ISO 8583 Message Processor (Free.iso8583)

Manual version 1.8

Written by
AT Mulyana (atmulyana@yahoo.com)

Last Edited on September 10, 2015

List of Content

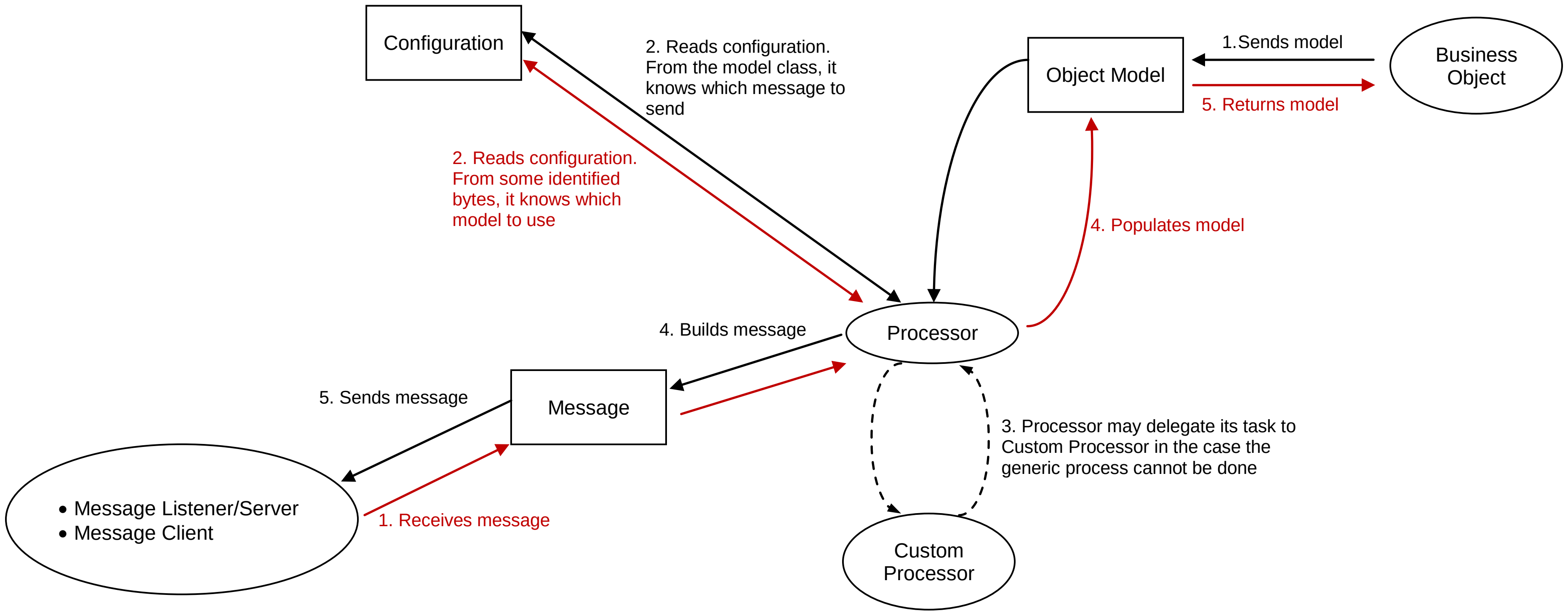
DONATION

If you feel this library is useful, please make a donation via PayPal by clicking this button/link:

https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=4ZKPC3URPZ24U

Process Diagram

Purpose: To make an abstraction from the business objects so that they only process a model object without realizing how ‘hard’ to parse/build an ISO 8583 message. Another purpose is to make generic process, all message types will be processed by one processor, only provides a configuration, no hard coding.


Configuration
2. Reads configuration. From the model class, it knows which message to send
2. Reads configuration. From some identified bytes, it knows which model to use
Object Model
1.Sends model
5. Returns model
Business Object
4. Populates model
Processor
4. Builds message
Message
5. Sends message
• Message Listener/Server
• Message Client
1. Receives message
3. Processor may delegate its task to Custom Processor in the case the generic process cannot be done
Custom Processor

Message Processor

Processor should have the life cycle at the application scope. It is started when the application starts and destroyed when the application stops. When it starts, it loads the message mappings configuration and maintains it as long as the processor alive. Loading/parsing configuration process only once is a benefit because it will speed up the process of message parsing/compiling.

The processor acts as a server which serves the requests from message listener, message client and business objects. The processor will create as many threads as needed to serve all request. To limit how many thread can run is the task of the message listener and the business objects.

When it starts and then parses the configuration and if it finds any bad configuration, it will throw an error and be shutdown. The processor will check all object classes mentioned in the configuration. After successfully started, it will not be shutdown if there is any bad ISO 8583 message comes because of no match configuration or the other reasons. It will only log the error. It should be never shutdown except the application is shutdown.

The processor will maintain some fields of a request so that it can be sent back as response, such as system trace number, or generate automatically a field value, such as transaction system. The processor will calculate the value of message length field for a response. If a request message comes and the message length field records a different length value from the actual length then the processor logs the error and no further process executed (message is ignored) . Actually, these tasks will be delegated to message parser and message compiler.

Class `MessageProcessor`

It is the main class to/from which the ISO 8583 message comes and goes. It is the message gateway. It will instantiate the other objects as necessary to process the message. Only one instance of this class exists (a singleton object). This class cannot be instantiated directly but via method `GetInstance` instead. To make it works correctly, method `Load` must be called when the application starts to load the configuration. This method must be also called again when the configuration changes.

There are other complementary objects provided by this library, those are Message Listener (to accept a request and send its response) and Message Client (to send a request and accept its response). These objects, however, can be replaced by outside objects to fit the application's need, as long as they implement `IMessageStream`. These replacing objects must be passed as the parameter to method `Send` or `Receive`.

```
namespace Free.iso8583
public class MessageProcessor
```

- `public static MessageProcessor GetInstance()`
  Returns an instance of this class. Use this method instead of instantiating this class directly. In fact, only one instance exists and this method always returns the same instance.
- `internal void Load(IConfigParser configParser)`
  Reads the mappings configuration by invoking `Parse` method of `configParser`

parameter. It will clear the configuration that exists before and replace it by the new read configuration. This method should be invoked when the application starts or each time the configuration changes. It will throw an exception if there is any bad configuration.

- `public void Load(Stream fileConfig)`
  Call this method if we want to start the processor using XML configuration. The parameter of this method is the stream of XML configuration file. This method should be invoked when the application starts or each time the configuration changes. Do not often invoke this method because it is very time consuming to parse XML configuration. This method calls `Load(IConfigParser)` by passing an instance of `XmlConfigParser` as the parameter.

- `public void Load(Type messageToModelMapping)`
  Call this method if we want to use attribute configuration. XML configuration has element `message-to-model` to map an incoming message to a model class. It's difficult to mimic this element by attaching an attribute to a model class because it must contain a list of mask rules. To load the attribute configuration, this method needs a parameter of `Type` object of a class. This class must define some public static methods. Those methods must return `IMask` object and have `MessageToModel` attribute and no parameter should be defined. Those methods will be invoked by `AttributeConfigParser.Parse` method. This `Load` method calls `Load(IConfigParser)` by passing an instance of `AttributeConfigParser` as the parameter.

- `public void Load()`
  Looks for the configuration on its own rule and invokes `Load(Stream)`. It's in the same directory as where the DLL file exists and named `messagemap-config.xml`.

- `public void Shutdown()`
  Shutdown the processor. Releases all held resources.

- `public IProcessedMessage Send(Object model, IMessageStream stream, MessageCallback callback)`
  Sends a message as a request (to Message Server). Optionally it will process a response message if exists. The response message which is formatted using ISO 8583 format will be converted to a model object before passed to the business object.
  Parameters:
  - `model` is the message will be sent. It will be converted to be ISO 8583 format.
  - `stream` is the message client to which the message will be passed to be sent to the remote host (server) or from which the response will be read.
  - `callback` is a delegate object which will be called after the reply message comes and converted to the model object. This delegate object accepts a parameter as a model object and returns another model object as the reply. Within this delegate, the business process is executed. If the parameter passed to this delegate is `null` then the delegate should not execute the business process or do some default processes. If `callback` is set to `null` then the processor will try to find an appropriate `MessageCallback` object based on its configuration. If still not found then no process will be executed after the response comes.

Returns:
An `IProcessedMessage` object. This is the threaded object which processes the message.

- `public IProcessedMessage Send(Object model, IMessageStream stream)`
  It simply delegates its task to the above `Send` method with the `callback` parameter set to `null`. No further execution if the reply message comes (ignored).

- `public IProcessedMessage Receive(byte[] message, IMessageStream stream, MessageCallback callback)`
  Invoked when a message is received by the message server (a request) or message client (a response).
  Parameters:
    - `message` is the received message using ISO 8583 format packed in a bytes array. If it's `null` or empty then the message processor will try to read from `stream` directly.
    - `stream` is a message server to which a message response will be passed to be sent to the remote host (message client) or from which the request will be read. It will be set to `null` if no response needed.
    - `callback`, when the message has been converted to a model object, it will be passed to `callback` (a `delegate` object with one parameter that is the object model). If this parameter is set to `null` then the processor will try to find an appropriate `MessageCallback` object based on its configuration.

  Returns:
  An `IProcessedMessage` object. This is the threaded object which processes the message.

- `public IProcessedMessage Receive(byte[] message, MessageCallback callback)`
  Typically it's used by the message client when receiving a response message. The client won't send a response. It simply delegates its task to the above `Receive` method with the `stream` parameter set to `null`.

- `public IProcessedMessage Receive(byte[] message, IMessageStream stream)`
  Typically it's used by the message server when receiving a request message. It simply delegates its task to the first `Receive` method with the `callback` parameter set to `null`.

Interface `IMessageStream`

This interface describes the interfaces for sending/receiving the message. Message listener/server and message client implement this interface. These two objects must have capability to send/receive message.

```
namespace Free.iso8583
public interface IMessageStream
```

- `byte[] Send(byte[] message)`
  Sends an ISO 8583 message to the remote host and returns a reply message. Returns `null` if no reply. This method will be invoked by `MessageProcessorWorker` when sending a message either as a request or a response. If this method returns an empty array (not `null`) then `MessageProcessorWorker` will try to read the reply message by using this object's `Receive` method.
- `int Receive(byte[] buffer, int offset, int count)`
  Reads the message from the remote host. It returns the count of received bytes. This method will be invoked by `MessageProcessorWorker` when receiving a message either as a request or a response. It should only delegate the task to `Read` method of `Stream` object corresponding to this object.
- `public void Close()`
  Closes the connection to remote host, releases all used resources. This method will be invoked by `MessageProcessorWorker` after processing a message.

Delegate `MessageCallback`

This is a callback that will be invoked when a message is received and has been converted to a model object. In this callback, the business process will be executed and a reply will be returned if exists. This callback is used by method `Send` and `Receive` of `MessageProcessor`.

```
namespace Free.iso8583
public delegate Object MessageCallback(Object model)
```

This delegate takes a parameter of a model object and returns another object model as a reply. Parameter and returned object can be `null`. If the returned object is `null` then no reply returned. If parameter is `null` then there was an error in processing the message. If the target object of this callback implements `IModelParsingReceiver`, the incoming message and its corresponding model object can be sought from `ParsedMessage` property. This property, however, may be `null` if the error happens before the parsing process.

Interface `IModelParsingReceiver`

This interface, typically, is implemented by the object which provides `MessageCallback` delegate. It's not mandatory that the object must implement this interface. It defines the interface to set the message that has been parsed (the message has been divided becomes bitmap, headers, content etc.). The object implementing this interface tells the message processor that it wants the result of message parsing.

```
namespace Free.iso8583
public interface IModelParsingReceiver
```

- `IParsedMessage ParsedMessage { set; }`
  It will be set by message processor after the message has been parsed and before the callback is executed. The callback process will be able to inquire more deeply about the message when it is executing.

Interface `IProcessedMessage`

This interface defines interfaces to access the message which is sent or received both in format of ISO 8583 and model object. It is implemented by `MessageProcessorWorker`.

```
namespace Free.iso8583
public interface IProcessedMessage
```

- `Object ReceivedModel { get; }`
  The received message that has been converted to the model object. It's `null` if no received message.

- `Object SentModel { get; }`
  The sent message that has been converted to the model object. It's `null` if no sent message.

- `byte[] ReceivedMessage { get; }`
  The received message that is formatted using ISO 8583 packed in bytes array. It's `null` if no received message.

- `byte[] SentMessage { get; }`
  The sent message that is formatted using ISO 8583 packed in bytes array. It's `null` if no sent message.

Class `MessageProcessorWorker`

When `Send` or `Receive` method of `MessageProcessor` object is invoked, it will create a thread to serve the request. The thread is an instance object of `MessageProcessorWorker` class. This is actual class which implements `IProcessedMessage` interface.

```
namespace Free.iso8583
internal class MessageProcessorWorker : IProcessedMessage
```

- `public MessageProcessorWorker(byte[] receivedMessage, IMessageStream stream, MessageCallback callback)`
  This constructor is invoked by a message server when a request message comes from a message client such as EDC.
  Parameters:
  - `receivedMessage` is the request message accepted. It has ISO 8583 format. If it's `null` or empty then the message will be read from `stream` directly.
  - `stream` is the thread inside the message server which processes the request. It's needed because the response message will be passed to it to be sent to the remote host or it's needed to read the request.
  - `callback` is a delegate object which will be called after the request message

comes and converted to the model object. This delegate object accepts a parameter as a model object and returns another model object as the reply. Within this delegate, the business process is executed. If this delegate is `null` then the processor will look for the appropriate delegate based on the configuration.

- `public MessageProcessorWorker(Object sentModel, IMessageStream stream, MessageCallback callback)`
  This constructor is invoked when a business object will send a message to a message server via the message client.
  Parameters:
    - `sentModel` is a model object that will converted to ISO 8583 formatted message and then to be sent to the message server.
    - `stream` is a message client that is responsible to process the message.
    - `callback` is a delegate object which will be called after the reply message comes and converted to the model object. This delegate object accepts a parameter as a model object. Within this delegate, the business process is executed. If this delegate is `null` then the processor will look for the appropriate delegate based on the configuration.
- `public String Id { get; private set; }`
  The id of this object assigned by message processor. This id can be used to get the reference to this `MessageProcessorWorker` object via `GetThread` method.
- `public int ThreadId { get; internal set; }`
  The id of the thread associated to this object. This id assigned by .NET framework.
- `public Object ReceivedModel { get; private set; }`
  Implements `IProcessedMessage` interface.
- `public Object SentModel { get; private set; }`
  Implements `IProcessedMessage` interface.
- `public byte[] ReceivedMessage { get; private set; }`
  Implements `IProcessedMessage` interface.
- `public byte[] SentMessage { get; private set; }`
  Implements `IProcessedMessage` interface.
- `public IMessageStream MessageStream { get; set; }`
  The same as `IMessageStream` object passed to constructor. It can be reset after the instance object is created.
- `public MessageCallback Callback { get; set; }`
  The same as `MessageCallback` delegate passed to constructor. It can be reset after the instance object is created.
- `public config.MessageToModelConfig MessageToModelConfig { get; private set; }`
  It's set by `Receive` method. When the message comes, this method inquires to

`GetQualifiedMessageToModel` method of `MessageConfigs` to get the appropriate `MessageToModelConfig`. It will be used by the next created `MessageParser` object.

- `public void ReceiveMessage()`
  When a message is received and the thread to process it is created, this method will be executed. This method will check `ReceivedMessage` property whether it's `null` or empty. If yes, it will call `Receive` method. Then this method will call `Parse` method to convert ISO 8583 message to be an appropriate model object, then `ProcessModel` method to process the model object and get a reply model object and then call `Send` method to send the reply model object that is previously converted to ISO 8583 message. This method is coupled with the first constructor which accepts an array of bytes containing ISO 8583 message.

- `public void SendMessage()`
  When a message will be sent and the thread to process it is created, this method will be executed. This method will call `Send` method to send the model object that is previously converted to ISO 8583 message and get the reply message. If `Send` method returns an empty array (not `null`) then it calls `Receive` method. After that, it calls `Parse` method to convert the reply message to be an appropriate model object and then call `ProcessModel` method to process the reply model object. This method is coupled with the second constructor which accepts a model object to be sent.

- `private void WriteMessageProcessException(Exception e, String log)`
  Writes an exception to the log with additional note. Beside that, it will note the message bytes (in hexadecimal format) or the model object (in the format of object initializer) that the processor fails to process. This method is invoked by `ReceiveMessage` and `SendMessage` method when they fail. This method calls method `Write(e, log)` of `Logger` to write the log.
  Parameters:
  - `e` is the exception to write.
  - `log` is the additional note.

- `private void NotifyError()`
  Executes the callback function with `null` parameter. It's called by `ReceiveMessage` and `SendMessage` method when an error happens in processing the message. If the target object of the callback function is an `IModelParsingReceiver` object then `ParsedMessage` property will be set.

- `private void Parse()`
  Parses the received message from the remote host (referred by `ReceivedMessage` property) to be an appropriate model object. It's called by `ReceiveMessage` and `SendMessage` method.

- `private byte[] Send(IParsedMessage pMsg)`
  Accepts a model object to be converted to ISO 8583 message then sends this message to remote host and return the reply message (ISO 8583 formatted) from the remote host. The model object is taken from `SentModel` property. To send the message, the message will be passed to the `IMessageStream` object held by `MessageStream` property.

Parameters:

- `pMsg` is typically `MessageParser` object which parsed the incoming message before sending a new message as a response. It is `null` if sending a request message.

- `private byte[] Receive()`
  Tries to read the incoming message from `MessageStream`. It is called by `ReceiveMessage` and `SendMessage` method when they notice that the incoming message has not given by the running message listener or message client.

- `private Object ProcessModel()`
  Processes the model object obtained from the received message (as a request or response) from remote host, in which the business process is executed. It will execute the callback function. To find the callback function, it will check `Callback` property. If it is `null`, it will check `ProcessModel` property of `MessageParser` object that was created to parse the incoming message. If it is still `null` then this method will return `null` as a reply object. If it is not, the returned object by the callback function will be returned by this method. If the target object of the callback function is an `IModelParsingReceiver` object then `ParsedMessage` property will be set.

- `private void RegisterMe()`
  Maintains the reference of this object, so that it can be retrieved via `GetThread` method. This method is called at the beginning of method `SendMessage` and `ReceiveMessage`. One of purposes of why needs to retrieve the reference is to cancel the process.

- `private void UnregisterMe()`
  Tells the message processor to not maintain the reference of this object anymore, so that it cannot be retrieved anymore via `GetThread` method. This method is called at the end of method `SendMessage` and `ReceiveMessage` method.

- `public static MessageProcessorWorker GetThread(String id)`
  Gets the reference of a `MessageProcessorWorker` instance object, identified by its id. Note, the object reference can be retrieved as long as it is executing, that is after invoking `RegisterMe` method and before calling `UnregisterMe` method.
  Parameters:
  - `id` is the id of the retrieved object. It is the same as `Id` property.

Interface `ICustomMessageProcessor`

To be reviewed. There may be a message that cannot be processed by generic process as done by this message processor. Then it will need a custom processor. The object that acts as the custom processor must implement this interface. The processor will notice when the message needs a custom processor from the configuration. In this case, when the `Send` or `Receive` method is invoked, the process will be delegated to the custom processor. The `Send` and `Receive` are executed asynchronously. There will be many processes will call these methods.

```
namespace Free.iso8583
public interface ICustomMessageProcessor
```

- `IProcessedMessage Send(Object model, IMessageStream stream, MessageCallback callback)`
- `IProcessedMessage Send(Object model, IMessageStream stream)`
- `IProcessedMessage Receive(byte[] message, IMessageStream stream, MessageCallback callback)`
- `IProcessedMessage Receive(byte[] message, MessageCallback callback)`

Message Parser

This part is responsible to parse the incoming ISO 8583 message. By this message parser, the message will be divided to be message type header, bitmap and other various headers and, of course, its content. The message content will be converted to an appropriate model object. The message parser will try to understand the configuration to make the parsing process done. The message parser will be created by message processor when there is an incoming message.

Interface `IParsedMessage`

It defines interfaces from those we can retrieve the result of the parsing process previously done by the message parser. This interface is typically implemented by class `MessageParser`. For further explanation, see interface `IModelParsingReceiver` about the use of this interface.

```
namespace Free.iso8583
public interface IParsedMessage
```

- `ParsedMessageContainer ParsedMessage { get; }`
  Returns a message container object storing the message that has been parsed to separated headers and message fields. See about Message Container.
- `Object Model { get; }`
  The message content that has been converted to the corresponding model object.
- `MessageBitMapCollection BitMap { get; }`
  All bitmaps contained by this message.
- `MessageTypeHeader MessageType { get; }`
  The message type header of this message.
- `byte[] GetHeader(String name)`
  Gets the bytes of a header identified by its name. It should return `null` if the name of header is not found.
  Parameters:
  - `name` is the name of the header that we want to retrieve. This name is defined in

XML Configuration. See the name attribute of element `header`.

- `byte[] AllBytes { get; }`
  Whole bytes of the parsed message. It is useful when we want to know all bytes of the incoming message.

Class `MessageParser`

It is the core object which does parsing process. This class is created by the message processor as the follow-up of a request for sending/receiving an ISO 8583 message. This class takes a part to parse the bytes of the ISO 8583 message either as a response message (in sending process) or as a request message (in receiving process). Specifically, this object is created when the method `SendMessage` or `ReceiveMessage` of class `MessageProcessorWorker` is invoked.

```
namespace Free.iso8583
public class MessageParser : IParsedMessage
```

- `internal MessageParser(byte[] bytes, MessageProcessorWorker workerThread)`
  The instance object of this class must be created with an ISO 8583 message as the parameter. The ISO 8583 message is packed in an array of bytes.
  Parameters:
  - `bytes` is the ISO 8583 message that will be parsed.
  - `workerThread` is the `MessageProcessorWorker` object that creates this `MessageParser` object. The parser needs the reference held by `MessageToModelConfig` property to avoid rechecking the configuration.

- `public MessageParser(byte[] bytes)`
  Invokes the above constructor with null value for `workerThread` parameter.

- `public void Parse()`
  After the object of this class is created, the consumer object should invoke this method. This method is to instruct the object to start parsing the message. Specifically, this method will call method `GetMessageToModelConfig`, `CheckMessageLength`, `ParseHeaders`, `ParseFields` and `ConvertToModel` consecutively. In the beginning of process, it sets the position of reading to the first byte.

- `private MessageToModelConfig GetMessageToModelConfig()`
  This method will asses the being parsed message to get the correct configuration so that the parser can parse the message. Simply, this method will delegate the task to the method `GetQulifiedMessageToModel` of class `MessageConfigs`. It will throw an exception if the matching configuration is not found. For further information about the configuration, see the section XML Configuration and Configuration Objects.

- `private void CheckToken(int nextFieldLength, IMessageHeaderConfig`

`curHeader,` `MessageFieldConfig` `curField)`
Before each piece (a header or a content field of the message) is read, this parser will check the length of this piece and compare it to the count of unread bytes. If the piece's length is greater then it will raise an exception. This method is charged to do that task, so it will be invoked every time a piece will be read (specifically, it will be invoked repeatedly by method `ParseHeaders` and `ParseFields`). It uses the position of reading to get the count of unread bytes. This checking process must be done because some pieces have a variable length which will be known after being calculated at parsing process.
Parameters:

- `nextFieldLength` is the length of the piece that will be read. If it has a variable length, the length is calculated before invoking this method.
- `curHeader` is the configuration object for the header that will be read. It will be `null` if the piece is a message content field.
- `curField` is the configuration object for the message field that will be read. It will be `null` if the piece is a message header.

See also Configuration Objects.

- `private void CheckMessageLength()`
  An ISO 8583 message must have the header which records the length of the message. This method will check the value of this header and compare it with the actual length of the message. If the value is different, this method will throw an exception.

- `private void ParseHeaders()`
  As its names, this method will parse the header part of the message. It knows what headers contained by the message from the configuration. The configuration is obtained by method `GetMessageToModelConfig` that should be previously invoked. This method will keep the BitMap header to be returned by property `BitMap`, message type header to be returned by property `MessageType` and finally keep all other named headers to be returned by method `GetHeader`. Each time, it reads a header, it will advance the position of reading by the length of that header. Each header will be added to the message container object to be returned by property `ParsedMessage`.

- `private void ParseFields()`
  Parses the content part of the message, separates field by field. It knows what fields contained by the message from the configuration. The configuration is obtained by method `GetMessageToModelConfig` that should be previously invoked. It also uses the BitMap header to check whether the field exists or not. The BitMap header should have been parsed previously by method `ParseHeaders`. Each field will be added to the message container object to be returned by property `ParsedMessage`. Each time, it reads a field, it will advance the position of reading by the length of that field. If there is a field whose variable length, this method is responsible to calculate the actual length. It is easily done by calling method `GetBytesLengthFromActualLength` of each corresponding message field container object (see section Message Container).

- `private void ConvertToModel()`

Tries to convert the message content to an appropriate model object based on the configuration (see section Element `message-to-model`). The message content has been parsed and included into the message container by method `ParseFields`. It will iterate each model property configuration (see section Element `property`) and take the corresponding message field container to get the property value. It will call `ParseTlv` if the field is a TLV field (see section Element `tlv`) and will call `ParseBitContent` if the field is a BitContent field (see section Element `BitContent`). If the model property is set to get value from a `delegate`, the `delegate` will be called with parameter of the message field bytes. After the value is gotten, if it is `null` then the property value will be set to `null`. If it's not, it will be converted to an assignable value before assigned to the model property by utilizing method `GetAssignableValue`. Failing to do so will raise an exception.

- `private Object ParseBitContent(ModelPropertyConfig cfg, MessageField fld)`
  This method will delegate the task to another overloaded method by using parameter values taken from property `BitContent` of `cfg` and property `BytesValue` of `fld`.

- `private Object ParseBitContent(BitContentConfig bcc, byte[] bytesValue)`
  Parses a BitContent field that is a field that can be broken to be some sub fields. For further information about BitContent field, see section Element `BitContent` and Element `field` in XML Configuration. BitContent field has its own model object that is set via attribute `class` of element `BitContent`. In turn, each property of this model object is mapped to a sub field in the BitContent. It will iterate each object property based on the configuration to get the property value. The order of property will the same as defined in the configuration. A sub field can be a TLV field, so it will invoke `ParseTlv` to get the value. If the field is not a TLV field then it will try to interpret the bytes array which belongs to the sub field as a string (an ASCII array). If the bytes array cannot be interpreted as a string because of a non regular character inside the array, it will try to convert the bytes array to an assignable value for the property. If it fails to find the value then an exception will be thrown. A TLV sub field has a variable length and the other one has a fixed length (defined by attribute `length` of element `field`). The length of bytes array which belongs to this BitContent field does not necessarily have to the same as needed by all model object property. Therefore, the last assigned property possibly has sorter length than that defined in the configuration. Not assigned properties will remain `null`. This method returns the model object associated to the parsed BitContent field.

- `private Object ParseTlv(Type propType, TlvConfig cfg, String tagName, int lengthBytes, Type tlvType, byte[] bytesValue, String locMsg, bool isAlwaysOneTag, ref int tlvLength)`
  Parses a TLV field and returns the value yielded by the parsing process. As BitContent field, a TLV field is divided to be one or some sub fields. Each sub field has 2 headers, the first is named tag and the second is the length header. For further information about TLV field, see section Element `tlv` and Element `tag` in XML Configuration. A sub field can be a BitContent field, so it will call `ParseBitContent` to get the value.

Parameters:

- `propType` is the type of the model object property that is be mapped to this TLV field. This method will try to return an assignable value for this type. If it fails to find an assignable value, it will throw an exception.
- `cfg` is the configuration object corresponding to the TLV field that will be parsed.
- `tagName` is the tag name. It's usable when there is only one tag (one sub field). The method uses this parameter only for getting the length of tag if `cfg` is `null` or the tag name is not defined in `cfg`.
- `lengthBytes` is how many byte after tag which records the length of the sub field content. It's used when `cfg` is `null` or the length of bytes is not defined in `cfg`.
- `tlvType` is the type of the model object associated to this TLV field. The type is set via attribute `class` of element `tlv`. If it's `null` then the TLV field doesn't have an associated model object. If this TLV field has a model object and this model object is assignable to the assigned property then this model object will be returned. If those conditions are not met then if there is only one sub field and this sub field's value is assignable then this value will be returned. If those conditions are not still satisfied then it will create a dictionary and return it if it's assignable. The dictionary is keyed by sub field tag names. If still fails, this method will create an object of the same type as assigned property type. Then, each sub field value will be assigned to the object's property whose the same name as the sub field tag name. If this process fails, an exception will be thrown.
- `bytesValue` is the bytes array of the message field which belongs to the TLV field. Note, when invoked by method `ParseBitContent`, the length of the array will possibly exceed the actual length of the TLV sub field. It is because BitContent field doesn't have a note about the length of a TLV sub field inside it. The actual length will be returned via parameter `tlvLength`.
- `locMsg` is the message for the exception which possibly happens when executing this method.
- `isAlwaysOneTag` defines whether the TLV field always has one sub field or not. It's `true` if it's invoked from `ParseBitContent` and `false` if invoked from `ConvertToModel`.
- `tlvLength` returns the actual length of the TLV field.

- `private void CheckTlvToken(int token, int length, byte[] bytes, String locMsg)`
  It's similar to method `CheckToken` but for parsing a TLV field (specifically, invoked by method `ParseTlv`). When iterating the sub fields, this method is invoked three times for each sub field, those are for tag header, length header and sub field content.

- `public ParsedMessageContainer ParsedMessage { get; }`
  Implements `IParsedMessage` interface.

- `public object Model { get; }`

Implements `IParsedMessage` interface.

- `public MessageBitMapCollection BitMap { get; }`
  Implements `IParsedMessage` interface.

- `public MessageTypeHeader MessageType { get; }`
  Implements `IParsedMessage` interface.

- `public byte[] GetHeader(string name)`
  Implements `IParsedMessage` interface.

- `public byte[] AllBytes { get; }`
  Implements `IParsedMessage` interface.

- `public Delegate ProcessModel { get; }`
  It's the `delegate` object defined by attribute `delegate` of element `message-to-model`. It's where the business process is executed. It's used by method `ProcessModel` of `MessageProcessorWorker` object. It's the same as property `ProcessModel` of `MessageToModelConfig`.

- `internal ModelConfig ModelCfg { get; }`
  It's the model configuration object. It's set when executing method `Parse` which is taken from the returned object of method `GetMessageToModelConfig`.

Message Compiler

This part is responsible to compile the outgoing ISO 8583 message, that is, to convert a model object to be an appropriate message. The message compiler will try to understand the configuration to make the compiling process done. Actually, this message compiler only creates a message container object (which is an instance object of class `CompiledMessageContainer`). To get the complete bytes of the message, use method `GetAllBytes` of that message container object. The message compiler will be created by message processor when sending a message.

Interface `ICompiledMessage`

It defines interfaces from those we can retrieve the result of the compiling process previously done by the message compiler. This interface is typically implemented by class `MessageCompiler`.

```
namespace Free.iso8583
public interface ICompiledMessage
```

- `CompiledMessageContainer CompiledMessage { get; }`
  Returns a message container object storing the message that has been compiled (the translation result of the `Model` object). See about Message Container.

- `Object Model { get; }`
  The model object from which the message is compiled.

Class `MessageCompiler`

It is the core object which does compiling process. This class is created by the message processor as the follow-up of a request for sending/receiving an ISO 8583 message. This class takes a part to compile the bytes of the ISO 8583 message either as a response message (in receiving process) or as a request message (in sending process). Specifically, this object is created when the method `ReceiveMessage` or `SendMessage` of class `MessageProcessorWorker` is invoked.

```
namespace Free.iso8583
public class MessageCompiler : ICompiledMessage
```

- `public MessageCompiler(Object model, IParsedMessage reqMsg)`
  When compiling a message for sending a response, this constructor should be used with the second parameter is not `null`.
  Parameters:
  - `model` is the model object from which the message is compiled.
  - `reqMsg` is the request message that has been parsed. If it is to send a response, this request message is useful to retrieve some message field values that must be sent back to the client.
- `public MessageCompiler(Object model) : this(model, null)`
  This constructor should be used when sending a request. No request message provided to compile the new message.
  Parameters:
  - `model` is the model object from which the message is compiled.
- `public void Compile()`
  After the object of this class is created, the consumer object should invoke this method to instruct the object to start compiling the message. Specifically, this method will fetch the configuration object for this message, then call `CompileHeaders` and `CompileFields` consecutively. It knows which configuration must be fetched from the type of model object and then it inquires `ClassToModels` property of class `MessageConfigs`. If the matching configuration is not found, an exception will be thrown.
- `private void CompileHeaders()`
  Compiles the header part of the message. It will iterate the headers based on the configuration to create the header container (an `MessageElement` object). Each header container is inserted to the message container object. The order of headers will be the same as the order of their writing in the configuration (a header written first in the configuration will be the first header in the message after the message length header).
- `private void CompileFields()`

Compiles the content part of the message. It will iterate model properties configuration to get bytes value for each message field. From the configuration, it knows which message field container object that must be created. If the field value should be taken from the request (in the case the compiled message is a response message), it will inquire the corresponding field in the request message. It will take the value of that request message field if the corresponding property of the model object is `null`. If the property value is not `null`, the field value is still compiled from that property value. If the field is TLV field (see section Element `tlv`) then it will call `CompileTlv` to get the field value. It will call `CompileBitContent` if the field is a BitContent (see section Element `BitContent`). If the configuration defines a `delegate` for the field (see attribute `delegate` of element `bit`) then the `delegate` will be invoked. If those conditions are not satisfied, it will try to find the matching method `SetValue` of the corresponding message field container object. If still not satisfied, it is considered that the field is absent (in bitmap header, the corresponding bit is off). Also, the message field is considered to be absent if the property value is `null` and by the configuration, it is not substituted by the corresponding request message field if it's response message.
The gotten value from `CompileTlv`, `CompileBitContent` or the `delegate`, which is an array of bytes, its length will be checked. If it's different from defined in the configuration (if not variable length) then it will raise an exception.
Each message field will be inserted to the message container object. The order of fields is defined by attribute `seq` of element `bit`.
See also, section Configuration Objects and Message Container.

- `private byte[] CompileBitContent(BitContentConfig cfg, Object obj)`
  Compiles a BitContent field that is the field that can be broken into some sub fields (see section Element `BitContent`). It returns the bytes value for this BitContent field. A sub field can be a TLV field, so it will call `CompileTlv`.
  Parameters:
    - `cfg` is the configuration object for this BitContent field. For further information about configuration object, see section Configuration Objects.
    - `obj` is the property value associated to this BitContent field. This value must be an instance object of type which is determined by the configuration (parameter `cfg`). If not, an exception will happen.
- `private byte[] CompileTlv(TlvConfig cfg, String tagName, int lengthBytes, Object obj, String locMsg)`
  Compiles a TLV field and returns the bytes value of the field. As BitContent field, a TLV field consists of one or some sub fields. Each sub field has 2 headers, the first is named tag and the second is the length header. For further information about TLV field, see section Element `tlv` and Element `tag` in XML Configuration. A sub field can be a BitContent field, so it will call `CompileBitContent` to get the field value.
  Parameters:
    - `cfg` is the configuration object for this TLV field. See section Configuration Objects for further information.
    - `tagName` is the tag name for the sub field. It's usable if there is only one sub field.

It's used if `cfg` is `null` or the tag name is not defined in `cfg`.

  - `lengthBytes` is how many bytes after the tag which records the length of the sub field content. It's used when `cfg` is `null` or the length of bytes is not defined in `cfg`.
  - `obj` is the value of the model property associated to this TLV field.
  - `locMsg` is the message for the exception which possibly happens when executing this method.

- `public CompiledMessageContainer CompiledMessage { get; }`
  Implements `ICompiledMessage` interface.

- `public object Model { get; }`
  Implements `ICompiledMessage` interface.

- `internal ModelConfig ModelCfg { get; }`
  It's the model configuration object. It is set when executing method `Compile`.

Message Container

Message container is the intermediate object to make easy when converting message bytes to a model object and vice versa. It unifies interfaces among different headers and fields (About the kind of fields, consult attribute `type` of element `bit)`. The important thing is concerning different kind of fields which each must be treated differently when getting their value. Another concern is how to make this field value assignable to the mapped model property. Another one is how to convert the value of the mapped model property to be the correct array of bytes for the corresponding message field. There are three groups of classes among message container objects. Those are header classes, field classes and message classes. The last group maintains separated header and field containers in the right order.

Class `MessageElement`

It is the base class for all message element container classes, either as a header or a data element (message field). There are two interfaces that should be had by an element container. Those are the length and the value (an array of bytes).

```
namespace Free.iso8583
public abstract class MessageElement
```

- `public virtual int Length { get; internal set; }`
  It is the length of this element, that is how long data inside this container. It's not necessarily the same as the count of bytes contained by this element. It depends on the data type. For example, if type is `B` then the length will be eight times of the count of bytes.

- `internal virtual byte[] BytesValue { get; }`

Array of bytes inside this element. In the other words, it will be include into the raw message.

- `internal virtual void SetValue(byte[] value)`
  Sets a new array of bytes for this element container. The new array should be returned by property `BytesValue`.
  Parameters:
    - `value` is new bytes for this container.

- `public byte[] RoBytesValue { get; }`
  A read-only version of property `BytesValue` that can be accessed by *outside world*. Read-only means that any change to the array returned by this property will not change the array contained by this element. Technically, it's a new copy of `BytesValue`.

Header Container Classes

These classes are container for header entity. The header part of message consists of some entities. It excludes the message length header that must exist as the first header. It has been defined three container classes. The first is for BitMap header, the second is for message type header and the last one is for the other headers.

Class `MessageBitMap`

It is the container of the BitMap header (primary bitmap) or the message field whose type "`BitMap`" (secondary and tertiary bitmap). Bitmap determines the presence of fields in the message. It is related to element `bitmap` or element `bit` whose type is "`BitMap`" in XML Configuration. This class type is auto cast to the type of `byte[]`.

```
namespace Free.iso8583
public class MessageBitMap : MessageField
```

- `public override int Length { get; internal set; }`
  Overrides property `Length` of parent class. When setting this property, it will resize `BytesValue` value (bitmap bytes array). If the length is less than before then the array will truncated. If greater than before then the new elements in array will set to 0.

- `public int StartBit { get; internal set; }`
  Ordinal number of message field corresponding to the first bit in this bitmap. Ordinal number for the other bits will follow as appropriate. For primary bitmap, it should be 1. For the other bitmaps, it should be greater than 1. It's set by method `Add` of `MessageBitMapCollection`.

- `public int FieldSeq { get; internal set; }`
  If this bitmap is a message field, it's the ordinal number of this message field. It's 0 if

primary bitmap.

- `public override int GetBytesLengthFromActualLength(int actualLength)`
  Overrides method `GetBytesLengthFromActualLength` of parent class.

- `internal void SetValue(byte[] value)`
  Overrides method `SetValue` of parent class. It will replace the bitmap bytes contained by this object.
  Parameters:
    - `value` is new bitmap bytes.

- `internal void SetBit(int bitPos, bool isOn = true)`
  Sets a bit on/off.
  Parameters:
    - `bitPos` is the ordinal number of bit to be set. This method does nothing if the ordinal number is not covered by this bitmap.
    - `isOn` specifies whether the bit is on or off. It is `true` if on.

- `public static void SetBit(int bitPos, byte[] bitMap, int startBit = 1, bool isOn = true)`
  Sets a bit on/off in the specified bitmap.
  Parameters:
    - `bitPos` is the ordinal number of bit to be set. This method does nothing if the ordinal number is not covered by the specified bitmap.
    - `bitmap` is the specified bitmap where the bit to be set.
    - `startBit` is the ordinal number of the first bit. The ordinal number for the other bits will follow as appropriate. This method does nothing if `startBit` is not multiple of 8 plus 1.
    - `isOn` specifies whether the bit is on or off. It is `true` if on.

- `internal void SetBitOn(ICollection<int> bitOnList)`
  Sets some bits in this bitmap on and the others off. It does not necessarily set the presence of message fields when compiling a message. Setting the presence of message fields is the message compiler's responsibility. Instead, the parameter for invocation of this method depends on the presence of message fields.
  Parameters:
    - `bitOnList` is a list of bit ordinal numbers that must be set on. The other bits which their ordinal number are not listed must be set off. The bit ordinal number starts from `StartBit`. If there is an invalid position (< `StartBit` or > maximum ordinal number) then it will be ignored.

- `public void SetBitOn(ICollection<int> bitOnList, byte[] bitMap, int startBit = 1)`
  Sets some bits in the specified bitmap on and the others off.
  Parameters:
    - `bitOnList` is a list of bit ordinal numbers that must be set on. The other bits

which their ordinal numbers are not listed must be set off. The bit ordinal number starts from value specified by parameter `startBit`. If there is an invalid position (< `startBit` or > maximum ordinal number) then it will be ignored.

  - `bitmap` is the specified bitmap where the bits will be set.
  - `startBit` is the ordinal number for the first bit.

- `public bool IsBitOn(int bitPos)`
  Checks if a bit is on or off. Returns `true` if it is on.
  Parameters:
  - `bitPos` is the ordinal number of bit that will be checked. If the ordinal number is invalid (< `StartBit` or > maximum ordinal number) then this method will return `false`.
- `public static bool IsBitOn(int bitPos, byte[] bitMap, int startBit = 1)`
  The same as above method but not using internal bitmap array. The bitmap array is passed as the second parameter.
  Parameters:
  - `bitPos` is the ordinal number of bit that will be checked. If the ordinal number is invalid (< `startBit` or > maximum ordinal number) then this method will return `false`.
  - `bitMap` is the bitmap array in which it will check the on/off bit.
  - `startBit` is ordinal number for the first bit.
- `public bool IsOutOfRange(int bitPos)`
  Checks an ordinal number whether out of range for this bitmap or not. If out of range (< `StartBit` or > maximum ordinal number), this method return `true`.
  Parameters:
  - `bitPos` is the ordinal number that will be checked.
- `public bool IsNull { get; }`
  Returns `true` if all bits in this bitmap are off.

Class `MessageBitMapCollection`

It contains all BitMap elements in the message (primary, secondary and tertiary). This class is to make easy when setting a bit or checking a bit whether it's on or off. All bitmaps are considered as one bitmap. It doesn't need to set/check individual bitmap. This class type is auto cast to the type of `byte[]`. Message parser and message compiler will use this object to check/set bitmap bits.

```
namespace Free.iso8583
public class MessageBitMapCollection
```

- `internal void Add(MessageBitMap bitMap)`

Adds a bitmap to this collection. It also sets property `StartBit` of the added bitmap.
Parameters:

- `bitMap` is the added bitmap.

- `internal void SetBit(int bitPos, bool isOn = true)`
  Sets a bit on/off.
  Parameters:
  - `bitPos` is the ordinal number of bit to be set. This method does nothing if the ordinal number is out of range.
  - `isOn` specifies whether the bit on or off. It is `true` if on.
- `internal void SetBitOn(ICollection<int> bitOnList)`
  Sets some bits in the bitmaps in this collection to be on and the others off.
  Parameters:
  - `bitOnList` is a list of bit ordinal numbers that must be set on. The other bits which their ordinal numbers are not listed must be set off. The bit ordinal number starts from 1. If there is an invalid position (< 1 or > maximum position) then it will be ignored.
- `public bool IsBitOn(int bitPos)`
  Checks if a bit is on or off. Returns `true` if it is on.
  Parameters:
  - `bitPos` is the ordinal number of bit that will be checked. If the ordinal number is invalid (< 1 or > maximum position) then this method will return `false`.
- `public List<MessageBitMap> ToList()`
  Returns all bitmaps contained by this collection as a list.
- `public byte[] BytesValue { get; }`
  Concatenates all bitmaps contained by this collection and returns it as an array.
- `public int Length { get; internal set; }`
  Sum of all length of bitmaps contained by this collection.

Class `MessageTypeHeader`

It is the container class for message type header (message type indicator). It relates to element `message-type` in XML Configuration. This class type is auto cast to the type of `byte[]`.

```
namespace Free.iso8583
public class MessageTypeHeader : MessageElement
```

- `public MessageTypeHeader(byte[] bytes)`
  This constructor determines that the value of this header must be provided when creating an object of this class.
  Parameters:

  - `bytes` is the value of the message type header. It must be defined in the configuration. See attribute `value` of element `message-type`.

- `public override int Length { get; internal set; }`
  Overrides property `Length` of parent class. It overrides the setting part by doing nothing. In the other words, it is read only. The value is taken from the length of bytes array passed to constructor.

- `internal override void SetValue(byte[] value)`
  Overrides method `SetValue` of parent class. This method does nothing. The value cannot be changed. The value is taken from the parameter passed to constructor.

- `public String StringValue { get; private set; }`
  The hexadecimal string of `BytesValue`.

- `public MessageVersion MessageVersion { get; private set; }`
  Indicates the message version.

- `public MessageClass MessageClass { get; private set; }`
  Indicates the message class.

- `public MessageFunction MessageFunction { get; private set; }`
  Indicates the message function.

- `public MessageOrigin MessageOrigin { get; private set; }`
  Indicates the message origin.

Enum `MessageVersion`, `MessageClass`, `MessageFunction` and `MessageOrigin`

Message type header (message type indicator) consists of four digits. Each digit is in a nibble. So, the count of bytes is two. The first digit indicates the message version. Three following digits consecutively indicate message class, message function and message origin. The enum types below enumerate the possible value for those four components of message type.

```
namespace Free.iso8583
public enum MessageVersion
```

- `v1987 = 0`
  ISO 8583-1:1987 version.

- `v1993 = 1`
  ISO 8583-2:1993 version.

- `v2003 = 2`
  ISO 8583-3:2003 version.

- Reseved3 = 3
- Reserved4 = 4
- Reserved5 = 5
- Reserved6 = 6
- Reserved7 = 7
- NationalUse = 8
- PrivateUse = 9

```
namespace Free.iso8583
public enum MessageClass
```

- Reserved0 = 0
- Authorization = 1
- Financial = 2
- FileAction = 3
- Reversal = 4
- Reconciliation = 5
- Administrative = 6
- FeeCollection = 7
- NetworkManagement = 8
- Reserved9 = 9

```
namespace Free.iso8583
public enum MessageFunction
```

- Request = 0
- RequestResponse = 1

- `Advice = 2`
- `AdviceResponse = 3`
- `Notification = 4`
- `NotificationAcknowledgment = 5`
- `Instruction = 6`
- `InstructionAcknowledgment = 7`
- `ResponseAcknowledgment = 8`
- `NegativeAcknowledgment = 9`

```
namespace Free.iso8583
public enum MessageOrigin
```

- `Acquirer = 0`
- `AcquirerRepeat = 1`
- `Issuer = 2`
- `IssuerRepeat = 3`
- `Other = 4`
- `OtherRepeat = 5`
- `Reserved6 = 6`
- `Reserved7 = 7`
- `Reserved8 = 8`
- `Reserved9 = 9`

Class `MessageHeader`

It is the container class for the headers other than message type and bitmap. It relates to element `header` in XML Configuration.

```
namespace Free.iso8583
```

```
public class MessageHeader : MessageElement
```

- `public override int Length { get; internal set; }`
  Overrides property `Length` of parent class. When setting this property, if the value is less than the length of `BytesValue` then `BytesValue` will be truncated. If it is greater than the length then `BytesValue` array will be expanded with the added elements are set to 0.

- `internal override byte[] BytesValue { get; }`
  Overrides property `BytesValue` of parent class. It cannot be accessed if value is still `null` (has not been defined yet). If `delegate` is defined, it will call the `delegate` function referred by property `GetFieldBytesFunct` to get the bytes value.

- `internal GetHeaderBytes GetFieldBytesFunc { get; internal set; }`
  The `delegate` object defined by attribute `delegate` of element `header` in the XML configuration.

Field Container Classes

Some container classes have been defined to contain a message field. It is not only for the value of the field but also some dispositions of that field, such as how many digits after decimal point for a numeric field, the count of bytes should be owned by the field etc. There are properties to get field's value as a bytes array, numeric or string. Of course, the returned value depends on the nature of that field. There are also some overloaded `SetValue` methods which accepts parameter as bytes array, numeric or string to set the field's value.

Class `MessageField`

It suits for container of a field whose type "`B`" (see attribute `type` of element `bit`). But because it's the base class for all message field containers and all message field containers will be recognized as an instance object of this class when parsing and compiling, this class defines members needed by all message field type for general operation.

```
namespace Free.iso8583
public class MessageField : MessageElement
```

- `public int Length { get; internal set; }`
  The count of digits/character. It's not the count of bytes because for a nibble field, one byte has two digits. Especially for field whose type "`B`", one byte has a length of 8 because one byte has 8 bits. It is negative if the field has a variable length.

- `public bool VarLength { get; internal set; }`
  `true` if the field has a variable length. If it is `true` the property `Length` should have negative value.

- `public int BytesLength { get; }`
  The count of bytes which stores the value of this field.

- `internal bool IsOdd { get; set; }`
  It's useful for a field which consists of the nibble (four bits) entities, such as type "`N`" and "`NS`" (see attribute `type` of element `bit`). This property is `true` if the count of nibble entities is odd. It causes a half (4 bits) of the first or the last byte in the bytes array of this field will be unused.

- `public static int GetBytesLengthFromActualLength(int actualLength, int divider)`
  Returns the count of bytes for a field if it has a length of `actualLength`.
  Parameters:
  - `actualLength` is the count of characters/digits. If this parameter is taken from property `Length`, the result should be the same as the value of property `BytesLength`.
  - `divider` is how many data contained in a byte. It should be 8 for type "`B`", 2 for type "`N`" and 1 for type "`ANS`".

- `public virtual int GetBytesLengthFromActualLength(int actualLength)`
  Returns the count of bytes for this field if it has a length of `actualLength`.
  Parameters:
  - `actualLength` is the count of characters/digits. If this parameter is taken from property `Length`, the result should be the same as the value of property `BytesLength`.

- `public int FracDigits { get; internal set; }`
  It's meaningful if this field is numeric (type "`N`" or "`AN`", see attribute `type` of element `bit`). It tells how many digits after decimal point. This property is needed because the decimal point is not included in the message. It will throw an exception if this property is set to a value less than zero or greater than the number of digits owned by this field.

- `internal override void SetValue(byte[] value)`
  Sets the field value using a bytes value. Before invoking this method, if `VarLength` is `false` property `Length` must be set first and its value must be same as the length of parameter `value`. If not, it will throw an exception.
  Parameters:
  - `value` is the new value for this field.

- `internal virtual void SetValue(String value)`
  always throws an exception, not implemented yet. A descendant class should implement this method if possible.
  Parameters:
  - `value` is the new value for this field.

- `internal virtual void SetValue(decimal value)`

always throws an exception, not implemented yet. A descendant class should implement this method if possible.
Parameters:

  - `value` is the new value for this field.

- `internal virtual void SetValue(int value)`
  calls `SetValue(decimal value)` with parameter `value` casted to `decimal`. Because of that, for this class, this method always throws an exception.
  Parameters:
  - `value` is the new value for this field.

- `internal byte[] BytesValue { get; }`
  Returns the value of this field as an array of bytes. Inherited from `MessageElement`.

- `public byte[] RoBytesValue { get; }`
  A read-only version of property `BytesValue` that can be accessed by *outside world*. Inherited from `MessageElement`. Even if property `BytesValue` is defined as read-only (no setting part), but still an element inside this array can be changed.

- `public String StringValue { get; }`
  Returns the value of this field as a string.

- `public decimal? DecimalValue { get; }`
  Returns the value of this field as a decimal. Returns `null` if this field is not numeric.

- `public int? IntValue { get; }`
  Returns the value of this field as an integer. Returns `null` if this field is not numeric.

Class `NMessageField`

It is suitable for the container of a field whose type “`N`” (see attribute `type` of element `bit`). After the value is set, property `DecimalValue` and `IntValue` will return a not `null` value because this field is numeric.

```
namespace Free.iso8583
public class NMessageField : MessageField
```

- `public override int GetBytesLengthFromActualLength(int actualLength)`
  Overrides the parent method. Because it’s for a field containing the nibble entities, one byte has a length of 2.
  Parameters:
  - `actualLength` is the count of characters/digits. If this parameter is taken from property `Length`, the result should be the same as the value of property `BytesLength`.

- `internal override void SetValue(byte[] value)`
  Overrides the parent method. Sets the field value using a bytes value. Before invoking this method, property `Length` must be set first and its value may not be less than the count of nibble in parameter `value`. If not, it will throw an exception. If the count of nibble is less than property `Length` value, it will be padded by '0' at the left. It also throws an exception if there is an invalid character (other than '0' to '9').
  Parameters:
    - `value` is the new value for this field.

- `internal override void SetValue(String value)`
  Overrides the parent method. It converts the string parameter to be an array of bytes and calls `SetValue(byte[] value)`.
  Parameters:
    - `value` is the new value for this field.

- `internal override void SetValue(decimal value)`
  Overrides the parent method. It converts the decimal parameter to be an array of bytes and calls `SetValue(byte[] value)`. Before invoking this method, property `FracDigits` should be set first, because it influences the value of `BytesValue`.
  Parameters:
    - `value` is the new value for this field.

Class **`NsMessageField`**

It should be the container of a field whose type "`NS`" (see attribute `type` of element `bit`).

```
namespace Free.iso8583
public class NsMessageField : NMessageField
```

- `internal override void SetValue(byte[] value)`
  Overrides the parent method. Sets the field value using a bytes value. Before invoking this method, property `Length` must be set first and its value may not be less than the count of nibble in parameter `value`. If not, it will throw an exception. If the count of nibble is less than property `Length` value, it will be padded by '0' at the left. There is no invalid character.
  Parameters:
    - `value` is the new value for this field.

Class **`AnMessageField`**

It should be the container of a field whose type "`AN`" (see attribute `type` of element `bit`). After the value is set, property `DecimalValue` and `IntValue` will return a not `null` value because this field is numeric.

```
namespace Free.iso8583
```

```
public class AnMessageField : MessageField
```

- `public override int GetBytesLengthFromActualLength(int actualLength)`
  Overrides the parent method. Because it contains ASCII characters, one byte has a length of 1.
  Parameters:
    - `actualLength` is the count of characters/digits. If this parameter is taken from property `Length`, the result should be the same as the value of property `BytesLength`.

- `internal override void SetValue(byte[] value)`
  Overrides the parent method. Sets the field value using a bytes value. Before invoking this method, property `Length` must be set first and its value may not be less than the length of parameter `value`. If not, it will throw an exception. If the length of parameter `value` is less than property `Length` value, it will be padded by space at the right.
  Parameters:
    - `value` is the new value for this field.

- `internal override void SetValue(String value)`
  Overrides the parent method. It converts the string parameter to be an array of bytes and calls `SetValue(byte[] value)`. It will throw an exception if there is an invalid character (other than '0' to '9').
  Parameters:
    - `value` is the new value for this field.

- `internal override void SetValue(decimal value)`
  Overrides the parent method. It converts the decimal parameter to be an array of bytes and calls `SetValue(byte[] value)`. Before invoking this method, property `FracDigits` should be set first, because it influences the value of `BytesValue`.
  Parameters:
    - `value` is the new value for this field.

Class **`AnsMessageField`**

It is intended for the container of a field whose type "`ANS`" (see attribute `type` of element `bit`).

```
namespace Free.iso8583
public class AnsMessageField : MessageField
```

- `public override int GetBytesLengthFromActualLength(int actualLength)`
  Overrides the parent method. Because it contains ASCII characters, one byte has a length of 1.
  Parameters:
    - `actualLength` is the count of characters/digits. If this parameter is taken from

property `Length`, the result should be the same as the value of property `BytesLength`.

- `internal override void SetValue(byte[] value)`
  Overrides the parent method. Sets the field value using a bytes value. Before invoking this method, property `Length` must be set first and its value may not be less than the length of parameter `value`. If not, it will throw an exception. If the length of parameter `value` is less than property `Length` value, it will be padded by space at the right.
  Parameters:
  - `value` is the new value for this field.

- `internal override void SetValue(String value)`
  Overrides the parent method. It converts the string parameter to be an array of bytes and calls `SetValue(byte[] value)`. There is no invalid character.
  Parameters:
  - `value` is the new value for this field.

- `internal override void SetValue(decimal value)`
  Overrides the parent method. It converts the decimal parameter to be an array of bytes and calls `SetValue(byte[] value)`.
  Parameters:
  - `value` is the new value for this field.

Class `NullMessageField`

It is intended for the container of a field whose type "`NULL`" (see attribute `type` of element `bit`). This container always saves `null` value whatever the value set to it.

```
namespace Free.iso8583
public class NullMessageField : MessageField
```

- `public override int GetBytesLengthFromActualLength(int actualLength)`
  Overrides the parent method. It always return 0.

- `private void SetValue()`
  It sets the value contained by this container to `null`. It's called by the other `SetValue` methods.

- `internal override void SetValue(byte[] value)`
  Overrides the parent method. It calls `SetValue` method.

- `internal override void SetValue(String value)`
  Overrides the parent method. It calls `SetValue` method.

- `internal override void SetValue(decimal value)`

Overrides the parent method. It calls `SetValue` method.

- `public override byte[] RoBytesValue { get; }`
  Overrides the parent property. It always returns `null`.

Message Container Classes

The message container maintains all header and field container in the right order which construct the whole message.

Class `ParsedMessageContainer`

It defines the object to maintain the parsed message. The message parser creates an object of this class to save the result of parsing process.

```
namespace Free.iso8583
public class ParsedMessageContainer
```

- `internal IList<MessageElement> Headers { get; }`
  A list that contains the header items of message. The message parser is responsible to insert all needed `MessageElement` objects.

- `internal IDictionary<int, MessageField> Fields { get; }`
  A list that contains the message content fields. The message parser is responsible to insert all needed `MessageField` objects.

- `public IList<MessageElement> RoHeaders { get; }`
  It's the read-only version of property `Headers` that can be accessed by *outside world*. An item cannot be inserted to or removed from this list.

- `public IDictionary<int, MessageField> RoFields { get; }`
  It's the read-only version of property `Fields` that can be accessed by *outside world*. An item cannot be inserted to or removed from this list.

Class `CompiledMessageField`

It is a wrapper of `MessageField` when compiling a message, especially for the fields whose a variable length that must be created their length header. This wrapper stores a header part for the field. Note, this length header is not the part of message header. It records the actual length of the field and is placed before the field value.

```
namespace Free.iso8583
public class CompiledMessageField
```

- `internal byte[] Header { get; set; }`
  The header bytes of this field. The message compiler is responsible to set this property.

- `public byte[] RoHeader { get; }`
  A read-only version of property `Header` that can be accessed by *outside world*.

- `public MessageField Content { get; internal set; }`
  The container which contains the field value.

- `public int Length { get; }`
  The count of bytes that constructs this message field including the `Header` bytes.

- `public int HeaderLength { get; }`
  The length of the `Header` bytes.

Class `CompiledMessageContainer`

It's the container the whole message parts when compiling the message. The message compiler will create an object of this class.

```
namespace Free.iso8583
public class CompiledMessageContainer
```

- `public int LengthHeader { get; internal set; }`
  The count of bytes of the message length header. The message length header bytes is always be placed in the beginning of message. The message compiler is responsible to set this property.

- `public MessageTypeHeader MessageType { get; private set; }`
  Message type header of this message.

- `public MessageBitMapCollection BitMap { get; }`
  All bitmaps in this message.

- `internal IList<MessageElement> Headers { get; }`
  A list containing the header items of this message.

- `internal IDictionary<int, CompiledMessageField> Fields { get; }`
  A list containing the content fields of this message. The message compiler is responsible to fill this list.

- `public IList<MessageElement> RoHeaders { get; }`
  A read-only version of property `Headers` that can be accessed by *outside world*.

- `public IDictionary<int, CompiledMessageField> RoFields { get; }`

A read-only version of property `Fields` that can be accessed by *outside world*.

- `internal void AddHeader(MessageElement hdr)`
  Adds an item to `Headers` list. It should be invoked to add a header item. This method will identify if the added header is bitmap header or message type header.
  Parameters:
    - `hdr` is an header item to be added.

- `internal void AddField(int seq, CompiledMessageField cmf)`
  Adds an item to `Fields` list. It should be invoked to add a field item. This method will identify if the added field is bitmap.
  Parameters:
    - `seq` is the ordinal number of the added field.
    - `cmf` is the added field.

- `public byte[] GetAllBytes()`
  Returns the complete bytes of the message including the message length header. This bytes array is to be sent to the remote host.

Message Listener/Server

Message listener will accept a request message then it asks the message processor to parse the incoming message and execute a business process after the message is converted to a model object. Message listener provided by this library passes `null` for `callback` parameter when calling method `Receive` of `MessageProcessor`. In the other words, it relies on the configuration to get the callback (consult method `ProcessModel` of `MessageProcessorWorker`).

The other listener (created outside this library) may determine which business process to be executed by passing a `MessageCallback` delegate to method `Receive` of `MessageProcessor`. So it may have a configuration to determine which business process to be executed when a message comes. The configuration can be based on processing code and/or message type and/or terminal ID and/or merchant ID.

Class `ListeningEventArgs`

This class defines the data object that will be used as the argument for `StartListeningEvent` event handler. The event is raised when the message server starts listening the request message.

```
namespace Free.iso8583
public class ListeningEventArgs : EventArgs
```

- `public ListeningEventArgs(IPAddress ipAddress, int port, DateTime startTime)`

To create this object, it must be provided all data needed for the event argument.
Parameters:

- `ipAddress` is IP address to which the server listens. It's "0.0.0.0" if the server listens all available addresses.
- `port` is port number to which the server listens.
- `startTime` is the time when the server starts listening.

- `public IPAddress IPAddress { get; }`
  It is IP address to which the server listens. It's "0.0.0.0" if the server listens all available addresses.
- `public int Port { get; }`
  It is port number to which the server listens.
- `public DateTime StartTime { get; }`
  It is the time when the server starts listening.

Class `MessageListenerSync`

This class defines properties that must be synchronized among threads serving the request. Typically, it's intended for property `Sync` of `MessageListener`. This property will be locked by `lock` statement to synchronize among threads.

```
namespace Free.iso8583
internal class MessageListenerSync
```

- `internal int NumOfConn { get; set;}`
  It is the count of concurrent connections which are being processed. When a connection is made by a request, a thread, that is an instance object of `MessageListenerWorker`, will be created to process the request. Therefore, it is also the count of `MessageListenerWorker` objects which are processing the request.
- `internal bool IsAcceptingConn { get; set; }`
  It will be `true` if the message listener is waiting a new connection from a request. Meanwhile, there may be some `MessageListenerWorker` objects which are still processing the request (each object is maintaining a connection). It will be set to `false` if the number of `MessageListenerWorker` objects reaches the maximum number of connections. It will be set to `true` again if a connection closes.

Class `MessageListener`

This class is for main thread which listening the request. When a request comes, it will create a child thread (an instance object of `MessageListenerWorker`) to process the request. Because of asynchronous process, it will not wait the request processing finishes. It will start waiting a new request instead. Therefore, there may be some concurrent processing of requests. Message listener must handle any failure pertaining to the message sending process.

```
namespace Free.iso8583
public class MessageListener
```

- `public MessageListener()`
  Creates an instance without certificate. The communication is intended not to use SSL/TLS if `SslCertificate` is not set later.
- `public MessageListener(String slCertificateFile)`
  Creates an instance along with a SSL certificate. It's used if the used PKCS #12 certificate doesn't use any password.
  Parameters:
  - `slCertificateFile` is the path of certificate file. See `SslCertificate` property for further information.
- `public MessageListener(String sslCertificateFile, SecureString certificatePassword)`
  Creates an instance along with a SSL certificate.
  Parameters:
  - `slCertificateFile` is the path of certificate file. See `SslCertificate` property for further information.
  - `certificatePassword` is the used password when exporting the certificate to PKCS #12 format.
- `public const int PORT`
  The default port for the server listens to.
- `public const int MAX_CONNECTIONS`
  Default maximum concurrent connections.
- `internal MessageListenerSync Sync`
  Holds some variables that must be synchronized among threads involved in serving requests. When updating this property state, a thread must lock it to get exclusive access.
- `internal ManualResetEvent TcpClientConnectedEvent`
  An event object for signaling between child thread and main thread. A child thread signals the main thread when it has started handling a request, so the main thread can restart waiting a new request. A child thread will also signal the main thread if it closes and notices that the main thread is not waiting a new request because the maximum number of connections has been reached.
- `public static String ConfigPath { get; private set; }`
  The full path of XML configuration file used by the message listener. It's read only for outside world. It can be set by method `SetConfigPath`.
- `public event EventHandler<ListeningEventArgs> StartListeningEvent;`
  The event handler that handles the event that is raised when the server starts listening the request message.
- `public String IPAddress { get; set; }`
  The IP address to which the server listens to. If it's `null` (by default), the server will listen to all IP address available on the hardware where the server runs.
- `public int Port { get; set; }`
  The port for the server listens to. If not set, it is the same as `PORT`.

- `public int MaxConnections { get; set; }`
  Maximum concurrent connections that the server can handle and also the maximum pending connections. If there are more clients want to connect to, it will throw an exception. If not set, it is the same as `MAX_CONNECTIONS`.

- `public X509Certificate SslCertificate { get; set; }`
  The certificate to be used for SSL/TLS communication. This certificate must include the private key. Use PKCS #12 format to store the certificate along with its private key. If this object is created by supplying the certificate to the constructor, this property will be set accordingly. But this property can be altered afterward. By default, this property is `null`, especially if no certificate was passed to the constructor. If this property is not `null` then the communication with the client will use SSL/TLS.

- `public SslProtocols SslProtocol { get; set; }`
  Specifies which version of SSL/TLS protocol that will be used. By default, it accepts SSL 3.0 or TLS 1.0.

- `public bool IsClientCertificateRequired { get; set; }`
  Set to `true` if this server requires the client has a certificate. By default, it's `false`. If it's `true` then `CertificateValidationCallback` property must be set for validating the client's certificate.

- `public bool IsCertificateRevocationChecked { get; set; }`
  Specifies whether the certificate revocation is checked. By default, it's `true` that will check the revocation.

- `public RemoteCertificateValidationCallback CertificateValidationCallback { get; set; }`
  The callback to validate the client's certificate. The callback must return `true` to make communication continues.

- `public int ReadTimeout { get; set; }`
  The time, in millisecond, how long this server can wait the data from the client before it decides that the connection has broken. By default, it's infinite.

- `public int WriteTimeout { get; set; }`
  When the server writes the data to the socket (sends the data to the client), it will wait the notification that the written data is accepted successfully. This property specifies the time, in millisecond, how long the server can wait before it decides that the connection has broken. By default, it's infinite.

- `public static void SetConfigPath(String path, String baseDir)`
  To set property `ConfigPath` that is the path for configuration file.
  Parameters:
  - `path` is the configuration file path. It can be absolute or relative.
  - `baseDir` is the base directory to which `path` parameter relative to. If it's `null` then `path` is relative to the current working directory. If `path` is absolute then this parameter is ignored.

- `public static void SetConfigPath(String path)`
  Call method `SetConfigPath` by passing `null` to `baseDir` parameter.
  Parameters:
  - `path` is the configuration file path. It can be absolute or relative to the current working directory.

- `private static void CreateXmlConfigParser()`
  Creates an instance of `XmlConfigParser` using an XML file referred by `ConfigPath`. The instance will be used by `Start` method.

- `public static void SetConfig(String fileConfigPath, String baseDir)`
  Calls `SetConfigPath(String, String)` method to set `ConfigPath` property and then calls `CreateXmlConfigParser` method. See also `SetConfig(Type)` method.
  Parameters:
  - `fileConfigPath` is the configuration file path. It can be absolute or relative.
  - `baseDir` is the base directory to which `fileConfigPath` parameter relative to. If it's `null` then `fileConfigPath` is relative to the current working directory. If `fileConfigPath` is absolute then this parameter is ignored.

- `public static void SetConfig(String fileConfigPath)`
  Calls `SetConfig(String, String)` method by passing `fileConfigPath` as the first parameter and `null` as the second one.
  Parameters:
  - `fileConfigPath` is the configuration file path. It can be absolute or to the current working directory.

- `public static void SetConfig(Type messageToModelMapping)`
  Create an instance of `AttributeConfigParser`. This instance will be used by `Start` method. This method and `SetConfig(String, String)` method can override each other. The last one will win that is, the `IConfigParser` instance created by it will be used by `Start` method.
  Parameters:
  - `messageToModelMapping` is the parameter that will passed when creating `AttributeConfigParser` instance.

- `private void DoBeginAcceptTcpClient()`
  Starts waiting a request. This method will hang until a request comes. When a request comes, method `DoAcceptTcpClientCallback` will be called asynchronously to handle the request. After the request is handled, it will restart waiting a new request. This loop keeps the message listener alive. It is invoked firstly by method `Start`. To indicate that it is waiting a request, it sets property `IsAcceptingConn` of `Sync` to `true`. It stops waiting a new request after the number of connection has reached `MaxConnections` number. It will restart waiting a new request if one or more connections close.

- `private void DoAcceptTcpClientCallback(Object clientObject)`
  Executing this method is triggered when a request comes. It will increase the value of property `NumOfConn` of `Sync` that is the number of concurrent connections being processed. After modifying `Sync`, it will create a `MessageListenerWorker` object to handle the request. The parameter of this method is a `TcpClient` object for communicating with the client. This method is executed asynchronously by `DoBeginAcceptTcpClient` method.

- `public void Start()`
  Firstly, it loads the configuration. If one of `SetConfig` methods has been called before, it will call `Load(IConfigParser)` method of `MessageProcessor`. If not then if

`ConfigPath` property has been set, it calls `CreateXmlConfigParser` method and then calls `Load(IConfigParser)` method of `MessageProcessor`. Otherwise, it calls `Load` method of `MessageProcessor`. If there is bad configuration, the listener will stop. After loading the configuration, it starts the listener to listen to the specified `IPAddress` and `Port` and then calls `DoBeginAcceptTcpClient` method.

- `public void Stop()`
  Stops listening `Port` and closes all connections.

- `private void RequestStop()`
  Stops method `DoBeginAcceptTcpClient` from looping process, that is to stop waiting a new request until method `Start` is invoked again.

Class **`MessageListenerWorker`**

The instance object of this class will be created to process a request. There may be some instances running concurrently that maintain different connection. The object of this class is created by method `DoAcceptTcpClientCallback` of `MessageListener`. The count of instance object alive will not be more than `MaxConnections`.

```
namespace Free.iso8583
internal class MessageListenerWorker : IMessageStream
```

- `public MessageListenerWorker(TcpClient client, MessageListener messageListener)`
  The only public constructor requires a parameter.
  Parameters:
    - `client` is an object that can be used to communicate to the client host. This `TcpClient` object is created when a connection has been created (happens in method `DoAcceptTcpClientCallback`).
    - `messageListener` is a `MessageListener` instance that creates this `MessageListenerWorker` object.

- `private void RegisterMe()`
  Registers this object to be maintained. The reference of this object will be saved for the use in method `CloseAll`. It also makes sure that this object will not be destroyed by *garbage collector*. This method invoked as soon as this object is just created.

- `private void UnregisterMe()`
  Releases this object to not be maintained anymore. It should be invoked if method `RegisterMe` has been invoked previously.

- `public void AcceptMessage()`
  Reads the request bytes from the network stream. The network stream is gotten from the `TcpClient` object passed to constructor. After the whole bytes are read, it invokes method `Receive` of `MessageProcessor` to process the request. The first parameter passed to the method is the request bytes which are just read and the second one is this object itself. This method invoked by method `DoAcceptTcpClientCallback` of `MessageListener`.

- `public byte[] Send(byte[] message)`
  Implements interface `IMessageStream`. It sends an ISO 8583 message to the client. This method always returns `null` because a server will always send a response and doesn't need a new request back except a new request-response sequence.
- `public int Receive(byte[] buffer, int offset, int count)`
  Implements interface `IMessageStream`. It reads the data from the client.
- `public void Close()`
  Implements interface `IMessageStream`. It closes the connection and calls method `UnregisterMe`. It decreases the value of property `NumOfConn` of `Sync`. If the main thread is not in the state of waiting a request because the maximum number of connection has been reached, it will signal the main thread to start waiting a new request because one connection has been released.
- `public static void CloseAll()`
  Invokes method `Close` of all maintained objects (see method `RegisterMe`).

Message Client

The message client will send a request message asked by the message processor. The message processor itself may get a request from a business object. By invoking method `Send` of `MessageProcessor`, a model object will be sent as a request message. This model object will be converted to an array of bytes (ISO 8583 message bytes) and in turn, method `Send` of message client will be called to send the message bytes to the server. The message client itself may invoke method `Send` of `MessageProcessor`, which in turn, it will call method `Send` of itself.

Class `MessageClient`

This is a basic message client that can send a request message as a model object (using method `SendModel`) or as an array of bytes (using method `SendBytes`). Method `Send`, which is needed by `MessageProcessor`, will connect to the server and send the message bytes and read the response bytes.

```
namespace Free.iso8583
public class MessageClient : IMessageStream
```

- `public MessageClient(String server, int port, Object model, MessageCallback callback)`
  The constructor.
  Parameters:
    - `server` is hostname/IP address to which this client will connect to.
    - `port` is the server port to which this client connects to.
    - `model` is a model object as the request message.
    - `callback` is the function will be executed after the response message comes and has been converted to a model object. The function itself must accept the

response model object as its parameter.

- `public MessageClient(String server, int port, Object model) : this(server, port, model, null)`
  The constructor.
  Parameters:
  - `server` is hostname/IP address to which this client will connect to.
  - `port` is the server port to which this client connects to.
  - `model` is a model object as the request message.
- `public String Server { get; set; }`
  The same as `server` parameter passed to constructor. By this property, the server hostname/IP address can be set again later.
- `public int Port { get; set; }`
  The same as `port` parameter passed to constructor. By this property, the server port can be set again later.
- `public Object Model { get; set; }`
  The same as `model` parameter passed to constructor. By this property, the request model can be set again later.
- `public MessageCallback Callback { get; set; }`
  The same as `callback` parameter passed to constructor. By this property, the callback can be set again later.
- `public bool IsSslEnabled { get; set; }`
  If it's `true` then the communication with server will use SSL/TLS. By default, it is `false`.
- `public X509CertificateCollection ClientCertificates { get; set; }`
  The collection of certificates it has. It is used when communicating using SSL/TLS and the server requires the client's certificate. This certificate must include the private key. Use PKCS #12 format to store the certificate along with its private key. By default, this property is `null`.
- `public SslProtocols SslProtocol { get; set; }`
  Specifies which version of SSL/TLS protocol that will be used. By default, it accepts SSL 3.0 or TLS 1.0.
- `public bool IsCertificateRevocationChecked { get; set; }`
  Specifies whether the certificate revocation is checked. By default, it's `true` that will check the revocation.
- `public RemoteCertificateValidationCallback CertificateValidationCallback { get; set; }`
  The callback to validate the server's certificate. The callback must return `true` to make communication continues. By default, it's `null` that will allow any certificate.
- `public LocalCertificateSelectionCallback CertificateSelectionCallback { get; set; }`
  The callback to choose which certificate which will be used. Remember it has a collection of certificates (see `ClientCertificates` property).
- `public int ReadTimeout { get; set; }`
  The time, in millisecond, how long this client can wait the data from the server before it

decides that the connection has broken. By default, it's infinite.

- `public int WriteTimeout { get; set; }`
  When the client writes the data to the socket (sends the data to the server), it will wait the notification that the written data is accepted successfully. This property specifies the time, in millisecond, how long this client can wait before it decides that the connection has broken. By default, it's infinite.

- `public IProcessedMessage SendModel()`
  Sends a request to the server by invoking method `Send` of `MessageProcessor` using `Model` as the first parameter, this object as the second parameter and `Callback` as the last one. It returns the response message as an array of bytes and a model object. Note, this method runs asynchronously, take advantage of `Callback` to know that the process has been done. As soon as this method exits, the response message, both as a bytes array and a model object, has not been set yet in the returned `IProcessedMessage` object (`ReceivedModel` and `ReceivedMessage` property). They are set just before `Callback` is called. It's because asynchronous process.

- `public IProcessedMessage SendBytes(byte[] message)`
  Sends a request message as a bytes array. It returns the response message as an array of bytes and a model object. By favor of `Thread` object, sending message can be processed asynchronously, for example: `new Thread(msgClient.SendBytes)`. By using property `Callback`, asynchronous process can be known when it's done.
  Parameters:
    - `message` is the request message.

- `private statis bool RemoteCertificateValidationCallback()`
  The default callback to validate the server's certificate if `CertificateValidationCallback` property is `null`. It allows any certificate.

- `private void CheckSocket()`
  Checks the connection to the server. If it is not open yet then it will create the new one. This method is called by `Send` and `Receive` method.

- `public byte[] Send(byte[] message)`
  Implements `IMessageStream`. It sends an ISO 8583 message. It will use a connection that has been opened if exists. If not, it tries to connect to the server. It returns the response message bytes. The calling process (in the message processor) will wait until a reply message comes. It will not block the process because the calling process has run in its own thread and each thread holds one instance of `MessageClient` (still processed in multi-threaded process). .
  Parameters:
    - `message` is the request message.

- `public int Receive(byte[] buffer, int offset, int count)`
  Implements `IMessageStream`. It reads the data from the server. It calls `Read(byte[] buffer, int offset, int count)` method of `Stream` object it has.

- `public void Close()`
  Implements `IMessageStream`. It closes the connection to the server.

Utility Classes

Utility class defines the public functions that can be used by all processes without concerning an object state. In technical words, they are signed as `static`.

Class `MessageUtility`

This static class provides the functions for manipulating the message. A lot of functions among them deal with converting message field value.

```
namespace Free.iso8583
public static class MessageUtility
```

- `public static decimal HexToDecimal(byte[] hex, int fracDigits)`
  Converts a hexadecimal in a bytes array to a decimal value. One byte contains two hexadecimal digits. Hexadecimal "10" means "10" decimal, not "16". All zero values on the left side will be removed. The `fracDigits` parameter determines how many digits after decimal point. No decimal point will be included in the bytes array, ex. "10.11" contains two digits (one byte) after decimal point, so the last byte in `hex` parameter contains 17 value (11 hexadecimal) and the other bytes will contain the mantisa.
- `public static ulong HexToInt(byte[] hex)`
  The same as `HexToDecimal` but return integer value.
- `public static String HexToReadableString(byte[] hex, int maxCharsPerLine = 0)`
  Converts a hexadecimal in a bytes array to be a string. One byte will produce two characters. Each two characters will be separated by space. Second parameter determines how many characters per line. If the produced string is longer than `maxCharsPerLine` then the string will be broken into some lines. If `maxCharsPerLine` is 0 or negative then the string will be one line.
- `public static String HexToString(byte[] hex)`
  Converts a hexadecimal in a bytes array to be a string. One byte will produce two characters.
- `public static String HexToString(byte val)`
  Converts a byte to hexadecimal string.
- `public static String HexToString(byte[] hex, bool isOdd)`
  Converts a hexadecimal in a bytes array to be a string. One byte will produce two characters. If there should be an odd number of characters (`isOdd` parameter value is `true`) then the first character will be removed.
- `public static String HexToString(byte[] hex, bool isOdd, bool isLeftJustified)`
  Converts a hexadecimal in a bytes array to be a string. One byte will produce two characters. If there should be an odd number of characters (`isOdd` parameter value is `true`) then one character will be removed. If `isLeftJutified` is `true` then the last character wil be removed, otherwise the first one.
- `private static void HexToChars(byte hex, char[] chars, int idx)`
  Converts a `byte` value (two digits hexadecimal) to be two `char` values. These `char` values will be placed in array `chars` at index `idx`-th and (`idx+1`)-th.

- `public static byte[] DecimalToHex(decimal val, int fracDigits, int length)`
  The reverse of method `HexToDecimal`. The `length` parameter determines how many bytes to be returned, will be padded by zero on left side.
- `public static byte[] DecimalToHex(decimal val, int fracDigits)`
  The same as the above method but returns the bytes as many as needed.
- `public static byte[] IntToHex(ulong val, int length)`
  The reverse of method `HexToInt`. The `length` parameter determines how many bytes to be returned, will be padded by zero on left side.
- `public static byte[] IntToHexLeftAligned(ulong val, int length)`
  The reverse of method `HexToInt`. The `length` parameter determines how many bytes to be returned, will be padded by zero on right side (left justified).
- `public static byte[] IntToHex(ulong val)`
  The same as the above `IntToHex` method but returns the bytes as many as needed.
- `public static byte[] StringToHex(String val)`
  The reverse of method `HexToString`. If there is an odd number of digits then padded by zero at the left side.
- `public static byte[] StringToHex(String val, bool isLeftAligned)`
  The reverse of method `HexToString`. If there is an odd number of digits then padded by zero at the left side (if `isLeftAligned` is `false`) or at the right side (if `isLeftAligned` is `true`).
- `public static byte[] StringToHex(String val, int length, bool isLeftAligned)`
  The reverse of method `HexToString`. It will return `length` digits. If there are more than `length` characters in the string (`val` parameter) then string is truncated. If the number of characters less than `length` then it will be padded by zero at the left side (if `isLeftAligned` is `false`) or at the right side (if `isLeftAligned` is `true`).
- `public static int AsciiArrayToInt(byte[] val)`
  Converts a bytes array to be an integer value. One byte represents an ANSI code, one character. Valid characters are “0” to “9”. If there is an invalid character then -1 will be returned.
- `public static String AsciiArrayToString(byte[] ansiChars)`
  Converts a bytes array to be a string. One byte represents an ANSI code, one character.
- `public static byte[] StringToAsciiArray(String val, int length)`
  The reverse of method `AsciiArrayToString`. The parameter `length` determines how many bytes to be returned, will be padded by space on the right side.
- `public static byte[] StringToAsciiArray(String val)`
  The same as the above method but returns the bytes as many as needed.
- `public static String GetBitString(byte[] bytes)`
  Returns a string which represents the value of each bit, 0 or 1.
- `public static byte[] DecimalToAsciiArray(decimal val, int fracDigits)`
  Converts the `decimal` value of parameter `val` to be an array of ASCII codes of each digits. Parameter `fracDigits` defines the number of digits after decimal period. If there are more digits than it then the value will be rounded. Decimal period will not be inserted

into the returned ASCII codes array.

- `public static decimal? StringToDecimal(String val)`
  Converts a string to `decimal` value. If the string has invalid value then `null` is returned. Negative value is invalid.

- `public static String HexVarlenToString(byte[] hex, int headerLength)`
  Returns a string representing the hexadecimal value inside the parameter `hex`. The length of the `hex` array is variable. To determine how many entries inside this array, there are some bytes in the beginning to note the length after. The count of how many these beginning bytes is passed via parameter `headerLength`. Example: parameter `headerLength` is 1 and the first byte in the parameter `hex` is "37" hexadecimal, then there are 37 digits hexadecimal (19 bytes) after the first byte. Thus, parameter `hex` contains 20 bytes, 1 byte of header and 19 bytes of content. Notice in the example before, "37" hexadecimal is not translated to "55" decimal, except there is a digit greater than or equal "A" hexadecimal. Note, if the content has the odd number of digits, it must be padded by zero at the right side (left justified).

- `public static String AsciiVarlenToString(byte[] ansi, int headerLength)`
  Similar with method `HexVarlenToString` but for ansi code (each code 1 byte). The parameter `headerLength` is still in hexadecimal digit. Example: the parameter `headerLength` is 2 and two first bytes of `ansi` array contains "0156" then `ansi` array has 158 entries (2 bytes of header and 156 bytes of content).

- `public static byte[] StringToHexVarlen(String val, int headerLength)`
  The reverse of method `HexVarlenToString`.

- `public static byte[] StringToAsciiVarlen(String val, int headerLength)`
  The reverse of method `AsciiVarlenToString`.

- `public static uint HexToInt(byte val)`
  Converts a byte value contains two digits hexadecimal to integer. Example: parameter `val` is "15" hexadecimal then this method must return "15" integer, "15" hexadecimal is not translated to "21" decimal. If the either digit is greater than or equal "A" hexadecimal then it will be translated as is, example "A5" will be "165" integer.

- `public static byte IntToHex2Digit(uint hex)`
  The reverse of method `HexToInt(byte val)`. Assumes the `hex` value lower than 100 and no hexadimal digit between "A" and "F".

- `public static ulong HexNDigitsToInt(byte[] val)`
  The same as calling repeatedly method `HexToInt` and calculate their result as if all entries in the array are a consecutive digits. If there is a digit greater than or equal "A" hexadecimal then it will calculate from the true value of each byte.

- `public static ulong BytesToInt(byte[] val)`
  Converts an array of byte values to `ulong` value as if they are the consecutive bytes for the `ulong` type.

- `public static byte[] IntToBytes(ulong val)`
  Reverse of method `BytesToInt`.

- `public static byte CharToByte(char ch)`

Returns the ASCII code of a character. Note, the passed parameter must be an ASCII character, not Unicode for example.

Class `Util`

Contains some useful functions that are generally needed by some processes.

```
namespace Free.iso8583
public static class Util
```

- `public static String[] PrintableClassPrefixes`
  This is used by method `GetReadableStringFromModel` as the default value for the second parameter that is when the second value is set to `null`. It is useful for the `Logger` so that it doesn't need to pass the second parameter everytime an exception happens (in fact, it can't). So, set this field in the beginning of the application runs and then `Logger` will take the advantage of this.

- `public static String GetAssemblyDir(Object obj)`
  Gets the directory where the assembly file containing a type exists.
  Parameters:
  - `obj` is an object whose type we search the directory where its assembly exists. If this is a `Type` object then we search the directory of this type's assembly file.

- `public static String Join(IEnumerable<Object> list, String separator)`
  Concatenates all item values inside a collection to be one string and returns the yielded string. All item values will be converted to string.
  Parameters:
  - `list` is the concatenated collection.
  - `separator` is a string value to separate each entry when it is concatenated.

- `public static String GetReadableStringFromModel(Object model)`
  Tries to write and returns a string representing a model object value in a readable format. All property names and values will be included. It calls another `GetReadableStringFromModel` method by passing `null` as the second parameter. This method is called by `Logger` when an exception happens to write the model object which is failed to process.
  Parameters:
  - `model` is a model to try to write.

- `public static String GetReadableStringFromModel(Object model, String[] printableClassPrefixes)`
  Some properties may have a weird value when it is converted to string (using method `ToString`). To make meaningful value of these properties, the process must be done recursively (re-execute `GetReadableStringFromModel`) to these properties. The process, however, can be infinite. This overloaded method of `GetReadableStringFromModel(Object model)` adds the second parameter to tell

what properties should be processed recursively.
Parameters:

- `model` is a model to try to write.
- `printableClassPrefixes` is an array of type prefixes should be processed recursively. If there is a property whose type prefixed by a string included in this array, the process will be processed recursively for this property. The strings inside this array are supposed to be the namespace of the types. Be careful to use this parameter to avoid a long process. If this parameter is `null` then the value is taken from field `PrintableClassPrefixes`.

- `private static void PrintModel(Object model, StringBuilder str, String indent, String[] printableClassPrefixes)`
  It is called by method `GetReadableStringFromModel` to execute the actual process of writing readable string. This method makes the recursive process done.
  Parameters:
  - `model` is a model to try to write.
  - `str` will contain the result string when the process finishes.
  - `indent` contains some tab characters to indent the properties to make more readable format. It is useful when doing recursive process. It should be empty string when this method is called first time before recursive process.
  - `printableClassPrefixes` is an array of type prefixes should be processed recursively.
- `private static bool IsPrintableClass(Type type, String[] printableClassPrefixes)`
  It is called by method `PrintModel` to check if the recursive process should be done to a property.
  Parameters:
  - `type` is the type of the checked property.
  - `printableClassPrefixes` is an array of type prefixes should be processed recursively.
- `public static MethodInfo GetConvertOperator(Type hostType, Type guestType, bool guestToHost)`
  Gets the conversion operator (implicit/explicit) as `System.Reflection.MethodInfo` object. The conversion operator exists in C# language. This operator is to convert from a type to another type when it's assigned to. This method is defined because the reflection methods don't aware the defined conversion operator so that it will throw an exception when assigning a value from the type matching the conversion operator. But actually, it should be assignable because of the existence of the conversion operator. This method is invoked by method `GetAssignableValue`.
  Parameters:
  - `hostType` is the type that defines the conversion operator.
  - `guestType` is another type that will be assigned to/from `hostType`.

    - `guestToHost` is `true` when we want to get the conversion operator to assign a value of `guestType` to a variable of `hostType`. Otherwise, it's the reverse assignment.

  Returns:
  An `MethodInfo` object representing the conversion method or `null` if it's not found.

- `public static Object GetAssignableValue(Type type, Object value)`
  Returns an assignable value of `value` (second parameter) for assigning it to an object whose type of `type` (first parameter). It will return `null` if it fails to find an appropriate value. It will be used when converting the message content to be a model object and vice versa. Each message field must be assignable to the corresponding property of the model object.

Configuration

To map between an ISO 8583 message to a model object, it's needed a configuration. The configuration will be read by the configuration parser before processing the messages. The parser should be executed only once at the application is starting.

This library supports two kind of configuration: XML and Attribute classes. Next sections will explain them more detail.

XML Configuration

The XML configuration will look like:

```
<MessageMap …>
    <message id="…" length-header="…" …>
        <header name="…" … />
        <message-type length="…" value="…" />
        <bitmap length="…" />
        <bit seq="…"  … />
        <bit seq="…"  … />
        ……
    </message>
    <message …>
        ……
    </message>

    <model id="…" class="…" message="…">
        <property name="…" bit="…" type="…" frac-digits="…" />
        <property name="…" bit="…" delegate="…" />
        <property name="…" bit="…" bitcontent="…" />
        <property name="…" bit="…" tlv="…" tlv-tag-name="…"
            tlv-length-bytes="…" tlv-class="…" />
        ……
    </model>
    <model id=”…” class=”…” message=”…”>
        ……
    </model>

    <message-to-model model=”…” delegate=”…”>
        <mask start-byte=”…” length=”…” value=”…” />
        <mask start-byte=”…” length=”…” mask=”…” result=”…” />
        <and>
            <mask start-byte=”…” length=”…” value=”…” />
            <mask start-byte=”…” length=”…” mask=”…” result=”…” />
            <or>
                <mask start-byte=”…” length=”…” value=”…” />
                <mask start-byte=”…” length=”…” mask=”…” result=”…” />
            </or>
        </and>
    </message-to-model>
    <message-to-model model=”…” delegate=”…”>
        ………
    </message-to-model>

    <BitContent id=”…” class=”…”>
        <field name=”…” length=”…” pad=”…” align=”…” null=”…” trim=”…” />
        <field name=”…” length=”…” pad=”…” align=”…” null=”…” trim=”…” />
        <field name=”…” tlv=”…” tlv-tag-name=”…” tlv-length-bytes=”…”
           tlv-class=”…” />
        ……
    </BitContent>
    <BitContent id=”…” class=”…”>
        ……
    </BitContent>

    <tlv id=”…” tag-length=”…” length-bytes=”…” class=”…”>
        <tag name=”…” />
        <tag name=”…” type="array" splitter=”…” />
        <tag name=”…” type="bitcontent" bitcontent=”…” />
        ……
    </tlv>

    <delegate id=”…” class=”…” method=”…” />
    <delegate id=”…” class=”…” method=”…” />

    <type id=”…” class=”…” />
    <type id=”…” class=”…” />
</MessageMap>
```

The root element is `MessageMap`. It will contain some `message` elements to define ISO 8583 messages, some `model` elements to define the model objects and some `message-to-model` elements to map the incoming messages to the appropriate model. Some complementary elements, such as `BitContent` and `delegate`, may exist.

Element `message`

This element defines an ISO 8583 message, explains all fields inside the message. Each field is represented by a child element inside `message` element. The child elements other than `bit` element are categorized as the header message element and will be placed at the beginning of the message. These header elements must be placed at the same order as their order in the message. It doesn't apply to element `bit` because it has attribute `seq`.

Attributes:

- `id`
  Mandatory. The id of this message. It will be referred by the element `model` to which message it is mapped.
- `length-header`
  Mandatory. How many bytes at the beginning message will record the message length. The binary value in these bytes from left to right (the left most is the highest) if converted to decimal value will tell how many bytes of message will come after the length header bytes.

Element `header`

It's the child element of element `message`. Defines the header of message and must be placed in the same order as their order in the message.

Attributes:

- `name`
  Optional. The name of this header. It can be a description of what header it is.
- `length`
  Mandatory. Determines how many bytes of this header.
- `value`
  Optional but must exist if attribute `delegate` is absent. If this attribute is specified then all message will have the same value of this header. The value of this attribute is a hexadecimal string.
- `delegate`
  Optional. Refers to an id of element `delegate`. This delegate will be executed to fill this header. The delegate function accepts no parameter and returns an array of bytes. If this attribute is specified then attribute `value` is ignored. See also element `delegate`.

Element `message-type`

Child element of element `message`. Represents the message type header of this ISO 8583 message.

Attributes:

- `value`
  Mandatory. Hexadecimal string for message type value. It's useful when compiling a message.
- `length`
  Mandatory. How many bytes the length of this message type header.

Element `bitmap`

It's the child element of element `message`. This element is to specify the length of the primary bitmap header. Bitmap header determines the presence of fields in the message. Example, if the first bit in this bitmap header is 1 then the first field of message will be present. Otherwise, it will be absent. Beside this bitmap header, there may be a message field behaves as a bitmap for extending message fields. This message field has type of `BitMap`.

Attributes:

- `length`
  Mandatory. How many bytes the length of this bitmap header is.

Element `bit`

It's the child element of element `message`. It represents a field in the message. This field may be absent depends on the value of bitmap.

Attributes:

- `seq`
  Mandatory. The ordinal number of this field in the message. It's the same as the ordinal number of the corresponding bit in the bitmap. A message may not specify the fields as many bit as in the bitmap header. Some bits in bitmap may be always 0. By the favor of this attribute, the element `bit` doesn't have to be placed in the true sequence.

- `type`
  Mandatory. The data type of this field. Supported data types are:
  - `B` represents a sequence of bits, on (1) or off (0). It can only be mapped to a model property of type bytes array.
  - `N` represents a numeric and each digit has four bits long (nibble). It's like a hexadecimal number but only digit 1 to 9 exist. "0x19" means "19" decimal, not "25" ("19" hexadecimal).
  - `NS` represents a hexadecimal string. It should be suitable to be mapped to a model property of type bytes array or string.
  - `ANS` represents an ANSI character, thus each character has 1 byte.
  - `AN` represents an ANSI numeric character ("0" up to "9"). Therefore it can be converted to a numeric data type.
  - `NULL` is for a dummy field. It exists to set the bit in bitmap header on `seq` position to be 1. However, this field doesn't exist in the message body. This type, especially, is intended for the secondary bitmap (bit 65 up to 128). Generally, secondary bitmap is considered as the first field. When it's defined the primary bitmap to cover secondary bitmap (by enlarging `length` value), this field type is used to set the first bit in primary bitmap on.
  - `BitMap` is an additional bitmap for extending message fields. It's intended for secondary and tertiary bitmap. This field type cannot be mapped to a model property.

- `length`
  Invalid if attribute `length-header` exists. How many characters or digits in the data. Especially for

type `N`, because one digit is four bits, if there are odd numbers of digits then it will be added one digit on the left filled by 0 so that the number of bytes is rounded. This attribute is invalid if `type` is `NULL`.

- `length-header`
  Optional. Determines how many bytes of field header recorded the bytes count of this field. The value inside this header bytes is interpreted like type `N` (four bits digits) and replaces the value of attribute `length`. If this attribute specifies one byte and the value in that byte records “0x15” and the data type is `ANS` then the actual length of this field is 16 bytes (1 byte header and 15 bytes content). This attribute is invalid if attribute `length` presents or `type` is `NULL` or `BitMap`.

- `from-request`
  Optional. Typically used by a response message. Gets the value of field on the same position in the request message that is previously saved by the processor if the value of the corresponding the model object property is `null`. The possible value of this attribute is `true` or `false`. The default value is `false`. This attribute is invalid if `type` is `NULL` or `BitMap`.

- `delegate`
  Optional. Refers to an id of element `delegate`. This delegate will be executed to fill this field when translating the corresponding model object property. The delegate function accepts the property value as the parameter and returns an array of bytes. See element `delegate`. This attribute is invalid if `type` is `NULL` or `BitMap`.

Element `model`

It defines a model object that will be used to save a message. The object model is more understandable by a business object to process. The message processor will translate a model object to/from an ISO 8583 message. A model object must be mapped to an ISO 8583 message for the processor to understand. A model object has some properties that are mapped to a field (element `bit`) in the message. Some properties may be generated automatically by the favor of a delegate function.

Attributes:

- `id`
  Optional. The id that will be referred by a message map (see element `message-to-model`) so that the model can be created when an incoming message comes. Note, it is optional because you may define a model that is just used by a sent message.

- `class`
  Mandatory if attribute `extend` is not specified. The assembly-qualified name of the type of the model object. It can also be the id of element `type`. To create an instance of the model object, it will get the object type first by using method `Type.GeType`.

- `message`
  Mandatory if attribute `extend` is not specified. The id of element `message` to which this model is mapped.

- `extend`
  Optional. If this model is inherited from another model, this attribute refers the id of the parent model. All attributes and properties different from those defined for the parent model, must be redefined by child model. Redefined attributes and properties will override the same ones which are already defined for the parent model.

Element `property`

It's the child element of element `model`. It represents a property of the model object. Perhaps not all properties of the model object are mapped (for internal use of the object model). If a mapped property is `null` then the corresponding field in the message will be absent (the corresponding bit in the bitmap header will be 0) or the field value will be generated automatically.

Attributes:

- `name`
  Mandatory. The name of this property. It's declared variable name for this property in the object model code.

- `bit`
  Mandatory. The order bit of the message field to which this property is mapped. The value will be the same as the value of attribute `seq` of element `bit`.

- `type`
  Mandatory if attribute `delegate` and `bitcontent` are not specified. It's the data type of this property. The possible values are `string` (for string value), `int` (for integer value), `decimal` (for currency value) and `bytes` (array of bytes value). If another type is needed, use attribute `delegate`. The actual type of the property in the code must be able to be set by using the data type specified by this attribute (it will use `GetAssignableValue` method). The property type in the code must be nullable to avoid error if the assigned value is `null`.

- `frac-digits`
  Optional. How many digits after decimal point. The message will not contain decimal point. It will be ignored if the value of attribute `type` is other than `decimal`. By default, the value is `0` (no digit after decimal point).

- `delegate`
  Optional. Refers to an id of element `delegate`. This delegate will be executed to get the value of this property. This delegate function accepts an array of byte values (will be taken from the message field value to which this property is mapped) and returns an expected type for this property. If this attribute is specified then the attribute `type` will be ignored. If this attribute is not ignored then attribute `delegate` of the mapped element `bit` (linked by attribute `bit`) must be specified. See also element `delegate`.

- `bitcontent`
  Optional. Refers to an id of element `BitContent`. Typically, it's used by the field containing additional information which has a variable length. If this attribute is specified then attribute `delegate` and `type` will be ignored. See element `BitContent`.

- `tlv`
  Optional. Refers to an id of element `tlv`. Typically, it's used by the field containing additional information which has a variable length. If this attribute is specified then attribute `bitcontent`, `delegate` and `type` will be ignored. See element `tlv`.

- `tlv-tag-name`
  Mandatory if this property does not refers to an element `tlv` which defines the tag. The tag id. Typically, it's used by the field containing additional information called TLV (Tag Length Value) data.

See element tlv for further information. This attribute indirectly specifies how many bytes will be occupied by the tag id. Note, when parsing an ISO 8583 message, the tag id will be ignored, it only notices the length of tag. Only when creating an ISO 8583 message from a model, the tag id will be included into message. If this attribute is specified then attribute bitcontent, delegate and type will be ignored. Note, this attribute will be usable if there is only one tag in this field. If there is more than one tag, use element tlv.

- tlv-length-bytes
  Optional. How many bytes will contain the data length value. Typically, it's used by the field containing additional information called TLV (Tag Length Value) data. This attribute is meaningful if attribute tlv or tlv-tag-name is defined. The default value is 3. See element tlv.

- tlv-class
  Optional. This attribute is meaningful if attribute tlv or tlv-tag-name is defined. See attribute class of element tlv.

Element **message-to-model**

This element is to map an incoming message to a model. When a message comes, the appropriate model will be instantiated. This mapping configuration will be evaluated when executing method Receive of MessageProcessor. This method is invoked by the message server or client when they receive a message.

The technique used to determine which model will be used is bit-masking operation. Element message-to-model contains one or more element mask. The element mask defines which bytes in the message to be masked and the result value of masking. If there are more than one element specified then it will be operated by logical "or" operation. Element message-to-model may also contain element and and element and itself may contain some element mask. The element and is to make logical "and" operation. Inside element and may also contain element or and element or itself may contain some element mask. The element or will operate all contained mask by logical "or" operation.

There may be more than one message-to-model which matches an incoming message. If it happens, the first defined one will be chosen. So, place the less general filter before another one if there is a possibility that they will match the same message. It will make the less general filter never misses a message. A message-to-model is considered less general if it has more bytes to check.

Attributes:

- model
  Mandatory. Refers to an id of the element model to which the incoming message will be mapped.

- delegate
  Optional. Refers to an id of the element delegate that will be executed when an appropriate message comes. The delegate function must implement MessageCallback. If this attribute is not specified then another process (message client or server) that utilize this message processor, should invoke method Receive of MessageProcessor that takes a delegate (MessageCallback object) for its parameter.

Element `mask`

It's the child element of element `message-to-model`. This element defines all parameter for masking operation when mapping an incoming message to a model.

Attributes:

- `start-byte`
  Mandatory. Defines the starting byte to be masked. The first index is 1, not zero.
- `length`
  Mandatory. Determines how many bytes to be masked.
- `value`
  Mandatory if attribute `mask` is absent. If attribute `mask` is present then this attribute must be absent. The value of the masked bytes must be equal a hexadecimal string written in this attribute.
- `mask`
  Mandatory if attribute `value` is absent. If attribute `value` is present then this attribute must be absent. This attribute value will be operated with the masked bytes by bitwise 'and' operation. This attribute value is a hexadecimal string. The result of masking operation is determined by attribute `result`.
- `result`
  Mandatory if attribute `mask` is defined. The possible value is 'equals' which defines the result of masking operation must be equal with the value of attribute `mask`. Another possible value is 'notZero' which defines the result of masking operation must be not zero (at least one bit is 1). The value 'notEquals' and 'zero' can also be set which are the reverse of 'equals' and 'notZero' consecutively.

Attribute `mask` and `result` are useful to mask the bit map header.

Element `BitContent`

Some fields in the message may store an additional information whose a variable length. This additional information can be broken down to be some sub fields. Thus, we can define a new model object whose some properties to be mapped to this message field. The element `BitContent` defines the model object needed for this purpose that can be referred by such sub fields. All properties of the object defined here must be a string or an array of bytes.

This element has a child element that is `field`. The first element is to determine how many header bytes that store the length of the field. The length value itself excludes the count of bytes of the header. The second one is to define the property of the defined object (sub field).

Attributes:

- `id`
  Mandatory. The id of this element that is referred by a message field needing it.
- `class`
  Mandatory. The assembly-qualified name of the type of this object. It can also be an id of element `type`. To create an instance of the object, it will get the object type first by using method `Type.GetType`.

Element `field`

It's the child element of element `BitContent`. It defines a sub field.

Attributes:

- `name`
  Mandatory. The property name of the object defined by the element `BitContent`. This property is mapped to this sub field.
- `length`
  Mandatory if `tlv` and `tlv-tag-name` are not specified. Determines how many characters (bytes) inside this sub fields.
- `pad`
  Optional. Determines the character will be padded if the length of the sub field value is less than the length defined by attribute `length`. It's usually space or zero. The default value is space.
- `align`
  Optional. Determines where to put the pad character. The possible values are `left` and `right`. If `left` is specified then the pad characters are put on the right side, otherwise on the left side. The default value is `left`.
- `null`
  Optional. Determines what character to be used when the mapped property is `null`. The sub field must have the length as defined by the attribute `length`. If this attribute is not present, the value is the same as the attribute `pad` value.
- `trim`
  Optional. It's to determine whether the sub field value must be trimmed (all leading and trailing spaces are removed) before assigned to the mapped property. The possible values are `true` and `false`. The default value is `true` which determines that the value must be trimmed.
- `optional`
  Optional. The possible values are `true` and `false`. The default value is `false` which determines that the field must exist. The field that defines it's optional, must be the last field. Thus, not more than one field that will be optional.
- `tlv`
  Optional. Refers to an id of element `tlv`. If this attribute is specified then all above attributes except `name` will be ignored. See element `tlv`.
- `tlv-tag-name`
  Mandatory if this field does not refer to an element `tlv` which defines the tag. The tag id. Typically, it's used by the field containing additional information called TLV (Tag Length Value) data. See element `tlv` for further information. This attribute indirectly specifies how many bytes will be occupied by the tag id. Note, when parsing an ISO 8583 message, the tag id will be ignored. It only notices the length of tag. Only when creating an ISO 8583 message from a model, the tag id will be included into message. If this attribute is specified then the all above attributes except `name` and `tlv` will be ignored. Note, this attribute will be usable if there is only one tag in this field. If there are more than one tag, use element `tlv`.

- `tlv-length-bytes`
  Optional. How many bytes will contain the data length value. Typically, it's used by the field containing additional information called TLV (Tag Length Value) data. This atribute is meaningful if attribute `tlv` or `tlv-tag-name` is defined. The default value is 3. See element `tlv`.
- `tlv-class`
  Optional. This atribute is meaningful if attribute `tlv` or `tlv-tag-name` is defined. See attribute `class` of element `tlv`.

Note that attribute `pad`, `align` and `null` are needed when converting the model object to the message field.

Element `tlv`

TLV stands for Tag Length Value. It's similar with `BitContent` to store additional data but has special format. It uses a tag to identify the data following the tag. All data in TLV represents ASCII characters. The format is started by a tag id which occupies some bytes defined by configuration, followed by pre-determined length of bytes which specifies the length of data following, like described below:

Tag ID	Data Length	Data

Example: if tag ID is "TD" (occupies 2 bytes) and the data is "ABCDEFG" (has 7 bytes length) and the bytes count of data length is 3 then the all data will be "TD007ABCDEFG". Note, there may be more than one 'Tag ID – Data Length – Data' sequence that construct whole message field.

Attributes:

- `id`
  Mandatory. The id of this element that is referred by a model property or `BitContent` field needing it.
- `length-bytes`
  Optional. How many bytes will store data length value. If defined, it will supersede the value defined by `tlv-length-bytes` attribute in element `property` of model and element `field` of `BitContent`.
- `class`
  Optional. The assembly-qualified name of the type of `tlv` object. It can also be the id of element `type`. It supersedes attribute `tlv-class` of `property` and `field` element. It will be used when parsing an ISO 8583 message to be converted to a model object. If this attribute is not specified then the processor will try to interpret what the appropriate type for this `tlv` object that is assignable to the property it is mapped to. The rule of interpretation is describes as follows:
  When collecting the sub field value of the `tlv`, the processor will create a map (type of `IDictionary<String, Object>`) keyed by tag ID. There may be more than one tag. The possible type of the value of sub field is string, array of string or a type defined by a `BitContent`.
  If there is only one entry in the map, the processor will try to assign the only value to the mapped property. If it fails then it will try to search a constructor of the type of the property that accepts a parameter whose type of value type and then creates an instance using that constructor. If not found and the type of value is string then it will search a `Parse` method which is public and static to parse the string value to an appropriate type. If still fails then throws an exception.
  If there are more than one entry in the map then it will try to assign the map to the property. If fails then it will interpret that the mapped property is an object whose properties named as the map key names

(tag IDs). Each value will be interpreted like interpreting the only one value described above.

Element **`tag`**

Element `tlv` should have some child element `tag`s which define all tags inside this message field. If not defined one, the element `property` and `field` referring to this element `tlv` must define attribute `tlv-tag-name`.

Attributes:

- `name`
  Mandatory. The tag ID.
- `type`
  Optional. The possible values are '`array`' and '`bitcontent`'. If specified '`array`' then attribute `splitter` should be defined. The value is considered as a string array, each element is separated by `splitter`. If specified '`bitcontent`' the attribute `bitcontent` must be defined. The value following this tag will be treated as `BitContent` field.
- `splitter`
  Optional. It's meaningful if `type` is '`array`'. This is a string will be used to separate each element of array. The default value is ';' (semi-colon).
- `bitcontent`
  Mandatory if `type` is '`bitcontent`'. The id of the referred element `BitContent`.

Note, all tag ID must have the same length.

Element **`delegate`**

It's to define a delegate function that will be executed for some purposes. The signature of the delegate function must fit with the purpose whether it's for mapping a field message to a property of the model object, to generate a message field value or to be called to process a model object after it's converted from a message.

Attributes:

- `id`
  Mandatory. The id of this element.
- `class`
  Mandatory. The assembly-qualified name of the type in which the method used for delegate is defined. It can also be the id of element `type`. To create an instance of the object, it will get the object type first by using method `Type.GeType`. If the method is not static, the message processor will create a new instance object of this class. Each delegate has different instance of object. If it wants to share the same instance then it must define a static method named `GetInstance` which returns the same instance object.
- `method`
  Mandatory. The name of the used method.
- `param-type`

Optional. The type of the parameter of this delegate. The delegate has one parameter at maximum. If the delegate has no parameter, this attribute is not defined. The value of this attribute is the assembly-qualified name of the type. It can also be an id of element `type`.

Element `type`

To define a type or class that furthermore can be referred by an attribute which defines a type, such as attribute `class`. It's worth if the type is referred more than one time. As we know, to get a reference to a type using a string as its identifier, we need assembly-qualified name. This name, if it's not a core type or not placed in the same assembly, is rather complicated, which includes full name space and assembly name. Using this element `type`, we can create an alias for a type, so we can refer a simple name.

Attributes:

- `id`
  Mandatory. The id of this element. This attribute value will be used to refer to this type.
- `class`
  Mandatory. The assembly-qualified name of the defined type. To create an instance of the object, it will get the object type first by using method `Type.GeType`.

Attribute Configuration

To map a model class to an ISO 8583 message, it can be attached some attributes/annotations to the model class and its properties. This section will explain the comparable items in attributes if compared to the items in XML configuration. The detail explanation should be read in XML Configuration. To find how to use attribute configuration, consult the source code, especially under namespace `Free.iso8583.example`. See also `Load(Type)` method of `MessageProcessor`.

Class `MessageAttribute`

This attribute is attached to the model class. This attribute is comparable to element `message`. We don't need element `model` because we know on which model class, the attribute is placed.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class MessageAttribute : Attribute
```

- `public int LengthHeader { get; set; }`
  Comparable to `length-header` attribute of element `message`.

Class `BaseHeaderAttribute`

It is a base class for all attributes class of message headers. Message header doesn't have a corresponding model property, so it is attached to the model class. The order header in the message is determined by `Seq` property of the attribute object.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public abstract class BaseHeaderAttribute : Attribute
```

- `public int Seq { get; set; }`
  Determine the order of header in the message. Lesser value will be prior to the greater one.

Class `HeaderAttribute`

It's comparable to element `header`. See also class `BaseHeaderAttribute`.

```
namespace Free.iso8583.config.attribute
public class HeaderAttribute : BaseHeaderAttribute
```

- `public string Name { get; set; }`
  Comparable to `name` attribute of element `header`.
- `public byte[] Value { get; set; }`
  Comparable to `value` attribute of element `header`.
- `public Type DelegateClass { get; set; }`
  The class which defines the method represented by the delegate as defined by `delegate` attribute of element `header`.
- `public string DelegateMethod { get; set; }`
  The name of method represented by the delegate as defined by `delegate` attribute of element `header`.
- `public int Length { get; set; }`
  Comparable to `length` attribute of element `header`.

Class `MessageTypeAttribute`

It's comparable to element `message-type`. See also class `BaseHeaderAttribute`.

```
namespace Free.iso8583.config.attribute
public class MessageTypeAttribute : BaseHeaderAttribute
```

- `public byte[] Value { get; set; }`
  Comparable to `value` attribute of element `message-type`. The `length` attribute is implied by the length of this array of bytes.

Class `BitMapAttribute`

It's comparable to element `bitmap`. See also class `BaseHeaderAttribute`.

```
namespace Free.iso8583.config.attribute
public class BitMapAttribute: BaseHeaderAttribute
```

- `public int Length { get; set; }`
  Comparable to `length` attribute of element `bitmap`.

Class `FieldAttribute`

This attribute is comparable to element `bit`.

```
namespace Free.iso8583.config.attribute
public abstract class FieldAttribute: Attribute
```

- `public int Seq { get; set; }`
  Comparable to `seq` attribute of element `bit`.
- `public virtual FieldType FieldType { get; set; }`
  Comparable to `type` attribute of element `bit`.
- `public virtual Type FieldClass { get; }`
  The class of field container which reflects `FieldType`. This field container will be used for the message field represented by this `FieldAttribute`.

Class `NullFieldAttribute`

This attribute is comparable to element `bit` whose type "`NULL`". Because no model property will correspond to the message field, this attribute is attached to the model class.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class NullFieldAttribute : FieldAttribute
```

- `public override FieldType FieldType { get; }`
  Always returns `FieldType.Null`.

Class `BitMapFieldAttribute`

This attribute is comparable to element `bit` whose type "`BitMap`". Because no model property will correspond to the message field, this attribute is attached to the model class.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class BitMapFieldAttribute : FieldAttribute
```

- `public override FieldType FieldType { get; }`

Always returns `FieldType.BitMap`.

Class `PropertyFieldAttribute`

This attribute is comparable to element `bit` whose type other than “`Null`” and “`BitMap`”. Because there is no attribute class comparable to element `model` then there is also no attribute class comparable to element `property`. So, the information of element `property` and element `bit` are merged to be one attribute class. This attribute class contains the information comparable to both XML element.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class PropertyFieldAttribute : FieldAttribute
```

- `public override FieldType FieldType { get; set; }`
  Has value other than `FieldType.Null` and `FieldType.BitMap`. It will throw an exception if set to an invalid value.
- `public int LengthHeader { get; set; }`
  Comparable to `length-header` attribute of element `bit`.
- `public int Length { get; set; }`
  Comparable to `length` attribute of element `bit`.
- `public PropertyType PropertyType { get; set; }`
  Comparable to `type` and `bitcontent` attribute of element `property`. If its value is `PropertyType.BitContent`, the BitContent class is the same as the return type of the attributed model property.
- `public bool FromRequest { get; set; }`
  Comparable to `from-request` attribute of element `bit`.
- `public Type FieldDelegateClass { get; set; }`
  The class which defines the method represented by the delegate as defined by `delegate` attribute of element `bit`.
- `public string FieldDelegateMethod { get; set; }`
  The name of method represented by the delegate as defined by `delegate` attribute of element `bit`.
- `public int FracDigits { get; set; }`
  Comparable to `frac-digits` attribute of element `property`.
- `public Type PropertyDelegateClass { get; set; }`
  The class which defines the method represented by the delegate as defined by `delegate` attribute of element `property`.
- `public string PropertyDelegateMethod { get; set; }`
  The name of method represented by the delegate as defined by `delegate` attribute of element `property`.
- `public Type Tlv { get; set; }`
  Comparable to `tlv` attribute of element `property`.
- `public String TlvTagName { get; set; }`

Comparable to `tlv-tag-name` attribute of element `property`.

- `public int TlvLengthBytes { get; set; }`
  Comparable to `tlv-length-bytes` attribute of element `property`.

Enum `FieldType`

This enumeration specifies the type of message field.

```
namespace Free.iso8583.config.attribute
public enum FieldType
```

- `B = 0`
  For type "`B`" as defined by `type` attribute of element `bit`.
- `N = 1`
  For type "`N`" as defined by `type` attribute of element `bit`.
- `NS = 2`
  For type "`NS`" as defined by `type` attribute of element `bit`.
- `AN = 3`
  For type "`AN`" as defined by `type` attribute of element `bit`.
- `ANS = 4`
  For type "`ANS`" as defined by `type` attribute of element `bit`.
- `Null = 5`
  For type "`NULL`" as defined by `type` attribute of element `bit`.
- `BitMap = 6`
  For type "`BitMap`" as defined by `type` attribute of element `bit`.

Enum `PropertyType`

This enumeration specifies the type of model property.

```
namespace Free.iso8583.config.attribute
public enum PropertyType
```

- `String = 1`
  For type "`string`" as defined by `type` attribute of element `property`.
- `Int = 2`
  For type "`int`" as defined by `type` attribute of element `property`.
- `Decimal = 3`
  For type "`decimal`" as defined by `type` attribute of element `property`.

- `Bytes = 4`
  For type "`bytes`" as defined by `type` attribute of element `property`.
- `BitContent = 5`
  If the property is a BitContent. The BitContent class is implied by the return type of the property.

Class `BitContentFieldAttribute`

It's comparable to element `field`. It's attached to a class property. The class itself represents the BitContent class.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class BitContentFieldAttribute : Attribute
```

- `public int Length { get; set; }`
  Comparable to `length` attribute of element `field`.
- `public char PadChar { get; set; }`
  Comparable to `pad` attribute of element `field`.
- `public FieldAlignment Align { get; set; }`
  Comparable to `align` attribute of element `field`.
- `public char NullChar { get; set; }`
  Comparable to `null` attribute of element `field`.
- `public bool IsTrim { get; set; }`
  Comparable to `trim` attribute of element `field`.
- `public bool IsOptional { get; set; }`
  Comparable to `optional` attribute of element `field`.
- `public Type Tlv { get; set; }`
  Comparable to `tlv` attribute of element `field`. Defines tlv class.
- `public String TlvTagName { get; set; }`
  Comparable to `tlv-tag-name` attribute of element `field`.
- `public int TlvLengthBytes { get; set; }`
  Comparable to `tlv-length-bytes` attribute of element `field`.

Enum `FieldAlignment`

It specifies the alignment of a sub field.

```
namespace Free.iso8583.config.attribute
public enum FieldAlignment
```

- `Left = 0`

Left alignment. The pad characters are put on the right side.

- `Right = 1`
  Right alignment. The pad characters are put on the left side.

Class `TlvAttribute`

It's comparable to element `tlv`. This attribute is attached to a tlv class.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class TlvAttribute : Attribute
```

- `public int LengthBytes { get; set; }`
  Comparable to `length-bytes` attribute of element `tlv`.

Class `TlvTagAttribute`

This attribute is comparable to element `tag`. It's attached to a tlv class property.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class TlvTagAttribute : Attribute
```

- `public string Splitter { get; set; }`
  Comparable to `splitter` attribute of element `tag`.
- `public TlvTagType Type { get; set; }`
  Comparable to `type` attribute of element `tag`. If it is `TlvTagType.BitContent` then the return type of the attributed property is BitContent class.

Enum `TlvTagType`

It specifies the type of tlv property.

```
namespace Free.iso8583.config.attribute
public enum TlvTagType
```

- `Default = 0`
  Default type, e.g. if not specified.
- `Array = 1`
  array type. See `type` attribute of element `tag`.
- `BitContent = 2`
  bitcontent type. See `type` attribute of element `tag`.

Interface `IMask`

To map an incoming message to a model, it will be executed bit-masking operation. In attribute configuration, all classes that are responsible to map the incoming message must implement this interface. See also `Load(Type)` method of `MessageProcessor`.

```
namespace Free.iso8583.config.attribute
public interface IMask
```

- `IMaskConfig GetConfig(Type modelClass, MethodInfo processDelegate)`
  Returns the configuration object that matches to this configuration.
  Parameters:
  - `modelClass` is the model class to which the incoming message will be mapped to.
  - `processDelegate` is the delegate that will be executed after the incoming message has been converted to a model object, as described by `delegate` attribute of element `message-to-model`.
- `IList<IMask> Children { get; }`
  Returns child `IMask` objects. `IMask` object may constitute the combination of some `IMask` objects.

Class `Mask`

It's comparable to element `mask`. See also `Load(Type)` method of `MessageProcessor`.

```
namespace Free.iso8583.config.attribute
public class Mask : IMask
```

- `public int StartByte { get; set; }`
  Comparable `start-byte` attribute of element `mask` whereas how many bytes to be masked is implied from the length `Value` or `MaskBytes` array.
- `public byte[] Value { get; set; }`
  Comparable `value` attribute of element `mask`.
- `public byte[] MaskBytes { get; set; }`
  Comparable `mask` attribute of element `mask`.
- `public MaskResult Result { get; set; }`
  Comparable `result` attribute of element `mask`.
- `IMaskConfig GetConfig(Type modelClass, MethodInfo processDelegate)`
  Implements `IMask` interface. It returns `MaskConfig` object.
- `IList<IMask> Children { get; }`
  Always returns `null` because it has no child.

Enum `MaskResult`

This enumeration specifies the possible values for `Result` property of class `Mask`.

```
namespace Free.iso8583.config.attribute
public enum MaskResult
```

- `ValueEquals = 0`
  Default value which means the masked bytes must equal to `Value` property.
- `Equals = 2`
  For value "`equals`" as defined by `result` attribute of element `mask`.
- `NotEquals = 3`
  For value "`notEquals`" as defined by `result` attribute of element `mask`.
- `Zero = 4`
  For value "`zero`" as defined by `result` attribute of element `mask`.
- `NotZero = 5`
  For value "`notZero`" as defined by `result` attribute of element `mask`.

Class `MaskOr`

It's comparable to element `or`. See also `Load(Type)` method of `MessageProcessor`.

```
namespace Free.iso8583.config.attribute
public class MaskOr : IMask
```

- `IMaskConfig GetConfig(Type modelClass, MethodInfo processDelegate)`
  Implements `IMask` interface. It returns `MaskOrConfig` object.
- `IList<IMask> Children { get; set; }`
  Contains child `IMask` objects.

Class `MaskAnd`

It's comparable to element `and`. See also `Load(Type)` method of `MessageProcessor`.

```
namespace Free.iso8583.config.attribute
public class MaskAnd : IMask
```

- `IMaskConfig GetConfig(Type modelClass, MethodInfo processDelegate)`
  Implements `IMask` interface. It returns `MaskAndConfig` object.
- `IList<IMask> Children { get; set; }`
  Contains child `IMask` objects.

Class `MessageToModelAttribute`

This attributes is attached to a method which returns `IMask` object. It's comparable to element `message-to-model`. See also `Load(Type)` method of `MessageProcessor` for further information of how to map an incoming message to a model object.

```
namespace Free.iso8583.config.attribute
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class MessageToModelAttribute : Attribute
```

- `public Type Model { get; set; }`
  Comparable to `model` attribute of element `message-to-model`.
- `public Type ProcessClass { get; set; }`
  The class which defines the method represented by the delegate as defined by `delegate` attribute of element `message-to-model`.
- `public string ProcessMethod { get; set; }`
  The name of method represented by the delegate as defined by `delegate` attribute of element `message-to-model`.

Configuration Objects

After the configuration is parsed/read, the configuration will be stored in some more 'understandable' objects. All objects created during parsing process will be destroyed, such as XML objects if the configuration is read from an XML file. The task to parse the configuration is the responsibility of Configuration Parser. All configuration objects should be thread-safe that will be used by many threads, should not change anything inside them after the configuration is stored in them. The configuration parsing process should be done once when the application starts or each time the configuration changes. The objects yielded by the parsing process should be alive as long as the application is running. All configuration objects are kept by a static class named `MessageConfigs`. This class is always sought by the message parser and message compiler when processing a message.

Next, it will be explained the relation between configuration objects and XML Configuration. However, this explanation also applies to the other type of configurations. You should just find the comparability between those configuration types. See further information in Attribute Configuration.

Generally, an element in XML Configuration has a class associated to it. The class will define the properties associated to each attribute in the element. If the element has child elements, the class associated to it, will define a property whose type of a collection to include all objects associated to each child element.

All configuration classes/types are placed in namespace `Free.iso8583.config`. In this namespace also, it is defined four delegates (`GetHeaderBytes`, `GetFieldBytes`, `GetPropertyValue` and `ProcessModel`). These four delegates represent element `delegate`. Each delegate defines different signature that can be referred by different element in the configuration. If there is a delegate specified by an element `delegate` whose the signature other than these four signatures, an exception will be raised by XML Configuration Parser when parsing the configuration.

Class `BitContentFieldConfig`

It represents element `field`.

```
namespace Free.iso8583.config
public class BitContentFieldConfig
```

- `public PropertyInfo PropertyInfo { get; set; }`
  Represents attribute `name`. It has been a `PropertyInfo` object of the defined property. It will make easier to get/set the property value.
- `public int Length { get; set; }`
  Represents attribute `length`.
- `public byte PadChar { get; set; }`
  Represents attribute `pad`.
- `public String Align { get; set; }`
  Represents attribute `align`.
- `public byte NullChar { get; set; }`
  Represents attribute `null`.
- `public bool IsTrim { get; set; }`
  Represents attribute `trim`. The attribute value has been converted to be a Boolean value. String '`true`' is Boolean `true`.
- `public bool IsOptional { get; set; }`
  Represents attribute `optional`. The attribute value has been converted to be a Boolean value. String '`true`' is Boolean `true`.
- `public TlvConfig Tlv { get; set; }`
  Represents attribute `tlv`. Because the attribute refers to an element `tlv`, it has been converted to a configuration object associated with that element `tlv`.
- `public String TlvTagName { get; set; }`
  Represents attribute `tlv-tag-name`.
- `public int TlvLengthBytes { get; set; }`
  Represents attribute `tlv-length-bytes`.
- `public Type TlvClassType { get; set; }`
  Represents attribute `tlv-class`. This property gets/sets a type reference whose name is defined by the attribute `tlv-class`.

Class `BitContentConfig`

It represents element `BitContent`.

```
namespace Free.iso8583.config
```

```
public class BitContentConfig
```

- `public String Id { get; set; }`
  Represents attribute `id`.
- `public Type ClassType { get; set; }`
  Represents attribute `class`. It has been a `Type` object as meant by the attribute.
- `public IList<BitContentFieldConfig> Fields { get; }`
  Contains all configuration objects representing all child element `field`s inside the element `BitContent` represented by this configuration object.

Class `TlvTagConfig`

It represents element `tag`.

```
namespace Free.iso8583.config
public class TlvTagConfig
```

- `public String Name { get; set; }`
  Represents attribute `name`.
- `public String Splitter { get; set; }`
  Represents attribute `splitter`.
- `public BitContentConfig BitContent { get; set; }`
  The configuration object of the element `BitContent` referred by attribute `bitcontent`. It will be `null` if the value of attribute `type` is not ‘`bitcontent`’.

Class `TlvConfig`

It represents element `tlv`.

```
namespace Free.iso8583.config
public class TlvConfig
```

- `public String Id { get; set; }`
  Represents attribute `id`.
- `public int LengthBytes { get; set; }`
  Represents attribute `length-bytes`.
- `public Type ClassType { get; set; }`
  The `Type` object referred by attribute `class`.
- `public IList<TlvTagConfig> Tags { get; }`
  A list of all configuration objects of the element `tag`s inside the element `tlv` represented by this configuration object.

- `public void AddTag(TlvTagConfig tagConfig)`
  Adds a tag to list `Tags`. It throws an exception if there is a tag whose the same name.
  Parameters:
    - `tagConfig` is the configuration of the added tag.

- `public TlvTagConfig GetTag(String tag)`
  Returns a configuration object of a tag.
  Parameters:
    - `tag` is the sought tag name. Corresponds to attribute `name` of element `tag`.

- `public TlvTagConfig GetTag(int index)`
  Returns a configuration object of a tag.
  Parameters:
    - `index` is the index of the sought tag inside list `Tags`.

- `public int TagsCount { get; }`
  The count of the element `tag`s inside the element `tlv` represented by this configuration object.

Interface **`IMessageHeaderConfig`**

All configuration classes for header items should implement this interface. The header items are represented by element `bitmap`, `message-type` and `header`. The message parser and message compiler will recognize all header configuration object as `IMessageHeaderConfig` object.

```
namespace Free.iso8583.config
public interface IMessageHeaderConfig
```

- `int Length { get; }`
  The count of bytes contained by the associated header. It represents attribute `length` of the elements of header items (element `bitmap`, `message-type` and `header`).

- `MessageElement GetNewHeader()`
  Returns an appropriate message container object for the associated header item. For example, it will return a `MessageBitMap` object for a bitmap header. It's used by message parser and message compiler to create the container for each header item. An implementing class should utilize property `Length` to check/determine the header length.

Class **`MessageBitMapConfig`**

It represents element `bitmap` or element `bit` whose `type` of `BitMap`.

```
namespace Free.iso8583.config
public class MessageBitMapConfig : MessageFieldConfig,
IMessageHeaderConfig
```

- `public MessageElement GetNewHeader()`
  Implements method `GetNewHeader` of interface `IMessageHeaderConfig`. Returns a corresponding `MessageBitMap` object. Property `Length` of the `MessageBitMap` object will be assigned the value of property `Length` of this configuration object.

Class `MessageTypeConfig`

It represents element `message-type`.

```
namespace Free.iso8583.config
public class MessageTypeConfig : IMessageHeaderConfig
```

- `public byte[] Value { get; set; }`
  Represents attribute `value`.
- `public int Length { get; set; }`
  Represents attribute `length`. Implements property `Length` of interface `IMessageHeaderConfig`. It implements the setting part by doing nothing. In the other words, it's read-only. The length value is taken from the length of property `Value`.
- `public MessageElement GetNewHeader()`
  Implements method `GetNewHeader` of interface `IMessageHeaderConfig`. Returns a corresponding `MessageTypeHeader` object. Using property `Value` passed to the constructor when creating the `MessageTypeHeader` object.

Class `MessageHeaderConfig`

It represents element `header`.

```
namespace Free.iso8583.config
public class MessageHeaderConfig : IMessageHeaderConfig
```

- `public String Name { get; set; }`
  Represents attribute `name`.
- `public String StringValue { get; set; }`
  Represents attribute `value`. It will throw an exception if it's set to `null` or the length of the assigned string is not twice of the value of property `Length`.
- `public byte[] Value { get; set; }`
  The bytes value of hexadecimal string represented by `StringValue` property.
- `public GetHeaderBytes GetFieldBytesFunc { get; set; }`
  Represents attribute `delegate`.
- `int Length { get; set; }`

Represents attribute `length`. Implements property `Length` of interface `IMessageHeaderConfig`.

- `public MessageElement GetNewHeader()`
  Implements method `GetNewHeader` of interface `IMessageHeaderConfig`. Returns a corresponding `MessageHeader` object. The value of `BytesValue` property of the `MessageHeader` object will be the same as the value of `Value` property. Property `GetFieldBytesFunc` of the `MessageHeader` object will refer the same delegate as referred by `GetFieldBytesFunc` property of this configuration object.

Class **`MessageFieldConfig`**

It represents element `bit`.

```
namespace Free.iso8583.config
public class MessageFieldConfig
```

- `public int Seq { get; set; }`
  Represents attribute `seq`.
- `public Type FieldType { get; set; }`
  The `Type` object of the message field container which is suitable with the value defined by attribute `type`. The appropriate container `type` will be determined by XML Configuration Parser.
- `public int Length { get; set; }`
  Represents attribute `length`. If the attribute is not specified then the value is negative.
- `public int LengthHeader { get; set; }`
  Represents attribute `length-header`. Negative value indicates attribute `length` is specified.
- `public bool FromRequest { get; set; }`
  Represent attribute `from-request`. The string “`true`” is the same as Boolean `true`.
- `public Delegate GetFieldBytesFunc { get; set; }`
  It’s a `delegate` object referred by attribute `delegate`. It is `null`, if the attribute is not specified.

Class **`MessageConfig`**

It represents element `message`.

```
namespace Free.iso8583.config
public class MessageConfig
```

- `public String Id { get; set; }`
  Represents attribute `id`.
- `public int LengthHeader { get; set; }`
  Represents attribute `length-header`.
- `internal IList<IMessageHeaderConfig> Headers { get; }`
  Contains all header items inside this message (It collects all configuration objects for element `bitmap`, `message-type` and `header`).
- `public IList<IMessageHeaderConfig> RoHeaders { get; }`
  It's the public access for `Headers`. It's read-only (cannot add/remove/replace the items inside this collection).
- `internal IDictionary<int, MessageFieldConfig> Fields { get; }`
  Contains all configuration objects of the fields inside the message represented by this configuration object.
- `public ICollection<MessageFieldConfig> RoFields { get; }`
  It's the public access for `Fields`. It's read-only (cannot add/remove/replace the items inside this collection).

Class `ModelPropertyConfig`

It represents element `property`.

```
namespace Free.iso8583.config
public class ModelPropertyConfig
```

- `public PropertyInfo PropertyInfo { get; set; }`
  It's the `PropertyInfo` object representing the property as mentioned by attribute `name`.
- `public MessageFieldConfig FieldBit { get; set; }`
  It's the configuration object for the message field to which this property is mapped. It corresponds to attribute `bit`.
- `public PropertyInfo GetValueFromBytes { get; set; }`
  Refers to an approprite property of the message field container class. Which property it refers to, depends on the value of attribute `type`. If the value is '`string`' then it refers to property `StringValue`. If the value is '`int`', it refers to property `IntValue`. If the value is '`decimal`', it refers to property `DecimalValue`. If the value is '`bytes`', it refers to property `BytesValue`. Which message field container class it refers to is known from the mapped message field configuration (consult property `FieldType` of `MessageFieldConfig`) . These properties are used by message parser to get the value of the corresponding message field.
- `public MethodInfo SetValueFromProperty { get; set; }`
  Refers to method `SetValue` of the message field container class which accepts the parameter as defined by attribute `type`. Which message field container class it refers to is

known from the mapped message field configuration (consult property `FieldType` of `MessageFieldConfig`) . The method is used by message compiler to set the value of a message field based on the mapped model property value.

- `public int FracDigits { get; set; }`
  Represents attribute `frac-digits`.

- `public Delegate GetPropertyValueFunc { get; set; }`
  It's a `delegate` object referred by attribute `delegate`.

- `public BitContentConfig BitContent { get; set; }`
  Represents attribute `bitcontent`.

- `public TlvConfig Tlv { get; set; }`
  Represents attribute `tlv`.

- `public String TlvTagName { get; set; }`
  Represents attribute `tlv-tag-name`.

- `public int TlvLengthBytes { get; set; }`
  Represents attribute `tlv-length-bytes`.

- `public Type TlvClassType { get; set; }`
  Refers to a `Type` object as defined by attribute `tlv-class`.

Class **`ModelConfig`**

It represents element `model`.

```
namespace Free.iso8583.config
public class ModelConfig
```

- `public String Id { get; set; }`
  Represents attribute `id`.

- `public Type ClassType { get; set; }`
  It's the `Type` object referred by attribute `class`.

- `public MessageConfig MessageCfg { get; set; }`
  It's the configuration object of the element `message` referred by attribute `message`.

- `internal Dictionary<int, ModelPropertyConfig> Properties { get; }`
  Contains all configuration objects representing all element `property` inside the element `model` represented by this configuration object. It's indexed by `Seq` value of the `MessageFieldConfig` that is mapped to this property.

- `public ICollection<ModelPropertyConfig> RoProperties { get; }`
  It's the public access for `Properties`. It's read-only (cannot add/remove/replace the items inside this collection).

- `public ModelConfig Clone()`
  This method will be invoked by XML Configuration Parser when creating a configuration object for a child model. The invoked method belongs to the parent model. A model becomes a child model if attribute `extend` is specified. This method returns a copy of this configuration object. It implies that the child model will have its own configuration object independent from the parent model but has similar state to the configuration object owned by the parent model.

- `public void AddOrSubtituteProperty(ModelPropertyConfig propCfg)`
  When parsing a child model, this method will be invoked by XML Configuration Parser to insert an item to the list `Properties`. This method will replace the existing property whose the same name as parameter `propCfg`'s. If there is no such property, parameter `propCfg` will be added to the list.

- `public ModelPropertyConfig GetPropertyMappedToField(int seq)`
  Gets `ModelPropertyConfig` object owned by this `ModelConfig` that is mapped to the message field on `seq`-th position. It means to find `ModelPropertyConfig` object inside this object whose `Seq` value of its `FieldBit` property is the same as parameter `seq` value. It returns `null` if no such property.

Interface `IMaskConfig`

It defines the interface to get the result of masking bits operation. Masking bits operation is done when there is an incoming message. It is to determine which model should be used to represent the message. This interface is implemented by classes which represent element `message-to-model` and its child elements (element `and`, `or` and `mask`). Only element `mask` defines how to mask the message bits specifically. However, element `and`, `or` and `message-to-model`, that can contain some element `mask`s, will execute an operation involving all result of element `mask` inside them. Therefore, the classes representing these elements should also implement this interface.
Note, element `message-to-model` can also contain element `and` and `or`. Element `and` can also contain element `or` and vice versa. Therefore, element `mask` may also be operated with element `and` or `or`. By implementing this interface, the process will be easier because it only invokes method `IsQualified` regardless whether the operand is element `mask`, `and` or `or`.

```
namespace Free.iso8583.config
public interface IMaskConfig
```

- `bool IsQualified(byte[] message)`
  Returns `true` if the incoming message is qualified.
  Parameters:
    - `message` is the incoming message to be tested.

Interface `IMaskListConfig`

It defines a list of `IMaskConfig` object. It is implemented by the classes representing element `and`, `or` and `message-to-model`. These elements can include the other elements which implement interface `IMaskConfig`.

```
namespace Free.iso8583.config
public interface IMaskListConfig
```

- `IList<IMaskConfig> MaskList { get; }`
  Contains `IMaskConfig` objects which are the child elements of implementing object. This list is accessed by XML Configuration Parser to include all child elements contained by element `and`, `or` and `message-to-model`.

Class `MaskConfig`

It represents element `mask`.

```
namespace Free.iso8583.config
public class MaskConfig : IMaskConfig
```

- `public static int MinBytesCountToCheck { get; internal set; }`
  It's the minimum count of the beginning bytes of the incoming message that can be checked to find the appropriate `MessageToModelConfig` object for this incoming message. It is the highest number calculated from the value of attribute `start-byte` and attribute `length`. So, it doesn't need to wait the whole message to check, it only needs `MinBytesCountToCheck` bytes to find the appropriate `MessageToModelConfig`. Specifically, this property is used by method `Receive` of `MessageProcessorWorker`. This method needs to know the length of message. This length value is placed in some bytes (length header) in the beginning of message. The bytes count of this length header is known from the configuration. It doesn't need to wait the entire message (that it doesn't know how long the message) but it only needs to wait `MinBytesCountToCheck` bytes to get the true configuration. The length header of an ISO 8583 message should always have two bytes. It's specified in the configuration to prepare the future change.

- `public int StartByte { get; set; }`
  Represents attribute `start-byte`.

- `public int Length { get; set; }`
  Represents attribute `length`.

- `public bool ValueEquals(byte[] message)`
  Checks `Length` bytes in the message starting position `StartByte` if it is the same as `Value`. It will return `true` if the value are the same as each other.
  Parameters:
    - `message` is the message to be tested.

- `public bool MaskEquals(byte[] message)`
  Executes bitwise logical "and" operation between `Length` bytes in the message starting position of `StartByte` and `Value`. It will return `true` if the result is the same as `Value`.
  Parameters:

  - `message` is the message to be tested.

- `public bool MaskNotZero(byte[] message)`
  Executes bitwise logical "and" operation between `Length` bytes in the message starting position of `StartByte` and `Value`. It will return `true` if the result is not zero.
  Parameters:
  - `message` is the message to be tested.

- `public byte[] Value { get; set; }`
  The bytes value of hexadecimal string represented by `StringValue` property.

- `public String StringValue { get; set; }`
  Represents attribute `value` or attribute `mask`, depending on which one is specified. It will throw an exception if hexadecimal value represented by the assigned value has more digits than that is specified by property `Length` (greater than twice of `Length`). If its length is less than it should be then it will be padded by "0" at the left. It will also throw an exception if the assigned value is invalid hexadecimal value.

- `public void SetValue(String value, String modelId, String delegateId)`
  It will be invoked by XML Configuration Parser to set `StringValue`. It is used to set also the location of the associated element `mask`. This location will be used in the error message if one happens.
  Parameters:
  - `value` is the assigned value for `StringValue`.
  - `modelId` should be the same as attribute `model` value of element `message-to-model` in which this element `mask` exists.
  - `delegateId` should be the same as attribute `delegate` value of element `message-to-model` in which this element `mask` exists.

- `public String MaskResult { get; set; }`
  Represents attribute `result`.

- `public bool IsQualified(byte[] message)`
  Implements `IMaskConfig` interface. It will invoke `ValueEquals` if attribute `value` is specified. It will invoke `MaskEquals` if attribute `mask` is specified and attribute `result` value is "`equals`". It will invoke `MaskNotZero` if attribute `mask` is specified and attribute `result` value is "`notZero`".
  Parameters:
  - `message` is the message to be tested.

Class `MaskAndConfig`

It represents element `and` inside element `message-to-model`. It does logical "and" operation between all factors when testing a message.

```
namespace Free.iso8583.config
public class MaskAndConfig : IMaskConfig, IMaskListConfig
```

- `public IList<IMaskConfig> MaskList { get; }`
  Implements `IMaskListConfig` interface.
- `public bool IsQualified(byte[] message)`
  Implements `IMaskConfig` interface. Invokes method `IsQualified` of each item inside `MaskList` and then operates their returned values by using logical "and" operation. Returns the result of the operation.
  Parameters:
    - `message` is the message to be tested.

Class `MaskOrConfig`

It represents element `or` inside element `message-to-model`. It does logical "or" operation between all factors when testing a message.

```
namespace Free.iso8583.config
public class MaskOrConfig : IMaskConfig, IMaskListConfig
```

- `public IList<IMaskConfig> MaskList { get; }`
  Implements `IMaskListConfig` interface.
- `public bool IsQualified(byte[] message)`
  Implements `IMaskConfig` interface. Invokes method `IsQualified` of each item inside `MaskList` and then operates their returned values by using logical "or" operation. Returns the result of the operation.
  Parameters:
    - `message` is the message to be tested.

Class `MessageToModelConfig`

It represents element `message-to-model`. It behaves like `MaskOrConfig`, that is, it will also execute logical "or" operation. Therefore, it inherits `MaskOrConfig`. Extensions defined by this class are only the attributes which apply to element `message-to-model`.

```
namespace Free.iso8583.config
public class MessageToModelConfig : MaskOrConfig
```

- `public ModelConfig ModelCfg { get; set; }`
  Represents attribute `model`.
- `public Delegate ProcessModel { get; set; }`

Represents attribute `delegate`.

Class `MessageConfigs`

This class maintains all configuration objects so that they can be alive as long as the application is running. The configuration objects are stored in some collections. These collections are fed by XML Configuration Parser. These configuration objects' states represent the configuration written in XML Configuration. Using XML objects is inefficient (not fast) because the configuration is always read each time there is an incoming/outgoing message. Therefore, they must be converted to these configuration objects. This converting (parsing) process, that is very time consuming, should be done once when the application starts and each time the configuration changes. Once the configuration changes, reparsing process can be asked again by using method `Load` of `MessageProcessor` object.

```
namespace Free.iso8583
internal static class MessageConfigs
```

- `public static IDictionary<String, Type> Types { get; }`
  Contains all `Type` object defined by all element `type`. This dictionary is keyed by attribute `id` value of corresponding element `type`.

- `public static IDictionary<String, GetHeaderBytes> HeaderDelegates { get; }`
  Contains `delegate` object defined by all element `delegate` that can be referred by attribute `delegate` of element `header`. This dictionary is keyed by attribute `id` value of corresponding element `delegate`. Each `delegate` inside this collection is an instance object of `GetHeaderBytes`.

- `public static IDictionary<String, Delegate> FieldDelegates { get; }`
  Contains `delegate` object defined by all element `delegate` that can be referred by attribute `delegate` of element `bit`. This dictionary is keyed by attribute `id` value of corresponding element `delegate`. Each `delegate` inside this collection is an instance object of `GetFieldBytes<>`.

- `public static IDictionary<String, Delegate> PropertyDelegates { get; }`
  Contains `delegate` object defined by all element `delegate` that can be referred by attribute `delegate` of element `property`. This dictionary is keyed by attribute `id` value of corresponding element `delegate`. Each `delegate` inside this collection is an instance object of `GetPropertyValue<>`.

- `public static IDictionary<String, Delegate> ProcessDelegates { get; }`
  Contains `delegate` object defined by all element `delegate` that can be referred by attribute `delegate` of element `message-to-model`. This dictionary is keyed by attribute

id value of corresponding element delegate. Each delegate inside this collection is an instance object of ProcessModel<,>.

- public static IDictionary<String, BitContentConfig> BitContents { get; }
  Contains all BitContent field configuration defined by element BitContent. This dictionary is keyed by attribute id value of the corresponding element BitContent.

- public static IDictionary<String, TlvConfig> Tlvs { get; }
  Contains all TLV field configuration defined by element tlv. This dictionary is keyed by attribute id value of the corresponding element tlv.

- public static IDictionary<String, MessageConfig> Messages { get; }
  Contains all message configuration defined by element message. This dictionary is keyed by attribute id value of the corresponding element message.

- public static IDictionary<String, ModelConfig> Models { get; }
  Contains model configurations defined by element model which its attribute id is specified. This dictionary is keyed by attribute id value of the corresponding element model.

- public static IDictionary<Type,ModelConfig> ClassToModels { get; }
  Contains the mapping between the model Type object and its configuration object. It is used by message compiler to find the appropriate model configuration when compiling a model object to be a message. It accepts a model object and then uses the model Type as the key to find the right configuration.

- public static IList<MessageToModelConfig> MessageToModels { get; }
  Contains all configuration objects defined by element message-to-model.

- public static MessageToModelConfig GetQulifiedMessageToModel(byte[] message)
  Finds the appropriate MessageToModelConfig object for a message. Seeks an appropriate item in MessageToModels. It is used by message parser to parse an incoming message. Note that there may be more than one MessageToModelConfig which matches the message. If it happens, this method returns the first matched one. So, place the more general filter after the less general one in XML configuration. It will make the less general filter never misses the message.
  Parameters:
    - message is the message to be checked.

- public static void Clear()
  Removes all configuration object collections. This method is invoked by method Load of MessageProcessor before parsing the configuration. It makes sure that all configuration objects represent the parsed configuration.

Among all collections above, only `ClassToModels` and `MessageToModels` are used intensively by message compiler and message parser. The other collections are only used by XML Configuration Parser to ease its task for parsing, which is actually able to be removed safely after parsing.

Delegate `GetHeaderBytes`

It defines signature for a function that can be used to get bytes for a message header, especially used by message compiler. The function is referred by attribute `delegate` of element `header` in the configuration.

```
namespace Free.iso8583.config
public delegate byte[] GetHeaderBytes()
```

This function returns an array of bytes for the header value. This function may not return `null`.

Delegate `GetFieldBytes`

This delegate is used for getting the value of a message field. Specially, it's used by message compiler. The function implementing this delegate can be referred by attribute `delegate` of element `bit` in the configuration.

```
namespace Free.iso8583.config
public delegate byte[] GetFieldBytes<P>(P value)
```

This function takes a parameter which is the value of the mapped model property and returns an array of bytes for the field value.

Delegate `GetPropertyValue`

It defines signature for a function that can be used to get the value for a model property. Typically, it's used by message parser. The function can be referred by attribute `delegate` of element `property` in the configuration.

```
namespace Free.iso8583.config
public delegate R GetPropertyValue<R>(byte[] bytes)
```

This function takes a parameter which is the bytes of the mapped message field and returns the value for the model property.

Delegate `ProcessModel`

It defines signature for a function that can be used by message parser to process a model yielded from a message parsing. In this function, the business process is executed.

```
namespace Free.iso8583.config
public delegate R ProcessModel<R,P>(P model)
```

This function takes a model as the only parameter and returns another model for the reply message. It returns `null` if no reply.

Configuration Parser

This part is responsible to parse configuration. The result of the parsing is some configuration objects which are navigable much easier than the objects which are used to write the configuration such as XML or attribute classes. It causes much more efficient process in parsing and compiling message. The configuration objects are used intensively by message parser and message compiler. The configuration objects are stored in some collections maintained by class `MessageConfigs`. The configuration objects will be alive as long as class `MessageConfigs` is not destroyed (it's likely starting from this library is loaded up to the application stops).

The configuration parser should be invoked before using message processor. It's invoked by executing method `Load` of `MessageProcessor`. If the configuration parser finds any bad configuration (such as invalid value for an attribute, required attribute is not specified, a child XML element is placed inside an inappropriate parent element etc), it will throw an exception of class `ConfigParseException`. It should cause the application shuts down (it cannot process ISO 8583 message) because the exception will be propagated by method `Load` of `MessageProcessor`. The configuration parser will declare an error message as clearly as possible. It tries to locate where the bad configuration exists. Any rules and constraints about the configuration are explained in the section Configuration.

Interface `IConfigParser`

All configuration parser classes must implement this interface. This interface object is used by `Load` method of `MessageProcessor` in which the process of reading the configuration begins. The reading process begins when `Parse` method is invoked.

```
namespace Free.iso8583.config
internal interface IConfigParser
```

- `void Parse()`
  Reads the configuration and translates it to be the configuration objects.

Class `ConfigParser`

Instead of implementing `IConfigParser` interface, a configuration class may inherit this abstract class. This class defines some global methods that can be used by a configuration parser class.

```
namespace Free.iso8583.config
internal abstract class ConfigParser : IConfigParser
```

- `public abstract void Parse()`
  As defined by `IConfigParser.Parse` method.

- `protected static Object GetInstanceOf(Type type)`
  Creates an instance object of a type as defined by parameter. This method is usually used when creating a delegate object.
  Parameters:
  - `type` is the type of object that will be created.

- `protected static void HookPropertyToField(ModelPropertyConfig propertyConfig, MessageFieldConfig fieldConfig, string fieldType)`
  To connect a model property to the corresponding message field.
  Parameters:
  - `propertyConfig` is model property configuration object.
  - `fieldConfig` is message field configuration object.
  - `fieldType` is a string that represents the field type. The possible values are “string”, “int”, “decimal” and “bytes”.

XML Configuration Parser

XML configuration parser is to read the configuration using XML format. Further information how to configure the message processor using XML, see section XML Configuration.

Class `XmlConfigParser`

This class represents XML configuration parser. Method `Load` of `MessageProcessor` creates an instance object of this class and then calls its method `Parse` to execute XML configuration parsing.

```
namespace Free.iso8583.config
internal class XmlConfigParser : ConfigParser
```

- `public XmlConfigParser(Stream fileConfig)`
  This object must be created along with XML configuration file as the parameter.
  Parameters:
  - `fileConfig` is the file stream of XML configuration that will be parsed.

- `private XmlElement GetElementById(String id)`
  Gets the element with the specified id. It returns `null` if no element whose the specified id. This is created because `GetElementById` method of `XMLDocument` provided by .NET framework requires `id` attribute to be specified in DTD. This method avoids that requirement. Actually, this XML configuration parser collects all elements which has declared id and stores them in a dictionary keyed by their id at the beginning process. Afterwards, this method will search that dictionary. The id must be unique over all elements.
  Parameters:

  - `id` is the id of sought element.

- `public void Parse()`
  Triggers parsing process execution. After creating this object, the consumer must invoke this method to parse the configuration. It collects all elements which declare an id that will be used afterwards by method `GetElementById`. Then it invokes method `ParseTypes`, `ParseDelegates`, `ParseBitContents`, `ParseTlvs`, `ParseMessages`, `ParseModels` and `ParseMessageToModels` consecutively. This method throws an exception if either the root element is not `MessageMap`, there is an unknown tag or duplicate id of element. The id must be unique over all elements.

- `private static String GetRequiredAttribute(XmlElement elm, String attrName, String elmId)`
  Tries to get the value of a required attribute. If the value is not specified, this method will throw an exception. Defining and using this method reduces repetitive code to check the attribute's value and throw an exception.
  Parameters:
  - `elm` is the element object which wants to get its attribute value.
  - `attrName` is the attribute name which its value will be obtained.
  - `elmId` is the id of the element (parmeter `elm`). It's `null` if the element does not declares the id or it does gets the value of attribute `id`.

- `private static int GetNonNegativeIntValue(String str, String attrName, String elmName)`
  Tries to get a non-negative integer value of an attribute. It throws an exception if the value is invalid to be evaluated as integer or the value is negative. Defining and using this method reduces repetitive code to check the attribute's value and throw an exception. This method is usually called after method `GetRequiredAttribute`.
  Parameters:
  - `str` is the attribute's value as string. In XML, the attribute's value is always string. It's usually the value returned by method `GetRequiredAttribute`.
  - `attrName` is the attribute's name for error message.
  - `elmName` is the element name which owns the attribute. It's for error message.

- `private static int GetNonNegativeIntValue(XmlElement elm, String attrName, int? defaultVal)`
  Tries to get a non-negative integer value of an attribute. The attribute is not required. Therefore, the value may be undefined. Parameter `defaultVal` is the default value if the value is not defined and this parameter is not `null`. If `null`, this method will return -1. If the value is defined, it will call the above method `GetNonNegativeIntValue`.
  Parameters:
  - `elm` is the element object which owns the attribute that its value will be read.
  - `attrName` is attribute name that its value will be read.
  - `defaultVal` is the default value if the attribute is not defined. If this parameter is `null`, this method will return -1.

- `private static bool GetBoolValue(XmlElement elm, String attrName, bool defaultVal)`
  Converts the value of an attribute (that is a string) to be a Boolean value. It converts string 'true' to be Boolean `true` and string 'false' to be Boolean `false`. If the attribute value is other than 'true' and 'false', it will throw an exception.
  Parameters:
  - `elm` is the element object which wants to get its attribute value.
  - `attrName` is the attribute name which its value will be obtained.
  - `defaultVal` is default value if the attribute is not specified.
- `private static String GetStringValue(XmlElement elm, String attrName, String defaultVal)`
  Returns the value of an attribute as a string. It will not need any conversion because the attribute value is always string. Another task of this method returns a default value if the attribute is not defined.
  Parameters:
  - `elm` is the element object which wants to get its attribute value.
  - `attrName` is the attribute name which its value will be obtained.
  - `defaultVal` is the default value if the attribute is not defined.
- `private void ParseTypes()`
  Reads all element `type`s. It creates all `Type` objects defined by all element `type`s. These `Type` objects are inserted to collection `Types` of `MessageConfigs`.
- `private static Type GetTypeFromName(String name)`
  Gets a `Type` object whose name specified by parameter. Firstly, this method will search in collection `Types` of `MessageConfigs`. The name is considered as the id of an element `type`. If not found, it uses `Type.GetType` to try to get the `Type` object. If no matching `Type` object found, it throws an exception. This method is used to interpret various `class` attribute in XML configuration.
  Parameters:
  - `name` is the name of `type`.
- `private static PropertyInfo GetProperty(Type type, String propName, String locMsg)`
  Gets the `PropertyInfo` object of a property. This method is used for the element representing a property, such as element `property` and element `field`. It throws an exception if the specified property doesn't exist.
  Parameters:
  - `type` is the `Type` object owning the property. It's based on the attribute `class` of element `model` and `BitContent`.
  - `propName` is the property name. It's the same as the value of attribute `name` of element `property` or `field`.
  - `locMsg` is for the error message. It should be the location of the element

specifying property.

- `private static Object GetInstanceOf(Type type)`
  Creates and returns an instance object of a `type`. Firstly, it will try to find a static method named `GetInstance` owned by the `type`. It is to anticipate if the `type` is singleton. If not found, it uses method `Activator.CreateInstance` to create a new instance. This method is used when creating the delegate (invoked by method `ParseDelegates`). Sometimes the method specified by an element `delegate` is not static which needs an instance object to get this delegate.
  Parameters:
    - `type` is the `type` which will be created an instance of it.

- `private void ParseDelegates()`
  Reads all element `delegate`s. It creates `delegate` objects specified by those elements. There are four correct signatures for delegate that can be referred by attribute `delegate` of element `header`, `bit`, `property` or `message-to-model`. If a delegate doesn't have a matching signature, this method will raise an exception. This method includes the delegates into collection `HeaderDelegates`, `FieldDelegates`, `PropertyDelegates` and `ProcessDelegates` of `MessageConfigs`. Different signature will fill different collection.

- `private void ParseBitContents()`
  Creates `BitContentConfig` objects defined by all element `BitContent`s. For each element `BitContent`, it will invoke method `ParseBitContentFields` to read all element `field`s contained by this element. It feeds collection `BitContents` maintained by `MessageConfigs`.

- `private void ParseBitContentFields(XmlElement elm, BitContentConfig cfg)`
  Creates `BitContentFieldConfig` objects defined by element `field`s contained by an element `BitContent`. It is invoked by method `ParseBitContents` for each element `BitContent`. The created `BitContentFieldConfig` objects are inserted into collection `Fields` of parent `BitContentConfig`. It will throw an exception if there is a violated constraint of defining attributes (see element `field`). Attribute `pad` and `null` must define one character. Otherwise, it throws an exception.
  Parameters:
    - `elm` is the element `BitContent` that contains some element `field`s.
    - `cfg` is the `BitContentConfig` object representing element `BitContent` passed as parameter `elm`.

- `private void ParseTlvs()`
  Creates `TlvConfig` objects defined by all element `tlv`s. For each element `tlv`, method `ParseTlvTags` is invoked to interpret all element `tag`s contained by this element. It feeds collection `Tlvs` of `MessageConfigs`.

- `private void ParseTlvTags(XmlElement elm, TlvConfig cfg)`
  Creates `TlvTagConfig` objects defined by element `tag`s contained by an element `tlv`. It is invoked by method `ParseTlvs` for each element `tlv`. The created `TlvTagConfig` objects are inserted into collection `Tags` of parent `TlvConfig` by calling method

AddTag. All length of tags must be the same. If not, it will throw an exception. It also throws an exception if its type is 'bitcontent' but its bitcontent is not defined or it refers to not existing BitContent element. An exception is also raised if there is a duplicate tag.
Parameters:

- elm is the element tlv which contains some element tags.
- cfg is the TlvConfig object representing element tlv passed as parameter elm.

- `private void SetTlvForBitContentFields()`
  It's invoked by method ParseTlvs after creating all TlvConfig objects to set property Tlv of all BitContentFieldConfig objects that has been created. It's because all TlvConfig objects (needed by property Tlv) are created after all BitContentFieldConfig objects are created.

- `private void ParseMessages()`
  Reads all element messages and creates MessageConfig objects corresponding with each. It invokes ParseMessageHeaders and ParseMessageFields method for each element message to read all header and field elements inside this element. It feeds collection Messages of MessageConfigs. It throws an exception if a constraint is violated.

- `private void ParseMessageHeaders(IList<XmlElement> headers, IList<int> headerType, MessageConfig cfg)`
  Creates configuration objects for all header elements inside an element message (consult IMessageHeaderConfig, MessageBitMapConfig, MessageTypeConfig and MessageHeaderConfig). It is invoked by method ParseMessages. The created configuration objects are inserted into collection Headers of parent MessageConfig. It throws an exception if there is a violated constraint.
  Parameters:
  - headers is the list of header elements that their configuration objects will be created. It should only contain element header, message-type and bitmap.
  - headerType is the type of header element in parameter headers at the same index (0=header, 1=message-type, 2=bitmap). It's used to determine the appropriate header configuration class (MessageHeaderConfig for header, MessageTypeConfig for message-type, MessageBitMapConfig for bitmap).
  - cfg is the MessageConfig object representing the parent element message.

- `private void ParseMessageFields(IList<XmlElement> fields, MessageConfig cfg)`
  Creates MessageFieldConfig objects defined by element bits contained by an element message. It is invoked by method ParseMessages for each element message. The created MessageFieldConfig objects are inserted into collection Fields of parent MessageConfig. It throws an exception if a constraint is violated.
  Parameters:

    - fields is a list of all element bits which their MessageFieldConfig objects will be created.
    - cfg is the configuration object of the parent element message.

- private void ParseModels()
  Parses all element models that is not a child of another model (it doesn't define attribute extend). For each element model, it calls method ParseModel to create its ModelConfig object, method ParseModelProperties to parse all properties of it and method RegisterModel to keep its ModelConfig object alive. Actually, it will iterate all element models but when it finds a child model, it prepares it to be processed as a child model. At the end, it invokes method ParseChildModels to process all child models. It throws an exception if there is a violated constraint. A child model must refer to an existent parent model is a constraint.

- private void ParseChildModels()
  Creates ModelConfig objects for all child models (element models which define attribute extend). For each child model, it calls method Clone of parent ModelConfig to create its ModelConfig object. Then it executes method ParseModelProperties to parse all element property inside it. It will reconfigure all inherited property configuration objects (obtained from method Clone) in the case its attribute class or message value differs from its parent. Reconfiguration is needed because some attributes of the ModelPropertyConfig object for each property must change. Primary example is PropertyInfo which must change because the Type object (specified by attribute class) has changed. Calling method ParseModelProperties functions to add new properties or substitutes the same property of parent model. As ParseModels, it invokes method RegisterModel to make its ModelConfig object maintained. Because a parent model can be the child of another model, it uses algorithm to create the top most ancestor so that method Clone can be executed successfully. If it finds the circular reference, that is a child model refers to a parent model that is actually the descendant of that child model, it will throw an exception. It throws an exception if there is a violated constraint.

- private ModelConfig ParseModel(XmlElement elm)
  Interprets an element model to be ModelConfig object and returns it. This method is invoked by method ParseModels and ParseChildModels. It throws an exception if there is a violated constraint including when it refers to non-existent element message. Attribute class and message are optional if it is a child model. If not specified, it is inherited from parent model.
  Parameters:
    - elm is the element model to be interpreted.

- private void RegisterModel(ModelConfig cfg)
  It feeds collection ClassToModels of MessageConfigs, that is, it sets the mapping between the model Type object and its configuration object. The configuration object is passed as parameter and model Type is taken from property ClassType of the parameter. If attribute id is specified then it also feeds collection Models of MessageConfigs.
  Parameters:
    - cfg is ModelConfig object to be maintained.

- `private void ParseModelProperties(XmlElement elm, ModelConfig cfg, bool subtitute)`
  Creates `ModelPropertyConfig` objects for all element `property` inside an element `model`. All created objects are included into collection `Properties` of parent `ModelConfig`. This method is invoked by method `ParseModels` and `ParseChildModels`. It throws an exception if there is a violated constraint.
  Parameters:
    - `elm` is an element `model` which becomes the parent element of all processed element `property`.
    - `cfg` is the configuration object for element `model` specified by parameter `elm`.
    - `subtitue` must be `true` if the processed model is a child model. It is to make aware that it may overwrite the parent property. A child model must invoke method `AddOrSubtituteProperty`.
- `private void ParseMappedMessageField(IDictionary<int, ModelPropertyConfig> propCfgs, IDictionary<int, String> types, bool subtitute, ModelConfig cfg)`
  Sets property `GetValueFromBytes` and `SetValueFromProperty` of each `ModelPropertyConfig` object passed as parameter. It concerns attribute `type` and `bit`. It throws an exception if there is a property referring to non-existent message field. This method is invoked by method `ParseModelProperties` and `ParseChildModels`. This method is defined to reduce repetitive code in both calling method. Method `ParseChildModels` needs this method when reconfiguring all inherited properties.
  Parameters:
    - `propCfgs` is the list of `ModelPropertyConfig` objects to be set. This dictionary is indexed by `Seq`'s value of the corresponding `MessageFieldConfig`.
    - `types` is the values of attribute `type` corresponding to each `ModelPropertyConfig` object at the same index in parameter `propCfgs`.
    - `subtitute` must be `true` if the processed model is a child model. Otherwise, it's `false`.
    - `cfg` is the configuration object of the element `model` containing the processed element `property`. All `ModelPropertyConfig` objects in `propCfgs` will be added to property `Properties` of `cfg`.
- `private void ParseMessageToModels()`
  Creates `MessageToModelConfig` objects for all element `message-to-model`s. For each element `message-to-model`, it calls method `ParseMasks` to parse all elements contained by this element. It feeds collection `MessageToModels` of `MessageConfigs`. It raises an exception if a violated constraint exists, including it refers to non-existent element `model` or invalid element `delegate`.
- `private void ParseMasks(XmlElement elm, IMaskListConfig cfg, String modelId, String delId)`
  Creates the configuration objects that implement `IMaskConfig` interface for the elements contained by the element specified by parameter `elm`. If it finds element `and` or

element `or`, this calls itself (recursive process) to process all contained elements in this element `and` or element `or`. It raises an exception if there is a violated constraint.
Parameters:

- `elm` is the element that contains the elements which their configuration objects will be created. It can be element `message-to-model`, element `and` or element `or`.
- `cfg` is the configuration object for `elm`.
- `modelId` is the value of attribute `model` of containing element `message-to-model`.
- `delId` is the value of attribute `delegate` of containing element `message-to-model`.

Attribute Configuration Parser

This parser is to read the configuration in the format of attributes/annotations which are attached to the model classes. For further information about the configuration, see section Attribute Configuration.

Class `AttributeConfigParser`

This class is the engine of attribute configuration parser. To use this class, calls `Load(Type)` method of `MessageProcessor`.

```
namespace Free.iso8583.config
internal class AttributeConfigParser : ConfigParser
```

- `private AttributeConfigParser(Type messageToModelMapping)`
  To create this class, it needs parameter containing the configuration information.
  Parameters:
  - `messageToModelMapping` is a class whose attributes for starting point to read the configuration. See `Load` method of `MessageProcessor`.
- `public void Parse()`
  Implements `IConfigParser.Parse`.

Exception

This library provides some exception classes that will be used if there is any error pertaining to executing code inside this library. `MessageProcessorException` is the root exception. It inherits class `Exception` directly. The other exceptions inherit this exception. By this condition, it can be checked that all exceptions raised by this library are always `MessageProcessorException` object or its descendant.

Class `MessageProcessorException`

It will be thrown if there is any exception in executing message processor or in setting message container.

It is also the general exception class should be used if no specific exception class defined.

Class `ConfigParserException`

This exception will be thrown when any bad configuration is found. It happens when parsing configuration by configuration parser.

Class `MessageParserException`

The exception of this class will be thrown if any failure happens in parsing message process. It is thrown by message parser.

Class `MessageCompilerException`

The exception of this class will be thrown if any failure happens in compiling message process. It is thrown by message compiler.

Class `MessageListenerException`

It's used in message listener.

Logger

This logger is used to log errors happening when processing the message or to log events such as a request comes. When parsing the configuration, this logger is also used to log the error before the message processor closes. Class `Logger` is the main class that is assigned to write the log. The log can be written into console screen, file or another media based on the application setting. The application can set the logger output by using method `AddOutput` or `ReplaceOutput`. The application may have some logger output media.

Enum `LogLevel`

This enumeration is used by logger to determine the severity of log. By this severity, we know how important the message is. Generally, "error" (the highest severity) needs attention to be fixed because there is something not running in the system. Otherwise, "notice" (the lowest severity) is only the information message that can be ignored. By this severity, logger may also filter which logs that are displayed/written into logger output media (see property `Level`).

```
namespace Free.iso8583
public enum LogLevel
```

- `Notice = 0`
  For "notice" log. Just for information, can be ignored but may be useful for audit.

- `Warning = 1`
  For "warning" log. There is something which should not be. However, the system is still

running well.

- `Error = 2`
  For “error” log. There is something wrong that causes the system failed.
- `MuteLog = 99`
  to mute all logs. If it’s used for property `Level` then no message will be logged. If it’s used as the parameter for method `Write` or `WriteLine` then the log will be ignored.

Interface `ILoggerOutput`

All logger output objects must implement this interface so that class `Logger` can use them. This interface defines the methods to write the log. This interface also uses enum `LogLevel` to give the implementation class a chance to manage its own rule which level should be maintained. If there are more than one output media, an output may only maintain error messages while another one maintains warning messages. They are, however, still impacted by the setting of property `Level`.

```
namespace Free.iso8583
public interface ILoggerOutput
```

- `void WriteLine(String log, LogLevel level)`
  Writes the log to the output media and ended by new line character(s). The new line character(s) will be appended automatically at the end of log.
  Parameters:
  - `log` is the log to be written.
  - `level` is the severity level of log.
- `void WriteLine(String log)`
  It should call above `WriteLine` by passing `LogLevel.Notice` as second parameter.
- `void Write(String log, LogLevel level)`
  Writes the log to the output media without appending new line character(s). If parameter `log` is ended by new line character(s), this new line character(s) will still be written.
  Parameters:
  - `log` is the log to be written.
  - `level` is the severity level of log.
- `void Write(String log)`
  It should call above `Write` by passing `LogLevel.Notice` as second parameter.

Class `ConsoleLoggerOutput`

It is the basic/common class of logger output. It writes the log to console screen. By default, this class will be used by class `Logger` as the output.

```
namespace Free.iso8583
public class ConsoleLoggerOutput : ILoggerOutput
```

- `public void WriteLine(String log, LogLevel level)`
  Implements `ILoggerOutput` interface. The second parameter is ignored.

- `public void WriteLine(String log)`
  Implements `ILoggerOutput` interface.

- `public void Write(String log, LogLevel level)`
  Implements `ILoggerOutput` interface. The second parameter is ignored.

- `public void Write(String log)`
  Implements `ILoggerOutput` interface.

Class `LoggerOutputList`

As mentioned before, the logger may have some outputs. This class is to encapsulate all output objects owned by the logger. This class implements interface `ILoggerOutput`, that is, it can be used as output media which will write the log to all output objects contained by it. By favor of this object, `Logger` object will not make any difference when writing a log either to one or more output. It still invokes only one method `Write` or `WriteLine` of `ILoggerOutput`. An object of this class will be created automatically, if not exists, when method `AddOutput` is invoked. Invocation of method `AddOutput` afterwards is actually to add the output object to this `LoggerOutputList` object. `Logger` object itself only knows that the used output object is this `LoggerOutputList` object.

```
namespace Free.iso8583
public class LoggerOutputList : ILoggerOutput
```

- `public IList<ILoggerOutput> List { get; }`
  Contains all output objects encapsulated by this object.

- `public void WriteLine(String log, LogLevel level)`
  Implements `ILoggerOutput` interface. It writes the log to all output objects contained by this object.

- `public void WriteLine(String log)`
  Implements `ILoggerOutput` interface. It writes the log to all output objects contained by this object.

- `public void Write(String log, LogLevel level)`
  Implements `ILoggerOutput` interface. It writes the log to all output objects contained by this object.

- `public void Write(String log)`
  Implements `ILoggerOutput` interface. It writes the log to all output objects contained by

this object.

Class `Logger`

This is the main class in writing the log. The application must get an instance of this class to write the log. This class defines method `Write` and `WriteLine` to write the log. Only one instance may exist (a singleton object). The only instance can be retrieved from method `GetInstance`.

```
namespace Free.iso8583
public class Logger
```

- `public static Logger GetInstance()`
  Gets the instance object of this class. This class cannot be instantiated by `new` statement. Because of keeping a singleton object, this method always returns the same instance.

- `public LogLevel Level { get; set; }`
  Only logs whose level the same as this property or higher will be written to the output media. By default, this property value is `LogLevel.Error`.

- `public bool IsStackTraceShown { get; set; }`
  to determine whether the stack-trace will be logged when calling method `Write`. This method is called by message processor when an error happens to write the information from the generated `Exception` object. The stack-trace is useful for audit and debugging. However, if the application is a console application and the logger output media is console itself then it may be better not to show stack-trace to make neater display. By default, the value of this property is `true` that means the stack-trace will be shown.

- `private void WriteTime(LogLevel level)`
  Writes the timestamp at the current time. At the beginning of log, it is always written the timestamp when the log is written. This method is always called before writing the log, that is by method `WriteLine` and `Write`.
  Parameters:
  - `level` is the severity level of log.

- `void WriteLine(String log, LogLevel level)`
  Writes the log and ended by new line character(s). The new line character(s) is appended automatically.
  Parameters:
  - `log` is the log to be written.
  - `level` is the severity level of log.

- `public void WriteLine(String log)`
  Calls above `WriteLine` by using `LogLevel.Notice` as second parameter.
  Parameters:
  - `log` is the log to be written.

- `void Write(String log, LogLevel level)`

Writes the log without appending new line character(s). If parameter `log` is ended by new line character(s), this new line character(s) will still be written.
Parameters:

- `log` is the log to be written.
- `level` is the severity level of log.

- `public void Write(String log)`
  Calls above `Write` by using `LogLevel.Notice` as second parameter.
  Parameters:
  - `log` is the log to be written.
- `public void Write(Exception ex)`
  Writes the attributes of an exception (message, source and stack trace) to the log output. This method uses `LogLevel.Error` as the severity level.
  Parameters:
  - `ex` is the exception to be written.
- `public void Write(Exception ex, String log)`
  Writes the note about an exception and then writes the attributes of the exception itself (message, source and stack trace) to the log output. This method uses `LogLevel.Error` as the severity level.
  Parameters:
  - `ex` is the exception to be written.
  - `log` is the additional note about the exception written before the exception attributes.
- `public void AddOutput(ILoggerOutput output)`
  Adds an output media to which the log will be written. This logger may have more than one output media. The log will be written to all media has been added. .
  Parameters:
  - `output` is the output media object to be added.
- `public void ReplaceOutput(ILoggerOutput output)`
  Removes all output media has been added and is replaced by the new one.
  Parameters:
  - `ouput` is the replacing output media.
- `public ILoggerOutput Output { get; }`
  Returns `ILoggerOutput` object used by this logger.

Namespace `Free.iso8583.model`

Class `NibbleList`

Nibble is four bits, that is a half byte. In ISO 8583 message, there are several fields which its each character is stored in a nibble. It is to reduce the count of needed bytes to store data. To see what type of field which uses the nibble data, you may consult attribute `type` of element `bit` in XML Configuration. Class `NibbleList` defines a list of nibbles. This class inherits `List<byte>`. All method of `List` should apply. This class is assignable to type of `byte[]` and vice versa. This class can be used by a model as the type of its attribute and furthermore this attribute will be convertible to be an array of bytes that will be used as the value of the mapped message field.

This class extends `List` by an additional property, that is `IsOdd` whose type of `bool`. If this property is `true` then the count of nibbles is odd, that is, there is a half byte unused, at the first or the last.

Class **`DateTimeNibble`**

This class provides methods to convert a date/time value to be a bytes array and vice versa. In ISO 8583 message, a date/time value is usually stored in a group of nibble (each character is stored in a nibble). For example, "December 13" or "12/13" will be stored in four nibbles (2 bytes), the first two nibbles are for 2 digits of month and the rest are for two digits of day.

Date/time field must be specified as type "`N`" in XML Configuration (see attribute `type` of element `bit`) and use attribute `delegate` to convert nibble format to a `DateTime` object and attribute `delegate` for vice versa. Class `DateTimeNibble` provides some methods that can be used for these delegates.

```
namespace Free.iso8583.model
public class DateTimeNibble
```

- `public static DateTime GetMMDD(byte[] bytes)`
  Converts four nibbles (2 bytes) to a `DateTime` value. The first two nibbles are 2 digits of month and the second two nibbles are two digits of day. This method throws an exception if the length of parameter `bytes` is less than 2 or there is a nibble whose invalid value. The valid value is between 0 and 9. A nibble can represent a value of 0 to 15. If the length of parameter `bytes` is greater than 2 then only the first two bytes will be used.

- `public static byte[] GetBytesFromMMDD(DateTime dt)`
  Reverse of method `GetMMDD`. It returns two bytes.

- `public static DateTime GetYYMM(byte[] bytes)`
  It's the same as method `GetMMDD` except for year/month. The first byte is for year (2 digits) and the second one is for month (2 digits).

- `public static byte[] GetBytesFromYYMM(DateTime dt)`
  Reverse of method `GetYYMM`. It returns two bytes.

- `public static DateTime GetHHMMSS(byte[] bytes)`
  This method will convert a time value to a `DateTime` object. The date for the `DateTime` object is current date. The time value is represented by six digits (2 digits of hour, 2 digits of minute and 2 digits of second consecutively). Each digit is stored in a nibble. This method throws an exception if the length of parameter `bytes` is less than 3 or there is a nibble whose invalid value. The valid value is between 0 and 9. If the length of parameter `bytes` is greater than 3 then only the first three bytes will be used.

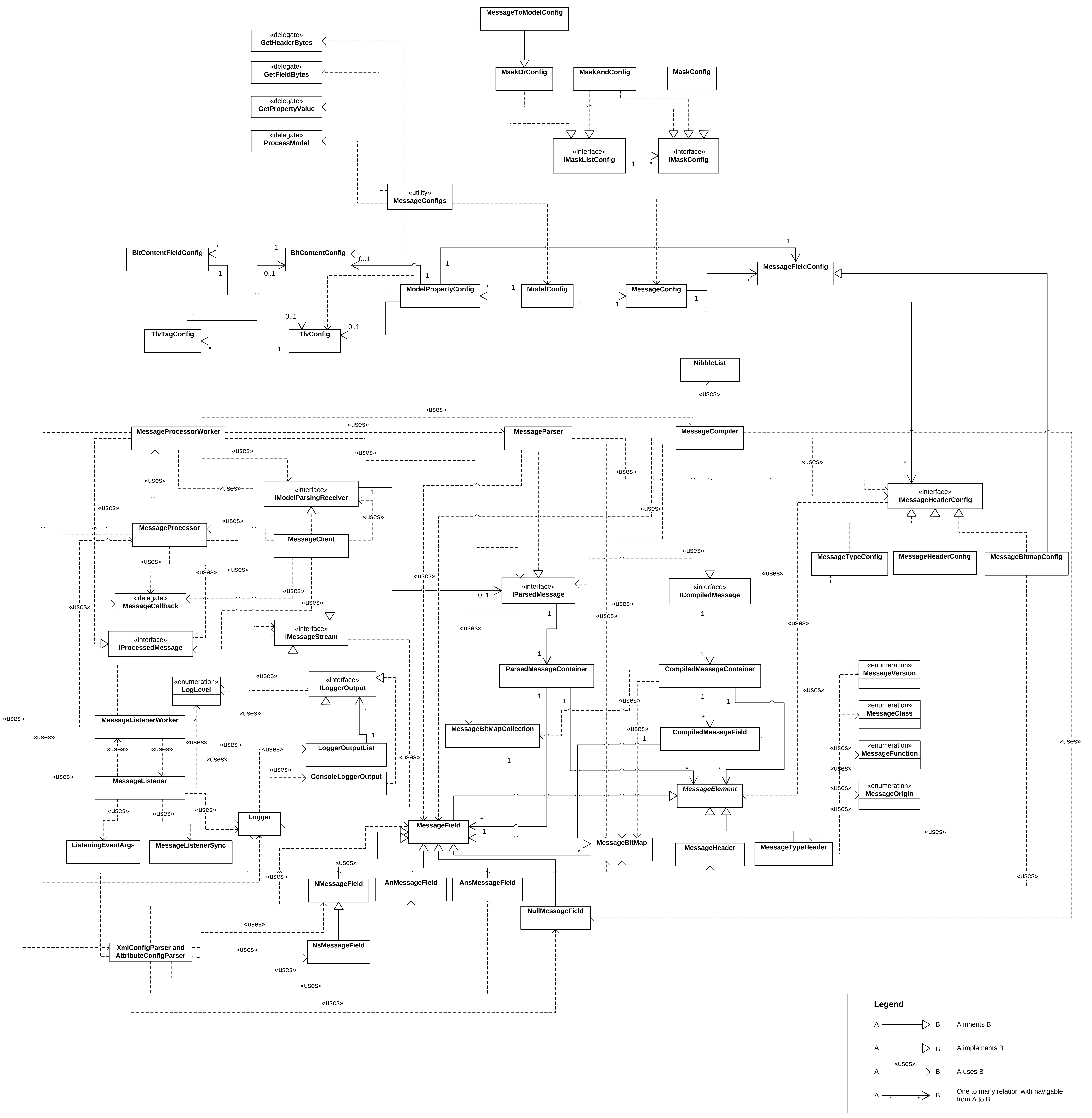
- `public static byte[] GetBytesFromHHMMSS(DateTime dt)`

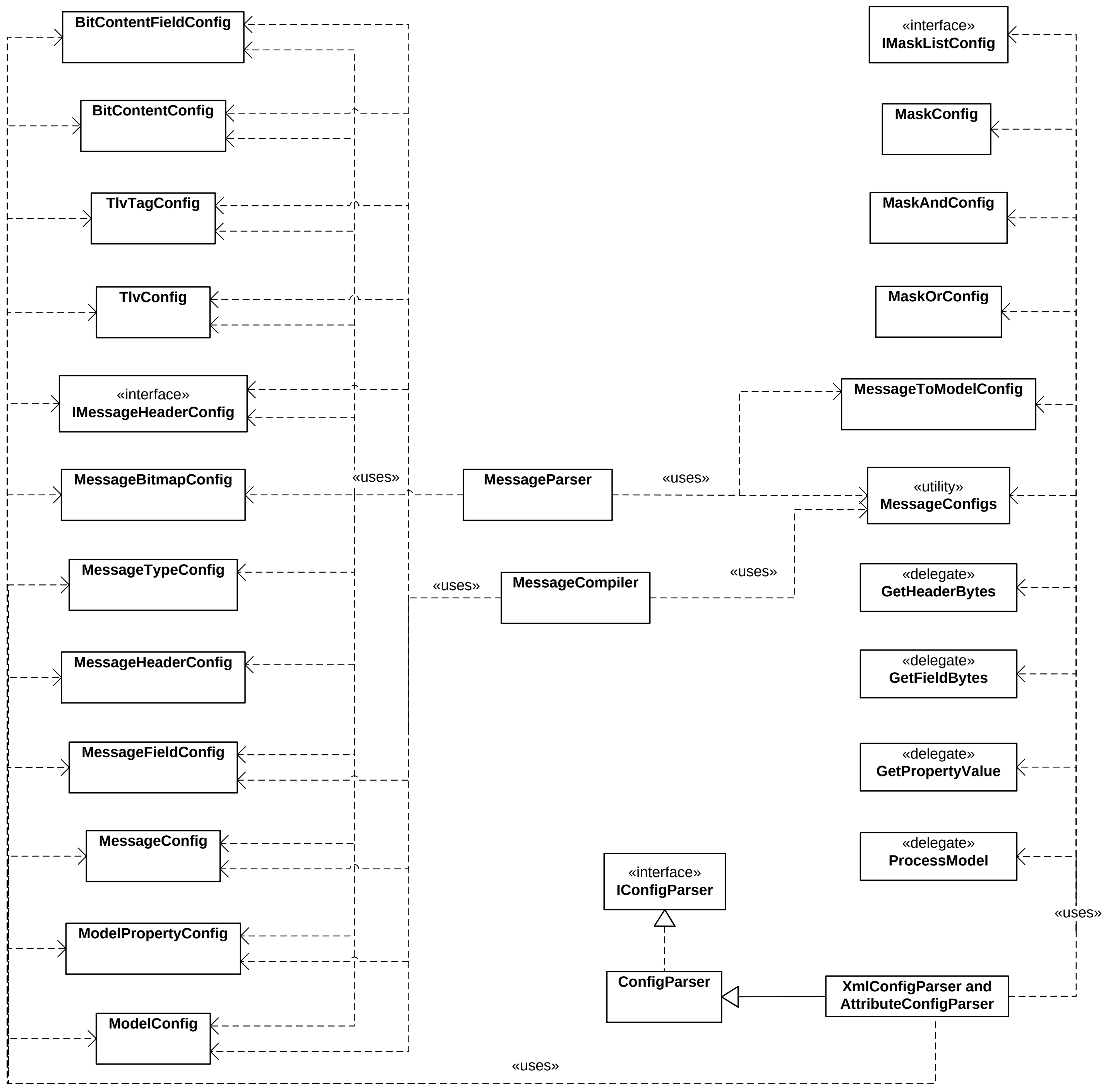
Reverse of method `GetHHMMSS`. It returns three bytes.

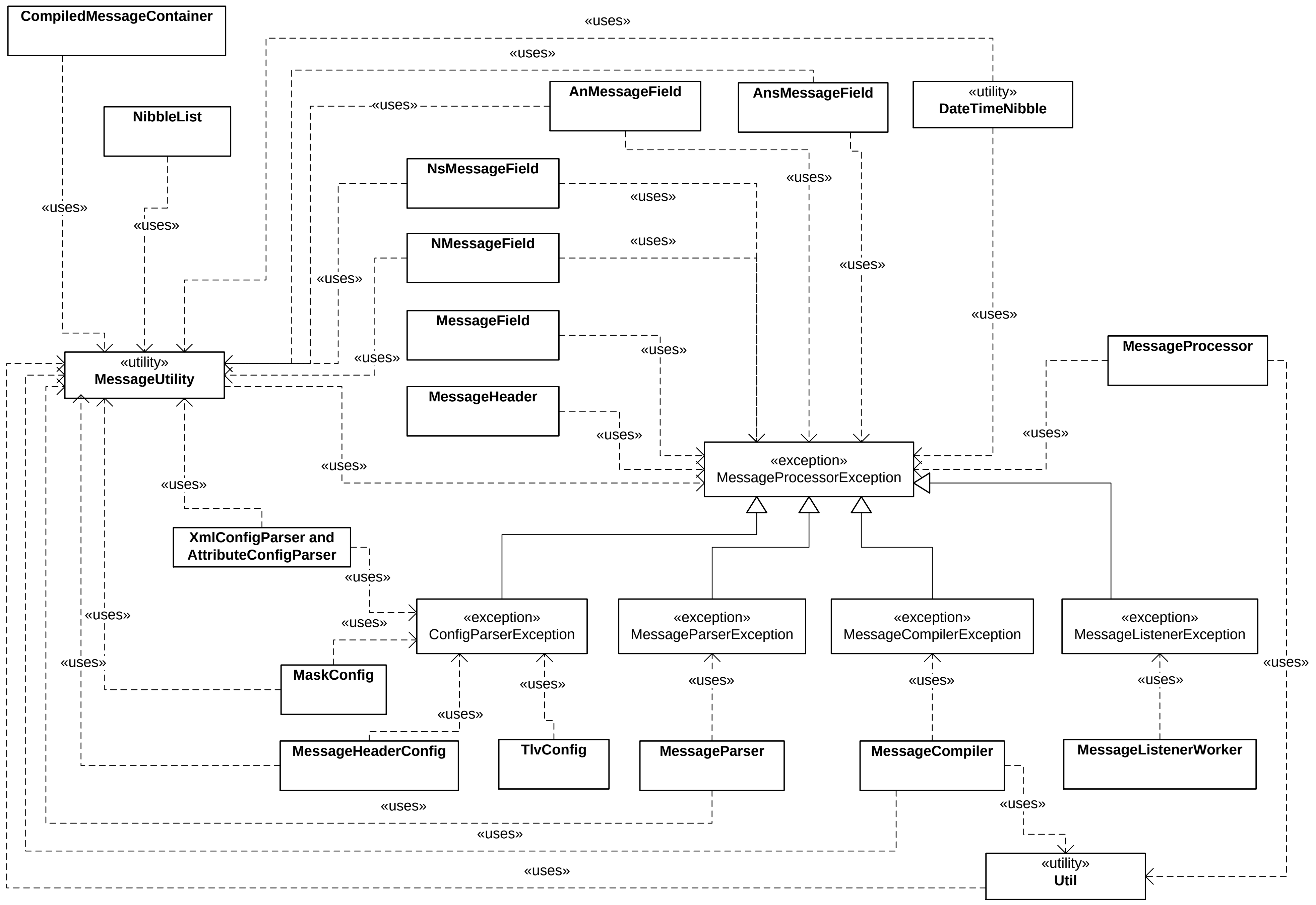
- `public static DateTime GetMMDDHHMMSS(byte[] bytes)`
  This method will convert a date-time value (in nibble format) to a `DateTime` object. The year for the `DateTime` object is current year. The month and day are represented by first two bytes of parameter (2 digits of month and 2 digits of day). The following three bytes represent time value (2 digits of hour, 2 digits of minute and 2 digits of second consecutively). Each digit is stored in a nibble. This method throws an exception if the length of parameter `bytes` is less than 5 or there is a nibble whose invalid value. The valid value is between 0 and 9. If the length of parameter `bytes` is greater than 5 then only the first five bytes will be used.

- `public static byte[] GetBytesFromMMDDHHMMSS(DateTime dt)`
  Reverse of method `GetMMDDHHMMSS`. It returns five bytes.

Class Diagram




BitContentFieldConfig
BitContentConfig
TlvTagConfig
TlvConfig
«interface» IMessageHeaderConfig
MessageBitmapConfig
MessageTypeConfig
MessageHeaderConfig
MessageFieldConfig
MessageConfig
ModelPropertyConfig
ModelConfig
«uses»
MessageParser
«uses»
MessageCompiler
«uses»
«uses»
«interface» IMaskListConfig
MaskConfig
MaskAndConfig
MaskOrConfig
MessageToModelConfig
«utility» MessageConfigs
«delegate» GetHeaderBytes
«delegate» GetFieldBytes
«delegate» GetPropertyValue
«delegate» ProcessModel
«interface» IConfigParser
ConfigParser
XmlConfigParser and AttributeConfigParser
«uses»
«uses»


CompiledMessageContainer
NibbleList
AnMessageField
AnsMessageField
«utility» DateTimeNibble
NsMessageField
NMessageField
MessageField
MessageHeader
MessageProcessor
«utility» MessageUtility
«exception» MessageProcessorException
XmlConfigParser and AttributeConfigParser
«exception» ConfigParserException
«exception» MessageParserException
«exception» MessageCompilerException
«exception» MessageListenerException
MaskConfig
MessageHeaderConfig
TlvConfig
MessageParser
MessageCompiler
MessageListenerWorker
«utility» Util
«uses»