Multimedia Data Exchange Agent

an Object Behavioral Pattern for Multimedia Programming

Chrystalla C. Alexandrou and George A. Papadopoulos
Multimedia Research and Development Lab (MRDL)
Department of Computer Science
University of Cyprus
75 Kallipoleos Str., P.O. Box 537
CY-1678 Nicosia, Cyprus
E-mail: {cschryst, george}@turing.cs.ucy.ac.cy

Abstract

This paper describes the Multimedia Data Exchange (MDE) Agent Pattern, which itself describes the decoupling of the communication mechanisms from the actual implementation of two multimedia components. The general model of Producer (Source), Filter, Consumer (Sink) that is used for the modeling of temporal media types as components is enhanced by this decoupling. The components must have the ability to connect to each other. This connection could be modeled as a simple relationship between the two components but this solution would require that the implementation of this relationship is the responsibility of either or both of the components and would also be visible by the application. This pattern proposes the separation and isolation of the implementation of this relationship into a different class and also proposes customization via a hierarchy that will produce different in usage and characteristics MDE Agents.

Name :

Multimedia Data Exchange Agent.

Indent:

Represents a directed channel connecting two directed ports of (multi)media components that are decoupled.

Motivation:

The temporal media types like digital video, music and animation are examples of time media sequences that have time characteristics. They have a starting time and duration. Usually Multimedia applications must have the ability to deal with such data. A multimedia system must have the mechanisms to produce, transform and consume temporal data based on the time constraints that the nature of this kind of data requires.

In modeling, the temporal media types can be viewed from two different perspectives: the passive and the active. The distinction between those two perspectives is the handling of time as a constraint. With temporal data we can apply non time critical operations like copying an audio file from one location to another or changing the luminance of a still image (passive). On the other hand audio playing back is a time critical operation (active). Since the time constraint adds complexity by putting additional requirements like synchronization and multithreading, the two different views of the temporal media are modeled with two different categories of classes: the *media* classes that deal with the passive characteristics of the temporal media and the *component* classes that deal with the time critical characteristics.

In a multimedia design, component classes encapsulate hardware devices and software services and represent resources that perform time-critical operations. Components are examples of active objects, objects that have state and behavior like ordinary objects but they can also have their own thread of control and they can perform actions without being requested to do so by another object.

In a multimedia system we can identify three different categories of components: the sources (producers), the filters and sinks (consumers). Sources are components that produce streams (time data sequences), sinks consume streams and filters transform streams. All the different components specify interfaces for importing and/or exporting of streams. Different protocols, stream types and direction can be considered as the main characteristics of those interfaces that in many cases where they simulate the hardware solutions are referred to as ports. Furthermore for simplicity we pay emphasis more to Sources and Sinks and we consider the Filters as Sources or Sinks since for the aspect of data exchange they can be viewed as being so.

A component must have the ability to connect with another component. Those connections in an object oriented model can be viewed as complex relationships that need to be modeled as separate classes. The Multimedia Data Exchange (MDE) Agent is a pattern for designing complex relationships that represent different communication mechanisms. Two components can communicate via a shared memory or via a network or via a buffer, etc.

If the MDE Agent was modeled as a simple relationship then the implementation details of the communication should have to be embedded in one of the communicating components and that would lead to the tight coupling of the communication channel with the component's implementation. The decision of what kind of connection the two components are going to have is taken at compile time although in reality it could be more effective for this decision to be taken at run-time during the instant availability of resources (virtual relationship).

In Multimedia systems components are involved with a great overload of producing/receiving streams among their other critical operations (e.g. play). The creation of MDE Agents as separate classes helps the components to be unaffected from the presence or absence of a connection since they can use different threads of control than those of the components. By definition the MDE Agents have state (connected, not connected) and behavior (Put, Get) and they should be modeled as active objects since they require their own thread of control. The MDE Agent is responsible for getting the streams the correct time from the Source, and for feeding the Source with the necessary input. This transaction justifies the use of "Agent" in the name of the pattern. When the Source is ready to produce time data it "asks" the application to "hire" an agent for this connection.

If we examine the system from the architectural perspective we observe that communication channels can be shared between two totally different components. Also, the system evolution forces the creation of the connection as a separate class. The type of connection can change due to certain requirements although the components remain the same or the opposite. The separation of the components from the MDE Agents also helps resource management, since the MDE Agent can exist only at run-time and when the communication channel is needed.

This pattern is specialized to Multimedia. As far as we are concerned there are two cases where a design pattern is specialized for Multimedia. In the first case, the design pattern addresses the idioms and characteristics of Multimedia; e.g. a design pattern for video data compression. In the second case, the design pattern addresses a non specific (although generic) Multimedia problem which, however, is solved from the point of view of Multimedia taking into account any particular characteristics. This latter approach is used by the proposed Pattern.

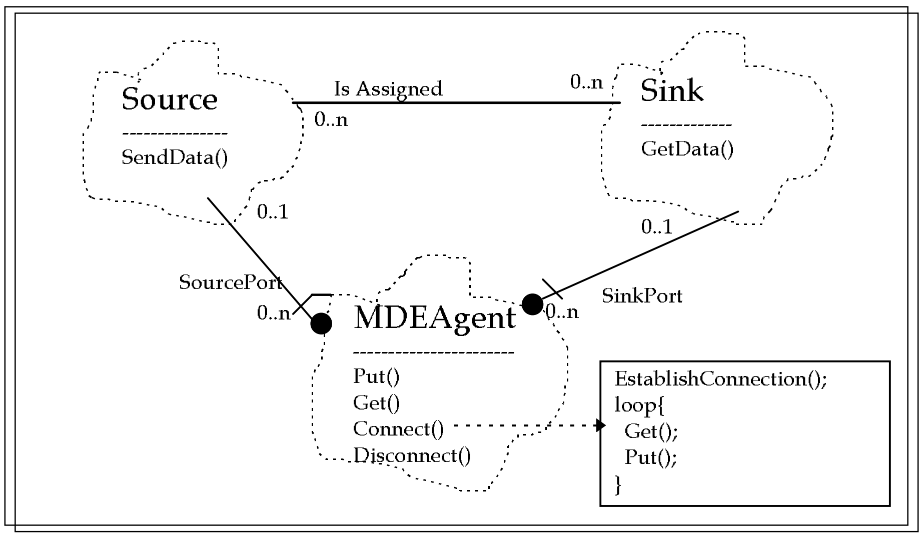
The issue of decoupling the input/output data thread from the main thread of control of an active component is generic and it can also be the case for non Multimedia systems, but since in Multimedia systems the importing and exporting of time based media and the synchronization of the Source / Sink pipeline is very much dependent on this data transferring, we suggest the creation of active objects that are responsible only for the data exchange and we try to keep the components unaffected by the presence or absence of connections. Another goal is to create modular and cohesive Source/ MDE Agent / Sink that can deal effectively with the evolution that is continuously taking place in Multimedia technology.

Applicability:

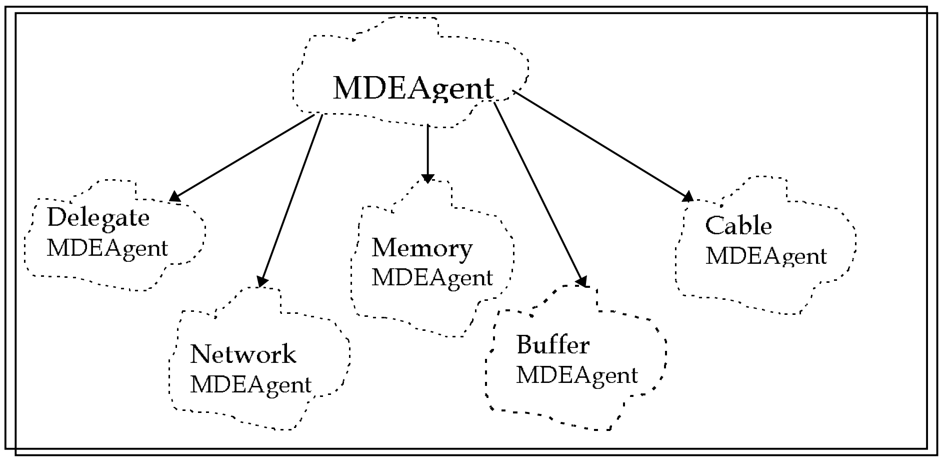
Use the MDE Agent Pattern:

- for modeling components that are involved with a great overload of producing/receiving streams,
- for modeling a directed channel of communication between two components that can speak the same "language",
- when the designer needs to hide the implementation details of the communication channel from the component,
- for the creation of reusable code for the communication channels between two active objects,
- when the designer does not want to commit to a particular channel of communication before run-time (virtual communication channels).

Structure:



Hierarchy Diagram:



Multimedia Data Exchange Agents:

- **MemoryMDEAgent** : A shared memory is used as a buffer and by tightly coupled components that run on the same machine. This MDE Agent owns and supervises a shared memory location where the SendData of the Source and Getdata of the Sink act.
- **BufferMDEAgent** : The Put and Get methods of the BufferMDEAgent add and remove respectively from a shared buffer that belongs to the Agent. They are used by loosely coupled components that run on the same machine.
- **NetworkMDEAgent :** The Put and the Get operate on a digital network by performing reads and writes. They are used as communication channels of two components running on different machines.
- **DelegateMDEAgent** : The Get method calls the SendData of the Source and the Put method calls the GetData of the Sink. The DelegateMDEAgent does not provide any storage capabilities for the data exchange.
- **CableMDEAgent :** It represents a physical cable connection. The necessary checks have to be implemented in the connect() method and the Get and Put methods are null methods.

Participants:

- **Source :** A component that has at least one Output Port. Output Port is the interface used by the component to export multimedia streams.
- **Sink:** A component that has at least one Input Port. Input Port is the interface used by the Sink in order to import data.
- **Multimedia Data Exchange Agent:** Connects an Output Port of a Source with an Input Port of a Sink and implements the required connection in a way that it frees the connected components from the implementation details and the handling of this communication channel.

Collaborations:

1. MDE Agent construction - In this phase the client application calls the MDE Agent's constructor and passes as parameters references to the two components that need to be connected. The constructor of the MDE Agent establishes the reference of the Source to the associated Sink and also the reference of the Sink to the associated Source. The main task of the constructor is to create the necessary conditions for this connection.
2. Establishment of the connection - In this phase the client application establishes the connection. It is necessary that beforehand the client application is ensured that the Source is ready to export data and the Sink is ready to import data. After that, it calls the MDE Agents method Connect that will establish the connection which returns an ErrorCode that identifies the state of the connection. In C++ the ErrorCode can be replaced by the necessary exception handling.
3. Disconnection - In this phase the connection is not necessary and the client application calls the Disconnect method of the MDE Agent that is responsible to nil the references from Source to Sink and from Sink to Source.

Consequences

The MDE Agent pattern has the following benefits and drawbacks:

1. Connects an Output port of a Source or a Filter to an Input port of a Sink or a Filter by making the necessary checks for the correctness of this connection and the availability of the Input port.
2. It offers the necessary sources for this connection.
3. It removes all the responsibility of this connection from the components. The connection runs using a different thread of control that deals with the complexity demanded like synchronization, mutual exclusion, etc. without overloading the components with this additional requirements. The fact that the MDE Agent localizes a behavior that otherwise it would be distributed among the other participants promotes the reusability of the participant components. This is possible because a change in the connection behavior and structure (e.g. from buffer connection to shared memory connection) is not going to affect the participant components but only the MDE Agent.
4. MDE Agents connect two components that speak the same "language" (communication protocol). In the case that there is incompatibility, the designer has to use an adapter pattern **[1]** to create compatible interface.
5. The creation of the communication channel as a separate part at runtime generates the risk of inability to create a communication channel due to the lack of resources since creating separate entities is more resource demanding. On the other hand, the static creation of communication channels that are not going to be utilized and they exist just in case they need to be used is possibly expensive.
6. The MDE Agents are unidirectional channels of communication. The reason for this is that by nature the time media have a specific direction. The same implementation is used for hardware solutions with input and output ports.
7. The details of how the components communicate is hidden from the application.

Implementation

The purpose of this pattern is not to show how the implementation of the different MDE Agents should be done but to show the separation of a relationship from the implementation. The implementation of the different descendants that deals, for example, with the network or the buffer behavior can follow existing patterns that are specialized for this behavior of the components. The MDE Agent Pattern defines the necessary interface for the different MDE Agents which is sufficient for replacing a complicate relationship with a class relationship.

1. From the structure diagram every MDE Agent has a "has" relationship to the Source via an attribute named SourcePort and a "has" relationship to Sink named SinkPort. Both of them have protected export control which means that they are available only to the descendant of the class MDE Agent.
2. Every Source must know every Sink it is connected to and vice versa if such a connection exists. This setting is done by the MDE Agent in the method Connect() which sets the necessary interface attributes of the Source and the Sink.
3. From the Source and the Sink there are no relationships to MDE Agent. The creation of the MDE Agent and the establishments of the SourcePort and SinkPort are done by the application.
4. Mutual Exclusion. In the case of BufferMDEAgent and MemoryMDEAgent the methods Get and Put must implement mutual exclusion mechanisms.
5. In the case of BufferMDEAgent a FIFO mutual exclusive buffer must be used. The allocation of the buffer is better to be done in the constructor of the BufferMDE Agent.
6. The MDE Agents are active objects. For the creation of the base mechanism of the MDE Agent we suggest the use of the Active Object **[2]**, an Object Behavioral Pattern for Concurrent Programming created by R. Greg Lavender and Douglas C. Schmidt.

Sample code

The following C++ code shows the definition of the MDE Agent.

```
Class Connector{
public
        // Public attributes
        …

        // constructor
        Connector (Components*   ProducerComp, Components*   ConsumerComp);

        // connection methods
        virtual ErrorCode   Connect();
        virtual ErrorCode   Disconnect();


        // destructor method
        virtual  ~Connector ();

        // other public methods
        ....

protected
        // state attributes
        Component *   SourcePort;
        Component *   SinkPort;


        // communication methods
        virtual ErrorCode    Get (void*  data, int nbytes);
        virtual ErrorCode    Put  (void*  data, int nbytes);

        // other protected methods
        ....
}
```

Known Uses

The whole development of the MDE Agent pattern was inspired from the Gibbs and Tsichritzis model **[3]** where the MDE Agents are defined as Connectors.

Related Patterns

Mediator is a similar pattern. The main difference is that Mediator deals with passive objects. The MDE Agent is an active object with the Connect method to create a separate thread of control. Another difference is that the Colleague classes knows about its Mediator and that can promote coupling between Colleague classes and Mediator. The Mediator supports also a multidirectional protocol in contrast to the MDE Agent that supports only unidirectional protocol.

Siemens Filters Pipes have a lot in common with our pattern. Pipes are similar to the MDE Agent. They both connect two active objects and they are both used for data exchange. On the other hand, there are some major differences:

1. It is clear that the control for the data exchange is in the Filter and not the Pipe. Pipes are passive and they act as an intermediate storage for the data exchange of two filters. In our pattern, the MDE Agent has its own thread of control; once the connection between a Source and a Sink is established by the application, the Connector is responsible for getting the data that the Source produces and passes it on to the Sink without the Source or Sink having to get involved in calling the EndData or GetData. Those two methods are invoked only by the Connector.

2. The only type of Connector covered by the Pipes and Filters is a FIFO buffer. In Multimedia systems this is not always the case. There are cases where a network Connector is needed, there are cases where the shared memory approach is very important, and there are even cases where there is only a hardware cable connection.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns- Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[2] Lavender, Schmidt. Active Object an Object Behavioral Pattern for Concurrent Programming. In proceedings of PLOP 1995.

[3] Simon J Gibbs, Dionyssios C. Tsichritzis. *Multimedia Programming- Objects, Environments and Frameworks*. Addison-Wesley, 1994.