Messages in `Mail::Box`

Mark A.C.J. Overmeer*
AT Computing bv†
Nijmegen, The Netherlands

March 12, 2002

Abstract

The `Mail::Box` module version 2 contains a new implementation for e-mail message handling. The functionality of `Mail::Box` covers that of many existing (old) Perl modules. The primary advantage over those, is improved processing and handling capabilities, required for modern e-mail. Besides, `Mail::Box` has a consequent naming-, documentation-, and programming style. With an optimal lazy approach, message parsing is delayed to improve performance in handling (the often huge) folders. However, the added functionality is invisible to the user.

The whole module is much too large to be documented in this conference paper, so only functionality related to handling one single simple mail message is discussed in detail. Other parts of the package are only listed.

Introduction

Electronic mail is a larger success on Internet than web-pages. Not that unexpected if you experience the complications of finding your way to the sites with useful information. E-mail is much easier: exchange your e-mail address with someone you know, and the practical use is there.

Take a look at CPAN[1], to find dozens of modules which are available to handle e-mail. Distinct purposes for those modules partially explain this high number. However, I expect the main reason behind this number of modules is caused by the general idea that *messages are so easy, it is a shame to use all those complicated existing modules.*

This observation is partially right: simple messages are simple to parse. A simple message:

```
From: me@example.com
To: you@somewhere.aq
Subject: This is a demo
```

*e-mail: `mark@overmeer.net`, web-site: `http://mark.overmeer.net`
†web-site: `http://www.ATComputing.nl`
[1]web-site: `http://www.CPAN.org`.

```
This is the message body.
Bye!
```

All you need to parse this is

```
my %head;
while(<>)
{   last if $_ eq "\n";
    my ($name, $content) = split /\:/;
    $head{$name} = $content;
}
my @body = <>;
```

When you look at `Mail::Internet`, the *mother* of all mail related modules, you see that it actually is implemented this way. Really straight forward.
Back in 1995, when `Mail::Internet` was created, life was this simple. But e-mail is certainly not that simple anymore! Nowadays, students post each other PowerPoint presentations to decorate jokes with images. My cousin posts the latest pictures of her children from New Zealand.

In recent years, e-mail changed from simple single-part and text-based messages into often large, multi-part and multi-media documents. The simplicity of `Mail::Internet` (with multi-part messages not even supported) is certainly not sufficient anymore. Of course, extensions do exist, like the `MIME::Entity` family, but they solve only a few of the complications which accompany harder messages. Using these packages, you also often need to know too much about mail.

During the year 2000, I needed a module to process mail folders and found out that the `Mail::Folder` was not supported anymore. `Mail::Folder` is (*was*) able to manage a set of `MIME::Entity` messages in various kinds of message folders. In need for a folder handler, `Mail::Box` version 1 was developed.

`Mail::Box` version 1 was based on the existing `MIME::Entity` messages. The further I went with my module, the more problems I met with these existing modules. Most problem reports I got were the result of these old modules I used.

Then in mid-2001, I decided that the e-mail modules needed a big clean-up. A clean-up from the bottom-up, fully object oriented, intensively documented, and uniformly structured. The result is `Mail::Box` version 2.

1 Overview

The main components of `Mail::Box` version 2 are listed below. Only the use of messages is discussed in detail in the next sections of this article. `Mail::Box` handles

messages A message object combines a message header and a message body. A message is received from someone, ready to be transmitted to other people (possibly on other systems) or to be archived in a file. A message has a text-only content, hence binary data is encoded (usually with `base64` encoding).

folders A folder[2] contains a set of messages. At the moment only MBOX and MH types of folders are supported, although future extension with EMH, IMAP, and MAILDIR folder types was kept in mind during the implementation.

The very common MBOX folder has one file containing multiple messages, each separated in the folder file by a special line. The MH folder has one file per message, and is a directory itself. They are quite different, but these differences are fully hidden for users of the `Mail::Box` package. For instance, even MBOX folders have (simulated) sub-folders now.

folder manager To avoid accidents, you may use the folder manager to open folders. This manager will avoid opening the same folder twice, and auto-detect folder type. Folders also will automatically get closed when the program terminates. The second task for the manager is to control the *message thread managers*: the administrators of relations between messages.

message thread manager Some messages are related; one is the start message of a discussion, and the replies refer to that message. And there may be replies on the replies, and so on. The thread manager collects these message relations, as you wish for the joined content of multiple folders. For instance, threads can be traced over the `InBox` and the `OutBox` together.

mail folder lockers Various applications use different methods to lock folders. Different locking mechanisms are implemented, but the user has to know which one to use him/herself. One locker combines all locking methods, hoping to work for sure, stopping all other applications.

folder via tie as ARRAY or HASH A folder looks like an array of messages, so it is obvious that a tie on the folder into an array may simplify the code. Simply push a new message on the `@outbox`, and the folder is extended.

Each message has a unique message-id, so a tie to a hash designed with the id as key is also a small step. For instance in case of message threads, the messages refer to one another based on the id. In this case, a hash is the choice of representation to simplify the discovery of relations.

sending messages A special section of `Mail::Box` is devoted to sending messages. Usually messages are automatically delivered into the system (by `sendmail` writing them to your `inbox` folder, or calling `procmail`).

For sending, you need to create a transport object explicitly, with the possibility to set configuration options, or simply call `$message->send`. In the latter case, `Mail::Box` does its best to find a way to deliver the message, implicitly creating a transport object.

logging and tracing For each single object, you may specify what to do with reports: warning, error, progress, notices, and trace can be logged in

[2] This package is named `Mail::Box` because the name-space `Mail::Folder` was already occupied. A mail box and a folder are synonyms, although a folder sounds better.

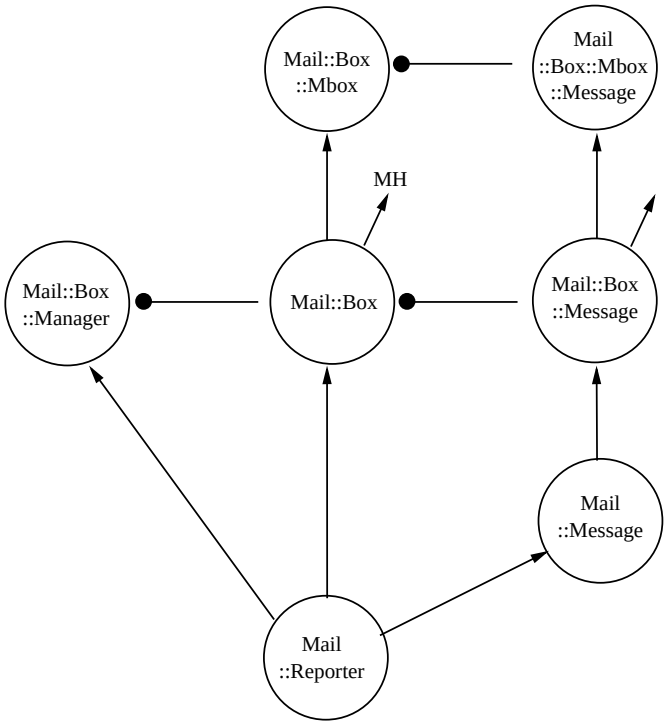

Figure 1: Relations between the main objects

the object and/or shown immediately. This is organised by the `Mail::Reporter` package, which is the base-class for all complicated objects in `Mail::Box`.

Figure 1 shows the relationships between the main objects. The mailbox manager maintains a set of mail boxes, which each maintain a set of mail box messages. The MBOX kind of mail boxes contain MBOX kind of messages. The MBOX messages extend the general folder messages by having a location within the folder file. A general mail box message is derived from a general message, which stands alone and is not persistent.

2 The data-structure of a message

The central object of `Mail::Box` is the `Mail::Message`, a general message. This message can be used for many purposes –even outside the scope of folders, for instance in other applications it could even be used for HTTP traffic.

A message object maintains one header and its related body. That is less straight forward than it looks, as there are three kinds of headers and seven kinds of bodies implemented!

2.1 Message headers

Three kinds of headers are defined. Of course: a message has one really well defined header –a steady set of header lines[3]. However, not all of these lines are known during the whole execution of the program.

The three distinguished states of a header:

all lines are known The `Mail::Message::Head::Complete` contains all header lines for the message, no line is missing for sure. Of course, a complete header consumes more memory than a header where not all lines are known. Do not underestimate the time which is needed to create the objects to store these header lines either.

some lines are known A `Mail::Message::Head::Subset` contains –as the name suggests– only a subset of the header lines. However, it knows where it can get the rest. Objects of this kind are often called *stubs*.

There are various situations which result in this kind of headers. For instance, some folder types have a fast index, which contains some header lines[4] for each message in the folder. Reading such fast indices will produce subset headers. Using this subset of lines, you postpone reading the whole message until unknown header lines are needed.

As long as you only use fields which are known, the subset header will stay. But on the moment you (accidentally?) access an unknown header line the full header is taken. The message's internal mechanisms will process the real header and put it in place of the subset header. Only the time delay is visible to the user.

no lines are known (yet) `Mail::Box` is lazy –as lazy as possible. If it costs time or resources to get information, and the data is not immediately needed, then it will be delayed. For instance, in an MH folder each file in the folder directory represents one single message. The order of the messages is defined by the name of the file (they are numbered sequentially), and with that the total number of messages is defined. It is cheap to find these facts out. On the other hand, it is really expensive to open each of these files to get some or all of the headers. So, as long as no-one needs any of the header lines, the header parsing will be postponed: a `Mail::Message::Head::Delayed` object is created as stub for the header for each message in the folder.

Both incomplete header types are introduced for one reason only: performance. Introducing lazy objects is cheap: stub creation costs as much as creating an object to store a single header line; it is much faster than parsing and creating a full header. It is profitable to create many lazy stubs in vain, just to accidentally save one message to get parsed too many.

[3] Each header line is stored in a separate `Mail::Message::Field` object.

[4] Usually, the often needed `Subject`, `To`, `From`, and `Message-ID` fields are included in these subsets. There may be more, there may be less.

2.2 Message bodies

Message headers only exist in three forms, but there are many more body types. Additional types are still under development or on the wish list. As there is only one complete header type, there are a few body types which contain all information, each maintaining the full content in a different way. Smart body storage is again mainly important for the performance. When you are not interested in the structures used to store the data, your programs will work but may be slightly slower than optimal.

By default, the `Mail::Box` folder readers make smart decisions when reading the data of messages. By opening a folder, you end up with messages with different kinds of body storage –and you do not have to care[5] at all! All types of bodies support methods to get the content of the body as a whole in one scalar, as list of lines, or as file-handle[6]. Some bodies provide more functionality. Especially the multi-part body is quite powerful.

The bodies store their data

as string The `Mail::Message::Body::String` stores the whole message body in one scalar. This can be useful when the data is binary: it can be written at once, or converted as a whole. Storage as lines is un-natural in such situations.

as lines For text processing, getting the body content as list of lines is easiest in most cases. When you store data as a `::String` object, the internal scalar must be split on new-line characters each time the separate lines are requested. For textual data, the `Mail::Message::Body::Lines` object is preferred as it stores the lines separately and supplies these lines without additional work.

as file Some messages are huge. Especially Word documents, PowerPoint presentations, PostScript files, and pictures can size into megabytes. In this case, it is very costly to keep this data in memory during the run of your program. The data is maintained in an external temporary file by the `Mail::Message::Body::File` object for as long as the folder is open. When the folder is closed, or the whole program terminates, these temporary files are automatically removed. Especially decoded binary bodies have a special affection to this kind of bodies.

external Like the previous kind the data is stored in a file. However, in this case not in a temporary file but a persistent file. The actual folder is reduced in size, so considerably faster to read and write. The message in the folder contains an indication where to find this external body. When the message is removed, the external file will be too.

The external body is under development. It still has to be decided how to transform a temporary body object into an external file with a decent unique name.

[5] You can control the type of the created bodies with the `body_type` option when the folder is opened. A code reference can be supplied, which must determine the type of the body to be created based on the known header information.

[6] A full list methods can be found in the `Mail::Message::Body` manual page. That package is the base for all body types.

in-folder In this situation, the body of the message is never taken from the folder, but kept as inner-file within the original folder using `IO::InnerFile`. This is a smart trick, but not always applicable: as we will see later, bodies can live without message and without folder.

In-folder bodies are best suited for MH folders, where each message has its separate file. The inner-file adds an offset to the location of bytes in the body, so simply skipping the header of the message.

This type is not yet implemented. When implemented, there may be two different body types to get it right: one for use in MBOX-like folders, and one for the MH-style folders.

All above cases are bodies with real data –these are relatively straight forward. However, there are other cases:

delayed body In this case, the location of a body is known but the data is not taken from the folder yet, just like the delayed header we saw in the previous chapter. Again `Mail::Box` tries to be maximally lazy to get optimal performance. A delayed body is by default the preferred body type for large bodies[7]. The stub is implemented by the `Mail::Message::Body::Delayed`.

multi-part body By far the most complicated body is the multi-part body, which is used to maintain messages with attachments. In many applications, you must implement special treatment for these bodies. In the current real-life e-mail usage, it is not possible to ignore these complex messages anymore.

The handling of multi-part bodies is implemented in the `Mail::Message::Body::Multipart` package. The main difference with normal bodies is that this one contains *parts* (which most people call *attachments*). Each part is a `Mail::Message::Part`, which extends the general `Mail::Message`. A part contains fewer header lines than a normal message, and has an enclosing (parental) body.

3 Implementing delayed objects

Parsing the text of the header may be delayed, and the body may be delayed as well. This is cause for some complications inside the module: one way or the other, the real header and body must get parsed when the data is needed without the user noticing this.

3.1 Stubbing delayed objects

In these situations, the developer's choice is between two solutions:

[7] To determine whether the body must be delayed or not, you can specify a code reference with the `extract` method when the folder is opened. The function is called by the parser with the complete header it has just read as argument. Based on that information, the decision is made between *true* (extract now) or *false* (delayed). If extracted, the `body_type` is used to determine which type of body will be created.

1. Create a wrapper around or within each method, which checks whether the data is already available, or still must be loaded:

   ```
   sub myMethod() {
      my $self = shift;
      $self->forceLoad unless $self->isLoaded;
      doMyThing();
   }
   ```

   It is the simpler approach when looked at it from the level of language knowledge required to implement it. However it is costly: when the data has been read, the checks will still take place. Even more important: it is risky because with inheritance it is not directly clear whether all wrappers are correctly implemented. When a new method is added to a base class, there may be a need for a wrapper in one or more of its sub-classes, which is hard to get tracked down.

2. Create an object which only implements methods which are do-able with the known dataset. When a 'missing' method is called, `AUTOLOAD` takes over, parses the real data into a real object, and replaces the stub by the real object (for instance, replaces the delayed header by the complete header in the administration of the message). Finally the method is called on the real object. Simplified:

   ```
   package IamReal;
   sub myMethod() {doMyThing()}

   package IamNotReal;
   sub AUTOLOAD() {
      my $self = shift;
      $self->readData();
      bless $self, 'IamReal';
      $self->myMethod();
   }

   package main;
   my $obj = IamNotReal->new();
   $obj->myMethod();  # Triggers realization
   ```

   In this example, `AUTOLOAD` transforms a `IamNotReal` into a `IamReal` object.

The second solution is without the penalty of continuous checking, but has a nasty side problem: for `AUTOLOAD` to work, it cannot inherit as the realized object does. This means in our case that

```
Mail::Message::Head::Subset->isa('Mail::Message::Head')
```

would fail. To solve this, the delayed objects inherit from `Object::Realize::Later`[8], which overrules `isa` and `can`. This class also supplies the right `AUTOLOAD`. This class is a powerful stub for delayed loading.

[8] `Object::Realize::Later` is separately available from CPAN.

3.2 Message status transition

The body which contains data can only be part of a message with full knowledge about the header. Figure 2 shows the stati which a message can pass through during its life within the program. When a folder is opened, the `extract` option

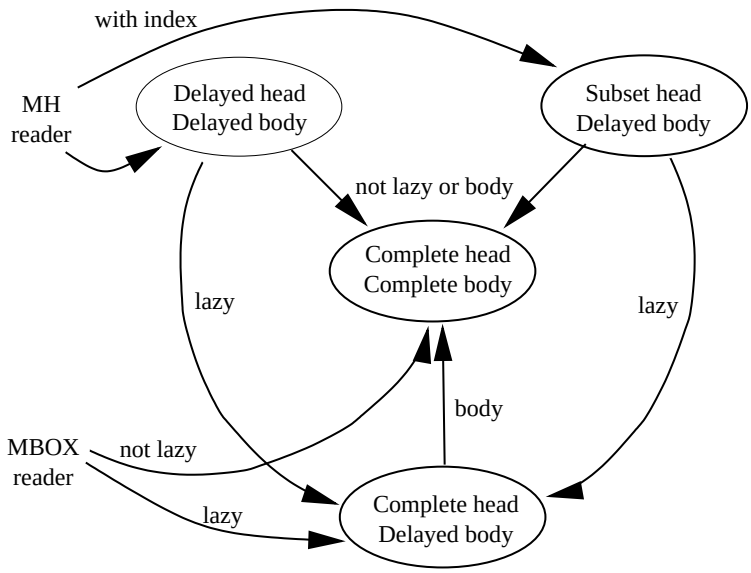

Figure 2: The message's life.

specifies when the mail box reader must be *lazy* –being delayed. By default, the reader will not be lazy on very small messages, but will be for large ones; it is purely an optimization issue. In figure 2, `body` means that there is a request for the content (body) of the message. A "complete body" is any body except the delayed version.

In any case, the transitions take place without knowledge by the user of `Mail::Box`: every move is handled by the internals of the module. The messages stay in the "complete header with complete body" state until the folder is closed and the last reference to the message within your program is removed.

4 Derived messages

All messages contain a header and a body, but still not all the messages are the same. Where the `Mail::Message` object can store any message, this functionality is extended for use in mail boxes by the `Mail::Box::Message`. Messages which are part of a mail box have a reference to that folder, and a sequence number. They also know in which file they reside. Figure 3 displays all objects which relate to a single message, and the class relationships.

For every folder type, this class is extended even further: for MBOX folders, the separator line which is used in the folder file is kept. For MH folders, the message's filename is stored. These message specializations are implemented in the `Mail::Box::Mbox::Message` and `Mail::Box::MH::Message`, respectively.

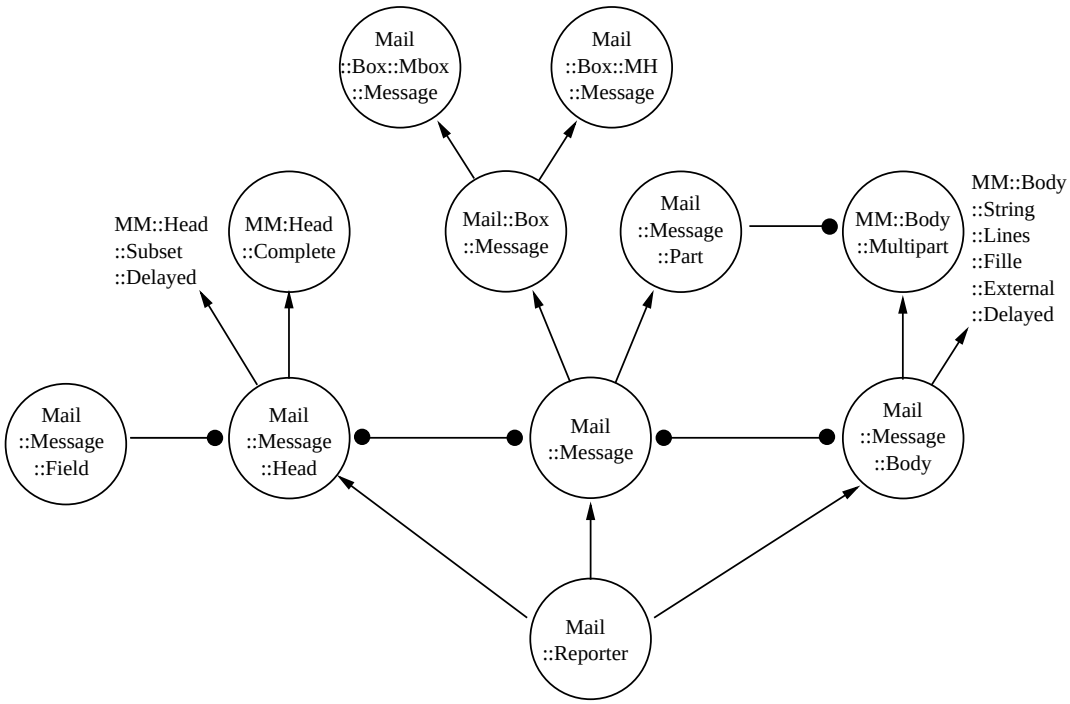

Figure 3: Message hierarchy with related objects

In multi-part messages, the body of the main message (which may, or may not be in folder) contains a set of parts (attachments). Each of these parts is a message by itself, although with little different rules than any general message. For instance, these `Mail::Message::Part` objects require fewer header lines.

4.1 Message type coercion

It is possible to add MH messages to MBOX folders: they are *coerced* from one type, via a normal message, into the next message type. Coercion is automatically done where needed.

Some methods in `Mail::Box` offer extra flexibility by accepting foreign message objects. As examples, for sending of a message or adding a message to a folder, the subjected message is coerced into an acceptable message object. `MIME::Entity` and `Mail::Internet` messages are coerceable without any problem. Reverse conversions are also implemented. This means that old Perl programs can easily be combined with `Mail::Box`.

A simple code example tells it all:

```
use Mail::Box::Manager;
use MIME::Entity;

my $mgr    = Mail::Box::Manager->new;
my $folder = $mgr->open('OutBox', access => 'a');
```

```
my $msg1   = Mail::Message->new(file => 'image.gif');
my $msg2   = MIME::Entity->build();

$folder->addMessage($msg1, $msg2);
$folder->close;
```

The folder type is auto-detected, but defaults to the creation of an empty MBOX type of folder. Both `$msg1` and `$msg2` are coerced into the right message type – `Mail::Box::Mbox::Message`. This means a complex conversion for the `MIME::Entity` foreigner, and simply re-blessing for the first message. Both messages will get linked to the folder (required for all `Mail::Box::Message`'s), and get a message separator to be used in the folder file (as required for `Mail::Box::Mbox::Message`'s).

5 Handling a message

The reason to send messages in the first place is to pass knowledge on. The message body contains this knowledge. The header contains some information about source and destination of the message, as well as delivery method and time stamps. The header is mainly accompanying information, and as such most lines are not of interest when you process the content.

Exceptions to this claim are the `Content-` lines in the header: they contain important details about the information in body.

5.1 Content information

The header lines which start with `Content-` are

Content-Type This field represents the MIME-type[9] and maybe some extra attributes. Examples of MIME-types are `text/plain` and `image/gif`. A whole `Content-Type` field may read:

```
Content-Type: text/plain; charset="us-ascii"
```

Content-Transfer-Encoding Data in a body is to be encoded during transport, to avoid conflicts with the protocols used in the transmission. Textual data is hardly encoded, usually `7bit` or `8bit` is used. Western languages are seldomly affected by these encodings, the Japanese and Chinese are less fortunate.

Binary data and data of unknown complexity is by default `base64` encoded. In former days, `uuencoding` was more popular. Each encoding–decoding pair of methods is implemented in a separate package based on `Mail::Message::TransferEnc`, and managed by `Mail::Message::Body::Encode`.

Content-Disposition This field tells how the data must be presented to the user. The data tells whether a mail user agent (MUA) should present this data directly when the message is opened (`inline`), or only on demand (`attachment`). Next to this, it may contain the default name of a file to unpack the body into.

[9] MIME-types are handled by the `MIME::Types` module, separately available on CPAN.

Of course when a body is passed around between methods, the kind of information which is stored in the body must be passed-on with it. These `Content-*` header lines are copied from the header object into the body object when the message is read from file. This means that you get the information –for instance the MIME-type– from the body object. Do not use the header object for this information: the body may have been decoded, recoded, or converted before you get your hands on it. The body's MIME information will reflect this, but the header of the message won't.

```
print $message->get('Content-Type');       # WRONG
print $message->head->get('Content-Type'); # WRONG
print $message->body->type;                # RIGHT
print $message->decoded->type;             # BETTER
```

5.2 From message to body

A body is more than just the data: it is also the knowledge about the data. From now on: keep in mind that the body is more important than the header, even more important than a message. Focus in your programs on treating the bodies.

You can retrieve the body of a message in two ways:

- `$message->body` returns the real body of the message, which may be encoded in any format. Unless you are sure what you are doing, this is not the way to go.
- `$message->decoded` returns a body which contains the decoded version of the message's body. This means that actions are taken to create a body with transfer encoding set to `none`.

Various ways to handle a body are implemented in two separate files which are automatically compiled when their methods are needed. They both add functionality to the `Mail::Message::Body` general body class. These packages are

`Mail::Message::Body::Construct` Complex body handling methods, often required for the complex message handling methods in `Mail::Message::Construct`.

`Mail::Message::Body::Encode` Control over body encoding and decoding. For now only transfer encodings are handled, but for instance html to text and vice versa is on the wishlist.

One of the methods which is offered is `concatenate`, which joins bodies and strings into a new body. It takes care of the conversions required to achieve this: all concatenated pieces must be unified[10] into the same MIME-type before gluing them into one new body. `Concatenate` is exemplary for all methods which process bodies: they take one or more bodies, and return a new body object with the result data.

Actions on bodies will never modify the body in place. They may however

[10] Unification is tried by the `unify` method.

return the source body if no changes were needed. For instance, if you call `concatenate` with only one body, that body will be returned unmodified.

Creating body object after body object may seem clumsy and *heavy*, but is in practice not that bad: a body object is in most cases just an array of lines plus the updated information from the `Content` lines. The auto destruction and clean-up of Perl will take care of intermediate body objects which are not used anymore.

5.3 From body to message

The general procedure to go from one message to another, for instance to create your own special reply[11] is to take the body from its source message, decode it, change it, and then add it to a new message.

So, let's first create the reply:

```
use Mail::Box::Manager;
my $mgr    = Mail::Box::Manager->new;
my $inbox  = $mgr->open($ENV{MAIL});
exit unless defined $inbox;

my $source = $inbox->message(6)->decoded;
my $reply  = $source->concatenate
 ( "This is an automatic reply.\n", "\n"
 , $source
 , "Please ignore (as we do with your message).\n"
 );
```

And now the (too simple) reply is prepared in a body referred to by `$reply`. The body type of `$source`, and its MIME-type and encoding, determines the shape of the `$reply`.

Then, there are three ways to put a body in a message:

Start with a message First instantiate a message, and then add a body like this:

```
my $head    = Mail::Message::Head::Complete->new;
$head->add(From => 'me@example.com');
$head->add(To   => 'you@anywhere.aq');

my $message = Mail::Message->new(head => $head);
$message->body($newbody);
```

At the moment you add the body, the body is converted into a transfer encoding which is sufficient to safely transport the message. Thereafter, the content information is copied from the body into the header. Finally, also the message-id in the header is changed to avoid the danger that the same message-id is used for two different messages.

[11] `Mail::Message::Construct` already defines a powerful `reply` method, but you may want some different solution.

Construct message from body A nicer route to follow is to take a body, and then build a message from it. This procedure is more straight forward than the previous:

```
my $message = Mail::Message->buildFromBody
 ( $body
 , From => 'me@example.com'
 , To   => 'you@anywhere.aq'
 );
```

The `From` and `To` lines are obligatory[12], because no message can do without them. The `buildFromBody` is described in `Mail::Message::Construct`, a package which extends the functionality of `Mail::Message` with complex message processing functionality.

Construct message with body A third approach creates a message with the body using the `build` method. In this case, the body plays a less important role.

```
my $message = Mail::Message->build
 ( From   => 'me@example.com'
 , To     => 'you@anywhere.aq'
 , attach => $body
 );
```

You may specify multiple `data` (text), `file` (file-name or file-handle), and `attach` (body and message objects) for the construction of a message. If only one kind of information is supplied, a single-part message is constructed, otherwise a multi-part is the result.

The last method is more powerful but less controlable then the second. The first method should be avoided, although it works.

5.4 From message to message

You didn't forget that the body is more important than the message? Don't let that idea out of your mind. You cannot go directly from message to message (unless you use some complex methods like `reply`, `forward`, or `bounce`), but need intermediate stages which process the body of the source message into the body for the destination message.

Figure 4 depicts the following program from the viewpoint of body transformation:

```
use Mail::Message;

my $source  = Mail::Message->new(...);
my $decoded = $message->decoded;

my $joined  = $decoded->concatenate
```

[12] The `From` and `To` lines are not required to build a `Mail::Message::Part`.

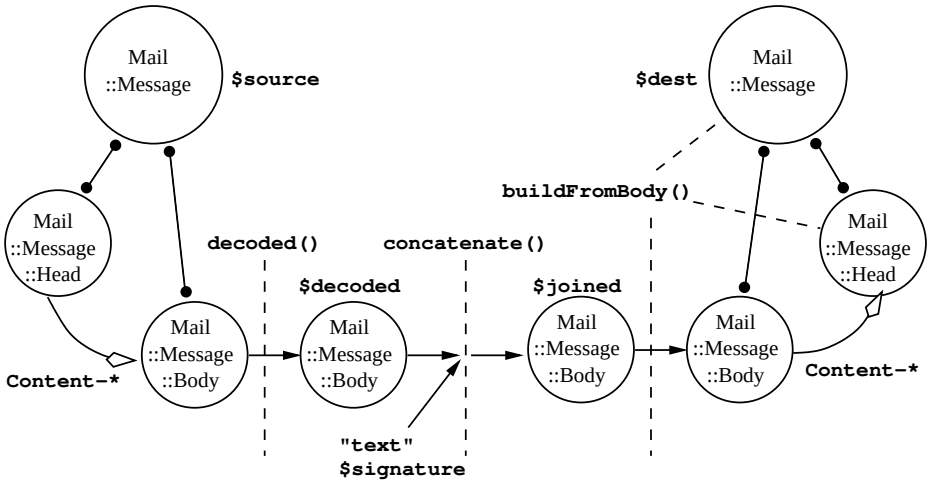

Figure 4: From message to message.

```
    ( "One line in front\n"
    , $decoded
    , $signature   # also a body...
    );

my $dest    = Mail::Message->buildFromBody($joined);
```

The code for the `Mail::Box` user stays simple. Different required conversions and handling the message organisation is hidden (as it should be!).

As a final step, the `$dest` message can be stored in a folder or transmitted. An example of message storage:

```
use Mail::Box::Manager;
my $mgr = Mail::Box::Manager->new;
$mgr->moveTo('OutBox', $dest);
$dest->send;
```

The manager keeps track on opened folders. It will also handle transport of messages between folders. Some actions can also be taken when the folder is not open. The manager first checks whether `OutBox` is opened. If so, the message is added to that folder using `$folder->addMessage($message)`. Coercion into the message type which is required for the folder takes place. For instance, if the folder is an MBOX, the message separating *From*-line is created.

When the folder is not open, the manager will auto-detect the folder type, and then will add the message without opening the folder. Of course, also in this case it will coerce the message to the right type before doing that.

6 Concluding

To finalise this overview on possibilities of messages offered by the `Mail::Box` module, the conclusion must be that handling messages is much harder than

just parsing a few header lines and the body. A valuable module must supply access and processing methods, and these are quite hard.

Only a very small part of the implemented functionality is discussed in this paper. We only discussed single-part messages deeply. For the other components of `Mail::Box` you have to use the normal manual pages and the example scripts which are included in the module. The `Mail::Box-Overview` and `Mail::-Box-Cookbook` manual pages are good points to continue the study.

About the Author

Mark Overmeer graduated as MSc in Computer Science at the University of Nijmegen in the Netherlands. After six years of experience as UNIX system administrator in a large data-center, he joined AT Computing to give training in UNIX, web-technologies, and programming languages.
Next to his professional activities, Mark maintains and develops a few web-sites and is actively contributing to the Perl community.
At the moment Mark maintains about quite a number of e-mail related Perl modules on CPAN: the `MailTools` collection, `MIME::Types`, and `Mail::Box`.