MessageManager Application Development Framework

Purpose:

The purpose of this document is to describe the design and installation of the MessageManager and the sample application: Museum Environmental Control System.

Installing, Building, and Running the Systems

To unpackage and execute the MessageManager and Museum Environmental Control system on a single computer do the following:

1. Copy the A3 Source.zip archive to a directory on your local machine.

2. Unzip the A3 Source.zip archive.

3. Open a command line prompt and go to the directory where you unzipped the archive.

4. Compile all of the applications by typing: javac *.java

5. Type rmic MessageManager to build the MessageManager interface stubs.

6. Before you start the MessageManager or Museum Environmental Control System, make sure that port 1099 is available on your system.

7. Start the MessageManager by typing: EMStart (note you can also double click EMStart.bat to start the MessageManager).

8. If the MessageManager successfully started you will see a command window named "MessageManager" and it will display the host machine's IP address and display a message that says "MessageManager ready." At this point you are ready use the MessageManager sent Messages and receive Messages to participants registered on the MessageManager. Note that only ONE Message Manger can run on a machine. At this point, you can start the Museum Environmental Control System. If you want, you can reduce the MessageManager window to save screen real estate.

9. Start the Museum Environmental Control System on the local system by typing: ECStart

10. Once the Museum Environmental Control System starts, the system should be up and running, collecting, and displaying data.

Once the Museum Environmental Control System is running you will see the monitoring console in the upper left showing system status. The temperature and humidity sensors will be in the right top corner of the screen. The humidity and temperature controllers will be on the left of the screen. The user console is a command window with the title: "MUSEUM ENVIRONMENTAL CONTROL SYSTEM CONSOLE." You should set the temperature and humidity control ranges by using console options 1 and 2. To stop the entire system, use the X option in the console. You may start the Museum Environmental Control System using a MessageManager running on another computer. To start the Museum Environmental Control System using a MessageManager running on another computer type: ECStart <ip address of the system

running the MessageManager>. This will cause the Museum Environmental Control System to register with the remote machine's MessageManager (of course assuming there is one running there). The Museum Environmental Control System is comprised of a set of cooperating applications that include:

ECSConsole.class - this application is the console
HumiditySensor.class - this application is the humidity sensor
HumidityController.class - this application is the humidity controller and turns on/off the humidifier/dehumidifier.
TemperatureSensor.class - this application is the temperature sensor
TemperatureController.class - this application is the temperature controller and turns on/off the heater/chiller.

These applications do not have to run on the same physical machine – they can run on any machine just so long they can all get IP access to the same MessageManager. If you want to run the system in a distributed fashion, you will have to start each of the above system applications one at a time. Each of these applications can be started separately by typing the following at a console window:

START /MIN java TemperatureSensor <MessageManager ip or blank if MessageManager is local>

START /MIN java HumiditySensor <MessageManager ip or blank if MessageManager is local>

START /MIN java TemperatureController <MessageManager ip or blank if MessageManager is local>

START /MIN java HumidityController <MessageManager ip or blank if MessageManager is local>

START java ECSConsole <MessageManager ip or blank if MessageManager is local>

No particular start up order is required, however the ECS will not work properly until all five system applications are up and running. For the Museum ECS, only one copy of each of these applications can be running. Multiple copies will run, but the system will not operate correctly. The system is unable to autonomously determine if all the required applications are running and will not report any errors due to duplicate or missing applications.

There is one more application that may be useful. There is a utility that lets you collect Messages and post Messages through the MessageManager. This can be a handy tool for debugging systems and application. The application is MessageUtility.class. This can be started by typing: java MessageUtility at the command line. When this application starts it will ask you for the IP address of the MessageManager, or just hit enter if the MessageManager is running on the local processor.

When shutting down a MessageManager, exit the MessageManager command window and also exit the MessageManager registry command window. If you fail to remember to shut down the MessageManager registry, you will not be able to start the MessageManager again.

Using the MessageManager

The MessageManager is essentially a process that facilitates very simple and flexible communications between registered processes and threads. As part of this assignment you will not have access to the MessageManager source code – consider this a technical constraint. A brief explanation of its operation is described here:

A process that is registered with a Message Manger is called a *participant*. When a process registers with the MessageManager, a queue is established for the participant. Participants can send Messages to a Message Manger and can retrieve their queue from the MessageManager. Upon receipt of a Message, the MessageManager stores the Message in the queue of every participant (including the sender). When a participant gets their queue from the MessageManager, a copy of all their current Messages that have been posted is sent to the participant and the participant's queue is cleared of Messages at the MessageManager. Highly decoupled and distributed systems can be built using the Message Manger software.

To utilize a MessageManager running on some machine, a process/thread must register with that MessageManager. Registration is automatically done by instantiating an object of type MessageManagerInterface. The **MessageManagerInterface** class provides access to the following methods:

Constructor: MessageMangerInterface(void)
Purpose: This method registers participants with the MessageManager. This instantiation should be used when the MessageManager is on the local machine, using the default port (1099).
Arguments: None
Returns: MessageManagerInterface object.
Exceptions: LocatingMessageManagerException, RegistrationException, ParticipantAlreadyRegisteredException

Constructor: MessageMangerInterface(String MessageManagerIPAddress)
Purpose: This method registers participants with the MessageManager. This instantiation should be used when the MessageManager is not the local machine. The default port (1099) is used.
Arguments: String IP address of the machine where the MessageManager of interested is running.
Returns: MessageManagerInterface object.
Exceptions: LocatingMessageManagerException, RegistrationException, ParticipantAlreadyRegisteredException

Method: GetMyId(void)
Purpose: This method allows participants to get their participant registration Id. This is set automatically when the MessageManagerInterface is instantiated. The registration Id is used by the MessageManager to establish and manage a participant's Message queue.
Arguments: None
Returns: long integer that is the participants id
Exceptions: ParticipantNotRegisteredException

Method: GetRegistrationTime(void)
Purpose: This method allows participants to obtain the participants time of registration.
Arguments: None.

Returns: String time stamp in the format: yyyy MM dd::hh:mm:ss:SSS where:
yyyy = year
MM = month
dd = day
hh = hour
mm = minutes
ss = seconds
SSS = milliseconds
Exceptions: ParticipantNotRegisteredException

Method: SendMessage(Message e)
Purpose: This method sends an Message to the MessageManager for posting.
Arguments: Message object.
Returns: None.
Exceptions: ParticipantNotRegisteredException, SendMessageException

Method: GetMessageQueue(void)
Purpose: This method retrieves a participant's Message queue from the MessageManager.
Arguments: None.
Returns: MessageQueue object.
Exceptions: ParticipantNotRegisteredException, GetMessageException

Method: UnRegister(void)
Purpose: This method is called when the object is no longer used. Essentially this method unregisters participants from the MessageManager and releases their queue from the MessageManager. ***It is important that participants unregister with the MessageManager. Failure to do so will cause unconnected queues to fill up with messages over time. This will result in a memory leak and Messageual failure of the MessageManager.***

There are two other important objects: Message objects and MessageQueue objects. Message objects encapsulate the Messages sent between participants. Messages objects consist of an arbitrary integer message number and a string message, both set by the caller. Messages are sent to the Message Manger when a participant calls SendMessage(Message e). Messages are stored in MessageQueues on the MessageManager. Each participant has a queue associated with it on the MessageManager. When a Message object is sent to the MessageManager, the MessageManager stores the Message object in each of the queues on the MessageManager. Participants can get their Messages by calling GetMessageQueue. When a participant calls the GetMessageQueue() method, a copy of the participant's Message queue is sent to the participant. The participant's Message queue is cleared of all Message objects on the MessageManager. The **Message** class provides access to the following methods:

Constructor Method: Message(int MessageId, String Text)
Purpose: To create an Message object.
Arguments: int Message ID. An integer that is determined by the participant. Message IDs have no semantic significance to the MessageManager. Participants must assign any meaning to Message IDs.
String Text. A text string that is determined by the participant. There is no semantic significance in this string to the MessageManager.
Returns: Message object

Exceptions: None

Constructor Method: Message(int MessageId)
Purpose: To create an Message object.
Arguments: int Message ID. An integer that is determined by the participant. There is no semantic significance in the Message ID to the MessageManager. Participants must assign any meaning to Message IDs.
Returns: Message object
Exceptions: None

Method: GetSenderID(void)
Purpose: This method returns the ID of the participant that posted this Message.
Arguments: None
Returns: long integer that is the participant's id this should not be confused with the Message Id. The participant id is used by the MessageManager to identify and manage queues.
Exceptions: None

Method: GetMessage(void)
Purpose: This method gets the Message embedded in this Message. If there is no message, then null is returned.
Arguments: None
Returns: String text
Exceptions: None

The MessageQueue class provides methods that allow participants to access Messages that have been stored for the participant by the MessageManager. The **MessageQueue** class provides access to the following methods:

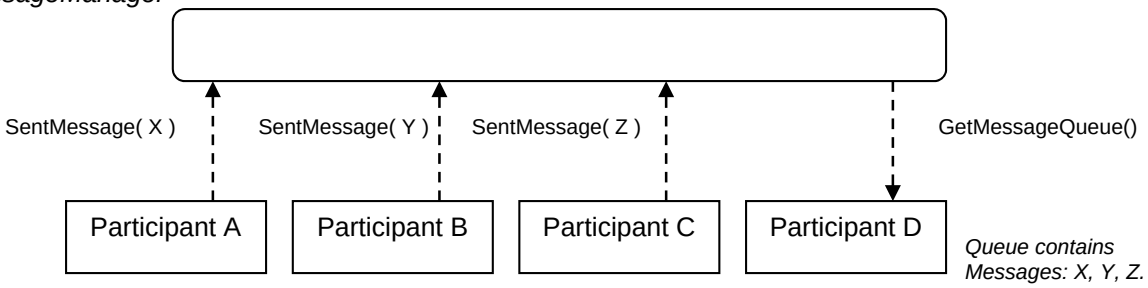
Method: GetSize(void)
Purpose: This method returns the size of the queue – this is the exact number of Messages stored in the queue.
Arguments: None.
Returns: int value indicated the size or number of elements in the queue (they are the same).
Exceptions: None

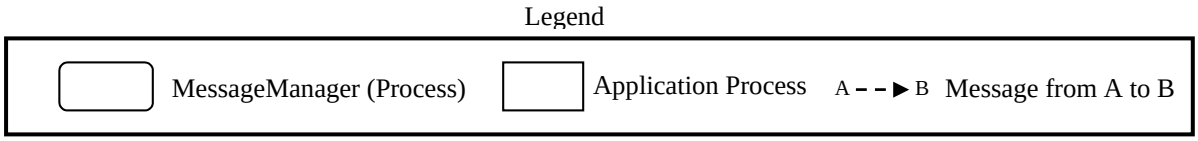
Method: GetMessage(void)
Purpose: This method gets the oldest Message off of the queue. Once you get a message from the Message queue, it is permanently removed. A call to GetMessage() reduces the size of the Message queue by 1, so be careful if you use GetSize() to control loops. The queue is a FIFO queue – that is the Message at the front of the queue is always the oldest Message to be posted in queue and is the first to be removed by GetMessage(). If the list is empty, null is returned. One way to control loops is the call GetMessage() until it returns a null.
Arguments: None.
Returns: Message object
Exceptions: None

A Dynamic Perspective of Systems Utilizing the MessageManager Software:

Highly decoupled and distributed systems can be built using the Message Manger software to facilitate very simple communications between applications. A simple interaction between two participants is illustrated below:

Example illustrating the sending of Messages and getting Message queues to and from an MessageManager





In this example, we assume that application (participants) A, B, C, and D are registered. Assume that participants A, B, and C send Messages X, Y, and Z respectively. At some point after Messages X, Y, and Z are sent, participant D gets its Message queue. This queue will contain Messages X, Y, and Z. If another get Message queue was issued by participant D, the queue would be empty. Further if participants A, B, and C were to issue get queues, they would also receive their respective Message queues that would also contain Messages X, Y, Z.

Each application process may have 2 or more threads that have connections to 1 or various MessageManagers on various machines. As a reminder, a registered participant is a MessageMangerInterface object residing in a thread/process or within another object. MessageMangerInterface objects or participants are automatically registered when they are instantiated. Again, a single process may have 1 or many MessageManagerInterface objects (participants) registered with the same or different MessageManagers. However each MessageMangerInterface object can only register with one MessageManager at a time. A simple component and connector viewtype illustrates the relationship between participants, processes/threads and objects:

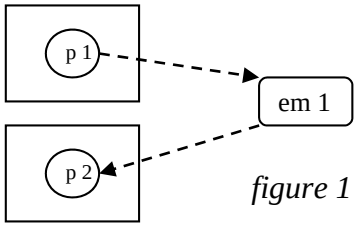


figure 1

Figure 1 shows two processes or threads both have participants registered with MessageManager em 1. In this case each process or thread has one participant in each communicating to one another via MessageManager em1.

Figure 2 shows two participants again registered with MessageManager em 1. Here both participants exist within a single process or thread, each communicating to one another via MessageManager em1.

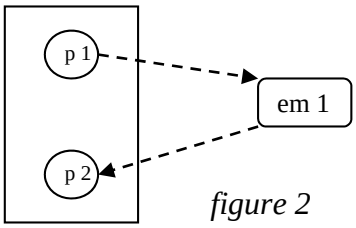


figure 2

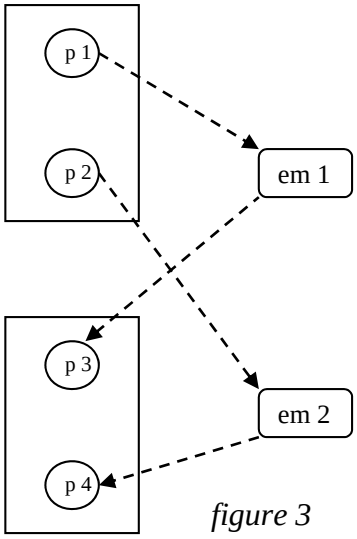
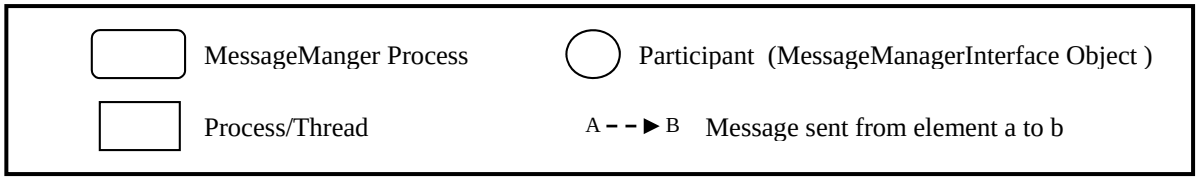


figure 3

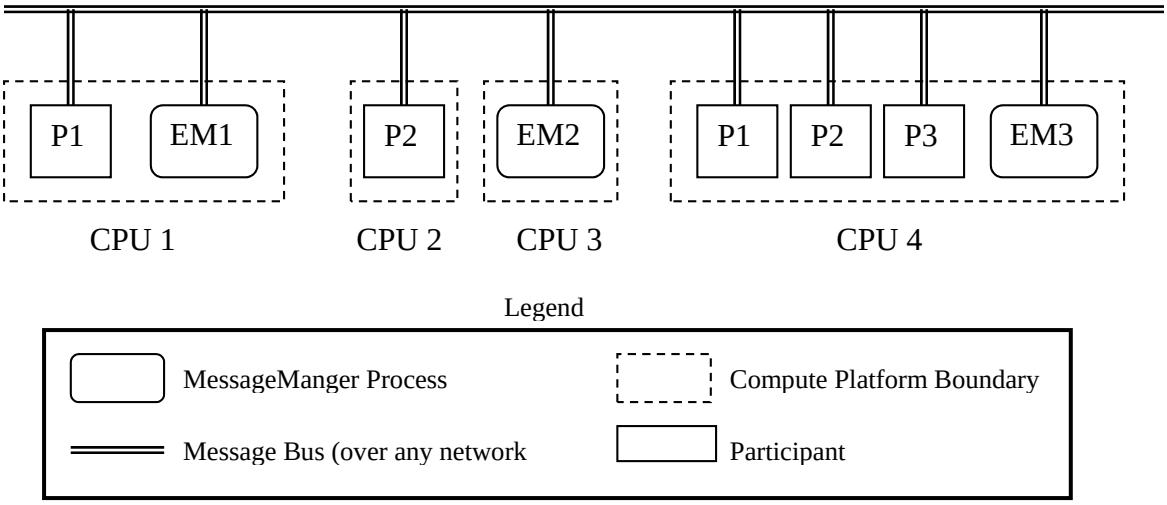
Figure three shows two processes or threads. Each process or thread has two participants. In the top process/thread there are two participants, p1 and p2. In the other participants p3 and p4. Note p1 and p3 have registered with MessageManager em 1; p2 and p4 have registered with em 2. In this case p1 and p3 can share Messages; p2 and p4 can share Messages. In all of these cases, participants can sent and receive Messages, although only one way Message flow is show in figures 1–3. Also note that figures 1-3 are not meant to indicate computer hardware boundaries. This relationship between participants, MessageManagers, and the hardware they execute on is explained more detail in the next section.

Legend



A Physical Perspective of Systems Utilizing the MessageManager Software:

Typical deployment scenarios are show below. Note that this figure shows runtime software elements and their mapping to hardware platform boundaries. The intent is to show typical, legal deployments of the system:

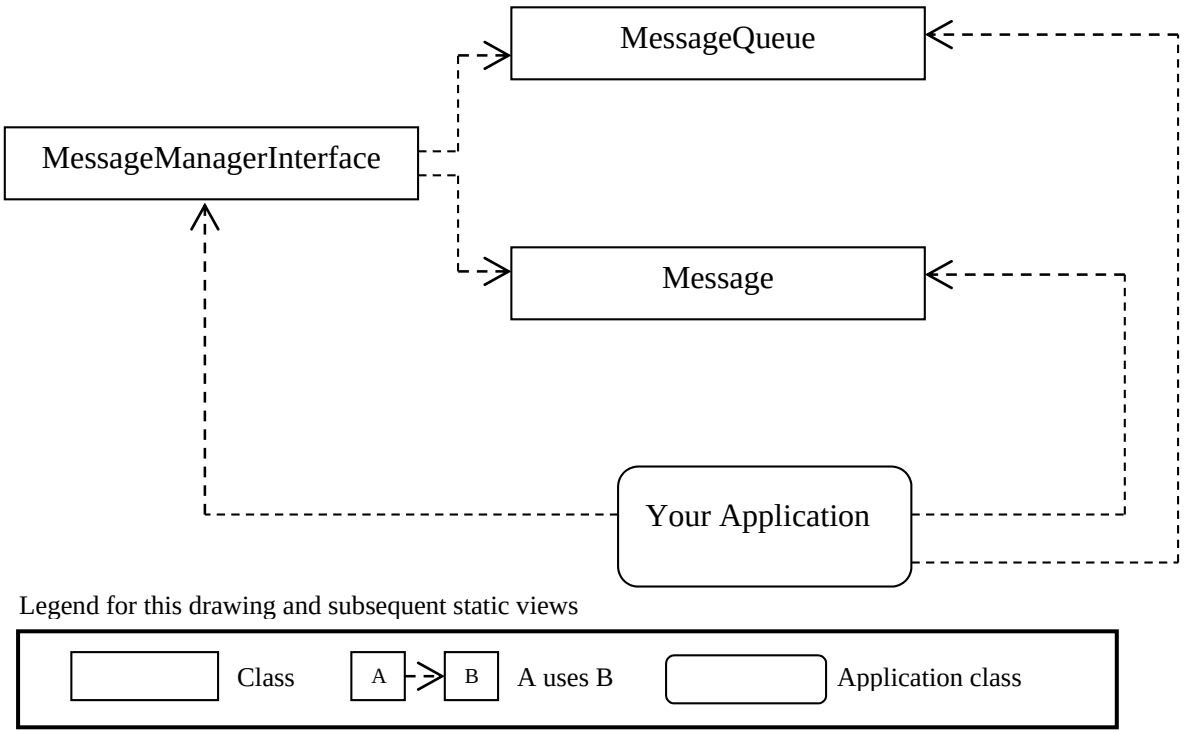


Note that the MessageManager does not depend on a particular network topology and ad hoc relationships can be established at runtime. All network mediums at this point are supported as well. Recall that a computer platform can have only one MessageManager process running at a time. Port 1099 is required by the MessageManager. A machine with a participant does not have to have a local copy of the MessageManager process running. However, there must be at least one MessageManger process running somewhere, on some machine that is reachable by participants. The figure above illustrates the key deployment scenarios supported by the MessageManager software. Note that there are 4 different compute platforms shown here - participant P1 can communicate with the local MessageManager EM1 where both are collocated on CPU1. However P1 could be registered with EM2 or EM3 as well. Note that any participant can only be bound to one MessageManager at a time. Note that CPU2 has a participant P2, but no local MessageManager. In this case participant P2 would have to use a non-local MessageManager. In the case of CPU3, there is only an MessageManager running (EM2) and no local participants. Finally on CPU4, there are multiple participants and one MessageManager (EM3) running. Note that P1-P3 on CPU4 may register with any of the MessageManagers EM1, EM2, or EM3 provided they are running on reachable hosts.

A Static Perspectives of Systems Utilizing the MessageManager Software:

Applications can be created to access the MessageManager by instantiating MessageManagerInterface objects. An MessageManagerInterface object provides access to various services required by applications and systems using the MessageManager to communicate Messages to one another. The figure below shows how an application is structured to use the MessageManager:

The general static structure of processes using the Message Manage.



Catalogue:

- **Message** – Stores the Message id and text string message. Provides methods (described above) to access the Message id and string message. This class is in the MessagePackage package.

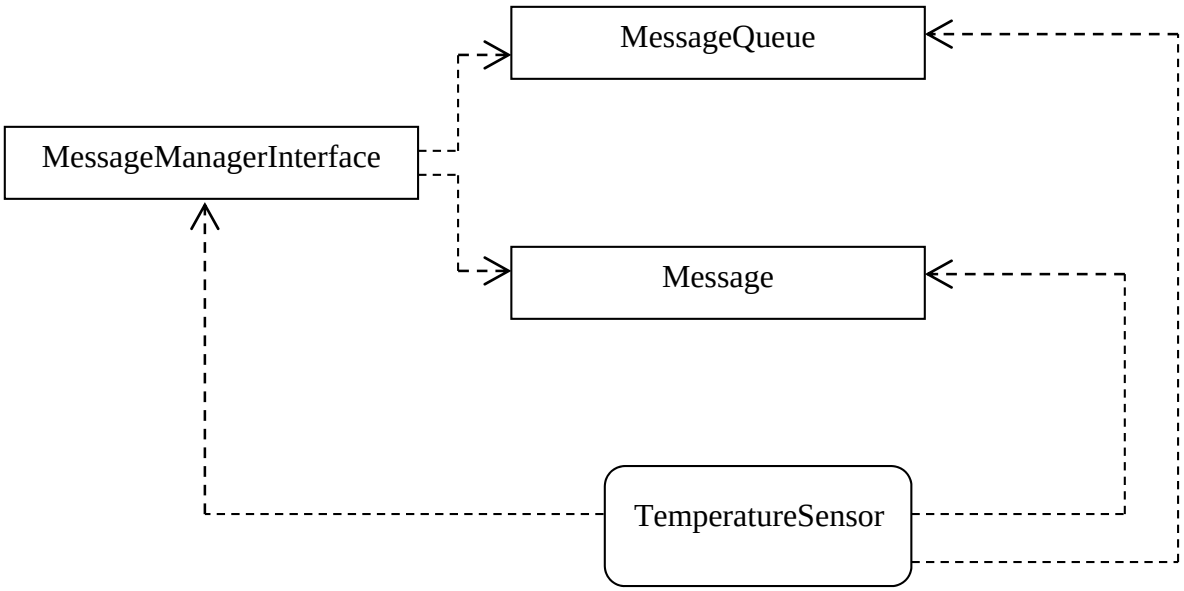
- **MessageQueue** – Gives participants access to Messages posted by other participants vis-à-vis the MessageManager. This class is in the MessagePackage package.

- **MessageManagerInterface** – This class provides applications with access to the various services of the MessageManager including the ability send to and receive Messages from other participants. This class is in the MessagePackage package.

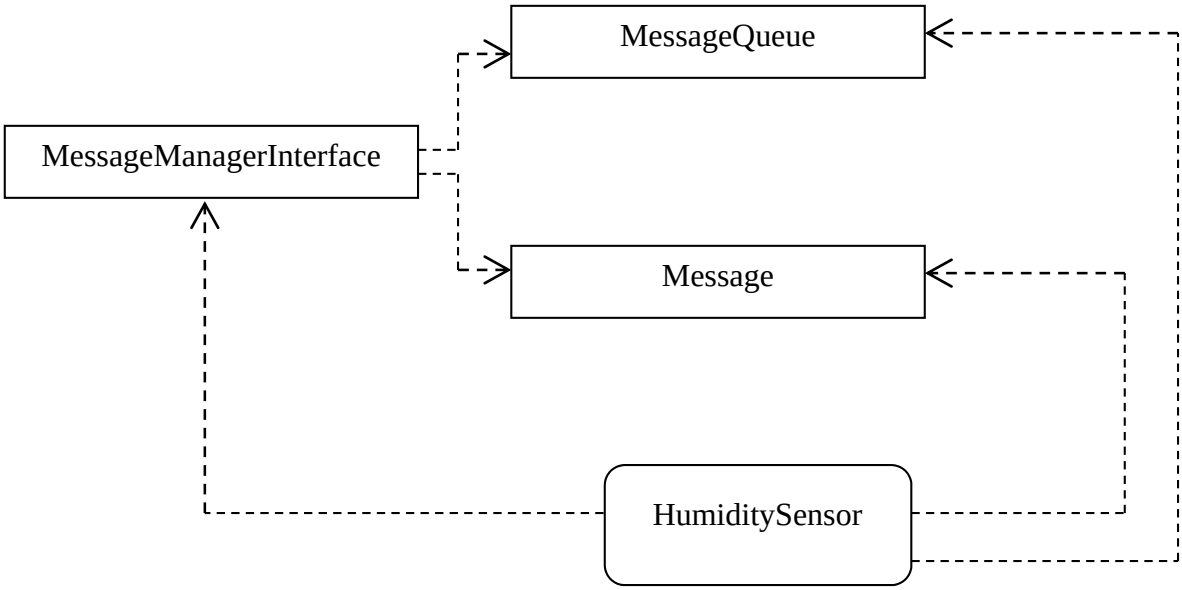
- **Application** – This class is differentiated because it contains the main thread. A process, thread, or object using the MessageManager.

Note that classes denoted by the square box are class whose source code you do not have access to. “Your application” in this case is a class you create that can make use of the MessageManager to communicate with other participants at runtime. All relationships here are uses. The general pattern above is used in all of the applications that comprise the Museum Environmental Control system are shown below. Note that the legend above also applies to the views that follow:

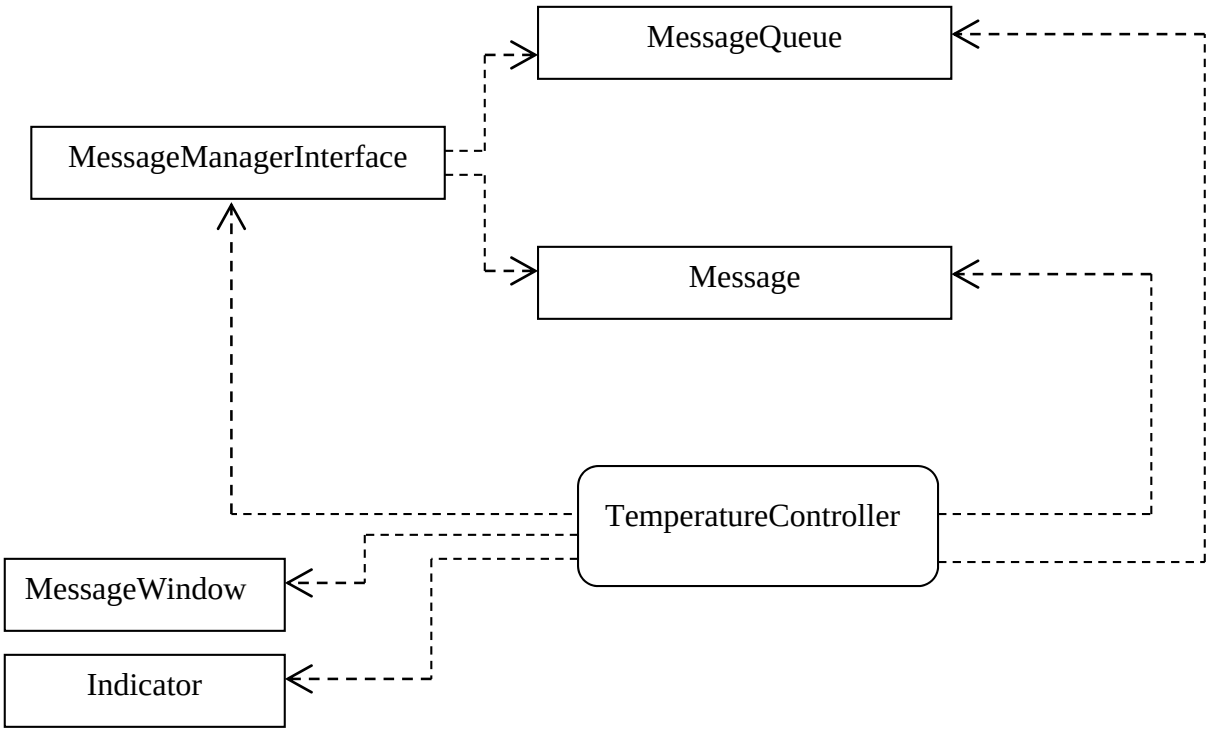
The static structure of the **TemperatureSensor** process.



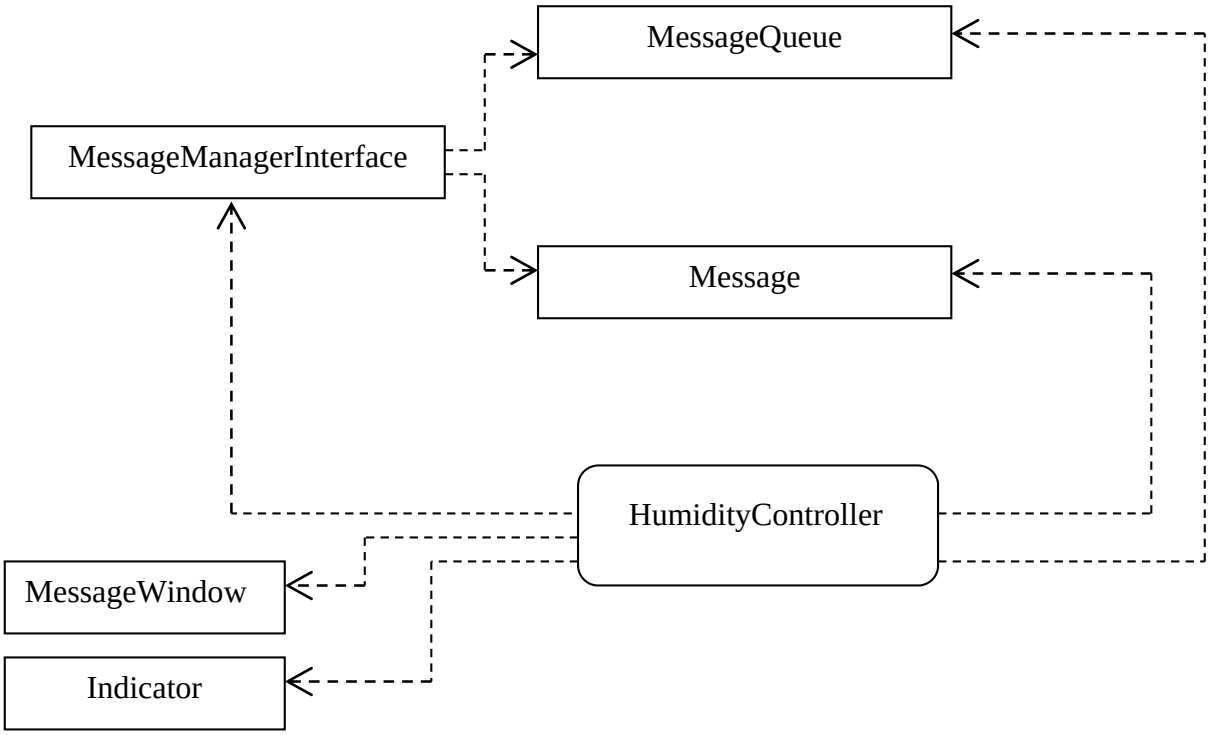
The static structure of the **HumiditySensor** process.



The static structure of the **TemperatureController** process.



The static structure of the **HumidityController** process.



Note that there are two additional classes shown here:

Catalogue:

- **MessageWindow** – This class provides a message window and allows message to be written to the window. The window includes a scroll bar, title, and each message is displayed with a time stamp. For simplicity the window size is fixed, however message windows can be placed anywhere on the terminal and can be moved at run time. This class is in the InstrumentationPackage package. You are free to use these class, but you do not have access to the source code. This class provides access to the following methods:

  Constructor Method: MessageWindow(String Title, int Xpos, int Ypos)
  Purpose: To create an MessageWindow object.
  Arguments: String Title – Message window title
  int Xpos – Message windows horizontal position on the screen
  int Ypos – Message windows vertical position on the screen
  Returns: MessageWindow object
  Exceptions: None

  Method: Height()
  Purpose: This method returns the height of the message widow
  Arguments: None
  Returns: int width in pixels.
  Exceptions: None

  Method: Width()
  Purpose: This method returns the width of the message widow
  Arguments: None
  Returns: int width in pixels.
  Exceptions: None

  Method: WriteMessage(String message)
  Purpose: This method writes a message with a timestamp to the window
  Arguments: String message.
  Returns: None
  Exceptions: None

- **Indicator** – This class simulates an indicator lamp on the screen. This can be used to indicate warning, error, status, and so forth. For simplicity the indicator size is fixed, however indicators can be placed anywhere on the terminal and can be moved at run time. This class is in the InstrumentationPackage package. You are free to use these class, but you do not have access to the source code. This class provides access to the following methods:

Constructor Method: Indicator(String Label, int Xpos, int Ypos, int InitialColor)
Purpose: To create an Indicator object.
Arguments: String Label – The indicator label
int Xpos – Horizontal position on the screen
int Ypos – Vertical position on the screen
int InitialColor – This is the lamp color as follows: 0=black, 1=green, 2=yellow, and 3=red

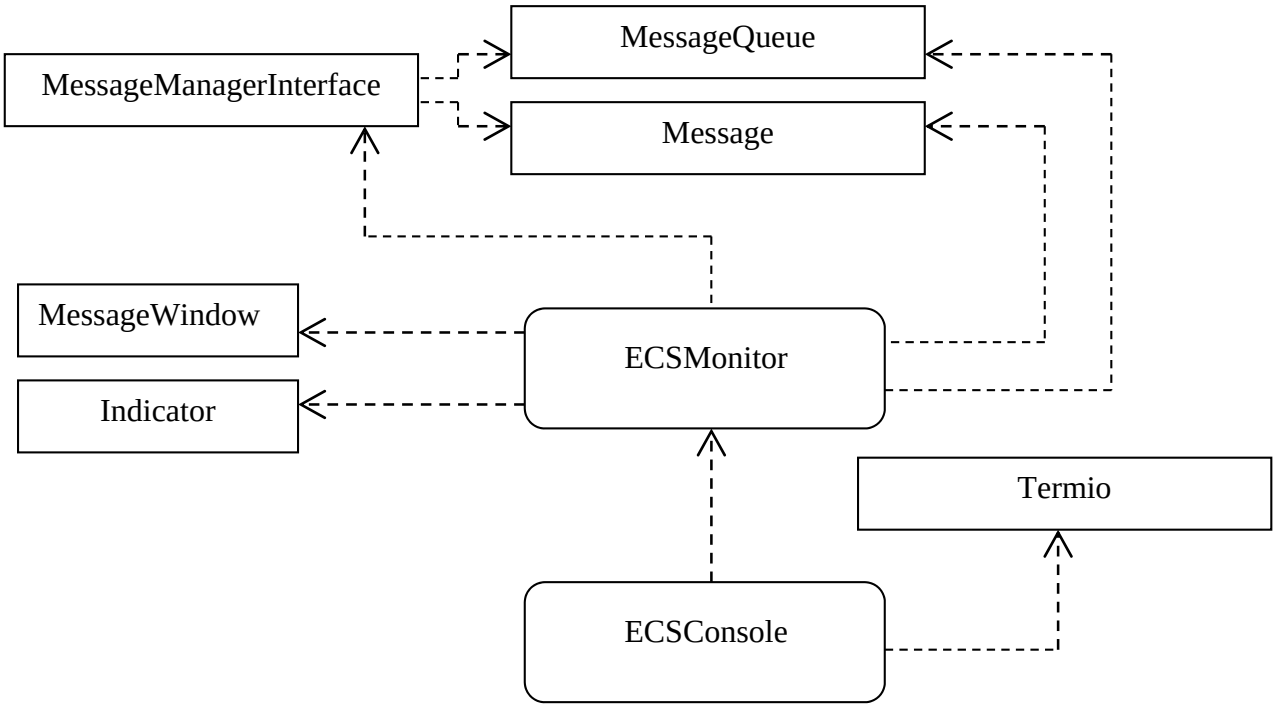
Returns: Indicator object
Exceptions: None

Method: void SetLampColorAndMessage(String s, int c)
Purpose: This method sets the lamp color and label
Arguments: String Label – The indicator label
int c – This is the lamp color as follows: 0=black, 1=green, 2=yellow, and 3=red
Returns: None
Exceptions: None

Method: void SetLampColor(int c)
Purpose: This method sets lamp color
Arguments: int c – This is the lamp color as follows: 0=black, 1=green, 2=yellow, and 3=red
Returns: None
Exceptions: None

Method: void SetMessage(String s)
Purpose: This method sets the lamp label
Arguments: String Label – The indicator label
Returns: None
Exceptions: None

The static structure of the **ECSConsole** process.



Note that the ECSConsole class is a bit different from the previous applications. This processes consists of two threads. The ECSMonitor interacts with the MessageManager by sending and receiving Messages and maintain the set temperature and humidity. The ECS console allows a user to enter set temperature and humidity values as well as shut down the system. You have access to these classes and it would be wise to use them as a model for applications that you have to create as part of assignment 3.