E-Mail processing with Perl

Mark A.C.J. Overmeer*
AT Computing† & NLnet Foundation‡

March 26, 2002

1 Introduction

Perl is a powerful language for automating administrative processes; whether it be for system administration, customer interaction via web-pages, or reporting purposes, it all can be done in Perl. Over 2200 modules which can help you with your tasks are available on Perl's Open Source archive CPAN[1]. At the moment of this writing, March 2002, about 60 of these modules are directly related to handling e-mail. This paper may help you use some of them.

1.1 Competing Implementations

Reading through the list of e-mail related modules, you get the impression of overlap: many modules implement the same functionality. For instance, there are at least five implementations for parsing a message from file into structures which can be handled by Perl. Why is there so much overlap?

There are various reasons why overlapping implementations are available in the Perl libraries – and those reasons apply to many other Open Source projects as well. They are caused by the following reasoning of programmers:

1. "The implementation is *bad*"

   Quite a few modules on CPAN are very old, and really should be replaced by modules which cover the same functionality but use improved

*e-mail: `mark@overmeer.net`, web-site: `http://mark.overmeer.net`
†`http://www.ATComputing.nl`
‡`http://www.NLnet.nl`
[1] `CPAN's main archive can be found at http://www.cpan.org`

syntax[2]. In searching for a useful module, the programmer gets a *bad* feeling about the existing code, and produces a new implementation for the same thing.

However, this is easily said, but harder to achieve: a replacement module passes through many phases:

- not at all functional
- partly functional, still many bugs
- partly functional – alpha
- partly functional – beta
- more functionality, fewer bugs
- acceptable functionality – release
- more functional than the original module

Although it is often easy to see *that* a module must be improved (users with little experience can see that), it requires more experience to determine *how* a module can be improved, and usually a lot of effort to get to the state that a rewritten module actually *is* improving the situation.

The conclussion must be that most modules look worse than they actually are: they are working with acceptable functionality. The qualification *bad* is only acceptable if you provide a *better* implementation.

2. "Oh, I didn't know there was an implementation already"

   CPAN has grown too large. It is very hard to find out which modules can satisfy your needs: On which versions of Perl does it run? On which platforms does it run? How functional is the module?

   If you start coding because nothing is available on CPAN yet, after a few months of work you may see someone else publishing his implementation addressing the same problem. So: publish early and get people involved that way. However, in early stages you do not supply sufficient functionality which drives people immediately away from your code never to come back. There is no easy answer to this dilemma.

[2] Perl's syntax is under continuous development, which makes the language more powerful with every new version. Most e-mail related core modules pre-date the time when real references and Object Orientation where added to the language, hence are full of tricks to circumvent these deficiencies.

3. "It is too complicated to use"

   The subjective feeling of *complicated* is caused by a few factors. Most complexity with modern programming languages is not in learning the language itself, but rather in learning how to use its libraries. Studying the libraries will consume most time. Some of Perl's libraries are much larger than Perl itself!

   Large libraries easily get marked as *too complicated*, just because the user underestimates the complications involved in a correct implementation. The average user is not aware of requirements in RFCs, compatibility issues with broken applications, environments of other users of the same library, or even what the needs of his own program are.

   Of course, the author of a module can help fight the feeling of complication by providing a simple interface to the library, good documentation, examples, and support. But the module will still be large and complex.

It is obvious that modules are in various conditions, and created with different objectives in mind, for different applications, and written by different programmers. In general, the e-mail related modules for Perl are too small, contain little to no documentation, and were created by many independent programmers all using their own programming style. Still people need to use them, and often they succeed. But how much effort do they have to put in it, and what is the quality of the result?

1.2 Message Parsing

But, message parsing is simple! Current implementations are *too complicated*? Yes, simple messages are simple to parse. A simple message:

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

This is the message body.
Bye!
```

A simple parser:

```
my %head;
while(<>)
{   last if $_ eq "\n";
    my ($name, $content) = split /\:/, $_, 2;
```

```
    $head{$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! My cousin posts the latest pictures of her children from New Zealand by e-mail. Students post to each other PowerPoint presentations to decorate jokes with animations and sound. 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 need to know too much about details of the mail protocols.

Some of the complications which e-mail related modules have to deal with:

- header lines can appear more than once in the same header. They should be treated case-preserving. Order preserving is nice. Wrapping long lines (*folding*) is only permitted for some of them;
- multi-parts and nested multi-part messages are quite hard to handle. Try to start thinking about how to construct them. Each part looks a bit like a message, but they are slightly different. What about deleting one part? Does it imply that the main message's identity is changed?
- the message body can be encoded (base64, quoted-printable, ...) for transfer, even though it contains plain text. The text can use varying character sets. What if we want to automate a reply included in a piece of our text encoded as ISO 8859-15, and the message we reply to is in UTF-8? Or it is structured in HTML?
- various folder types, some organizing the messages together in one file (like Mbox folders), others as separate files within one directory (like MH and Maildir), or separate entities on a remote machine (like IMAP);
- differentiated treatment of labels which relate to messages status (is it read, replied upon, etc). Sometimes these labels are stored in the messages header as `Status` and `X-Status` lines. However, there are

differences in implementation and interpretation of these header lines. Maildir encodes the labels in the filename of the message, while MH maintains a special label file per folder.

- how do we create a reply to a binary message? Or for a multi-part? Or one part within a multi-part?

Maybe this list will warn you not to create your own implementation for handling mail folders.

A decent mail handling module must hide as many of the complications as possible from the users. Of course, a module implementing everything that has to do with e-mail gets large and hence harder to use and harder to learn. However, hopefully it enables everyone to create powerful applications without the need to know all the details about correct mail.

1.3 Development of `Mail::Box`

During the year 2000, I needed a module to process mail folders and discovered that the `Mail::Folder` module 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 of 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 received were the result of these old modules I used.

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

The goal of `Mail::Box` is to be

- one consistent set of packages;
- documented on different levels of abstraction;
- easily extendible in an object-oriented way;
- hiding as much nastiness about messages as possible; and
- providing strong building blocks to construct new messages, and replies, forwards, and bounces of messages.

In the next chapters, we will see a few things that `Mail::Box` can do for you. Where applicable, some other e-mail related modules will be introduced as well.

2 `Mail::Address`

`Mail::Address` is part of the MailTools bundle of Perl modules. Most of the modules in that bundle are best avoided, but `Mail::Address` is a good exception. Use `Mail::Address` to convert text into e-mail address objects, or the reverse: objects into text. The `Mail::Address` object provides access to the knowledge in the address information.

2.1 Parsing a line with addresses

The following program demonstrates how `Mail::Address` is used to get e-mail addresses from a string:

```
use Mail::Address;

my $line  = 'info@example.com (Overmeer, Mr M.)';
my @addr  = Mail::Address->parse($line);
my $first = $addr[0];
print $first->comment,"\n";  # (Overmeer, Mr M.)
print $first->name,"\n";     # Mr M. Overmeer
print $first->address,"\n";  # info@example.com
```

Different textual representations of e-mail addresses are handled. The `parse` method returns all addresses which are contained on the line as list. Each address is converted to a separate `Mail::Address` object. The `name` method is most impressive, knowing how different cultures prefer to mangle personal names and then de-mangle them.

2.2 Creating an address

The following example shows the reverse process: create an address object yourself, and then `format` it into a correct string.

```
use Mail::Address;

my $a = Mail::Address->new('Mark Overmeer','me@example.com');
print $a->format,"\n";      # Mark Overmeer <me@example.com>
```

3 MIME::Types

The `MIME::Types` module knows how to handle *mime-types* right. A MIME compliant[3] message contains information about the kind of data which is stored in the body. That information is stored in the `Content-Type` line. For example:

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

In this example, the mime-type of the message is `text/plain`. Information about the character set used to compose the text is not part of the mime-type.

3.1 Information about a mime-type

Given the string describing a mime-type, the `MIME::Types` module can provide details about it.

```
use MIME::Types;
my $alltypes = MIME::Types->new;

my $plain    = $alltypes->type('text/plain');
print $plain->mediaType;    # text
print $plain->subType;      # plain
if($plain->isBinary) ...    # false
if($plain->isAscii) ...     # true
print $plain->encoding;     # quoted-printable
```

The module handles the case-insensitivity of the mime-type description, and provides a little more background information. As a service, it can tell you whether the body is binary, and how the data should preferably be encoded for transmission.

3.2 Information about filename extensions

The `MIME::Types` module also knows about filename extensions in relation to mime-types. Some of these notions are operating system specific.

```
use MIME::Types;
my $alltypes = MIME::Types->new;
```

[3]The MIME (Multipurpose Internet Mail Extensions) standard is described in RFCs 2045 up to 2049, and defines how the header lines are shaped. The RFCs also define acceptable content for specific lines.

```
my $h = $alltypes->mimeTypeOf("index.html");
print $h->simplified;     # text/html
print "$h";               # text/html

my $g = $alltypes->mimeTypeOf("GIF");
print $g->type;           # image/gif

my $z = $alltypes->mimeTypeOf("a.gz");
print $z->type;           # application/x-gzip
print $z->simplified;     # application/gzip
print $z->subType;        # gzip
```

In mime-types, a leading '`x-`' indicates the use of a name which is not officially registered. So, new names will be introduced with a `x-`, and after registration this flag will disappear. The `MIME::Types` module understands this.

4 `Mail::Message`

This is the first object from the `Mail::Box` module we will discuss. `Mail::Box` is all about messages which are stored in folders, but messages can also live (temporarily) outside a folder and therefore can be discussed separately first.

Every MIME compliant message contains

- a header: a few lines which describe the message's body, information about the transport of the message, and so on. This is also called the *meta information* about the message; and
- a body: the actual content.

In `Mail::Message`, these parts are represented by two separate objects: a `Mail::Message::Head` and a `Mail::Message::Body`. Let's take them apart.

4.1 A message header

The header of a message contains information about the message, excluding the *payload* which is stored in the body. This information is stored in well-described lines, for example:

```
From: you@example.com
Date: Wed, 4 Oct 2000 14:22:35 -0400 (EDT)
```

```
Message-Id: <200010041822.e94IMZr19712@example.com>
To: me@home.museum
Received: (from majordomo@localhost)
    by example.com (8.11.0.Beta3/8.9.3) id e94IMj420302
    for everyone; Wed, 4 Oct 2000 14:22:45 -0400
Subject: Font metrics
Status: RO
Content-Type: text/plain; charset="us-ascii"
Content-Length: 1620
Lines: 32
```

The label (*field*) of a header line is case-insensitive, but has a preferred notation: each word in the field should start with a single capital.

Some of the lines, like `Received` in the example, can be folded over multiple lines when they get too long. This is only allowed for the well-described *structured* lines, not for other lines.

Each (folded) line can have three parts:

```
<field> : <body> ; <comment>
```

The comment including its leading semi-colon is optional.

4.1.1 Get a header line

Once you have a message, you can ask about lines in the header:

```
my $msg  = ....;  # get it somewhere
my $date = $msg->head->get('date');

print "$date";    # Wed, 4 Oct 2000 14:22:35 -0400 (EDT)
$date->name;      # Date
$date->print;     # Date: Wed, 4 Oct 2000 14:2...
```

The `$msg` is constructed some way, or contained in a folder; we'll see later how to get one. The `$date` returned is either `undef` (not found) or a `Mail::Message::Field` object. This object stringifies to the body of the header line, **excluding** the comment.

To simplify access to the header lines, you can also directly ask the message for information in the header. Consider the following alternatives:

```
my $msg  = ....;  # get it somewhere
```

```
my $head = $msg->head;
print $head->get('DATE');
print $head->get('content-type');

print $msg->get('DATE');
print $msg->get('content-type');
```

It doesn't really matter which solution you take. The former will be a tiny bit faster though.

4.1.2 Special fields

Of course the previous section was valid, however it is much too low level. For instance, the `To` and `From` header lines contain e-mail addresses. And did you know that those lines are overruled by `Resent-To` respectively `Resent-From` lines if present?

A few fields get special treatment. Fields which start with `Content-`, like `Content-Type` and `Content-Length` are managed by the message's body (explained in 4.2.2). Fields which contain e-mail addresses have their own methods which return `Mail::Address` objects. Example:

```
my $from = $message->from;
print $from->name;   # Mr M. Overmeer

my @to   = $message->to;
my @cc   = $message->cc;
```

When there are `Resent-` lines present, only the data in those lines is returned, as it should be. Otherwise, the usual set of lines is parsed.

4.2 Message body

The body is the *payload* of the message. `Mail::Box` stores the body of each message in opened folders temporarily in an efficient way. As user of the module, you do not to know which way: it may be kept in memory or in a file. Independent of the way it is stored, the data is available in various ways.

When the message is created or read, some header lines are copied into the body object. This is especially important because we will process the body, which may change the `Content-Length` of the body, or the number of `Lines`. The changes must be reflected in the meta information. Some processing steps may even modify the `Content-Type`. Often you want to decode the body to be able to use it, which means that the `Content-Transfer-Encoding` changes to `none`.

4.2.1 Getting the body

A message can supply two different versions of its body: the encoded body, and a decoded version of the body. Of course, in an application it is often more useful to ask for the decoded version:

```
my $msg     = ...;          # get it somewhere

my $encoded = $msg->body;   # are you sure?
$encoded->print;            # oops, base64 encoded text?

my $decoded = $msg->decoded;
$decoded->print;            # perfect
```

Do not worry: if the body was not encoded in the first place, then no time or resources are consumed to "decode" it. However, if there is work to be done, the `$decoded` points to a different body object, which no longer has anything to do with the message from which it came.

4.2.2 Information about the body

Information about the body which was originally found in the header is copied into the body object. When you have a decoded body of a message, this data may (will) differ from what was found in the header.

The following examples may be useful:

```
print $decoded->transferEncoding;
             # was Content-Transfer-Encoding
print $decoded->nrLines;       # was Lines
print $decoded->type->body;    # text/html

my $type = $decoded->mimeType; # a MIME::Type!
print $type->subType;          # html

$decoded->print                # works best
    unless $decoded->mimeType->isBinary;
```

5 Constructing a message

There are a few ways to construct a new message. Think about whether the new message has a relation to an existing message or not. If so, what is that relationship? Although the methods are quite similar in use, their results can differ quite a lot.

The following message constructors are available:

`Mail::Message->read(\*STDIN)` **and** `Mail::Message->read(\@lines)`
: Let's start with the worst solutions. These methods are required only when an external source delivers the message data this way. For instance, when you write a `procmail`[4] application, `sendmail` will start your program and provides the message on `STDIN`. You are left without options, so need to `read`.

`Mail::Message->build(`*data*`)`
: Create a message based on a set of loose data: throw in a number of header lines, and some scalars, arrays, or file-handles with text, and the message is created.

`Mail::Message->buildFromBody($body)`
: A more careful approach to construct a message: first construct a body object, and then turn it into a full message with a header. Of course, the content related header lines are constructed based on the data which is stored in the body.

`$msg->reply`
: Create a reply based on a received message. This implies that the destination of the newly created message is the originator of the source message. Some header lines keep track of this game of answering answers, and they are updated accordingly.

`$msg->forward`
: Forwarding a message implies that you want someone else to read the message as well, but at the same time wish to add some notes to the message. The resulting message will contain (a part of) the original. Some header lines are automatically created as well.

---

[4] `procmail` is used to process incoming mail at the moment it arrives on the system. It can be used to remove spam at arrival, or channel messages to separate folders based on sender.

`$msg->bounce`

Use this when you need to forward a message to someone without modifying the content. Some minor modifications in the message header are made (specifically some `Resent-` headers are added), but that's all.

The next sections discuss small examples for each constructor. Look at the manual-page of `Mail::Message::Construct` for full details. The examples are taken from the `examples/` directory which is distributed in the `Mail::Box` module.

5.1 Constructing with `read`

To start off simple: `read` parses a message from a file handle, a string, or an array of lines. Be warned that all lines must end with a newline, including the last line.

General examples:

```
use Mail::Message;
my $msg1 = Mail::Message->read(\*STDIN);
my $msg2 = Mail::Message->read(\@lines, log => 'PROGRESS');

my $msg3 = Mail::Message->read(<<MSG);
Subject: hello world
To: you@example.com
                    # warning: empty line required !!!
Hi, greetings!
MSG
```

When you write your program in a way such that lines are first collected in an array, and then this `read` method is called to produce a `Mail::Message`, you are **not** on the right track. In such cases it is better to use two steps to produce the best result:

```
use Mail::Message;

my @lines = ... ;  # contains 8bit encoded html

my $body  = Mail::Message::Body->new
  ( data              => \@lines
  , mime_type         => 'text/html'
  , transfer_encoding => '8bit'
  );
```

```
my $msg   = Mail::Message->buildFromBody($body);
```

Have a look at the `Mail::Message::Body::Construct` manual page to learn more.

5.2 Constructing a message in one step

Building a whole message from components is more flexible than using `read`: specify data and some header lines, and get a fully dressed-up message in return.

A number of missing lines will be added for you; the `Content-` lines are calculated or predicted. When more than one source for data is specified, a multi-part message is produced. When the data needs to be encoded, it will automatically get encoded.

If one `Mail::Address` or an array of them are specified for a header line, they are flattened to a nicely formatted string.

A full program:

```
use Mail::Message;
use Mail::Address;

my $to  = Mail::Address->new('your name', 'you@tux.aq');
my $signature = Mail::Message::Body->new(...);

my $msg = Mail::Message->build
 ( From   => 'me@home.nl'
 , To     => $to
 , Cc     => 'everyone@example.com'

 , data   => [ "This is\n", "the first part of\n"
             , "the message\n" ]
 , file   => 'myself.gif'
 , file   => 'you.jpg'
 , attach => $signature
 );
```

The example above provides three header lines. Next to those, at least the obligatory `Message-ID` will be added, plus many lines about the type and size of the body. The resulting message is a multipart containing four parts: the first specified in-line, then two as external files, and the last as a prepared body object.

5.3 Constructing a message from a body

The buildFromBody method is related to build, but is used with a previously composed body object. This is better suited for creating (nested) multi-part messages, where more control on the produced message structure is required.

The overly complicated example below shows that header lines may also be passed as Mail::Message::Fields. In this case, you overrule the content-type as stored in the body: don't do that at home!

```
my $body = $some_message->decoded;   # or
my $body = Mail::Message::Body->new(...);

my $type = Mail::Message::Field->new
 ( 'Content-Type', 'text/html'
 , 'charset="us-ascii"'
 );

my @to   =
 ( Mail::Address->new('Your name', 'you@example.com')
 , 'you@example.info'
 );

my $msg = Mail::Message->buildFromBody
 ( $body

 , From => 'me@example.nl'
 , To   => \@to
 , $type
 );
```

5.4 Constructing a reply

A reply is based on a source message, and by default constructs a message ready to be sent back to the author of the source message.

There are many ways to get that to work. The complicating factor is the wish to utilize parts of the source message in the reply. This is especially hard when you want to created automated replies: think about the receipt of binary or multi-parts which have to be treated differently from simple plain text messages.

An example:

```
use Mail::Message;
my $msg     = ...;   # get it somewhere
my $pgp_key = Mail::Message::Body...

my $reply   = $msg->reply
 ( prelude         => "No spam, please!\n\n"
 , postlude        => "\nGreetings\n"
 , strip_signature => 1
 , signature       => $pgp_key
 , group_reply     => 1
 );
```

Please do not forget to terminate all lines in the prelude and postlude with a newline.

With the right options, `Mail::Box` tries to do its best for you, in any situation. When a reply is made on a multi-part message, it tries to figure out which part is the primary content to be used in the answer composition. When the source message's body is binary, it will be attached, and the prelude plus postlude are combined into a nice covering text.

5.5 Constructing a forward

Once you have seen `reply` you have seen `forward`: their interface is rather the same. The forwarded message also composes a new message based on the original, but it is encapsulated differently: the whole message body is taken to be forwarded by default, not only an extract.

Also some of the header lines will be updated differently. For instance, the subject line will start with "`Re:`" for the reply, and with "`Forw:`" for the forward. Besides, while a reply knows where the message must be sent, the forward requires that information to be specified.

5.6 Constructing a bounce

Too many automated applications `forward` or `reply` their messages when they really should use `bounce`. This may be caused by the fact that bouncing messages was a latter addition to the MIME specifications.

Examples for bounce can only be small:

```
use Mail::Message;
my $msg    = ...;
```

```
my $bounce = $msg->bounce
 ( To  => 'postmaster@home'
 , Bcc => \@the_whole_world
 ):

$bounce->send;
```

This example also contains a new feature:, by calling `send` on a message it will be... sent. The `Mail::Box` module contains packages which support various message transmission mechanisms. The easy to use `send` method tries hard to find a way which works in your environment. Of course, this is configurable as well.

6 Mail::Box

So far, only single messages have been discussed. Single messages are created in your program, and disappear when your program stops. To preserve messages they must be stored in folders. The `Mail::Folder` name-space is occupied by an implementation which is not maintained, so the implementation which can handle `Mail::Message` objects was named `Mail::Box`.

There are many kinds of mail folders, which all differ quite substantially in how they store a message. They all employ different run-time administration of the messages they contain. But in the general case you do not have to know these distictions if you are using the `Mail::Box` module.

6.1 The Mail::Box::Manager

The `Mail::Box::Manager` keeps track on all open folders, so you cannot accidentally open the same folder twice in your program. It also autodetects the folder type, which saves you work, and at the same time compiles the required modules for you. Your conclussion must be: always start with the manager.

In the next example, we open a folder via the manager:

```
use Mail::Box::Manager;

my $mgr   = Mail::Box::Manager->new;
my $inbox = $mgr->open('=Inbox'):
die unless defined $inbox;
```

```
$inbox->close;
```

As you see, no need to specify whether this is an Mbox, MH, or Maildir type of folder. When the name of the folder starts with an equals-sign (=), it will look for the name in a – folder type dependent – default directory.

Important to know: folders are opened read-only by default. Use

```
my $inbox = $mgr->open('=Inbox', access => 'rw');
```

to allow updates. When the folder is closed, it is automatically de-registered from the manager. When the folder gets closed, goes out of scope[5], or the program terminates, the changes will be written.

6.2 Getting the messages

Of course, the only reason to open folders is to get to the messages which are located in them. The main methods for this are

```
my $msg = $folder->message(3);        # fourth
my $msg = $folder->message(-1);       # last
my $msg = $folder->messageId($msgid); # that one

foreach my $msg ($folder->messages)   # all subjects
{   print $msg->subject, "\n";
}

print $_->subject, "\n"
    foreach $folder->messages;        # all subjects

print scalar $folder->messages;       # count

$msg->delete;                         # flag for deletion
$folder->message(3)->delete;          # same
```

There are many more methods available, which are described in the manual pages. The deletion of a message takes place when the folder is closed, not immediately.

[5] A `my` variable only has limited visibility, the *scope*. When the result of opening a folder is caught in such variable, it will get destroyed with the variable at the end of its scope.

6.3 Moving messages around

Next to retreiving messages from a folder, you are also able to add messages to a folder. Be warned that some of these methods work on the manager, others on a folder.

```
$mgr->moveMessage($otherfolder, $msg);
$mgr->copyMessage($otherfolder, $msg);
$mgr->appendMessage($otherfolder, $msg);
$folder->addMessage($msg);   # a Mail::Message
```

It is permitted to move, copy, or append more than one message at once. When the specified folder is open in the program, actions will take place on that opened folder. Otherwise, the updates are made as cheaply as possible, preferably without opening the destination folder.

7 Further reading

This paper only briefly introduced a few of the modules which can be used to handle e-mail with Perl. It is only an introductory story, and you certainly need to read more to write real-life applications.

Powerful libraries require study before they can be used. Sometimes they look too complex, but often that's because you do not understand the complications. Be glad for everything that is offered for free. Contribute with complaints and suggestions, so existing modules can improve. New implementations can only declare others 'bad' if they themselves are in a 'better' state.

As final remark: be very, very, very careful not to lose e-mail by mistakes in your program. People will hate you when e-mail gets lost.