Abstract

This paper describes the messaging design pattern (MDP). 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. Therefore it is only natural and intuitive to model, design and create software based on a messaging pattern (i.e. paradigm). 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. This is a key aspect. The messaging paradigm and design pattern provide a robust and accurate model to represent the real world. As a consequence, software engineering processes and techniques are substantially improved.

Messaging Design Pattern (MDP)

Intent: The messaging design pattern allows the interchange of information (i.e. messages) between components and applications.

Motivations (forces): 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, listen to music, talk over the phone, or communicate via the internet. Right now, you are reading this written message. Since computer applications seek to model the real world, it is only natural to design and create software using a messaging approach. We can argue that this approach provides a more complete and accurate representation (i.e. model) of the real world. Therefore software engineering processes are significantly improved by the use of the messaging pattern.

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.

**Messaging Interface**

**Messaging Design Pattern (synchronous mode)**

**Messaging Design Pattern (synchronous mode without messenger involved)**
Consequences:

- Encapsulation. The messaging design pattern maximizes encapsulation. 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. 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. 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. In general, 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 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 the messaging design pattern. 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. 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. 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.

- In order to take full advantage of this design pattern, people need to think in terms of messaging 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 libraries and method/procedure invocation (both local and remote).

- Transferring messages between components introduces a small overhead when compared with traditional method/procedure invocation. As computers become faster and faster this becomes a non-issue. Also, the benefits of messaging outweigh this small performance cost.

- 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.

- Remote proxies and application interfaces. MDP is particularly well suited for the implementation of remote access. 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 other 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. 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: 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 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. MDP and the “Lego” architecture make it possible.

- 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.

- MDP behaves like a state machine. Therefore it can be extended to provide fault-tolerant capabilities in a very natural and intuitive 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.
Implementation and Code Examples:

The messaging design pattern is implemented using the Jt messaging interface (JtInterface). This interface consists of a single method:

```java
public interface JtInterface {
    /**
     * Jt messaging interface used for the implementation
     * of the messaging design pattern.
     * Process an input message and return a reply (output message).
     */
    Object processMessage (Object message);
}
```

The messaging interface (JtInterface) is simple but powerful. The simplicity of this interface can be deceiving. One method is all that is needed. It acts as a universal messaging interface that applies to remote and local framework components. This interface handles any type of message (Object class). Although Java is used here, MDP can be implemented using any computer language.

The following example sends a message to a remote component/Restful service via a remote proxy. The messaging design pattern is able to handle any communication protocol/technology. To illustrate the implementation, several UML diagrams are included. Figure 1 shows how MDP is used to implement the Proxy design pattern. Figure 2 shows how MDP and several design patterns are combined to implement web services. Figure 3 demonstrates how other communication interfaces can be handled by plugging in the appropriate adapter. Figure 4 shows the implementation of secure web services by plugging in additional components. For clarity sake, the message sequence and the intrinsic processMessage() method have been removed from these diagrams. Only synchronous messaging is shown although MDP also supports asynchronous messaging.

![UML Diagram](image)

Figure 1. Proxy implementation using MDP. The proxy forwards the message to the subject.
Figure 2. Web Service implementation using MDP.

Figure 3. Implementation of remote communication interfaces using MDP.

Note: Additional communication protocols/technologies can be handled (Web services, EJB, sockets, SSL, etc). It is just a matter of replacing the HTTP Adapter and plugging in the appropriate adapter for the remote interface. The messaging design pattern and architecture make this possible. The other components remain unchanged. MDP solves a whole family of problems associated with remote APIs.

Figure 4. Secure web service implemented by plugging in a component to perform message encryption/decryption.
public static void main(String[] args) {
    JtFactory factory = new JtFactory();
    String sReply;
    JtRestProxy proxy;
    String url = "http://localhost:8080/JtPortal/JtRestService";
    // Create an instance of the remote Proxy.
    proxy = (JtRestProxy)
        factory.createObject (JtRestProxy.JtCLASS_NAME);
    proxy.setUrl(url);
    proxy.setClassname ("Jt.examples.Echo");
    // Specify that secure/encrypted messaging should be used.
    proxy.setEncrypted(true);
    // Send the message to the remote component/service via
    // the remote proxy.
    sReply = (String)
        factory.sendMessage (proxy, "Welcome to MDP messaging ...");
    // The remote component will echo the input message.
    System.out.println ("Reply:" + sReply);
}

Related design patterns:

- Enterprise Integration Patterns. Related literature has been published describing messaging patterns in the specific realm of Enterprise Application Integration (EAI). MDP is not limited to this realm. Our formalization of the messaging pattern, its consequences and applicability are more extensive and deeper. For instance, the application of MDP to local components, multithreading applications, pattern implementation, and software modeling/design can bring significant improvements to software related processes. This paper also proposes a direct messaging approach over 'hybrid' approaches based on a combination of messaging and remote method/procedure invocation. The additional layer and its complexities become redundant.

- GoF design patterns and design patterns in general. GoF design patterns and most of the other design patterns deal, at some level, with the interchange of information (i.e. messaging) between their constituent components. Therefore MDP is well suited for their design and implementation as discussed above.