Abstract Server Design

The MIS Server will need to process concurrent messages for both monitoring and information handling. For this reason the server architecture is abstracted away from either of these tasks so that it can be reused for both.

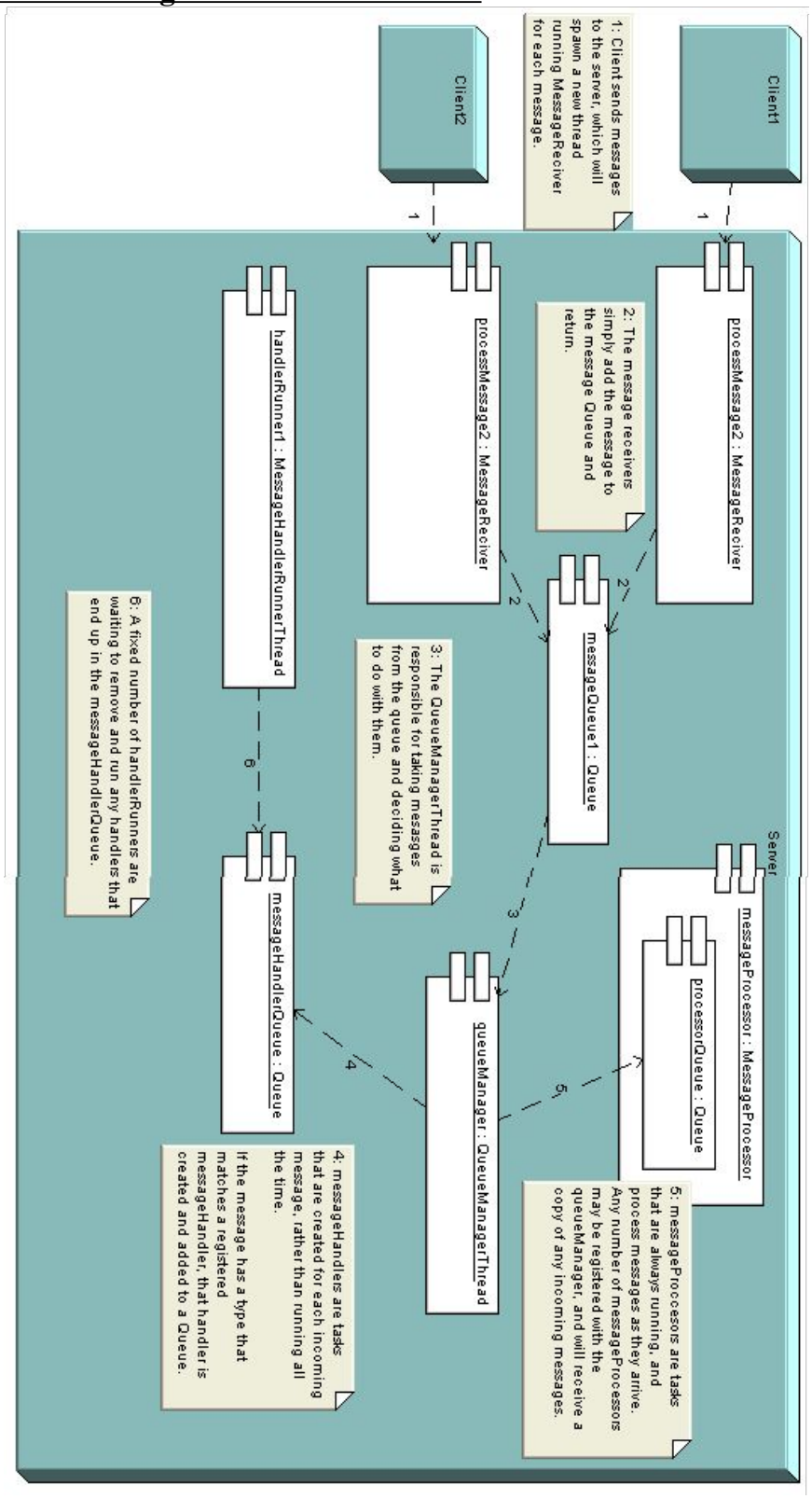
The structure revolves around using threadsafe queues to store messages until they are ready to be processed. Messages are received by a fast, non-blocking MessageReceiver, which simply puts them into the main queue.

The actual message handling is done either by dedicated threads (processors) or by code invoked only on the recipt of a message (handlers)

The queue manager handles the dispatch of messages to any interested handlers or processors. Each processor has its own queue of messages awaiting processing, so it can asynchronously work its way through the messages.

The handlers are added to a queue of handlers waiting to be run. This queue is monitored by a fixed number of handlerRunner threads, which run these messageHandlers.

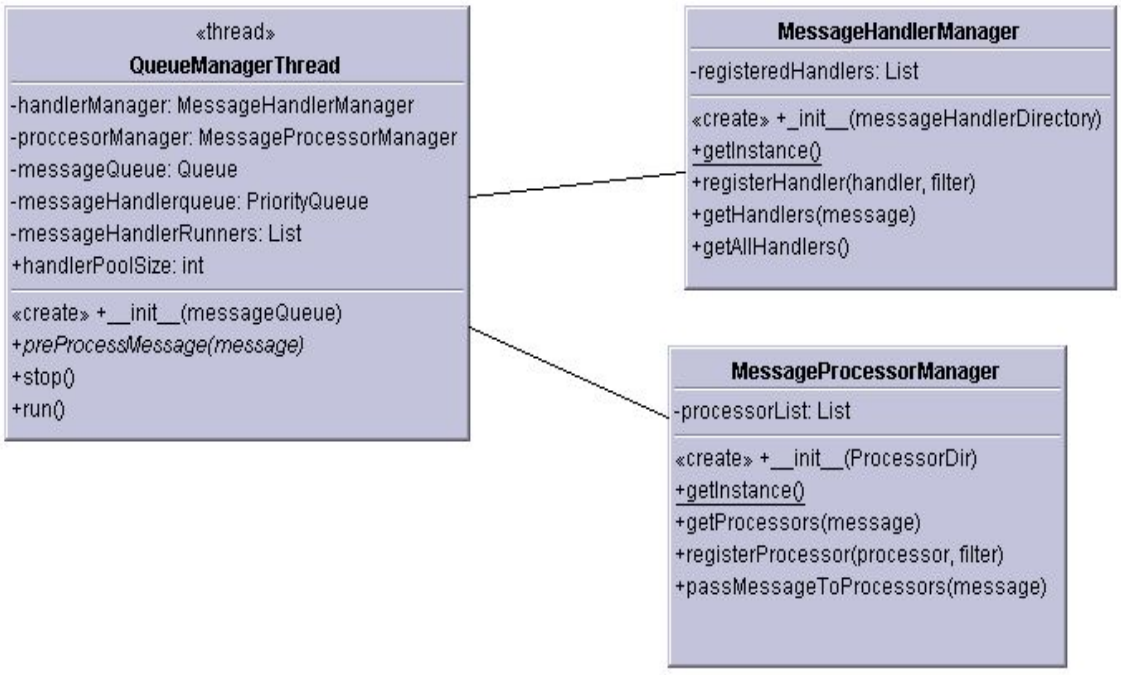
Message Flow Through The Abstract Server



The Queue Manager

The **QueueManager** pulls messages out of the queue and decides what to do with them. To do this, it must find which handlers and processors are interested in the message. The information on what handlers are available, and which messages they should receive is held by the singleton classes **MessageHandlerManager** and M**essageProcessorManager**.

These keep a list of handlers and processors, registered either through calling a *register...* method, or by scanning a directory for the relevant classes and register them all.



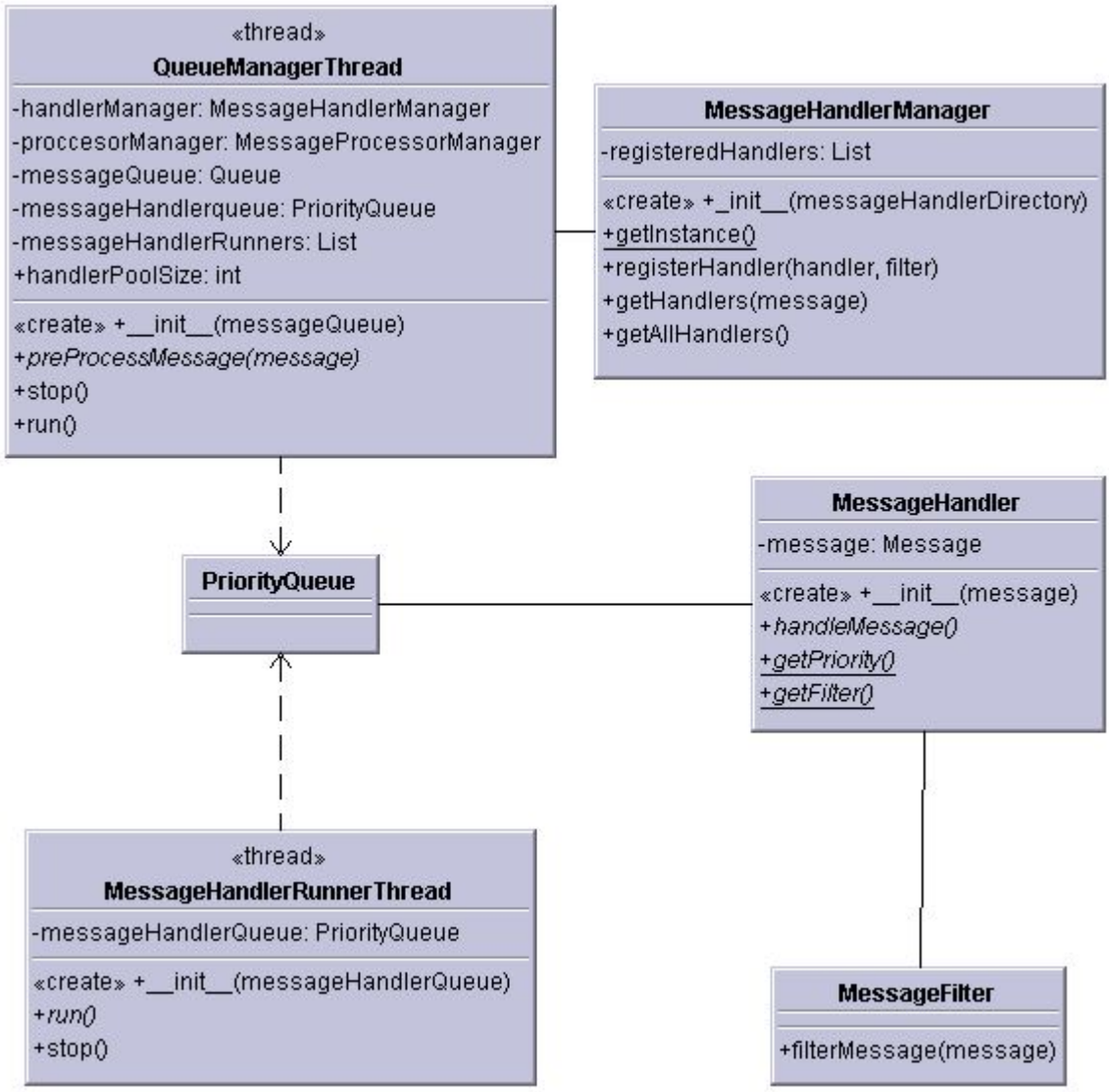
In order to determine which messages a handler or processor should receive, it defines a **MessageFilter**, this is an abstract class and has one method, *filterMessage*, which returns true if the handler is interested in this message.

The abstract method *preProcessEvent* is invoked before any handlers or processors, and could be used, for instance, to unpack a CORBA representation in the message.

The Message Handlers

The messageHandlers are dealt with first. So as not to hold up the main message queue, the handlers run in their own threads. A pool of these threads is started by the **QueueManagerThread**, and these monitor the messageHandlerQueue, taking **MessageHandler**s from it one at a time and running them.

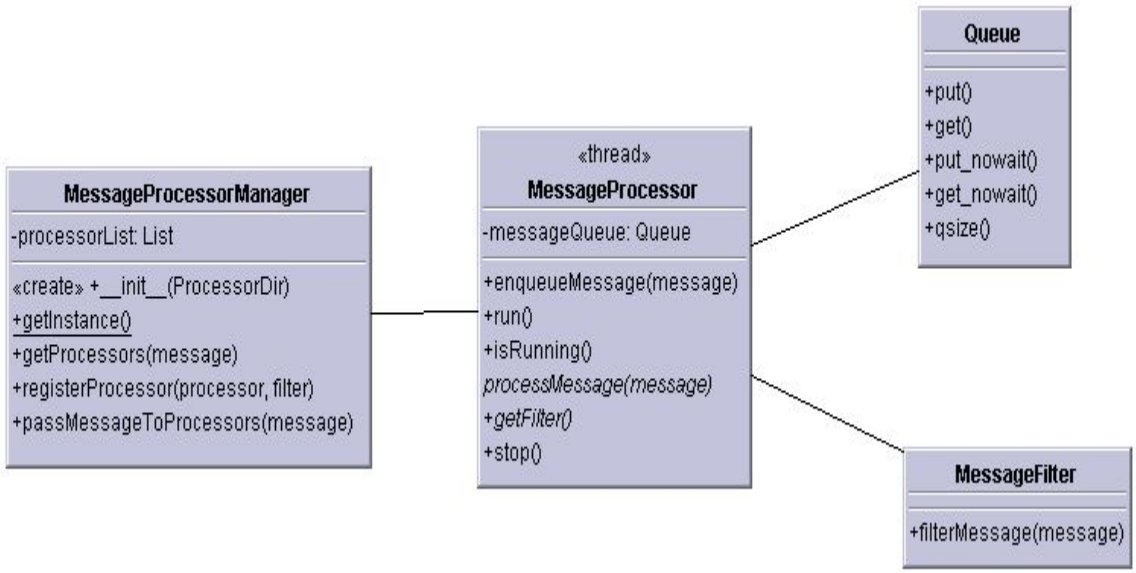
MessageHandlers are inserted into this queue by the **QueueManagerThread**, after being looked up in the **MessageHandlerManager**. The abstract method *handleMessage* does the actual work.



The handlers can be given an integer priority, those handlers with a high priority will be given to the runner thread first.

The Message Processors

MessageProcessors are given the message next. The QueueManagerThread passes the message to the messageProcessorManager, which in turn passes it to all relevant processors, by calling passMessageToProcessors.



The message then sits in the processors queue until it is ready for it – the *processMessage* abstract method is called by the thread as messages arrive. When this method terminates, the thread sleeps until a new message is ready to be processed.

The message processor threads are started as they are registered.

Order of execution

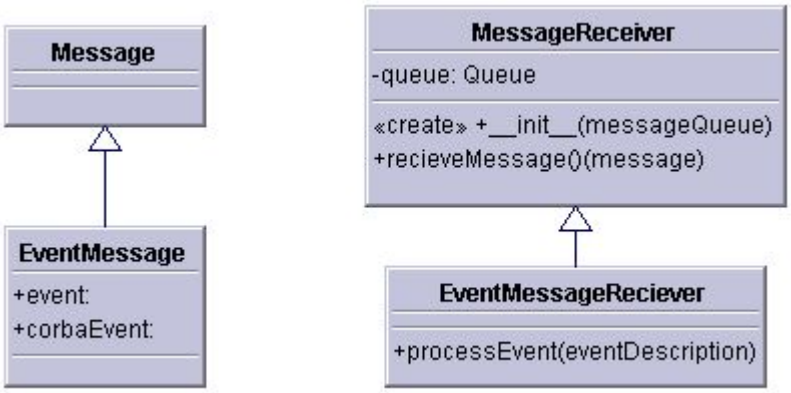
It should be noted that if the order in which message related code run is important, then it must be in the *preProcessEvent* method in the **queueManager.**
Because **eventProcessors** are completley asynchronous, even though they may be give messages in a particular order one may take longer than another to get around to processing that message, so that the order in which they actually begin processing is undefined.

EventHandlers are inserted into a priority queue, which means that although higher priority event handlers are started before lower priortity ones, they are not guaranteed to have finished first unless there is only one **handlerRunnerThread**

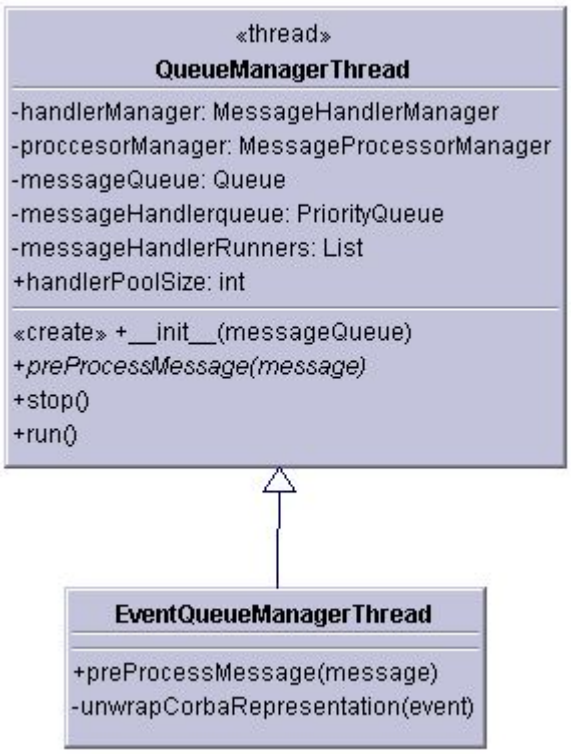
As a consequence, if any code depends critically on other code having run first, it must all be in one handler, or the first part of the code must be in the messageQueueManager's *preProcessEvent,* or there must be only 1 handlerRunner

Specializing the Server for the MIS Event Monitoring Server.

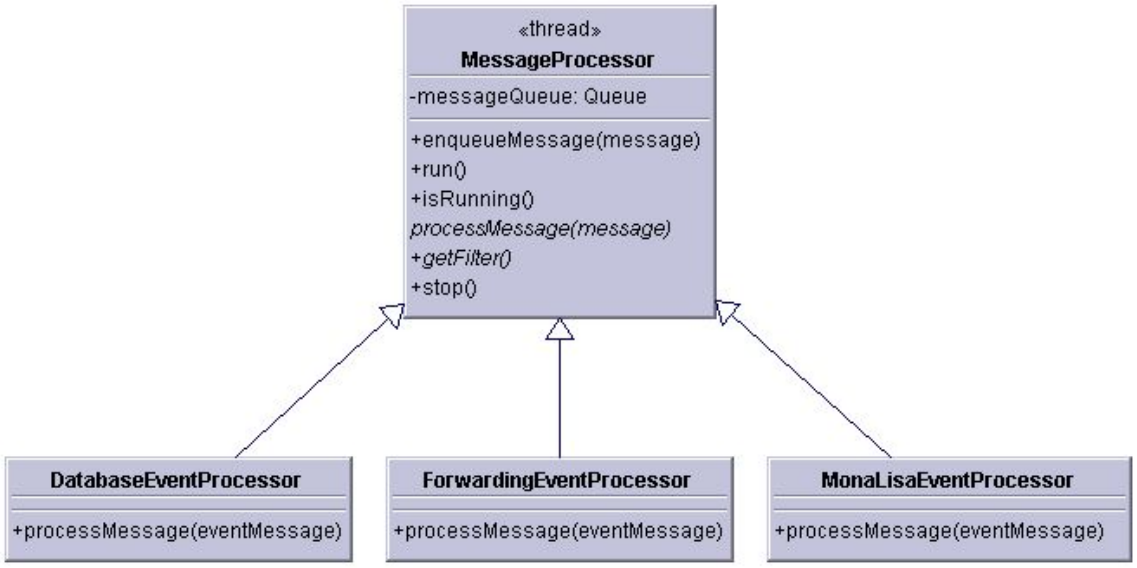
In the monitoring service, events are passed to the **EventMessageReceiver** remote interface's *processEvent* method. We make this class a subclass of the **MessageReceiver** server class. In the *processEvent* method we wrap the **EventDescription** in a **Message** object, and put it in the queue.



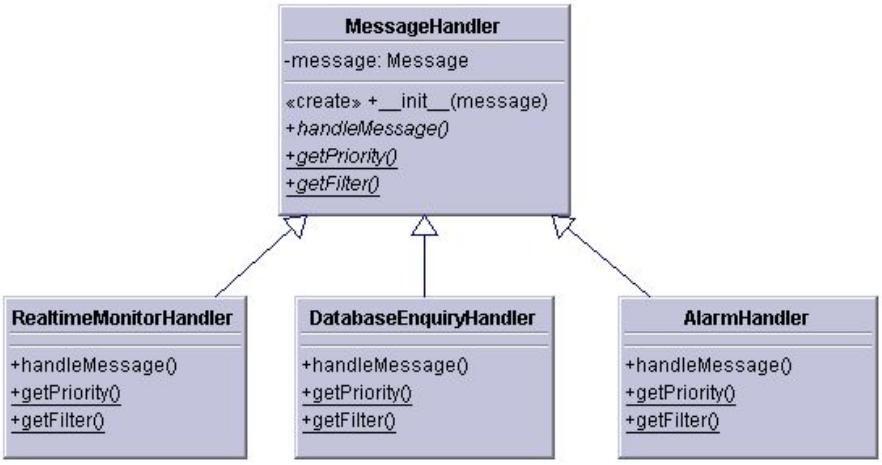
We subclass the **QueueManagerThread** class in order to provide a *preProcessEvent* method which can unwrap the corba representation of the event and provide a python version for use by handlers and processors.



We subclass **MessageProcessor** to provide the services which are applied to all events, the **DatabaseMessageProcessor**, as well as **MonaLisaEventProcessor** which will pass the message to other monitoring systems if necessary. We also provide a **ForwardingMessageProcessor** in order to forward events to upstream MIS servers so as to provide scalability should we want run the MIS across several machines, say using one for database logging, and another for real time monitoring.



Forwarding to real time monitoring will be handled by a messageHandler with a filter so that only relevant events are given to the handler. Other messageHandlers will handle requests from samTV for event histories or send out alarms. Alarms that send email should have rate controls and summarize repeated alarms in a single message.



Starting a MIS server.

The main method of the MIS server should perform the following actions.

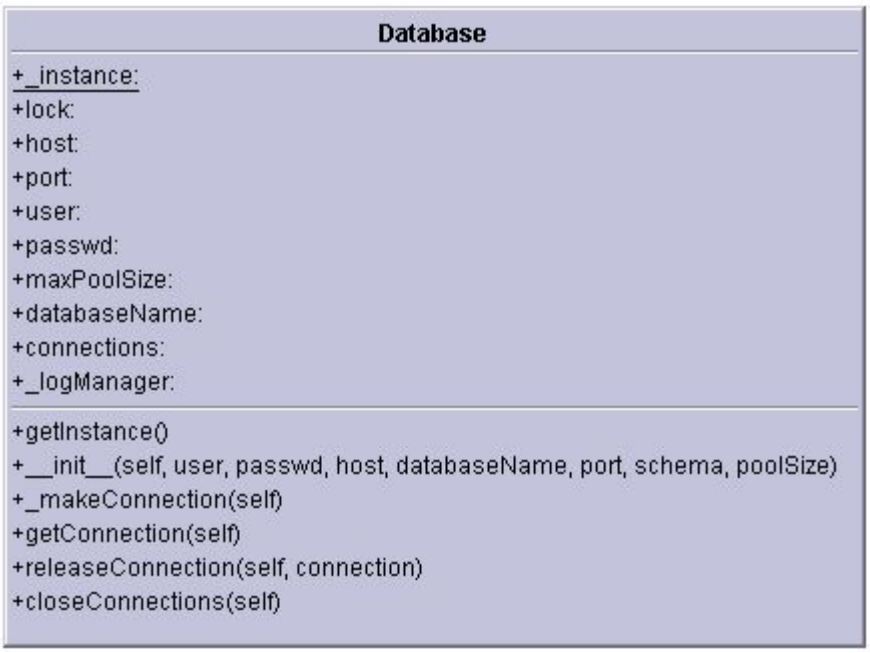
1) Create a Queue for the message queue.
2) Create an **EventMessageReceiver** receiver that puts messages into the Queue from step 1
3) Instatiate the singleton **HandlerManager** and **ProcessorManager** classes, providing a directory to be scanned for handlers to be registered.
4) Create the **messageQueueHandler** passing it a reference to the queue created in step 1
5) Start the **messageQueueHandler**
6) Register the **messageReceiver** created in step 2 with the corba naming service.
7) Wait for a signal to terminate the server, and shut it down cleanly.

Stopping

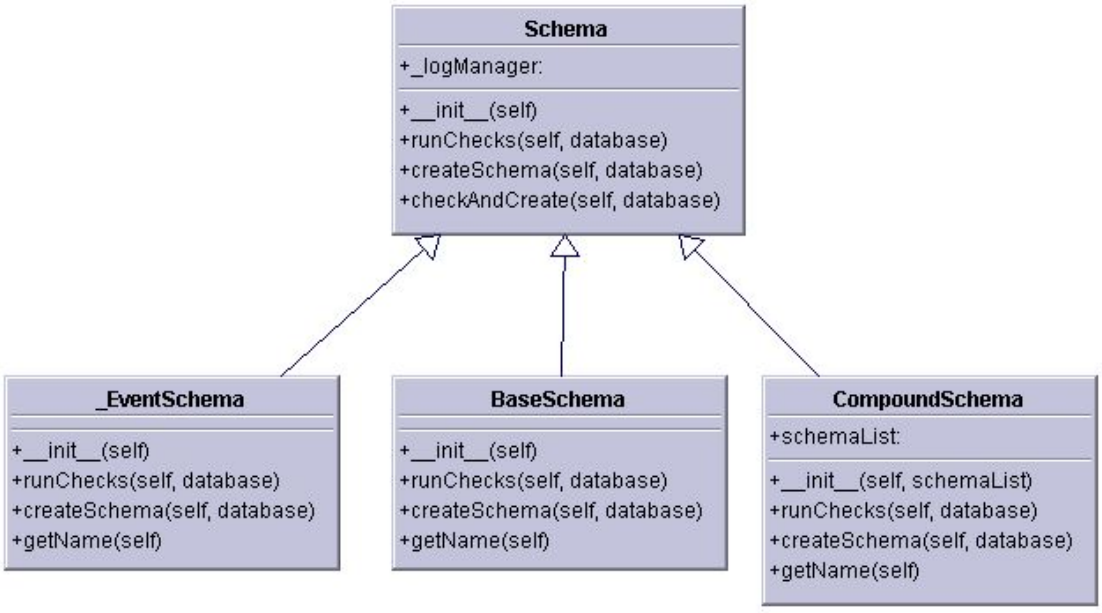
In order for the MIS to shutdown cleanly, the stop method of the messageQueueHandler should be called. This should call the stop methods of all processors, and all handlerRunners.

Database connections

Database connections are handled by a connection pooling database class, which threads may request connections from and to which connections are returned after use.



The **Database** class also allowes schema management, a **Schema** object has abstract methods to check whether a schema is present and create it if necessary. The compound schema checks for and creates any missing schemas in its schema list.



The MIS Schema is made from a compound of the base schema, which provides a version management table, and the Eventschema, which provides the necessary tables for storing events – below

EVENT table:

Field	Type	Null	Key	Default	Extra
ID	varchar(50)		PRI		
TYPE	varchar(50)				
TIME	datetime	YES		NULL	
PARENTID	varchar(50)	YES		NULL	
PRODUCERID	varchar(50)	YES		NULL	
DICTIONARYID	bigint(20)	YES	MUL	NULL	

EVENTDICTIONARY table:

Field	Type	Null	Key	Default	Extra
DICTIONARYID	bigint(20)			0	
KEYID	bigint(20)			0	
VALUE	text	YES		NULL	

DICTIONARYKEYS table:

Field	Type	Null	Key	Default	Extra
KEYID	bigint(20)		PRI	0	
DICTIONARYKEY	varchar(50)			NULL	

Although the framework is mainly database independent, the API drivers do sometimes differ subtly with the specification, and so there was some need to decide upon one database, at least initially. After some speed tests, we have decided on mysql as the optimal database so existing classes work with that. Small changes to connect strings and exception handling may have to be made to add compatiability with further databases. These should be subclasses of Database, such as PostGresDatabase or OracleDatabase.