Spam Filter Architecture

Strawman Architecture Description

Michal Young, 13 October 1998

Aside

This is not the "correct" solution to the exercise posed for the OMSE software architecture class; the exercise does not have a "correct" solution. It may not even be a particularly good solution, although I have not purposely introduced flaws. At the risk of embarassing myself, I am simply contributing one solution, which should be subject to the same critical evaluation as possible solutions contributed by other members of the class.

Introduction

This document is a sketch of an architecture for a spam filter application. More precisely, it describes an organization intended to cover a family of possible spam filter products; these products would have different execution architectures and feature sets, but would share a module dependence architecture with as few differences as possible.

Desiderata

The tentative requirements and characteristics that we have been informed of include several constraints that may be difficult to satisfy simultaneously with a single product. The organization described here is therefore motivated mainly by flexibility to produce variant products (a "product family") by combining basic parts in different ways. That is, the design decisions that are "hidden" in the major components include, as far as possible, decisions about how the components are combined to form a product with a particular run-time architecture. Of course, not all components can be ignorant of the overall organization; these decisions have therefore been centralized in a few components. The idea is that it will be cheaper and less error-prone to replace a few such components entirely, rather than distributing such changes through the product(s).

Key Separations

The architecture[1] is designed with certain key separations in mind. Each of these separations is motivated either by likely differences between two variant spam filter products, or by likely changes to a single spam filter product over time (e.g., adding new filtering features). The key separations are:

- Spam filter functionality is separated from "packaging." By "packaging," I mean whether the functionality is invoked through a mail program plug-in interface, by a stand-alone program running on a mail host computer, by a stand-alone program running on a mail client

1. I will write "the architecture" for brevity in this section; a later section will distinguish the overall organization of module dependencies (which is what "the architecture" refers to here) from other aspects of architecture, including run-time organization.

program, etc.

Rationale: Permit variant products to be packaged as stand-alone programs to be executed on mail server or client machines, as well as plug-ins for various mail programs.

- Classification is separated from manipulation of mail.

Rationale: Variant products that manipulate mail in different ways (e.g., by accessing mail storage directly, by invoking functionality of a mail program plug-in interface, or through POP3 or IMAP) may identify potential spam in the same manner.

- Classification policy from mechanism.

Rationale: The same basic mechanisms (regular expression matching, mail parsing, etc.) can support many different policies. Although not all product variants will support exactly the same mechanism, and new mechanism may be added over time, changes to policy may occur much more rapidly (by analogy to virus detection utilities that periodically download new tables of virus "signatures.") Moreover, policy could vary from user to user (e.g., through scripting), while we do not anticipate that it will be necessary to support dynamic addition of new classification mechanism by users (e.g., through a plug-in interface).

- Major classification mechanism components from each other.

Rationale: While we expect the basic mechanisms used in mail classification to change more slowly than the policy, it is likely that it will vary between products and possibly over time. For example, it might be desirable to omit some advanced capabilities like collaborative spam filtering from the initial version of the product, to reduce time-to-market.

Example invocation architectures

We found it useful to sketch the "invokes" architecture of product variants to evaluate and refine the overall module dependence architecture.

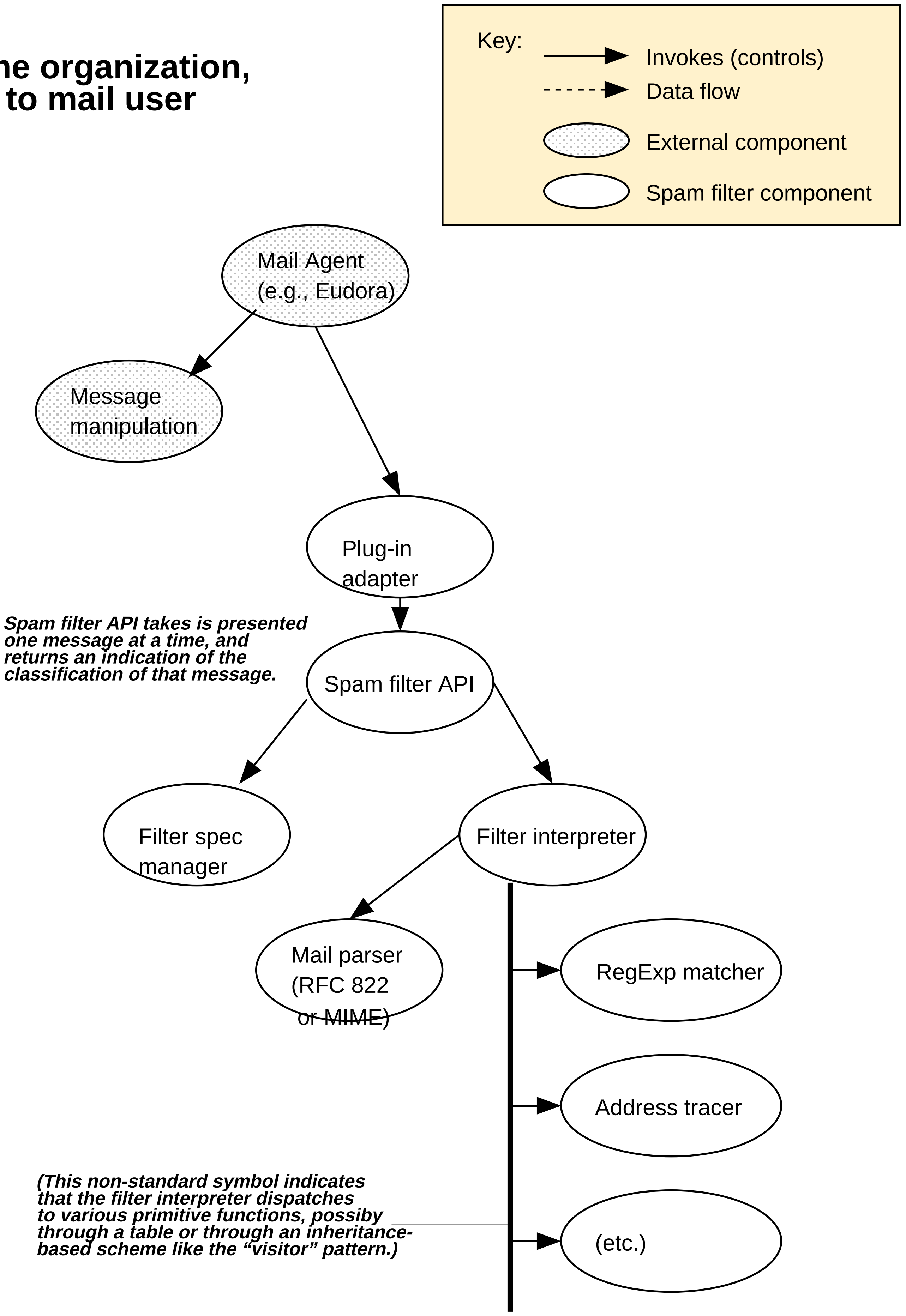
Mail agent plug-in

We begin with a product packaged as a plug-in for a mail agent such as Eudora. In this organization, we assume actual manipulation of the message is left to the mail agent. The mail agent may present one message at a time to the plug-in, or it could present a block of messages; the "adapter" would present one message at a time to the spam filter API, which would return an indicator of the message classification. (If message manipulation, e.g., refiling the message, were also to be performed by the plug-in, it would be invoked by the plug-in adapter.) This design is shown in Figure 1 below:

In this invocation architecture, the "mechanism" of classifying messages is divided mainly into two components, a message parser and an interpreter. The message parser is responsible for extracting named parts of messages (e.g., it can be asked to extract the "From:" field of a message, or each of the "Received: " fields). The message parser is the only component that depends on the rules of internet mail formatting, and should be (almost) the only the component that changes if,

Figure 1.

Run-time organization, plug-in to mail user agent



Spam filter API takes is presented one message at a time, and returns an indication of the classification of that message.

(This non-standard symbol indicates that the filter interpreter dispatches to various primitive functions, possiby through a table or through an inheritance-based scheme like the "visitor" pattern.)

for example, the classification mechanism were extended to allow filtering on the types of MIME attachments.

The filter spec manager returns a (possibly named) set of instructions to be executed by the filter interpreter. It is separated from the interpreter because of potential dependencies on the invoking environment, e.g., if it provided some user interface through the mail agent. The “invokes” relation shown in this figure fails to characterize two other important relations: The component shown here that retrieves a classification specification is part of a module or subsystem that also manages creating and editing such specifications (and perhaps also retriieving canned specifications from a server, during periodic product updates). It shares with the filter interpreter a dependence on a language for defining filtering criteria (which is not shown because it is not something that exists at execution time).

The filter interpreter is dependent on the mail parser for retrieving individual fields of messages as specified in a filter specification (e.g., it retrieves the “From: xxxx” field if a filter specification classifies all mail from certain addresses as spam). This dependence should be fairly stable, but nonetheless the interface of the mail parser does not prescribe a fixed set of fields that can be retrieved; rather, a field is specified as a string, and the contents of the field are returned as strings. Note that there must be an iterator-like interface to the mail parser, since a single field may appear many times (e.g., there may be many “To: xxxx” fields if the message has many recipients.)

While the filter interpreter may invoke several different components to perform its functionality, it is undesirable for this invocation relation to create a “uses” relation; that is, adding or removing a component like the address tracer should not require changing the interpreter. In practice this cannot be entirely true; actually what is shown here as a single “interpreter” component must be further factored into a core interpreter and a mechanism for extending that core interpreter with additional mechanism. This is a fairly common structure; it appears, for example, in the Emacs text editor through dispatch table, as well as the “visitor” design pattern [Gamma et al] for object-oriented systems. Since we can be fairly confident of being able to structure the interpreter in this manner, we will not break it down further in this architectural sketch.

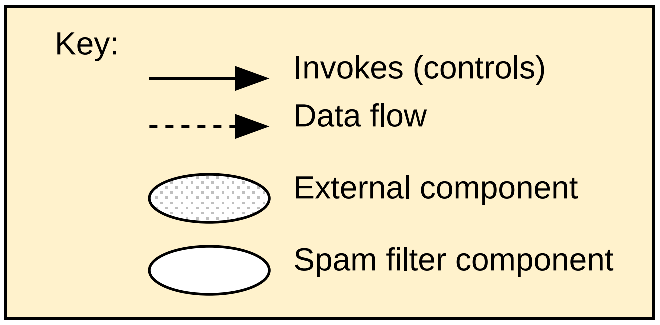
Remote POP3/SMTP forwarding-based filter

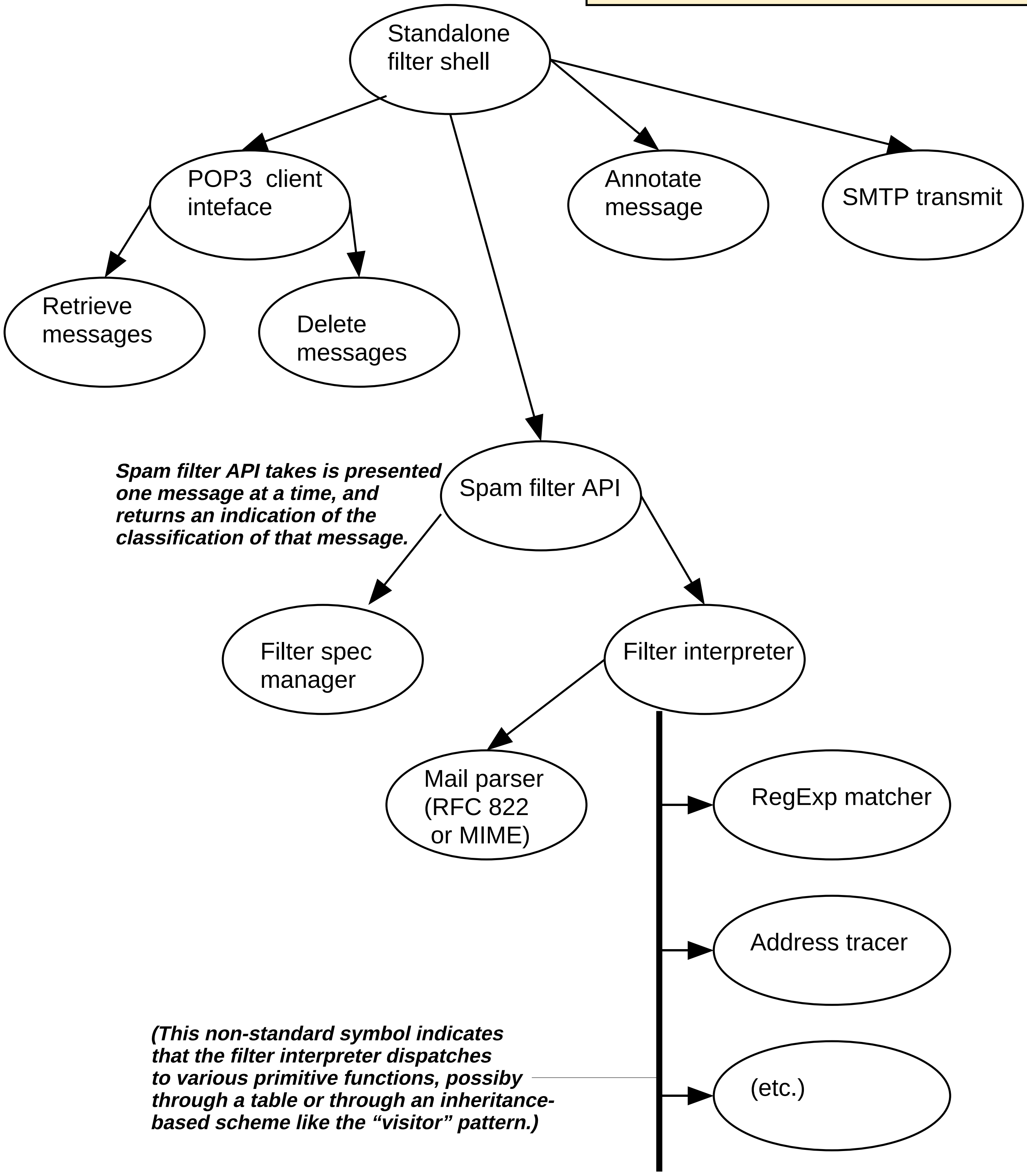
As a check on the structure above, we consider how it might be modified to produce a filter that operated as a stand-alone application program. We imagine a filter that works by “selective forwarding,” retrieving messages using POP3 and then forwarding to another or the same address those that are not classified as spam. A minor variation is that all messages are forwarded, and an “X-spam-class: xxxxx” field is added to each (e.g., “X-spam-class: probably-not”). This architecture is illustrated in Figure 2 below.

This change of “packaging” appears to be straightforward; some new components are added, and one (the plug-in adapter) is omitted, but there seem to be no changes to the remaining components. Unfortunately it isn’t quite as clean as one would guess from the “invokes” diagram. First, the “filter spec manager” component is likely to be changed, or possibly replaced entirely to suit a different environment. Second, the “annotate message” component shares knowledge of mail message formats with the mail parser; these two components should be part of the same module or subsystem.

Figure 2.

Run-time organization, POP/SMTP-based remote filter





Spam filter API takes is presented one message at a time, and returns an indication of the classification of that message.

(This non-standard symbol indicates that the filter interpreter dispatches to various primitive functions, possiby through a table or through an inheritance-based scheme like the "visitor" pattern.)

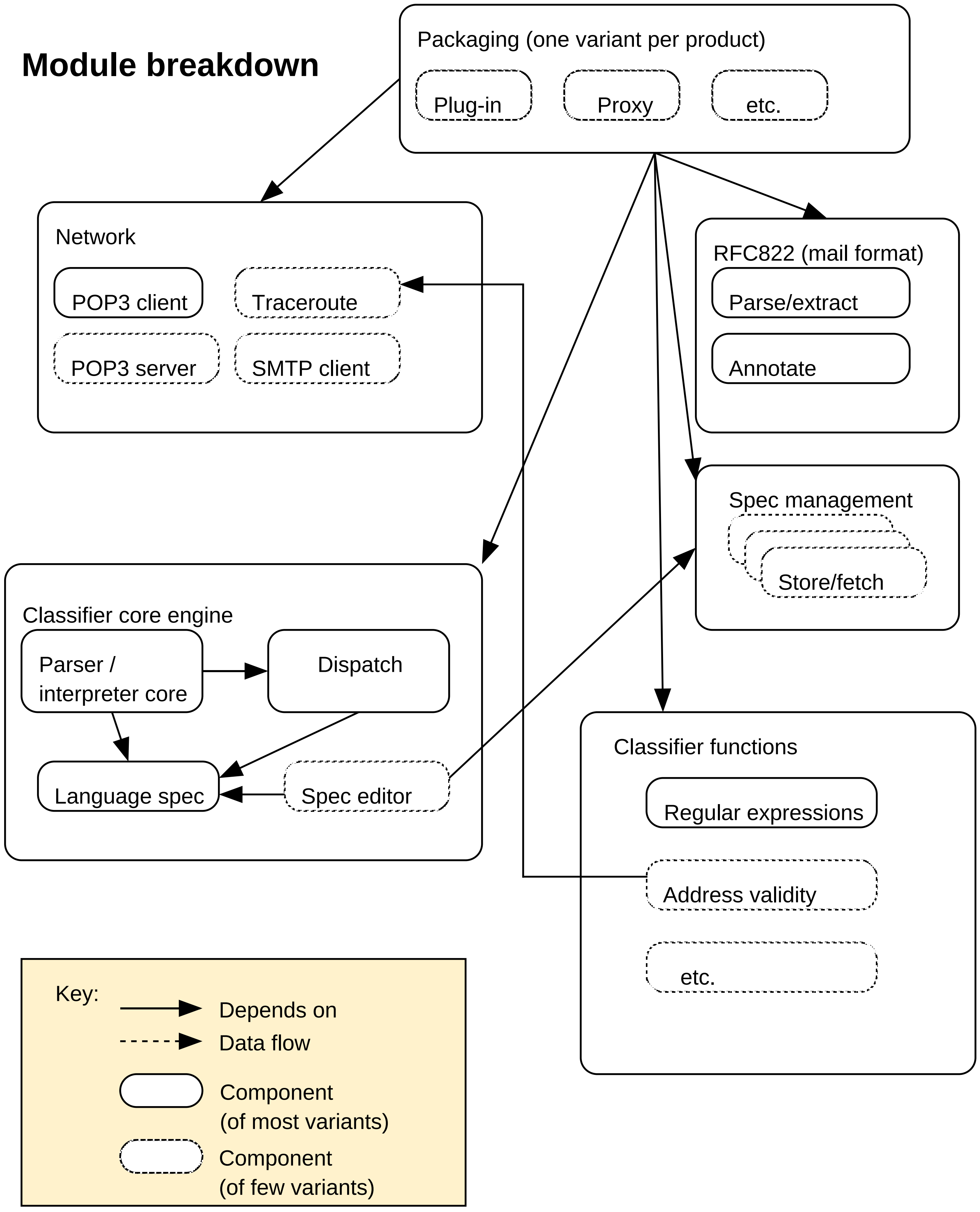
POP Proxy

One can easily imagine a variation on the “selective forwarding” or “forwarding with annotation” schemes of the previous example. Suppose we are checking email on a Palm Pilot or other PDA, with insufficient resources and interfaces to support a plug-in spam filter. We might instead implement the spam filter as an application on a host computer. The spam filter would present a POP interface to the PDA, but would provide that interface by communicating with another POP server (such arrangements are sometimes called “proxies”). It would, of course, apply filtering on messages before sending them on. It is apparent that a minor variation on the POP/SMTP filtering scenario suffices for this, with a new “POP3 server” component but with few other changes.

Modules and dependence

We now discuss an architecture of module dependence, rather than invocation relations. The module dependence architecture is more closely tied to a work breakdown structure than is the invocaton relation (e.g, it will reflect the fact that mail parsing and mail annotation should almost certainly be implemented by the same individual or team). However, it is somewhat more difficult to understand “at a glance” because it is a structure designed to accomodate several variations on the product. This is illustrated in Figure 3 below.

Module breakdown


Packaging (one variant per product)
Plug-in
Proxy
etc.
Network
POP3 client
Traceroute
POP3 server
SMTP client
RFC822 (mail format)
Parse/extract
Annotate
Spec management
Store/fetch
Classifier core engine
Parser / interpreter core
Dispatch
Language spec
Spec editor
Classifier functions
Regular expressions
Address validity
etc.
Key:
Depends on
Data flow
Component (of most variants)
Component (of few variants)