitingFor(Event* event);

  // Add this microthread to the list of microthreads interested in events
  void addDispatching(Microthread* microthread);

  // Remove this microthread from the list of microthreads interested in events
  void removeDispatching(Microthread* microthread);

private:
  List<Microthread *> mtList_;
};

bool
EventDispatcher::waitingFor(Event* event)
{
  // Iterate through the list of waiting microthreads to see if one of them
  //  wants to take control of the passed-in event
  for (mtList_.begin(); !mtList_.atEnd(); mtList_++) {
    if ((*mtList_)->takingOver(event)) {
      return true;
    }
  }
  return false;
}
```

In the code example above, the *EventDispatcher* does not handle an unconsumed event. Instead it returns whether or not the event was consumed. The default behavior of an unconsumed event is the responsibility of the calling object.

When a concrete *Microthread*'s *takingOver* method is called to query it about an incoming event it might handle the event in the following manner:

```
bool
ConcreteMicrothread::takingOver(Event* event)
{
  bool takenOver = false;

  if (interestedIn(event)) {
    // Remove dispatching for this microthread
    getEventDispatcher()->removeDispatching(this);

    // Resume running of the microthread - non-public helper method
    resume(event);

    takenOver = true;
  }
  return takenOver;
}
```

When a concrete *Microthread* resumes execution it can check its internal state and process the event accordingly. Note that *Microthread*s do their work in discrete chunks. The wake-up point (the *Microthread* version of a program counter) is a way of tracking the progress of the *Microthread* in completing its tasks. When moving to the next task requires information from another event, the *Microthread* suspends itself waiting for that event.

```
void
ConcreteMicrothread::resume(Event* event)
{
  switch (internalState) {
  case INTERNALSTATE1:
    processInternalState1Event(event);
    break;
  case INTERNALSTATE2:
    processInternalState2Event(event);
    break;
  default:
    throw "Bad internal state for ConcreteMicrothread";
    break;
  }
}
```

In this code example, the valid execution states for the *ConcreteMicrothread* is denoted by an enumeration.

12.0 Known Uses

12.1 Playground Distributed Programming Environment [1]

Often in distributed applications, protocols are used to negotiate functionality between distributed components. As part of a protocol one component receives a request for some distributed functionality. It will then collaborate with one or more other components to provide the requested functionality.

Protocol negotiations may be needed, for example, to determine which options (if any) are supported for a particular functionality. They may also be needed to determine if the functionality can be realized between the targeted components. The interaction of components to enable some specified functionality is illustrated in Figure 4.

One component sends a request to another component. To achieve the desired functionality the enabling component collaborates with a third component. It sends a message to this collaborating component saying "Here are all the options I support for the requested functionality. Which of these do you support if any?" The collaborating component may then respond "I do not support what you need", "There's not enough common functionality between us to do the job", or "I support what you need but with these qualifications." The enabling component needs to wait on the collaborating component's response to proceed.

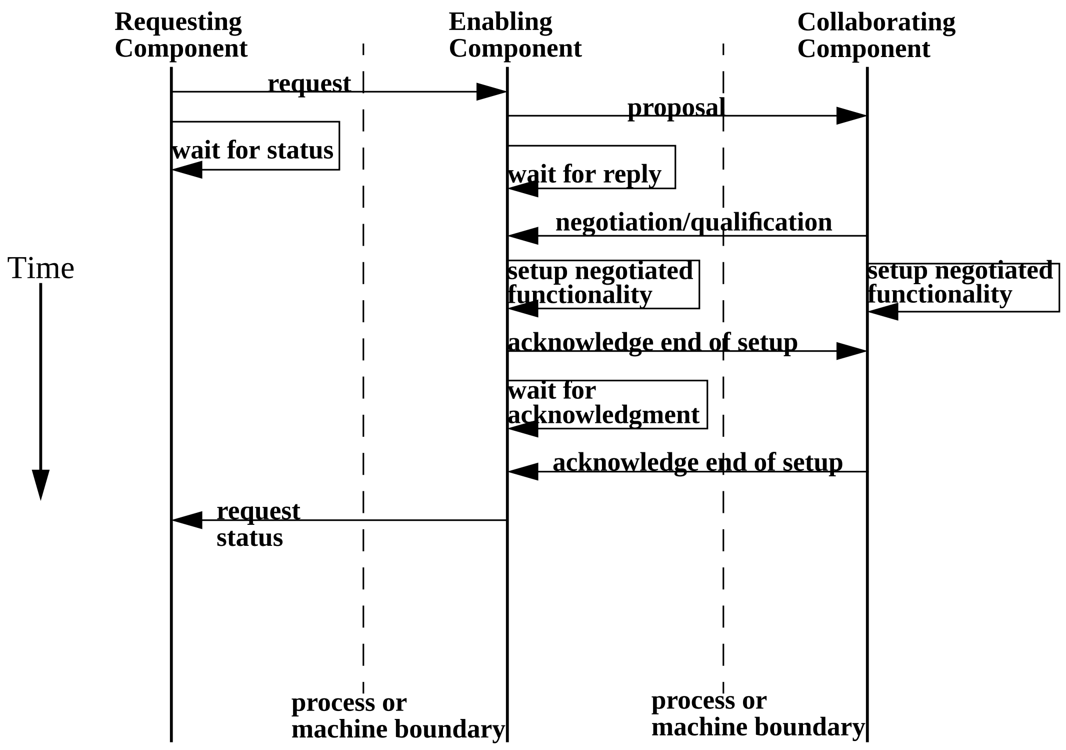

FIGURE 4. Interaction Diagram For Protocol Negotiation

There may be several negotiation or collaboration stages between distributed components to accommodate certain functionality. The distributed components may need to do some processing on their own, send off an event to the other coordinating components, and then wait for a reply. This cycle may occur several times before the desired capability is enabled.

Complex protocols can be simplified using the *Microthread* pattern. *Microthread*s are created and then sent to the appropriate components. When an initial *Microthread* is received by another component it is told to run itself (the default behavior for an event that is not consumed). The *Microthread* will then execute until it needs to coordinate with another component or components. At this point, it will suspend its execution waiting for the appropriate coordination event. The *Microthread* determines when it will be suspended and on which *EventDispatcher*(s) it will wait for events.

When a *Microthread* receives an event of interest it checks to see if it has all the information or resources needed to continue execution. If so, it will unregister itself from the *EventDispatcher* (by calling its *removeDispatching* method) and resume its execution. Otherwise, it will continue to be suspended waiting for events. Since the current execution state of the *Microthread* is kept it knows the appropriate context when resuming execution.

Some of the protocols supported by the Playground distributed programming environment are non-trivial and involve several different types of *Microthread*s (Figure 5). Some of the *Microthread*s need to wait for replies at several different stages of their executions. For example, in the lifetime of a *ProposedLinkMicrothread* it can wait on up to three different kinds of events (replies from its collaborating *NegotiatedLinkMicrothread*) depending on its current execution state. It also knows what types of responses are appropriate for its particular execution state. It

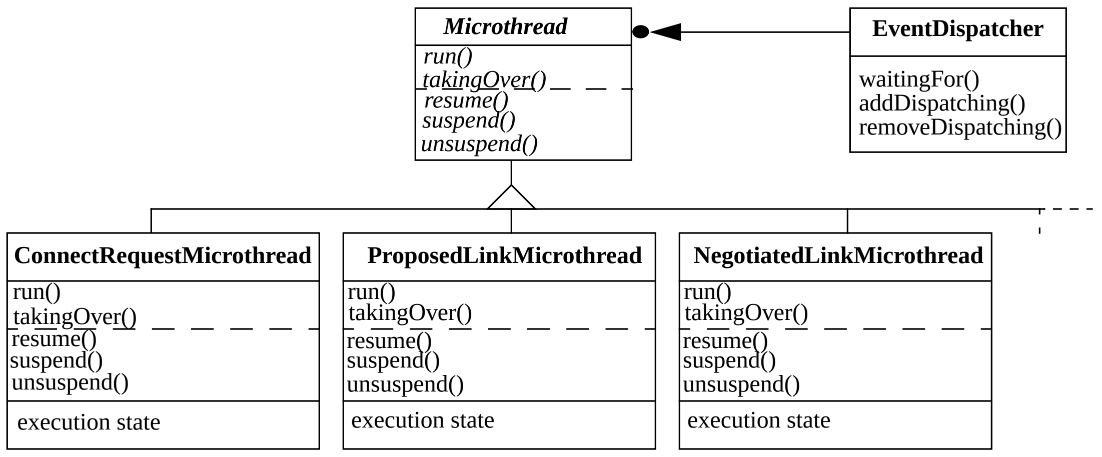

FIGURE 5. Playground Example

will report an error if the response is inappropriate for a given state. Moreover, several protocol negotiations may be occurring simultaneously.

12.2 Java Media Framework

The Java Media Framework (JMF) uses the Microthread pattern to control audio and video players. The event dispatching in JMF appears not to be as flexible as is described in the pattern but this flexibility may not be needed. Typically, events are broadcast to all potentially interested *Microthread*s rather than querying each interested *Microthread* one at a time.

12.3 Multitasking Operating Systems

Typically, multitasking operating systems use the Microthread pattern for process management in a non-object-oriented way. Operating systems suspend the execution of processes due to timeout or I/O events and then resume the processes at a later point until the process has completed its execution.

13.0 Related Patterns

The following patterns relate to the Microthread Pattern:

- The *Command* pattern [3] is used to make *Microthread*s executable. A key component of the *Microthread* pattern is to extend *Command*s to facilitate suspension and resumption of execution.
- The *External Chain of Responsibility* pattern [4] can be used to pass incoming events to potential receivers.
- The *Strategy* pattern [3] can be used to facilitate different algorithms that *EventDispatcher*s employ to dispatch events.
- The *Iterator* pattern [3] can be used in an *EventDispatcher* to iterate through the interested *Microthread*s.

References

[1] K. Goldman, B. Swaminathan, P. McCartney, M. Anderson, R. Sethuraman, “The Programmers'

Playground: I/O Abstraction for User-Configurable Distributed Applications”. *IEEE Transactions on Software Engineering*, 21(9):735-746, September 1995.

[2] D. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching,” in *Pattern Languages of Program Design* (J. Coplien and D. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995

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

[4] J. Hoffert, “Resolving Design Problems in Distributed Programming Environments”, *C++ Report*, SIGS, Vol. 10, No. 7, July/August 1998