SimDispatch Element

The dispatcher's primary function is to coordinate the timing of events across multiple processing elements in the simulation. Its role comprises of two parts:

1. The EventQueue and related operations
2. Message (Packet) communication

Event Queue

The event queue is responsible for the execution of tasks in the proper chronological order across multiple processing elements. It does so by maintaining a priority queue that is arranged by an arbitrary priority metric, most likely a global time.

The queue interface itself is straightforward, supporting enqueue(), dequeue(), and a peek() method (that does not dequeue the next element but returns it.) An isEmpty() method also returns if there are remaining elements.

Class signature

```
class SimEventQueue {
      public void enqueue(SimEvent argEvent, long priority);
      public SimEvent dequeue();
      public SimEvent peek();
      public boolean isEmpty();
}
```

Message Communication

In order for the dispatcher to gain control of the simulation at appropriate times, all messages generated be processing elements will be sent to the dispatcher, which will then forward the message along to the appropriate destination element when all other units in the simulation have caught up.

For instance, a PIM may implement *send* and *receive* primitives for communication. The *send* primitive should call a method that invokes the dispatcher, passing the message and a priority (time). If the PIM is done processing, it will send a message that indicates there is no more processing to be performed, which will then allow the dispatcher to pull the next event from the queue, and schedule the appropriate processing element to run. The *receive* primitive would put the executing processing element to sleep and request a notification from the dispatcher when the anticipated packet arrives. When a processing element has no work to perform, it is in a state of *receive* until the next message arrives.

In order for the callback mechanism to work, the processing elements must first register themselves with the dispatcher. All processing elements have a method that responds to messages from the dispatcher, and can be used to wake up from a *receive* call.

Class signatures

```
class SimDispatchElement extends SimElement {

      //For singleton design pattern
      private SimDispatchElement();
      private SimDispatchElement mySelf;
      public SimDispatchElement getInstance();

      //For processing element callback
      public boolean registerProcessingElement(SimProcessingElement argUnit)

      //Notification of a message from a processing element
      public void postEvent(SimEvent argEvent, long argTime);

}
```

```
class SimProcessingElement extends SimElement {
      public int processEvent(SimEvent argEvent);
}
```

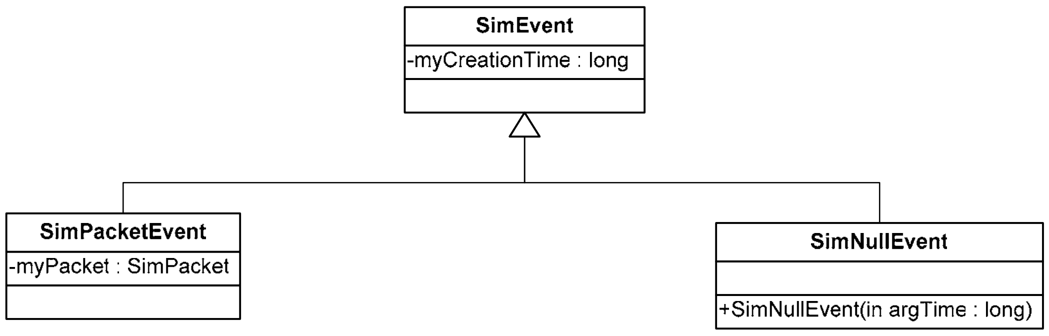
```
class SimPIM extends SimProcessingElement {

      //communications primitives
      public void send(SimPacket argPacket);
      public SimPacket receive();

      //execution at this PIM has finished
      public void finished();
}
```

Events

The simulation is guided by a chronological progression of events. There are two main classes of events in the simulation:

1. Events that are associated with data transfer and messages in the simulated architecture
2. Events that indicate transitions in the state of the simulation

This yields a simple class hierarchy of:



The SimEvent class is subclassed by the two subdivisions of messages classified by their payload. The precise details of the SimPacket class are left for external specification.

Note: *This subdivision could also be done by the division suggested above, with the two subclasses being SimSimulationEvent and SimSimulatedEvent to represent simulation control messages and simulated messages respectively.*

Currently, the implementation suggests two subclasses of SimPacketEvent to correspond with a message being sent from a processing element, and a message being received by a processing element. One subclass of SimNullEvent to notify the dispatcher that a processing element has finished its work would also be necessary. Additional subclasses can be added as appropriate.