Messaging Design Pattern and Pattern Implementation

Al Galvis.
Freedom Software
Email: egalvis2000@yahoo.com

Abstract

Interchange of information (i.e. messaging) is an inherent part of nature and man-made processes. Messaging is a ubiquitous part of the world around us. Conventional software methodologies and component technologies overlook messaging and therefore provide an incomplete model. On the other hand, a messaging paradigm and the associated messaging design pattern (MDP) address this gap and provide a more complete and accurate model of the real world. As a consequence, software engineering processes and techniques are improved significantly. While designing and manufacturing software, we need to think not only in terms of software components, but also in terms of the messaging being exchanged between these entities. Encapsulation, decoupling and reusability are improved while reducing complexity. This paper also discusses how the messaging design pattern is utilized to implement or help implement other well-known design patterns like Gang of Four design patterns (GoF), Data Access Objects (DAOs), and J2EE design patterns. Keep in mind that most of the design patterns are, at some level, responsible for interchanging information between participants. The overall design and UML diagrams are simplified and streamlined making them easier to understand and implement. The resulting software design and implementation are also more robust and straightforward. Design patterns implemented using MDP, can be reused to provide transparent and secure access to remote components/services as the basis for a complete distributed component model.

Messaging Design Pattern (MDP)

Intent: The messaging design pattern allows the interchange of information (i.e. messages) between components and applications. It improves decoupling, encapsulation and reusability by separating component communication from component functionality.

Messaging is ubiquitous. On the other hand, conventional software methodologies and models tend to overlook it. MDP addresses this problem by proposing and relying on a messaging model. While designing and manufacturing software, we need to think not only in terms of software components but also in terms of the messaging being exchanged between these entities. Components are one essential part of the object oriented methodologies. The interchange of information (i.e. messaging) between components is also an important part of a complete model. This helps justify the need for MDP and a model based on a messaging approach. It also illustrates the gap between conventional models and the reality that they seek to represent. MDP addresses this gap.

Applicability: This design pattern can be applied to solve a great variety of problems in many diverse scenarios. A messaging paradigm is widely used in the real world. Messages are interchanged all around us. Entities are constantly sending, receiving and processing messages. Human beings for instance: when we watch TV, when we talk to a friend, talk over the phone, or send an email message. Right now, you are reading this written message which is possible because of messaging. Since computer applications seek to model the real world, it is only natural to design

and write applications using a messaging approach. We can argue that messaging provides a better and more accurate representation (i.e. model) of the real world.

Motivations: Object oriented applications consist of a collection of components that need to communicate with each other. As proposed hereafter, this interaction can be accomplished via a messaging (MDP) model. This is consistent with the real world in which independent entities interface with each other via messaging. Each entity is a self-contained/independent unit. The mechanism of communication with other entities is via messaging. As a consequence, MDP minimizes coupling. MDP also improves encapsulation and reusability.

Conventional implementations, not based on a messaging model, are unable to provide this level of encapsulation, decoupling and reusability. MDP and the messaging model address these shortcomings while at the same time reducing complexity. Actually we can argue that messaging should be the principal mechanism of communication since components are independent entities. Therefore they should be modeled and treated as such, in order to accurately mimic reality. A later section will demonstrate how design patterns implemented using MDP are combined to implement access to distributed components. Conventional models not based on MDP present complexities and limitations. MDP addresses these problems.

Participants:

a) Message Sender: Component that sends the message.
b) Message Recipient (Receiver): Component that receives the input message and may produce a reply (output message) after processing it. The input message, general in nature, may contain any type of information. The component may be instructed to perform computations based on the input message.
c) Messenger: Intermediary that transfers the message from the sender to the recipient. The sender and the recipient don't need to be concerned about how the message is transferred (communication protocol, message format, encryption/security mechanism, etc.) and the transformations performed on the message along the way. This is the messenger's purpose and responsibility. Similar to the real world, it is often the case that the messenger is not required. The message can be sent directly to the message recipient. Several modes of communication are possible: synchronous, asynchronous and two-way messaging.
d) Message: any piece of information (i.e. data) that needs to be interchanged between sender and recipient. Two messages are usually involved: input message and output message (or reply message). The reply message is not required.

Structure:

The messaging design pattern is implemented using the messaging interface (JtInterface). This interface consists of a single method to process the input message and produce a reply message.

Figure 1. Messaging Interface

The MDP messaging interface is simple but powerful. The simplicity of this interface can be deceiving. One method with one parameter (message) is all that is needed. It acts as a universal messaging interface that applies to remote and local components. This interface handles any type of message (For instance, the Java Object class). It returns a reply (of any type). In Java, the messaging interface can be declared as follows:

public interface JtInterface { Object processMessage (Object message); }

The message receiver needs to implement this interface in order to receive and process incoming messages. Although Java is used here, MDP can be implemented using any comparable computer language. Languages that don't use interfaces can simply declare a processMessage() function or method in order to implement MDP.

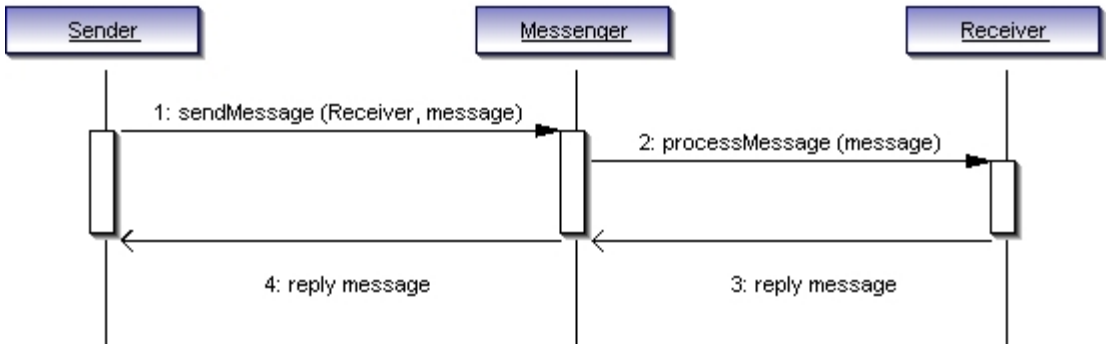

Figure 2. Messaging Design Pattern (synchronous mode).

Figure 3. Messaging Design Pattern (synchronous mode without messenger involved)

Consequences:

Encapsulation. The messaging design pattern maximizes encapsulation. As mentioned earlier, each component is a self-contained/independent unit. The only mechanism of communication with other components and applications is via messaging.

Decoupling. MDP minimizes coupling. Again each component is a self-contained unit that can perform independently from the rest of the system.

Reusability. MDP improves reusability. This is similar to the building blocks in a "Lego" set. Very complex models can be built based on simple pieces that share a simple way of interconnecting

them (i.e. common interface). The power of the approach is derived from the number of combinations in which these toy pieces can be assembled. Components that use MDP can be interchangeably plugged into complex applications. The components can be assembled in a limitless variety of configurations. The user of a component only needs to know the input/output messages that the component handles. Applications are also able to reuse components from other applications at the component level: a single component can be extracted from another application, provided that the messaging design pattern is being used.

QA/Testing process. MDP facilitates testing and debugging efforts. Components are tested as independent units by sending messages to the component and verifying the expected reply messages (black-box testing). In general, unit testing can be performed via a testing harness. No need to include testing code inside the component code which can be time consuming and lead to the unexpected introduction of software defects.

Design process. MDP improves and simplifies the design process. The bulk of the design work becomes defining the set of components needed to meet the system requirements and the input/output messages that each component needs to handle. There is a tight correspondence between UML design diagrams and the components needed for the implementation. Several UML diagrams are geared towards messaging (sequence and collaboration) although their implementation doesn't rely on it. The UML model and the implementation are disconnected when MDP is not used. Since all components share the same messaging interface, they can also be readily added to BPM/BPEL diagrams. As mentioned earlier, this is similar to building blocks that can be reused and connected in many different ways.

Development process. Since each component that relies on messaging is self-contained, a large team of people can cooperate in the development effort without stepping on each other's work/code. In the ideal situation, responsibility for one component/package can be given to an individual. The rest of the team only needs to know the input/output messages that someone else's component is designed to handle. In general, there is no need to change someone else's code. The need for creating, maintaining and merging several versions of the code is also minimized or eliminated. Testing/QA engineers can perform their testing independently via a testing harness. As a general rule, there is no need to add testing code to the component itself.

Logging and Debugging. Since all the components use the same messaging interface, messages can be logged automatically. This minimizes the need for print/logging statements inside the code which can be time consuming and error-prone. By taking a look at the messages being interchanged and automatically logged, the user is usually able to quickly track down the message/component that is causing the problem (with minimum or no extra effort).

Security. Well-known encryption and authentication mechanisms fit in well with the messaging design pattern. Strong security can be provided by the framework that implements MDP [3]. This is done by encrypting and authenticating the messages being interchanged. The sender and the recipient don't need to be too concerned with how secure messaging is implemented. This provides strong security while at the same time simplifying the implementation of security. If required, custom security mechanisms can also be incorporated: sender and receiver need to agree on and implement the message encryption/authentication mechanism to be used.

Multithreading and asynchronous messaging. MDP is able to handle the complexities associated with multithreading and asynchronous messaging. Components that implement MDP are able to execute in a separate/independent thread. This is a natural representation of the real world: each component (entity) is a self-contained unit and executes independently for the rest of the system.

Messages can be processed asynchronously using the component's own independent thread. This capability is usually implemented in the context of a component framework [3]. The component doesn't need to add separate logic to handle multithreading which is time consuming, complex and prone to error.

Speed of development and cost. Because of all the reasons outlined above, the messaging design pattern is able to substantially improve the speed of development and reduce cost.

Quality and software maintenance. Quality and software maintenance efforts are also improved as a result of the all of the above.

Messaging paradigm and learning curve. In order to take full advantage of this design pattern, people need to think in terms of a messaging paradigm when they model, design and build software applications: independent entities (i.e. components) interchanging messages among each other. This may require learning time and training. Although a messaging approach is natural, intuitive, and consistent with the real world, traditional approaches are based on method/procedure invocation (both local and remote).

Overhead. Transferring messages between components introduces a small overhead when compared with traditional method/procedure invocation. This is especially true when a messenger is used. As computers become faster and faster this becomes a non-issue. Also, the benefits of messaging outweigh this small performance cost. Notice that the messenger component is part of many real world problems and applications in which an intermediary is necessary for message interchange.

Disciplined approach. MDP encourages a disciplined approach that may have a small impact on the initial development time of a component. Messaging should be the only channel of communication between components. External class methods may still be invoked using the traditional approach. On the other hand, this should be used sparingly in order to minimize coupling and maximize encapsulation. An ideal component is a self-contained unit that interacts with the other components only via messaging. The additional development time is again outweighed by the benefits introduced by messaging. Moreover individual components based on messaging can be purchased or extracted from other applications.

Known uses

Design patterns. MDP has been used to implement and/or facilitate the implementation of other well-known design patterns like Gang of Four design patterns (GoF), DAO, J2EE Design patterns, etc. MDP also provides a more natural, streamlined and straightforward implementation of other design patterns. This topic is covered in detail later in this paper.

Distributed component and messaging model. MDP is particularly well suited for the implementation of a complete distributed component model. It is able to provide transparent access to remote components regardless of the protocol, technology and communication mechanism being used: remote objects are treated as local objects. Messages can be transferred via web services, REST, EJBs, HTTP, sockets, SSL or any comparable communication interface. Design patterns implemented using messaging (adapters, remote proxies and facades) make this possible by hiding the complexities associated with remote APIs. MDP solves a whole set of problems dealing with remote application interfaces and remote access to distributed components. Because of MDP, sender and recipient don't need to be concerned with how messages are transferred.

Component based frameworks and design/BPM/BPEL tools. MDP can be utilized to implement component based frameworks [3]: components can be interchangeably plugged into complex framework applications using the "Lego" architecture previously described. These components can also be readily incorporated into UML/BPM/BPEL diagrams in order to design and implement complex applications. Notice that for components to be used interchangeably, they need to share the same interface. MDP provides this common messaging interface.

Secure Web Services. MDP has been utilized to implement secure web services. This includes RESTful web services. A Web service is just another mechanism of communication between heterogeneous applications. Notice that the messaging design pattern doesn't place any restrictions on the message sender and recipient. These components can be running on multiple computers and operating systems. They can also be implemented using multiple computer languages and technologies.

Enterprise Service Bus (ESB) components and applications. Messaging has been used to implement ESB components and applications. Once all the building blocks are present (remote proxies, adapters, facades, etc), they can be assembled to create a new application in a fraction of the time required by traditional methods.

Secure and Multithreaded applications. MDP provides the building blocks required to assemble secure and multithreaded applications in a fraction of the time required by traditional methods. These building blocks are usually provided within the context of a messaging framework.

Fault-tolerant applications. MDP behaves like a state machine. Therefore it can be extended to provide fault-tolerant capabilities in a very natural fashion by replicating components and coordinating their interaction via consensus algorithms. Adding fault-tolerant characteristics to a program that uses a traditional approach is, in general, a difficult undertaking.

Related Patterns

Related literature [2] has been published describing messaging patterns in the specific realm of Enterprise Application Integration (EAI) and SOA. This work focuses on the communication between multiple applications. MDP is not limited to this realm. The approach used by MDP is distinctively different. Moreover, our formalization of the messaging design pattern, its consequences and applicability are more extensive and deeper. For instance, the application of MDP to pattern design/implementation, distributed component model, communication between local components, multithreaded applications, and software modeling/design. Related patterns don't focus on these areas.

Design Pattern implementation

As mentioned earlier, most design patterns have to deal with the interchange of information between pattern participants. The messaging design pattern has been used to implement and/or facilitate the implementation of other well-known design patterns like Gang of Four design patterns (GoF), DAO, MVC, J2EE design patterns, etc. Many design patterns will be used to illustrate how this is accomplished. The same concepts apply to the implementation of many others. MDP also provides a more accurate and straightforward implementation. Although synchronous messaging is shown, MDP also supports asynchronous messaging.

Design patterns implemented using MDP can be reused as building blocks. A generic pattern implementation becomes possible. Complex applications can be built based on these building blocks which share a simple way of interconnecting them (i.e. common messaging interface).

Proxy

MDP facilitates the implementation of Proxy. Under the messaging paradigm, Proxy is mainly responsible for forwarding the input message to the real subject. Notice that all the participants are completely independent (minimum coupling). Encapsulation is maximized. MDP messaging is the only mechanism of communication between participants.

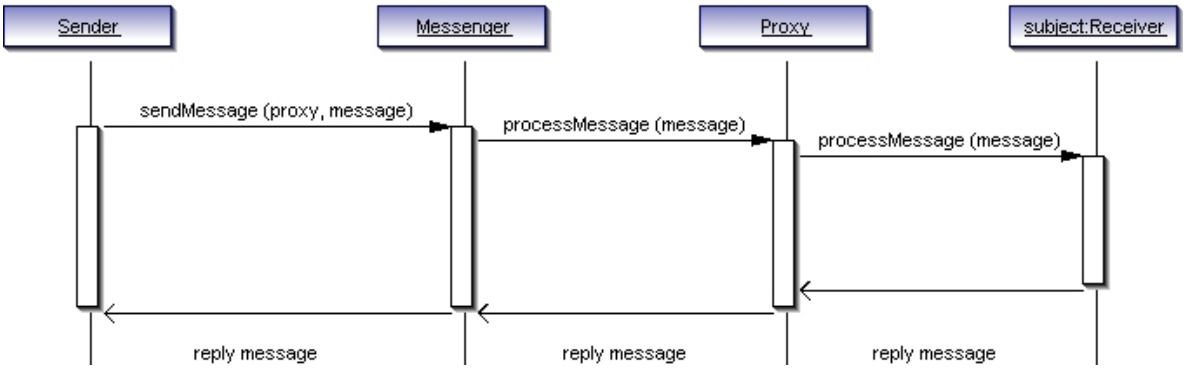

Figure 4. MDP implementation of Proxy

Comparison between MDP and traditional approaches

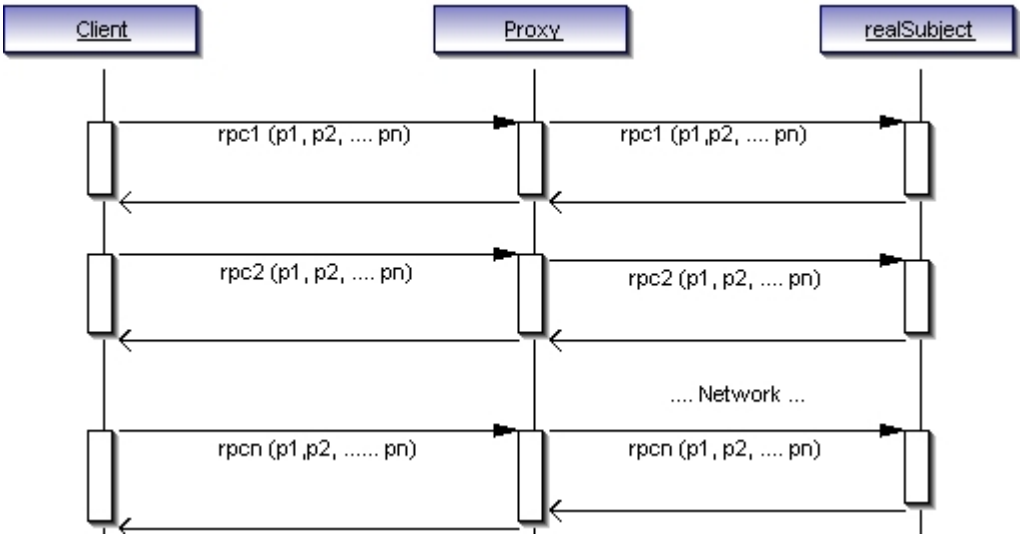

Figure 5. Traditional Proxy implementation (without messaging)

The diagram shown above (Figure 5) represents the conventional Proxy implementation. For the purposes of comparison, let's assume that a remote Proxy needs to be implemented. The conventional approach doesn't rely on messaging. The resulting shortcomings become evident. Similar shortcomings apply to the conventional implementation of other design patterns and component based applications. The gap is related to the lack of messaging in traditional models and methodologies. MDP bridges the gap and limitations:

Complexity: The conventional approach doesn't have a common interface. The Proxy needs to handle multiple remote procedure/method calls with multiple parameters each (rpc1 … rpcn). This becomes fairly complicated since we need to consider the stub/skeleton classes, IDLs or similar constructs that are required to implement remote procedure/method invocation. Parameter/method changes require a fair amount of work in terms of maintaining the additional constructs. Let's not forget that a production application probably needs to handle a substantial amount of proxies. The complexity becomes hard to handle under the traditional approach. In order to compare complexity, please refer to the code example close to the end.

Lack of Reusability: A conventional Proxy needs to be implemented for each remote component. This is in addition to the required stubs/skeleton classes or similar constructs.

Coupling: The conventional approach presents some dependencies as shown above. Changes to local parameters and/or methods have an impact on the remote methods/parameters and vice versa. Updates need to be made. Stub/Skeleton classes (or similar) need to be updated as well. Components are not completely decoupled as opposed to MDP.

Encapsulation: The conventional approach exposes some dependencies as discussed above. It can be argued that components are not totally encapsulated.

Secure Communication: Let's complicate the previous scenario. In a realistic situation, distributed components usually need to share information in a secure/encrypted fashion. Implementing security becomes more complicated under the conventional implementation because of coupling/dependencies between components. Transport level security is usually required.

Multithreading: Now let's assume that the remote component needs to use a separate thread to process the remote call. This is another situation that is often found in a real world scenario. The conventional approach requires the implementation of separate multithreading logic which is time consuming, complex and error prone.

BPM/BPEL diagrams: Components that don't rely on messaging cannot be readily added to BPM/BPEL diagrams because of the lack of a common messaging interface.

The problems outlined above become worse in terms of complexity when the scenarios are combined (secure communication and multithreading). On the other hand, MDP brings the following benefits and consequences:

Simplicity: MDP requires a simple messaging interface: one single method (processMessage) with one parameter (message). It also requires *only one* remote procedure/method call with one parameter. Intuitively this seems to be an optimum (minimum) number in terms of complexity and coupling. The messaging interface never changes. Therefore it doesn't require any maintenance. The overall design and UML diagrams are also simplified and streamlined making them easier to understand and implement (figure 4). In order to compare complexity, please refer to the code

example close to the end. A message can be sent to a remote component by simply using the following statement: *messenger.sendMessage (proxy, message);*
This can be done regardless of the component class, protocol, technology, message class/format. Traditional approaches are unable to accomplish this due to limitations, tight coupling, etc.

Reusability: The MDP proxy can be reused for any remote component implemented based on the common messaging interface. A generic implementation of the design pattern can be achieved by using MDP.

Coupling: MDP messaging maximizes decoupling. The components are completely independent. Messaging is the only channel of communication between participants. Since the MDP interface is generic, new messages, arbitrary message types and message changes can be handled without causing any major impact.

Encapsulation: MDP maximizes encapsulation. MDP components are totally encapsulated. Again, messaging is the only channel of communication.

Secure Messaging: MDP can handle this scenario by plugging in additional MDP components to handle security and encryption. Since MDP maximizes decoupling, this becomes possible. For instance, encryption/decryption components can be added to the MDP messaging pipeline (Figure 14). The local component may also encrypt the message before sending it. Specific messages or portions of a message may be encrypted.

Multithreading/Asynchronous messaging: MDP is able to handle this scenario with ease. A queuing mechanism can be added to store the incoming messages (Figure 13). A framework based on MDP is able to provide such logic.

BPM/BPEL diagrams: MDP patterns and components can be readily added to BPM/BPEL diagrams because they share a common messaging interface. This is part of ongoing and future work.

Adapter

Under messaging and MDP, the main purpose of Adapter becomes the transformation of messages between message sender and receiver so that these components can be interconnected. For instance, you may need to implement a HTTP Adapter so that your local component can communicate with a remote component via the HTTP protocol. The same principle applies to other communication technologies and protocols (sockets, web services, REST, etc.)

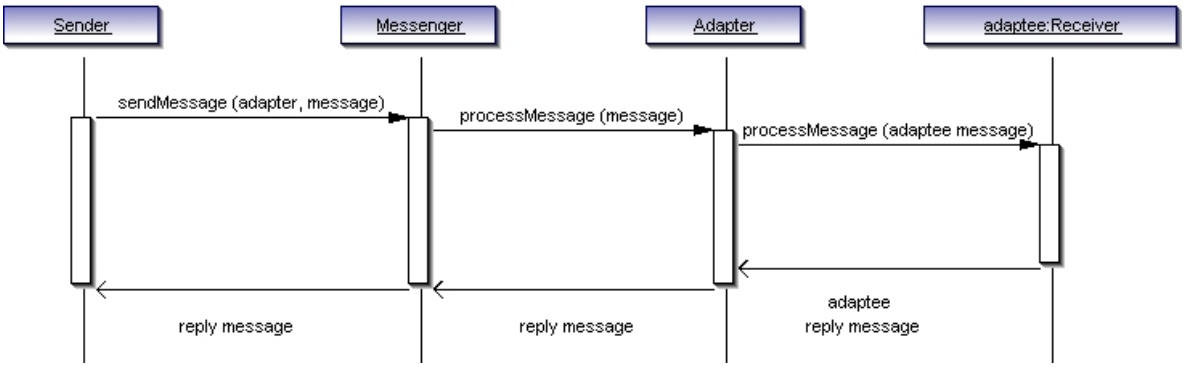

Figure 6. Implementation of Adapter via of MDP

Strategy

Under a messaging paradigm, Strategy is mainly responsible for forwarding the message to the component that implements the concrete strategy.

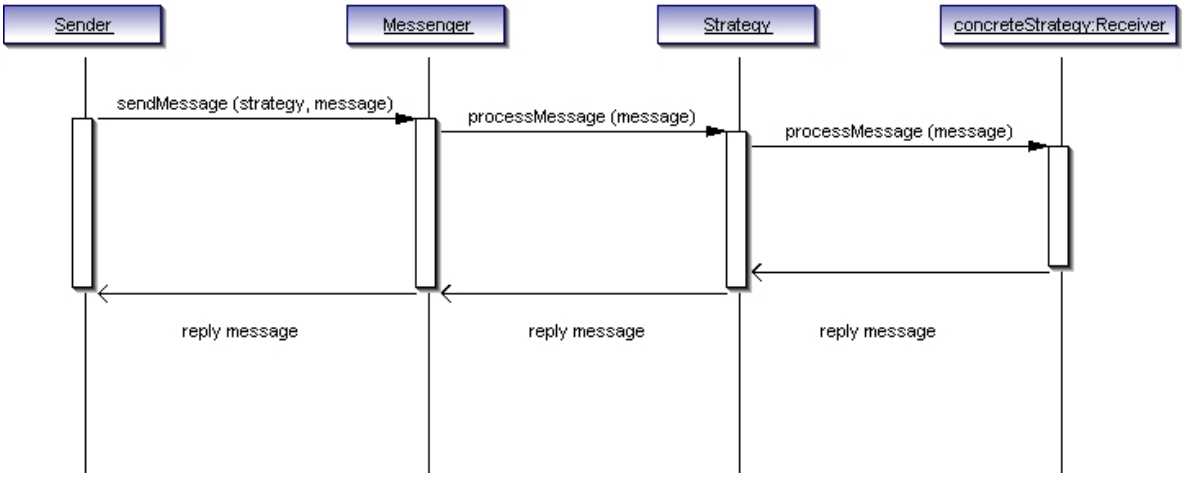

Figure 7. Strategy implementation using MDP

Façade

Under messaging and MDP, Façade is mainly responsible for forwarding the message to the appropriate subsystem.

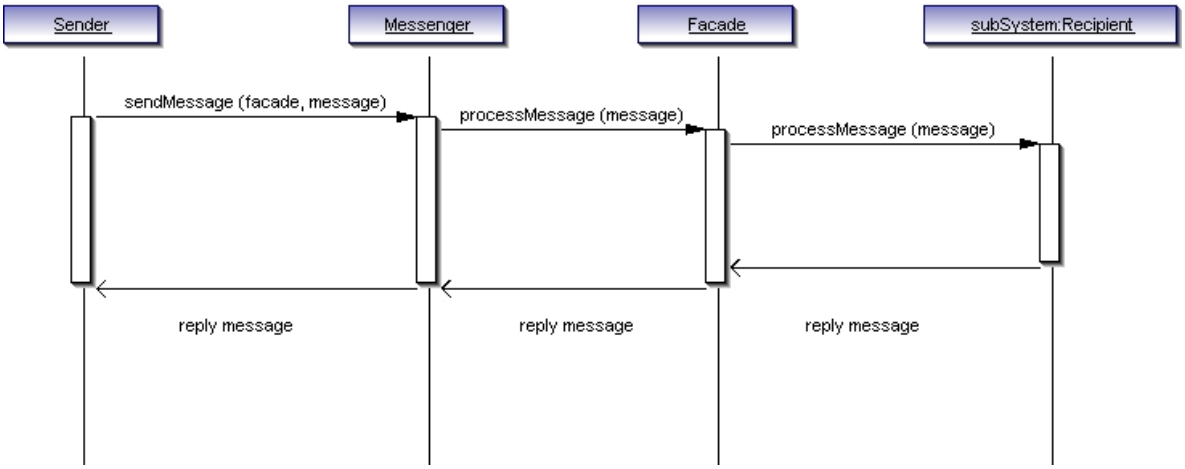

Figure 8. Façade implementation using MDP

Bridge

When MDP is used, Bridge is mainly responsible for forwarding the message to the concrete implementer.

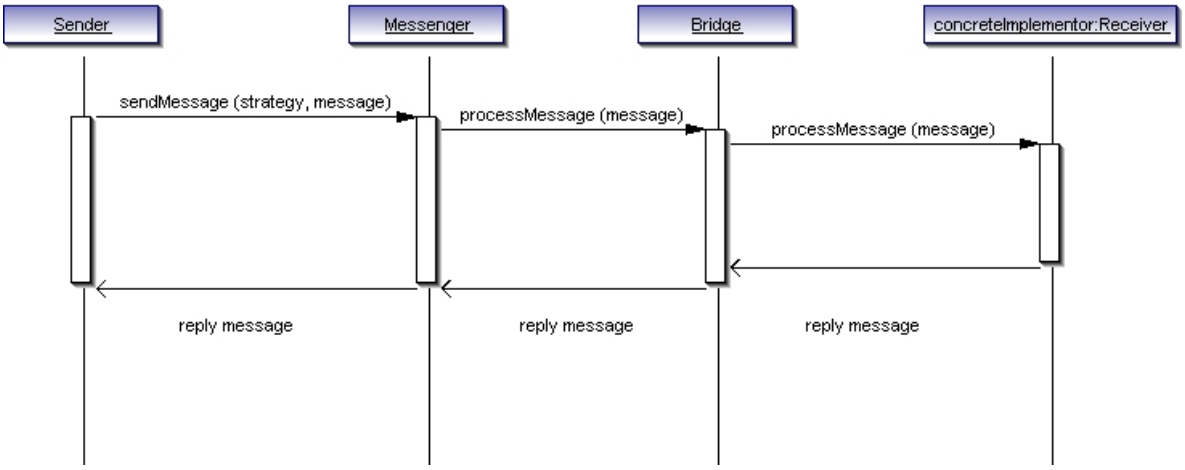

Figure 9. MDP implementation of Bridge

Command

MDP facilitates the implementation of Command (Figure 10). Under the messaging paradigm, Command is responsible for processing the request/message. It may also queue or log requests (RequestLogger).

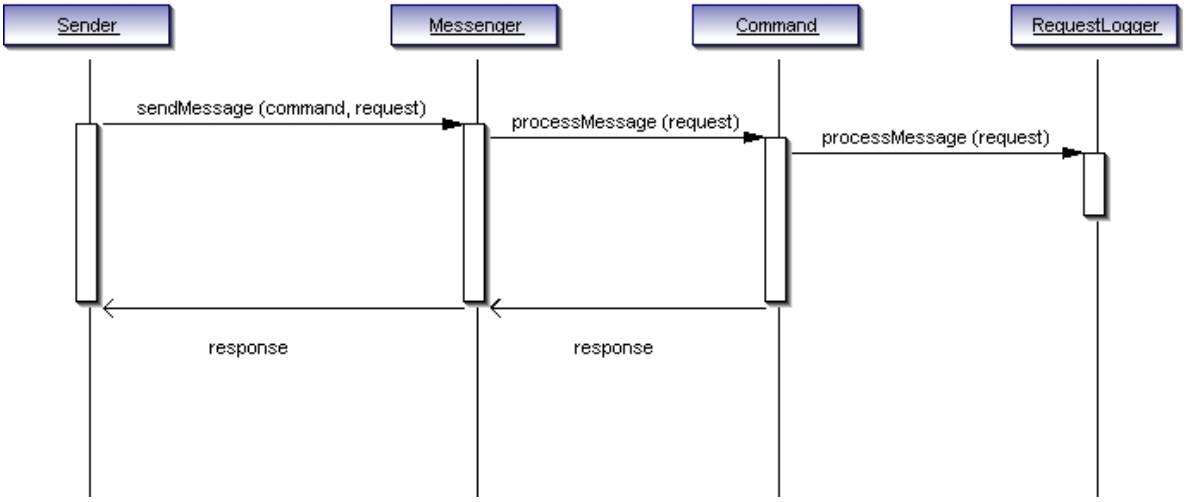

Figure 10. MDP implementation of command

Decorator

When MDP is used, Decorator is responsible for implementing new functionality. Decorator is also responsible for forwarding messages (related to the existing functionality) to the decorated component.

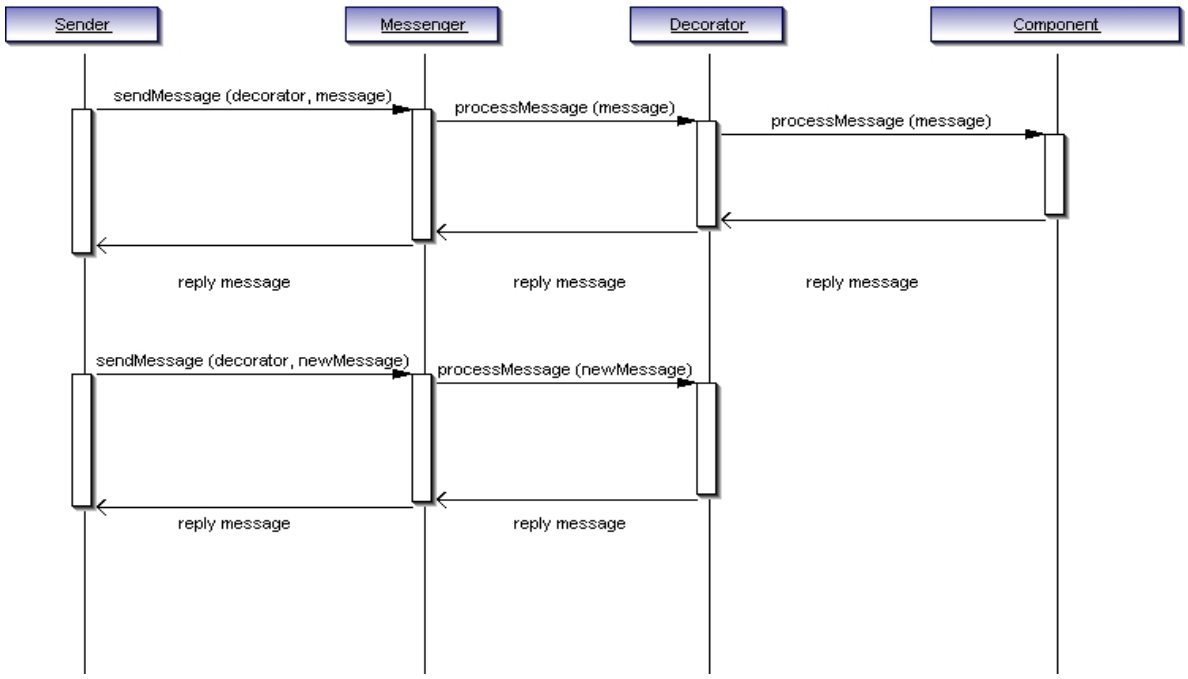

Figure 11. MDP implementation of Decorator

Prototype

When messaging is used, Prototype is responsible for implementing the ***clone*** functionality/message: Prototype returns a copy of the object when the ***clone*** message is received.

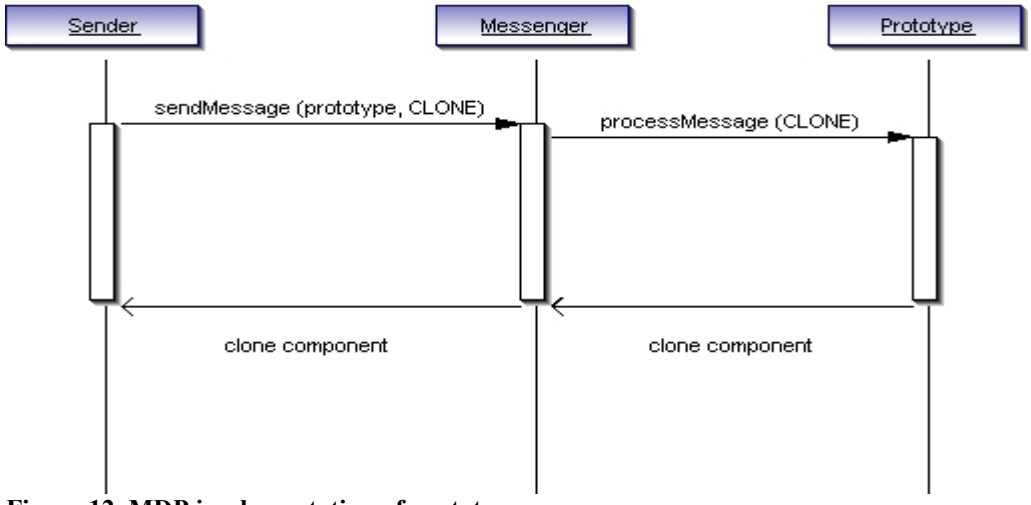

Figure 12. MDP implementation of prototype

Asynchronous messaging and multithreading

Now consider your short term memory, email system, postal mailbox or message recording machine. Messages can be sent asynchronously and placed in a message queue or pile until you are ready to "process" them. MDP is able to mimic this behavior and handle the complexities associated with asynchronous messaging and multithreading (figure 13). As mentioned earlier, components that implement MDP are able to execute in a separate/independent thread. This is an accurate representation of the real world: each component (entity) is a self-contained unit able to execute independently from the rest of the system. Messages can be processed asynchronously using the component's own independent thread. This capability is usually implemented in the context of a component framework via a messaging queue. MDP components don't need to add separate logic to manage multithreading. Conventional models require separate multithreading logic which is time consuming, complex and prone to error. The complexities associated with multithreading applications are addressed by the MDP model.

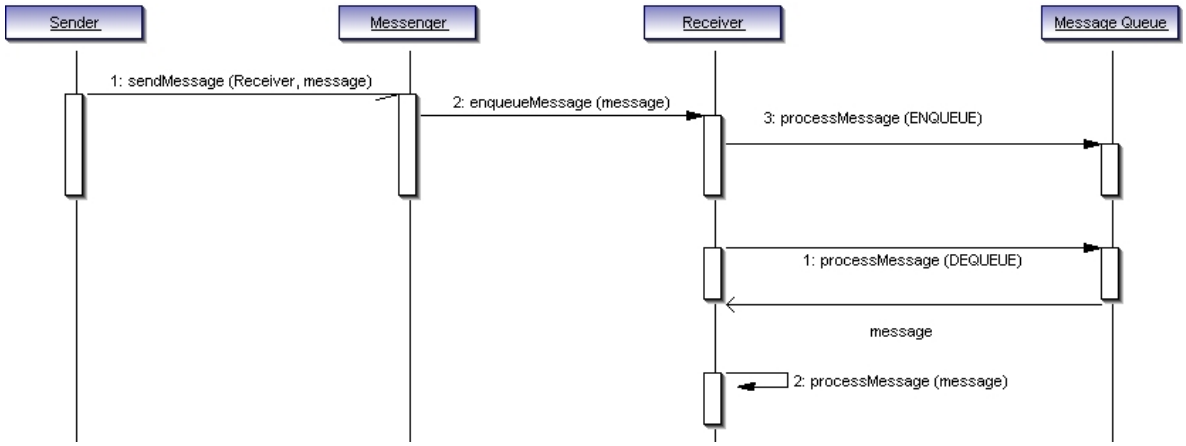

Figure 13. MDP Asynchronous messaging

Message Queue: internal subcomponent used to enqueue the incoming messages. The component needs to rely on some sort of internal memory or queuing mechanism to store the incoming messages. The component's thread dequeues the messages and processes them. A MDP framework usually implements the logic required to manage the component thread and the message queue. The MDP component only needs to inherit this functionality from a framework superclass. The MDP approach is not only realistic; it also handles most of the complexities associated with the implementation of multithreading.

MDP access to distributed components and services

MDP and several of the other design patterns discussed earlier can be combined to implement access to remote components and services. Notice that message sender and receiver don't need to be running on the same host. MDP is able to provide transparent and secure access to remote components regardless of the protocol and communication technology being used: remote components are treated as local components. Messages can be transferred via web services, EJBs, RMI, HTTP, REST, Sockets, SSL or any other comparable communication interface. The design patterns discussed above (messaging, adapters, proxies and facades) make this possible by hiding the complexities associated with remote APIs. The following UML diagram illustrates how this is accomplished. In the real world, you can communicate with a friend across the room or thousands of miles away via a phone conversation. You and your friend don't need to be concerned as to how the telephone conversation is transmitted. It is transparent to you. The phone company provides the

messaging framework to make it possible. By mimicking this behavior, MDP is able to provide transparent communication between local and remote components and services.

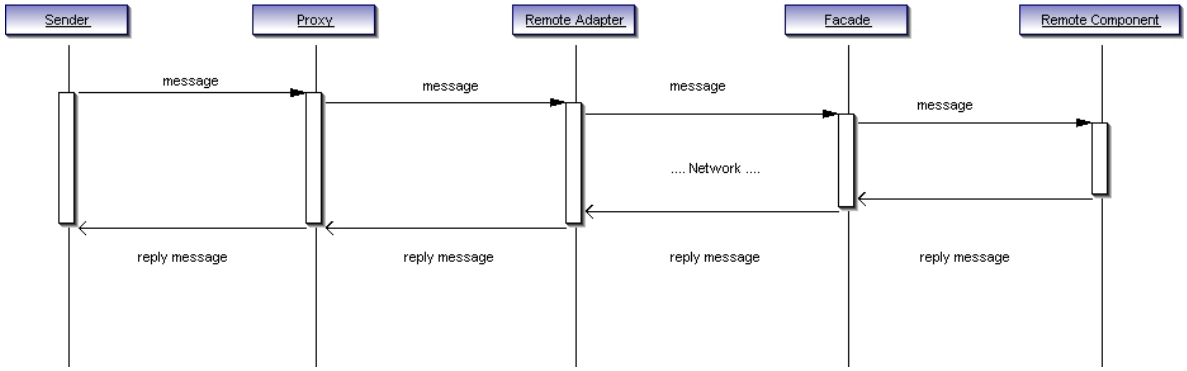

Figure 14. MDP access to remote components and services

The following are the design patterns involved. For clarity sake the messenger component and the intrinsic processMessage() method have been removed from the UML diagram.

1) Proxy: the message is sent to the remote component via its proxy.
2) Remote Adapter : adapter responsible for interfacing with the remote API.
3) Façade: forward the message to the appropriate remote component. It usually provides security capabilities as well.

Implementation of J2EE Design Pattern

Enterprise Java Beans (EJBs) is one of the technologies that can be used to access remote components. The next section will illustrate how the messaging design pattern and several other J2EE design patterns can be combined to implement transparent access to distributed EJB components. The same principles apply to other communication technologies and protocols.

J2EE Business Delegate

When MDP is used, Business Delegate is mainly responsible for forwarding the message to the remote component via EJBAdapter and J2EESessionFacade. The behavior is very similar to Proxy.

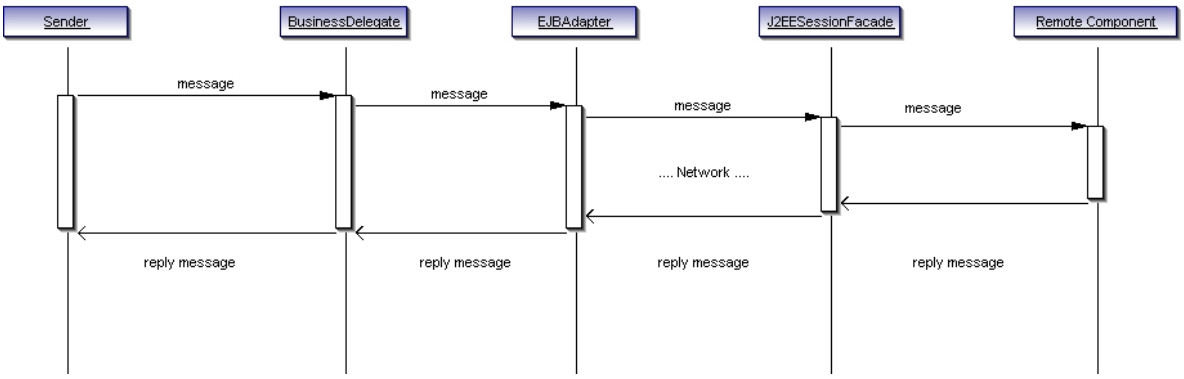

Figure 15. MDP implementation of J2EE Business Delegate

J2EE Session Façade

When MDP is used, Session Façade is mainly responsible for forwarding the message to the appropriate remote component.

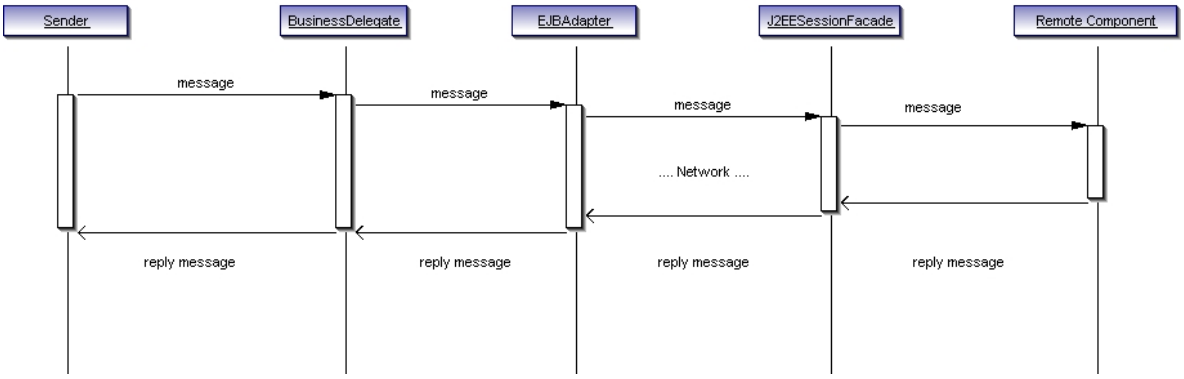

Figure 16. MDP implementation of J2EE Session Façade

The following are the design patterns involved. For clarity sake the messenger component and the intrinsic processMessage() method have been removed from the UML diagram.

1) Business Delegate: the message is sent to the remote component via the business delegate.
2) EJBAdapter : adapter responsible for interfacing with the EJB API. It transfers the message to J2EESessionFacade.
3) J2EESessionFacade: forward the message to the appropriate remote component.

The J2EE session façade is usually responsible for security as well (messaging authorization and authentication). Notice that under a messaging paradigm, this can be made transparent to message sender and receiver. In other words, sender and receiver don't need to be too concerned as to whether or not secure messaging is being used and how it is being implemented. The framework should provide the required security mechanisms ("plumbing"). Using a real world analogy, in general you and your friend should not be concerned if the service provider is encrypting your phone conversation because of privacy and security considerations.

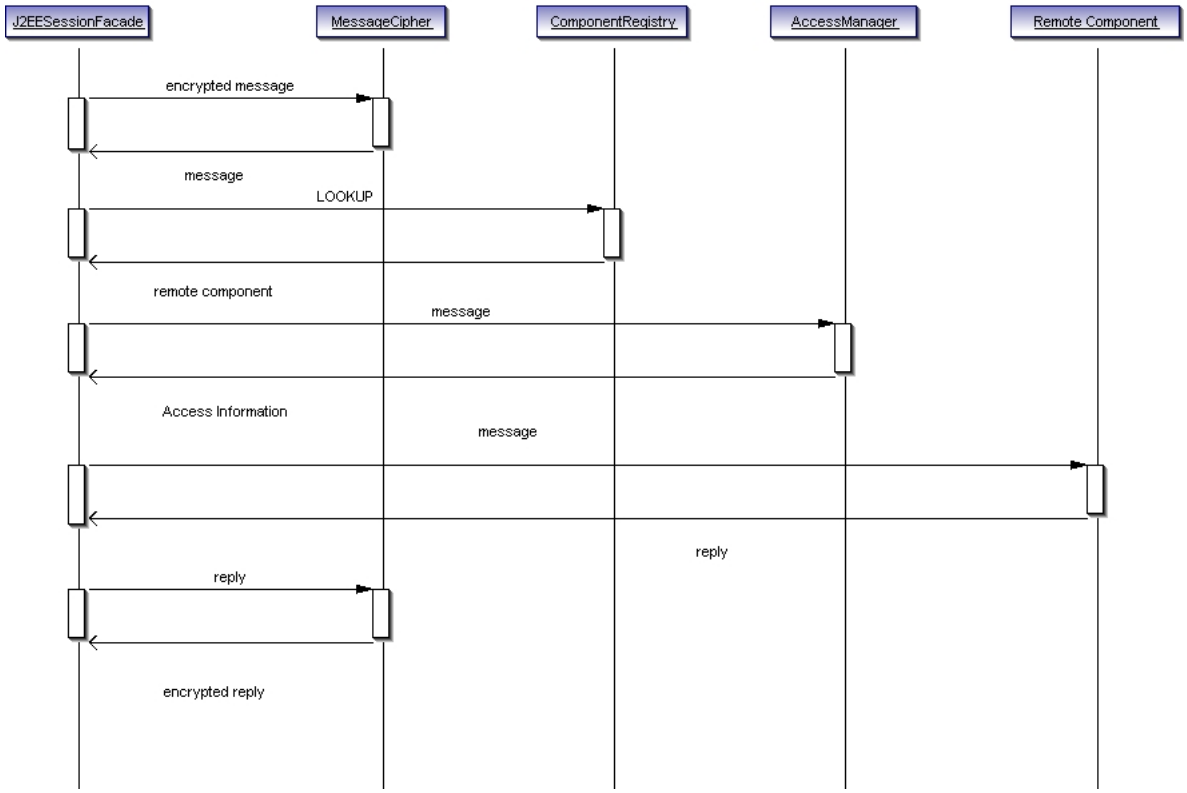

Figure 17. MDP implementation of J2EE session façade

Before the message is forwarded to the Receiver, J2EESessionFacade performs decryption, authorization and authentication on it. The following are the components involved:

MessageCipher: component responsible for decrypting the input message and encrypting the reply message.
Component Registry: allows the system to register and look up components by name.
AccessManager: responsible for granting/denying access to remote components. It authorizes and authenticates each message received.

J2EE Service Locator

When messaging is used, Service Locator is mainly responsible for locating the service (home interface) by interfacing with the JNDI Adapter. JNDIAdapter is a messaging adapter that interfaces with the JNDI API.

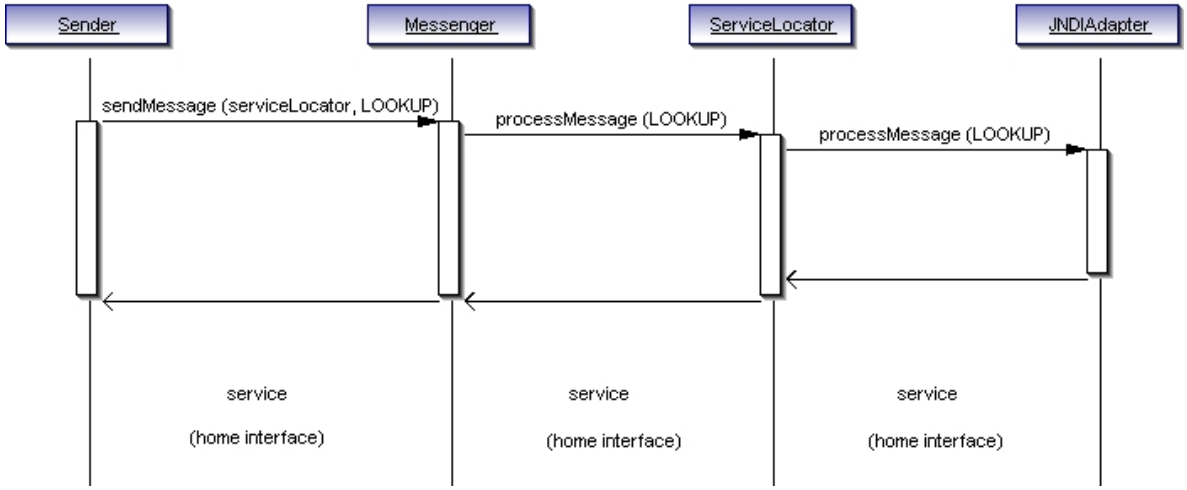

Figure 18. J2EE Service Locator

Related Work

MDP may be confused with message-oriented middleware (MOM) technologies like the Java Messaging Service (JMS). Although MOM solutions follow a messaging design pattern, they are not the same. MOM is just one of the many applications that utilize messaging. Your e-mail application also follows a messaging pattern. It is not unusual for a design pattern to be studied and documented after it is first utilized. MDP can be applied to many problems and technologies. MDP is not limited to MOM or EAI applications. On the other hand, MOM technologies cannot be used to handle the range of MDP applications mentioned above. For instance, MOM cannot be applied to provide pattern implementation or communication between local components.

The Jt design pattern framework[3] has been designed and built based on the principles and concepts outlined in this paper. The framework is conceived and implemented, from the ground up, based on design patterns. The framework architecture is based on MDP which is utilized to implement many well-known design patterns, in addition to SOA, ESB and BPEL technologies.

Code Examples

The following code example [3] illustrates the use of several of the design patterns discussed earlier. MDP is being used to implement web services and distributed access. A proxy is used to provide transparent access to a remote component or service. The message is forwarded to the remote component via the MDP Adapter and Façade (Figure 14).

```
/**
 * Send a message to a remote component/service using a remote MDP proxy.
 * Secure and asynchronous messaging is supported.
 * The messaging design pattern provides transparent access
 * to remote components. HTTP is used here. Any other comparable
 * communication protocol/technology can be used.
 */

public static void main(String[] args) {

      JtFactory factory = new JtFactory ();
      String sReply;
      JtHttpProxy proxy;
```

```
        String url = "http://localhost:8080/JtPortal/JtRestService";
        JtMessenger messenger = new JtMessenger ();

        // Create an instance of the remote Proxy.

        proxy = (JtHttpProxy)
            factory.createObject (JtHttpProxy.JtCLASS_NAME);

        proxy.setUrl(url);
        proxy.setClassname ("Jt.examples.Echo");

        // Specify that secure/encrypted messaging should be used

        messenger.setEncrypted(true);

        // Asynchronous messaging is supported
        //messenger.setSynchronous(false);

        // Send the message to the remote component/service.

        sReply = (String)
             messenger.sendMessage (proxy, "Welcome to MDP messaging ...");

        System.out.println ("Reply:" + sReply);


}
```

References

1. Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Reading, MA, 1994.
2. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Gregor Hohpe and Bobby Woolf, Addison-Wesley, 2004
3. Galvis, A. *Jt - Java Design Pattern Framework, An application of the Messaging Design Pattern*. IBM Technical Library, 2010.