Mailman – An Extensible Mailing List Manager Using Python

Ken Manheimer
Corporation for National Research Initiatives
klm@cnri.reston.va.us

Barry Warsaw
Corporation for National Research Initiatives
bwarsaw@cnri.reston.va.us

John Viega
Reliable Software Technologies
viega@rstcorp.com

Abstract

Email has a key role in the explosive growth of the Internet, calling for Mailing List Management Systems (MLMS) that can adapt to, and even foster, new forms of community organization as they emerge. A new MLMS, Mailman, is well suited to such evolution because it has been developed to be versatile and extensible. One factor contributing to these strengths is its implementation in Python.

In this paper we will look at various aspects of Mailman's extensibility. We will consider how the system's design and how features of its implementation language, Python, factor into that extensibility.

1. Introduction

1.1. What is Mailman?

Mailman is a Mailing List Management System, like Majordomo [1] and SmartList [2], used to manage email redistribution lists. Mailman gives each mailing list a Web page, and allows users to subscribe, unsubscribe, etc. over the Web. List managers can administer their lists entirely from the Web. Mailman also integrates most things people want to do with mailing lists, including archiving, mail to news gateways, and so on.

Mailman was originally developed by John Viega. Ken Manheimer picked up the ball to bring Mailman to 1.0b3. Currently, Mailman development is a Open Source [3] group effort, led by John Viega, Ken Manheimer and Barry Warsaw. Mailman has been designated by the Free Software Foundation as the GNU Mailing List Manager [4]. The Mailman home page, with distributions and background, is at [5].

See [6] for more details on the system, and visit the Mailman-developers mailing list [7] if you're interested in joining the Mailman development community.

1.2. Why Extensibility?

From the early days of the ARPAnet to today, email and Mailing List Management systems have played a central role in the formation and conduct of communities on the Internet. With the profound dynamism of the Internet, the infrastructures by which it organizes are continually evolving. Over time, the Internet's rapidly increasing scale and the advent of new and improved strategies for organization of its communities demand continuing development of the mechanisms that support them. New and different protocols may take up some types of load, as Usenet News has done, but email, as a medium, has proven to be particularly versatile and enduring. A good MLMS will help foster the evolution of the Internet communities, by growing with them.

Extensibility also offers an excellent opportunity having to do with the core constituency of mailing list users - the mailing list administrators. These administrators typically are near enough to the end-users to have clear impressions of their needs. Also, they often are technically savvy enough to implement improvements to accommodate those needs - provided the system they're changing doesn't present too high a threshold of comprehension. Here is a prime opportunity for exploiting the Bazaar-style of open-software development [8], enabling the administrators of the medium, themselves, to guide its development. This enables development more quickly and closely tailored to the needs of the user community.

Finally, most aspects of an MLMS do not require the kind of speed optimizations that force change-impeding hardening of the system. Performance critical aspects, like mail delivery to large numbers of users, are generally the purview of the underlying Mail Transport Agent, not the MLMS. Large-scale operations can impose some specialized performance demands on the MLMS, of course. Their specificity, however, enable isolating the optimizations to select components, and Python's compiled-language extensibility allows hardening of those specific components as needed This all enables a strategy of isolating rigidity to the particular subsystems

that need it – an approach for which Python is ideally suited. As it happens, we don't currently see need for this any of this kind of hardening, but we don't know what the future (or potential growth of Mailman's use) will bring.

1.3. Why Python?

Python is particularly well suited to implementing an extensive and changing system. Its combination of clean syntax and cogent semantics aids the programmer, all the more so in the process of changing existing code – that of others, as well as that of your own. It is eminently dynamic, enabling interaction with and programmatic handling of just about everything in the language. By satisfying prototyping and rapid development needs, as well as those of general programming, it can be seen to foster "continuous development", where a system continues to evolve to accommodate a changing world.

2. A Broad Overview of Mailman's Structure

Mailman's core component is the MailList object, a class instance that represents an individual mailing list. MailList instances are used in scripts to take input from an MTA, the Web, Cron, and direct operator articulation. Taking these inputs they may transmit messages via the MTA, e.g. list postings or administrative messages, change various databases, including ones for pending activity confirmations, Web page templates, and so forth, or just change mailing list characteristics settings. Figure 1 shows a block diagram of these relationships, with the MailList instance occupying the *Mailman Core Package* block at the center.

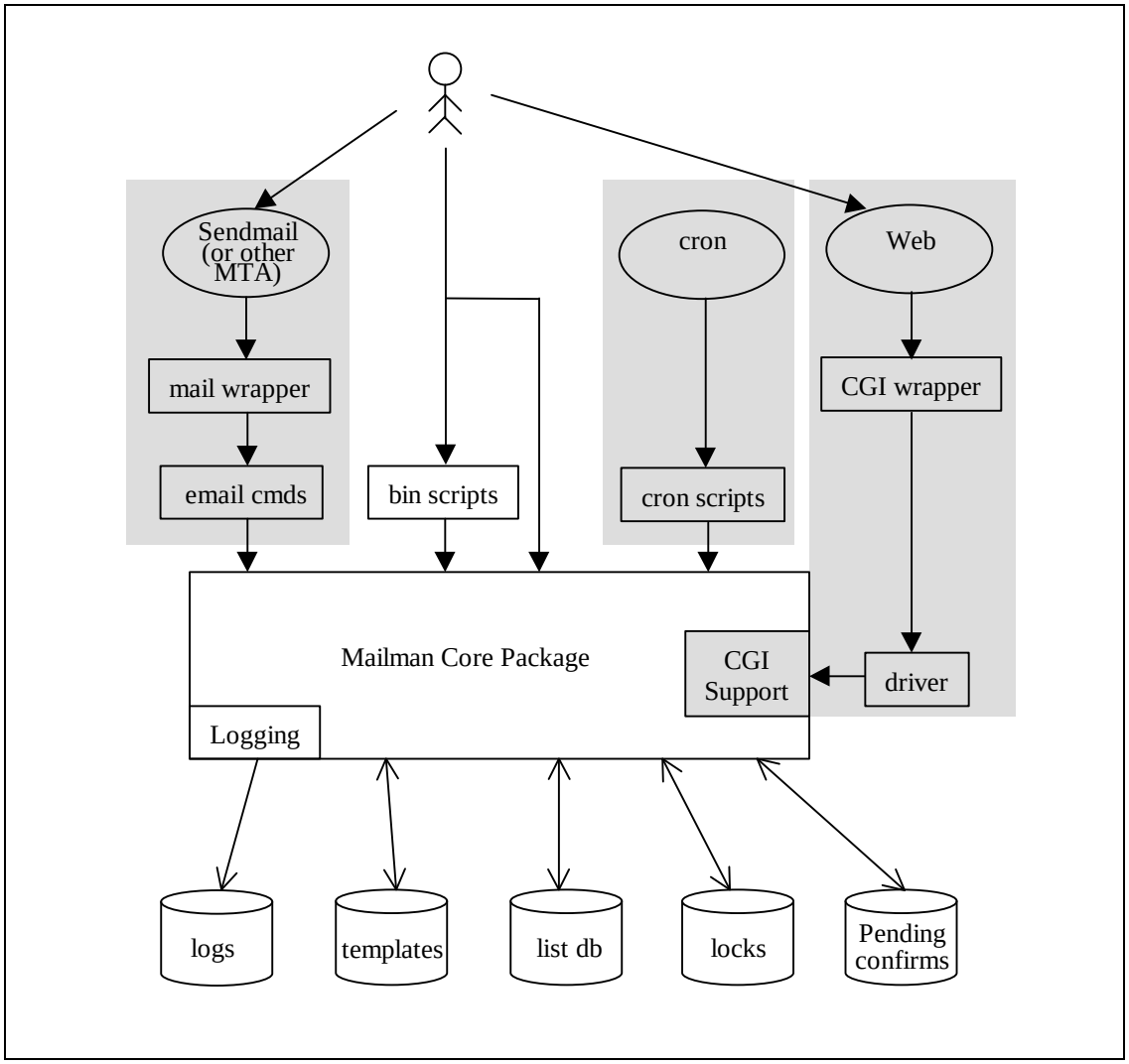

Figure 1. Mailman Block Diagram

See [6] for much more detail about the overarching organization. We focus on the operation of the MailList object in this paper.

The MailList class is composed by multiple inheritance from a number of task-oriented component classes, as mixins. The task oriented components contain the methods, variable declarations, and initializations related to the functionality of a particular subsystem; for example, that of the delivery mechanism or of the emailed-commands handler.

The code directly in the MailList class is responsible for coordination of the mixin classes initialization, central identification of the specific mailing list, creation of new mailing lists, and management of mailing lists' persistent data and locking. The internal MailList object code also handles the very top level of subscriptions and message posting, but the task-oriented base classes are responsible for the underpinnings of that and all the other functions of the mailing list object. The following base classes currently exist:

MailCommandHandler:
: This class implements the parsing and execution of Majordomo-style commands embedded in email to -request addresses. Although users more typically interact with mailing lists directly through the Web interface, for compatibility, user commands can be issued via email. Where appropriate, the commands have the same syntax and semantics as the corresponding Majordomo commands.

HTMLFormatter:
: This class is used to generate list-specific HTML for presentation via the World Wide Web interface. Primarily, it uses a widget library also included in Mailman. Together this class and library serves a purpose very similar to that of Robin Friedrich's HTMLgen [9] and Digital Creations, L.C. Document Template [10]. (These alternatives were not available during the early stages of mailman's development.)

Deliverer:
: This class conducts delivery of any of the email associated with a mailing list. This includes membership delivery of postings, subscription acknowledgments, announcements to the list administrator about list creation, list business pending approval, subscriber notices regarding their passwords, and myriad other things. Email is used for a lot of things by a mailing list system, even one with a comprehensive Web interface

ListAdmin:
: This class manages the queuing and notification of mailing list submissions - postings and subscriptions requests - that require administrator decision (approval or rejection). For example, a list may be set to require administrator approval for any postings, or a posting may be held due to triggering a filter intended to catch undesired commercial messages (can you say spam?).

Archiver:
: This class handles the archival of posted messages. Mailman mailing lists can have public or private archives, and this class places the posted message in the appropriate location. It also interfaces with external Hypertext archivers such as Andrew Kuchling's Pipermail [11], which is bundled with Mailman.

Digester:
: Mailing list members can receive posting immediately, or they can opt to have cumulative "digests" of the list traffic sent to them periodically. This class manages accumulation of the digests, formulation of the plain and MIME formats (when there are subscribers to the respective types), and dispatching of the digests to the respective subscribers.

SecurityManager:
: This class primarily verifies authorization passwords for the site administrator, list administrators, and users. It also performs the task of sanitizing the Majordomo-style approval passwords from the headers of administrator approvals submitted via email.

Bouncer:
: Mailman catches email delivery bounce notices, and accumulates tallies of bounce scores for the mailing list members. For scores that exceed designated thresholds within designated timeout conditions, the bouncer triggers list-prescribed actions, including disabling of mail delivery or, if set by the list administrator, unsubscription of the member from the list.

GatewayManager:
: This class handles optional email-to-Usenet gateways for mailing lists.

3. A Selective Tour of Mailman's Facilities, regarding Versatility

3.1. Programming and Interacting With MailList Objects

Mailman's central structure is the MailList class object, whose instances represent individual mailing lists. These classes are designed to be easily instantiated by external programs, providing a single, easy to handle package for access to almost all mailing list functionality. They constitute a compact, self-contained API for articulation of mailing lists from programs, and also for interaction with them using the interactive Python interpreter shell.

Managing their own locks, MailList instances can be used simultaneously from many processes without conflict. Programs manipulating MailList instances run under CGI for Web interfaces or via command-pipe email aliases for email interfaces. Cron scripts instantiate MailList instances to do a job. MailList-instance based scripts are used to automate any routine procedures, such as conversion of subscriber lists from established Majordomo mailing lists.

An interactive session with MailList instances provides an eminently useful development and debugging tool. MailList objects are easy to instantiate - all that's necessary is inclusion of the Mailman package directory on the PYTHONPATH, and knowledge of the right module to import. With interactive instantiation, we are able to exercise and test isolated subsystems, as well as the behavior of the MailList as a whole, and we can employ exploratory tools, like the Python debugger, along the way.

We can also use interactive sessions to do mailing list "surgery" - to operate the MailList in ways not provided for in existing scripts. Using a utility function, Utils.map_maillists(), we can apply arbitrary functions to all or to selected Mailman mailing lists at the site. This enables us to do wholesale conversions of the MailLists to accommodate, for instance, changes in the address of the site, or to search for particular members of any of the mailing lists and then do some processing on their subscriptions.

In general, in the context of Open Implementation [12], this versatility fits within the category of the "Invocation" style of opening.

3.2. MailList Object Inherits from Task-Oriented Components

The MailList object is composed from task-oriented component classes using inheritance, as "mixins". This approach enables easy sharing of component classes methods and data throughout the composite MailList object. In contrast to using explicit delegates for the components, it avoids the need to explicitly identify and pass around delegate instances in order to use the data and methods of those components.

Having all the methods and data inhabit the namespace of the primary MailList instance can lead to inadvertent name collisions. However, we feel that the system would have to get much bigger before that would become a practical concern - and at that point we could use naming conventions to prevent the collisions, while still enjoying the easy sharing. Use of multiple inheritance provides this direct sharing, along with organization of the system into distinct, conceptually motivated modules, easing debugging and development.

New major modules are still being added as task-specific mixin classes, and the process is exceptionally uncomplicated. For instance, as of this writing one of the primary authors added bi-directional mail/news gatewaying capability to Mailman. This module required knowledge of some boilerplate structure, and only minor changes to existing modules, providing a major functionality with almost plugin-style ease.

3.3. MailList Instance State Persistence

This direct sharing afforded by inheritance-based composition of the MailList's components also simplifies the MailList object's persistence mechanism. By identifying its own data members via self.__dict__, the MailList object's persistence mechanism saves and restores MailList state using a marshal. (Members that should not be saved are distinguished with a leading "_" underscore.) This exploits Python's introspection capabilities, as well as a standard, simple persistent storage facility. (The higher level, standard persistent storage mechanism, pickle, would do more work than we want or need, so we were able to avoid its overhead.) As with the sharing mechanism itself, the persistence arrangement is uncomplicated, easing approach and acquaintance by newcomers.

3.4. Logging Mechanism

Interaction with MailList instances commonly are triggered remotely - via the Web or email - or from periodically firing cron jobs. The lack of an operator or a console can make system failures in these contexts hard to trace. Of course, every program *ought* to behave perfectly, or at least fail gracefully. However, when programming in an environment where change is frequent, we need to provide some defensive mechanisms that aid the capture of the errors that inevitably slip by. Mailman's logging mechanism provides that coverage.

Reliable logging is also essential for tracking the occurrence of common events that otherwise take place "behind the scenes". This can include mailing list subscription activity, automated change of subscriptions due to delivery failures, and so forth. It also is useful to be able to use "flag-printing" debugging, even when stdout does not go anywhere useful - e.g., when running under CGI, or in disconnected forked processes, or via email.

The crux of the Mailman logging scheme is a Logger class with the job of directing messages to log files with minimal chance of disrupting operation when the logging process, itself, fails – for instance, when the log files are inaccessible. Logger objects are implemented to be defensive about what might go wrong in the logging process, and to attempt suitable alternative actions short of raising exceptions. For instance, logger objects open their log files "lazily", and avoid interrupting operations if they are unable to access them, trying alternative reporting avenues, instead

Logger instances obey the conventional Python file-like object interface protocol. Thus, they can be used to write messages like standard file objects would be. Logger objects can also be substituted for standard output streams like sys.stderr and sys.stdout, enabling, for instance, blanket capture of error tracebacks from within the modules where they occur. Time-stamped logger objects and multi-stream output variants are commonly used within Mailman scripts that run disconnected from a terminal, to capture errors.

Loggers are applied in Mailman Web-associated components with another dandy refinement. All Web CGI scripts are launched via a driver script. The driver script launches the intended, job-specific scripts within the context of an unqualified try-except statement. If any exception escapes the job-specific script - including ones that simply cannot be caught within a script, for instance syntax errors - then the driver catches the exception and handles them in a useful way. The driver produces the traceback and a listing of all the HTTP environment variable settings both to stdout (HTML formatted, for rendition on the Web), and to the error log file. This way, the Web visitor is provided with informative feedback, including e.g. instructions about contacting the site administrator, if they are inclined, and also the site has a detailed record of the error. (See Figure 2. Code Excerpt from CGI Driver Script, showing the use of error loggers and the comprehensive exception guard.)

The CGI driver script, itself, is small and carefully hardened, in order to minimize the chance that it will introduce errors where they won't be caught. The same driver is used for all underlying CGI scripts, increasing the complexity of the driver a bit, but ultimately reducing the overall complexity, and increasing the exercise, and ultimately the hardening, of the driver script overall.

Structural integration of error logging within Mailman eliminates the need for every CGI or mail handling script to itself code for logging, and it increases the detection and pinpointing of faults early in the development cycle. This incorporation depends on Python's high-level exception mechanism, polymorphism, and standard file-object protocols for a thorough, low-hassle implementation.

3.5. Web Interface

Mailman provides an interface to MailList objects via CGI, extending programmatic access to the World Wide Web. The MailList base class, HTMLFormatter, contains MailList-specific HTML widgets, built upon an HTML widget library which is also part of

```
try:
  logger = StampedLogger('error', label='admin', manual_reprime=1,nofail=0)
  multi = MultiLogger(sys.__stdout__, logger)
  scriptname = sys.argv[1]
  pkg = __import__('Mailman.Cgi', globals(), locals(),'scriptname')
  module = getattr(pkg, scriptname)
  main = getattr(module, 'main')
  try:
      main()
  except SystemExit:
      # a valid exit
      pass
  except:
     print_traceback(logger, multi)
     print_environment(logger)
```

Figure 2. Code Excerpt from CGI Driver Script

Mailman. The underlying library provides a modest range of HTML document presentation and CGI form widgets, as well as cookie handling for authorization. Together with complete access to Mailman mailing lists via the MailList object, this general mechanism enables publishing to the Web of access to any aspect of MailList.

On this we build typical Web-related functionality, such as an overview of the mailing lists on the site, and review and subscription to particular lists, available via the Web. (See Figure 3. Mailing List Home Page.) In addition, we also extend administrative customization of MailList operation (see the Configuration Options section, below), administrative action on the disposition of subscriptions and postings being held for approval, and subscriber control of their subscription status, customization options, and password, among other things.

The elaborateness of programmatically generating Web documents, and the lack of a local operator and error console when the generation typically happens, can complicate development and debugging of the process. Use of Mailman's logging utilities, as described above, provides reporting of unexpected errors, and also provides convenient means for debugging flag "printouts" when exercising Mailman's Web interfaces via the Web.

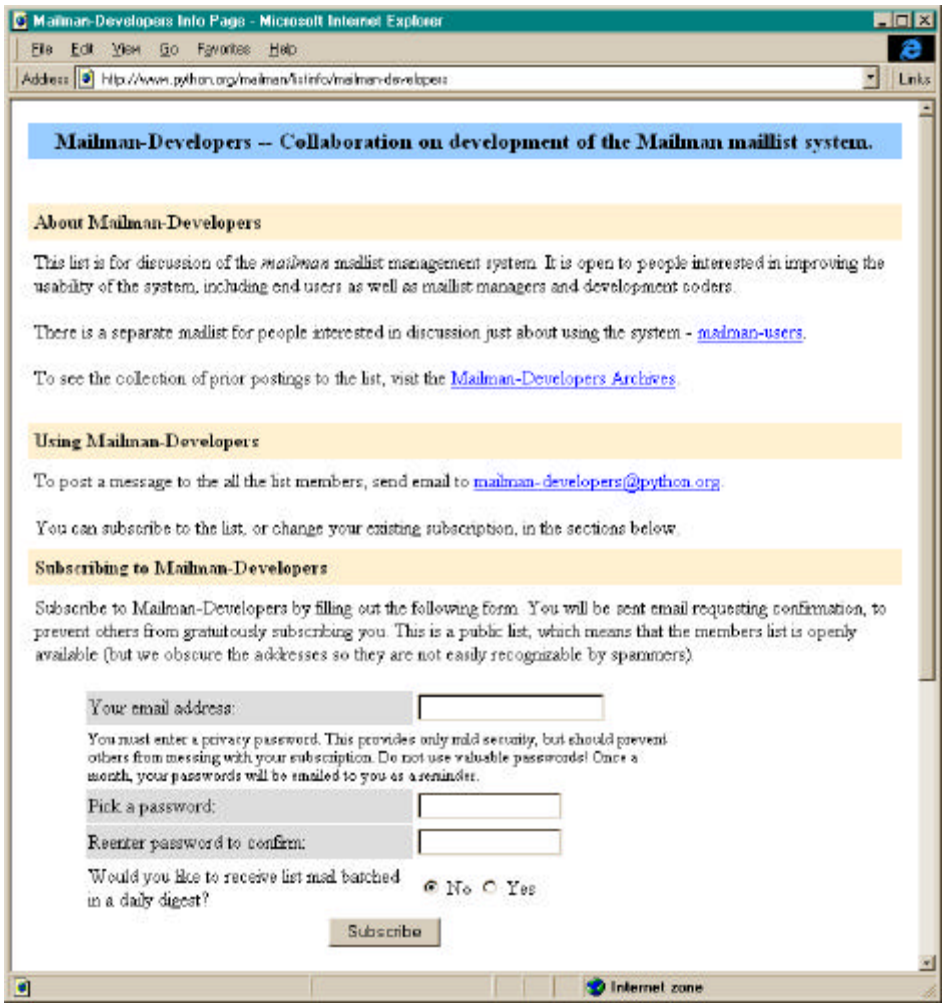

Figure 3. Mailing List Home Page

3.6. Configuration Options Mechanism Exploits Namespace Dynamism

One significant subsystem demonstrating the power of the interface between MailList objects and the Web is the mailing list customization-options mechanism. (See Figure 4. General Administrative Options Page.)

MailList configuration options are expressed as simple data structures (specifically, as tuples) specifying the

name of the MailList's data member which contains the underlying setting, the type and layout of the HTML user interface element for the option, a brief description, and an optional elaborate description. These options are collected into lists according to rough categories, e.g. list-privacy specific options, or digest specific settings. (The option lists also include string entries that are used to annotate their presentation, typically including at least a header describing the category of the set.)

These option descriptors dictate the contents of Web pages by which the mailing list administrators customize the behaviors of their mailing lists - coupling the CGI widgets on the pages with the underlying settings in the MailList objects. Python's dynamic namespaces and high-level data structures, among other things, enables this simple mechanism to couple user interface with the underlying data members.

Its elementary nature, in turn, simplifies the process of adding new configuration variables or changing existing ones - a common occurrence when new features are added or existing ones are changed.

The early formal structuring of the options has provided another benefit - it enables central enhancement of the options mechanism as a whole. One recent example is addition of a help mechanism, which entailed adding the optional slot for elaborate descriptions and a corresponding addition to the presentation mechanism to offer help for those

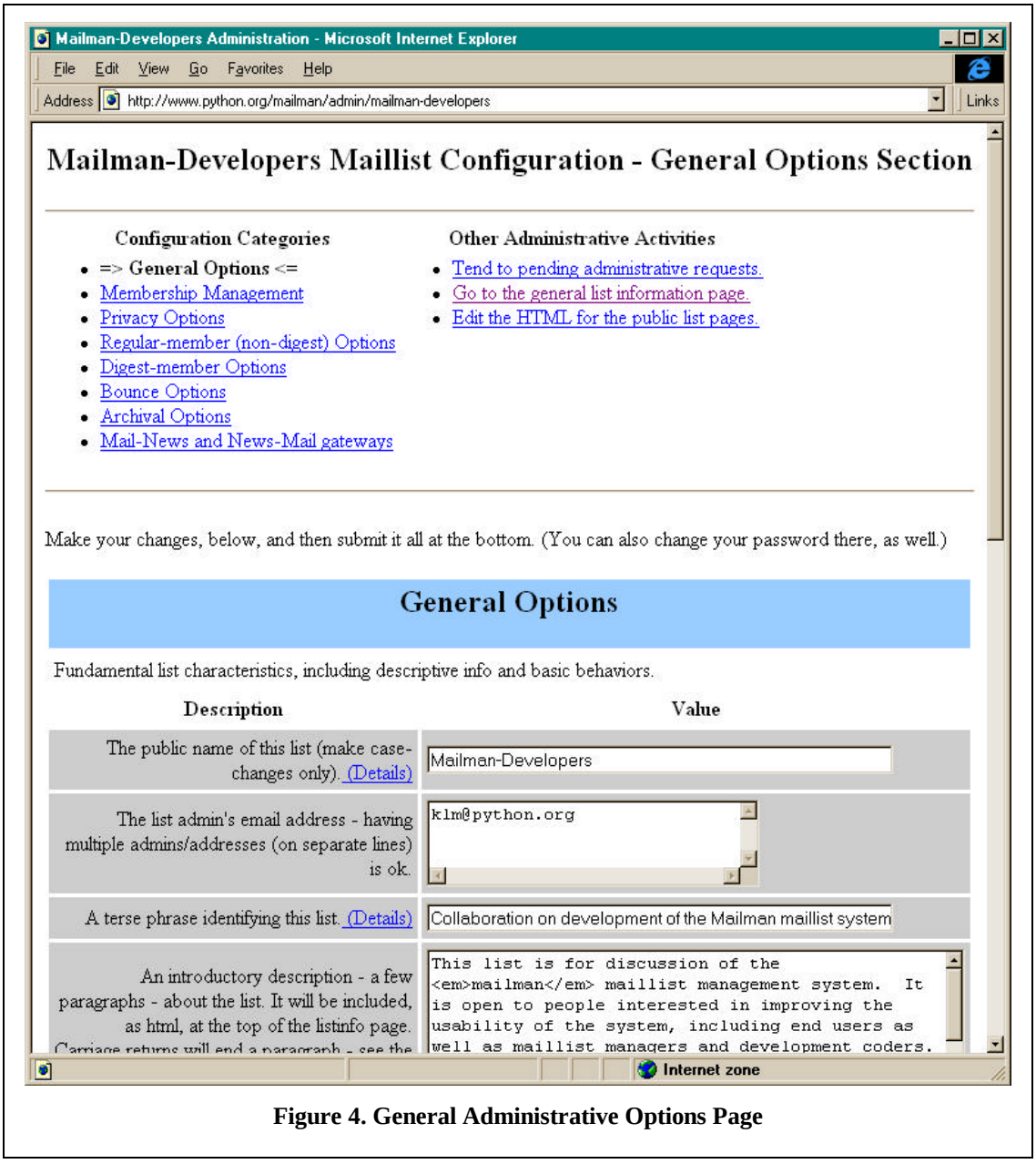

Figure 4. General Administrative Options Page

variables that contain the elaborate description.

These option description tables could and should be divided into plugin directories, to further isolate the introduction of new options from the main body of the program. This would afford two benefits:

- Isolation of the program from disruption due to faults in the option descriptions (which tend to be changed more commonly than other parts of the program).
- Reevaluation of the option descriptions while the program is running (which will be particularly useful when the program is able to run as a persistent daemon)

Languages lacking the ability to directly access and effect runtime namespaces could not do any of this without significant and cumbersome indirection.

4. Drawbacks, Lessons and Open Questions

We discussed a small sample of some key Mailman features exhibiting the versatility of the design and implementation. Below we discuss some inherent drawbacks, and also some lessons learned and open questions we're still pondering.

The MailList object use of mixins means that it gathers all method and data member names in the same namespace. This requires some consideration of the code for the other components to avoid collisions. Considering that our aim is for these components to use eachother's facilities, this is not a problem.

The Mailman configuration options compound this danger by directly populating the list object with numerous data members representing the options values. We should reduce this load by encapsulating the options within a class object tailored to getting and setting the options as attributes. This would also afford additional functionality on options, such as better defaulting relationships. In this approach, MailLists that do not customize an option would track changes to the central setting, while currently the MailList's lock-in separate copies of the settings when they are created.

Early versions of Mailman used broad, unqualified except clauses, masking unintended exceptions and making it extremely difficult to track down the origin of faults contained therein. In practice, unqualified except clauses should never be used unless the intention is to catch and actually handle any contained failures. (Code that does general failure handling can be seen as an executive of the code being handled. For example, the single CGI driver script, which directs fault and debugging info to the appropriate destinations, plays this role with respect to the CGI scripts.) In general, except clauses should be as completely qualified as possible, and should be moved as close to the exception they're meant to catch as can be handled.

One fundamental question concerns the pitfalls of dynamic typing in a large system. In statically typed systems, the requirement for explicit variable declaration exposes most typos and many types of interface misuse at compile time. Python's handling of dynamic types exposes such errors only at runtime, and only when the faulty code is executed. This can mean that errors in obscure branches can stay hidden for a long time - and pop up when least expected. As systems get larger it becomes harder to exercise all the code paths, increasing the difficulty of exposing such errors. At what point do the benefits of static typing outweigh the versatility of dynamic typing? Are the respective benefits even directly comparable?

5. Conclusion

The continuing dynamic evolution of the communities served by Mailing List Management systems suggests that these systems are perpetually unfinished, with at least some aspects undergoing evolution. Prototyping and rapid development are among Python's clear strengths, and invaluable in this regard. (A uniform, flexible architecture is pretty essential, too.)

Mailman exploits many of Python's features, including native object orientation, multiple inheritance, polymorphism, high-level control structures like exceptions, conventional protocols, dynamic access to namespaces, cogent data structures, and a wealth of standard libraries. The power of the language, combined with its tendency to readability, enables development of sophisticated systems with approachable, untortured code. Inherent introspection and organizational features like classes and modules promote the flexibility of open implementation.

We have discussed a few of the ways all this has paid off, including experiences with easy integration of valuable new subsystems, and increasing incidence of contributions to the development effort from the user community, even this early in Mailman's existence. We consider Python to be instrumental in these benefits, and a very good choice for this kind of project.

References

[1] D. Barr. The Majordomo FAQ. http://www.greatcircle.com/majordomo/majordomo-faq.html

[2] The SmartList FAQ. http://www.mindwell.com/smartlist/

[3] OpenSource.Org. http://www.opensource.org

[4] Free Software Foundation. GNU's Not Unix. http://www.gnu.org

[5] Mailman Web Site, list.org. http://www.list.org/

[6] John Viega, Barry Warsaw, Ken Manheimer. Mailman: The GNU Mailing List Manager. In Proceedings of the 12th Large Installation Systems Administration Conference (LISA '98), Dec. 1998.

[7] Mailman-Developers MailList; mailman-developers@python.org; subscribe: http://www.python.org/mailman/listinfo/mailman-developers

[8] Eric Raymond. The Cathedral and The Bazaar. http://www.earthspace.net/~esr/writings/cathedral-bazaar

[9] Robin Friedrich. HTMLgen. http://starship.skyport.net/crew/friedrich/HTMLgen/html/main.html

[10] Digital Creations, L.C. Document Template. http://www.digicool.com/releases/bobo/DocumentTemplate-rn.html

[11] Andrew M. Kuchling. Pipermail. http://starship.skyport.net/crew/amk/maintained/pipermail.html

[12] Gregor Kiczales, Andreas Paepke. Open Implementations and Metaobject Protocols. (Work in progress.) http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Kiczales-TUT95