Open Source Collaboration Codified

Diplomarbeit im Fach Informatik

vorgelegt von

Ke Chang

geb. *19.09.1981* in *Chengdu, VR China*

angefertigt am

Department Informatik
Lehrstuhl für Informatik 2
Programmiersysteme
Friedrich-Alexander-Universität Erlangen–Nürnberg
(Prof. Dr. M. Philippsen)

Betreuer: *Prof. Dr. Dirk Riehle, Dipl.-Inf. (FH) Carsten Kolassa, M. Sc.*

Beginn der Arbeit: *16.08.2010*
Abgabe der Arbeit: *16.02.2011*

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 2 (Programmiersysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den *16.08.2010*

Ke Chang

Abstract

When using mailing list as a collaboration tool, (open source) software developers are following various usage patterns. In order to improve the efficiency of open source collaboration, this thesis tries to identity these existing patterns by analyzing the mailing lists of popular open source projects, then proposes an annotation schema to codify these patterns. A mailing list archiver application is also implemented, which applies the codifications to handle email messages, provides tool supporting for the improvement.

Keywords: Open Source Software Development, Collaboration, Mailing List, Conversation Action, Usage Pattern, Email Message, JavaMail API, Google Web Toolkit (GWT), Hibernate, PostgreSQL

Contents

1 Introduction

The term “open source”, when used on software, stands not only for a characteristic – the source code of these software are publicly available, but also stands for a development method or practice, which utilizes distributed peer collaboration and transparency of process [15].

1.1 Open Source Software Development

Open source software development describes the process, in which open source software is developed [31]. Nowadays, compared to the development process of most commercial software, open source software development shows some clear distinctions. One well known metaphor is treating the two processes as “The Cathedral and the Bazaar”, respectively [23]. Most commercial software, as well as free software like Emacs and GCC, are developed using a “cathedral” model, where a small group of developers employ a top-down design method to craft the product, the source code is restricted to them only; In the contrary, most free and open source software is developed using a “bazaar” model, where the source code are publicly available, lots of people, including developers and users, participate in the project from distributed locations, and employ a bottom-up design method.

1.1.1 Characteristics and Collaboration

As is summarized in [24, 30], open source software development should show the following characteristics:

- **Users should be treated as co-developers.** They could access the source code and are encouraged to contribute, such as giving feedbacks, submitting bug reports and patches, write documentations etc. Quoting the “Linus’s Law” — “Given enough eyeballs all bugs are shallow” — If lots of developers and users collaborate on a project, that project will eventually obtain a very good quality.
- **Early releases.** Open source software tend to be released early and often, in order to get more potential co-developers to contribute early.
- **Frequent integration.** Patches and/or code changes are merged into code base as often as possible. So bugs could be fixed in a relative short time.

- **Several versions.** Such as a stable release version and a buggier development version. Users can choose to use the latest features and help the development by reporting bugs etc.
- **High modularization.** Modular structure enables parallel development of independent components.
- **Dynamic decision making structure.** There is usually an organizational structure behind the project to make strategic decisions.

From those characteristics listed above, it can be concluded that one important aspect of open source software development is **"collaboration"**. Whether the "enough eyeballs" inspect bugs, potential co-developers contribute project assets, or parallel components development and decision making, to ensure all these activities to run correctly and effectively, a well organized cooperation is required. Also, geographically distributed development could lead to misunderstanding, miscommunication and coordination problems, because the awareness of the activities of developers at remote sites is significantly reduced [1].

1.1.2 Supporting Tools

Open source software development, and it is supported by various tools. These tools include [31]:

- **Communication channels.** Some electronic means of communication are required to overcome the lack of face-to-face meetings. E-mail is one of the mostly used medium among open source developers and users. Usually mailing lists are set up to deliver emails to all interested parties at once. Mailing list is often the most active place for communicating in the project, it also serves as the "medium of record" [6]. For real time communication, many projects use an instant messaging method such as IRC (Internet Relay Chat). Recently, web-based forums and wikis also become a popular way for users to get support information and to interact with developers.
- **Software engineering tools**
  - **Version control systems.** These tools enable open source software developers to manage code changes, such as version reverting, code forking/merging, conveniently. They also enable the public to get access to the source code. Examples of such systems are Concurrent Versions System (CVS), Subversion (SVN) and Git.
  - **Bug trackers and task lists.** For large-scale projects a bug tracking system is needed to keep track of the status of various issues in the development of the project. It also enables developers to coordinate with each other and plan

releases. Some commonly used bug tracker systems include: Bugzilla, Trac, GNATS and Mantis.

- **Testing tools.** Because open source projects undergo frequent integration, tools that can help automate testing during system integration are used.
- **Package management.** This type of system is a set of tools that automate the process of installing, upgrading, configuring and removing software from the operating system. Examples are the Red Hat Package Manager (RPM) and Advanced Packaging Tool (APT). Both of which are commonly used by Linux distributions.

Since collaboration is vital in open source software development, the tools that support communication are especially important, one of which is the mailing list.

1.1.3 Mailing List and Its Usage

Mailing lists are essential for project communications. Basically, it is a special usage of email that allows for widespread distribution of information to many Internet users. Modern mailing list systems usually provide features like email- and web-based subscription, digest mode, moderation, administration interface, header manipulation and archiving [6, 28].

As stated in the sections above, mailing list is the place where open source software developers communicate most actively. There are various usages, for example: general discussions, release announcements, questions and answers, vote for decisions, etc. The archive function of mailing list software provides a way to collect and store past messages, indexing and searching for these messages can also be supported.

1.2 Problem and Motivation

In this thesis, two problems will be concerned.

First. In order that a complex project be successful, its team member should be able to interact productively, so that relevant knowledge can be acquired, generated and circulated effectively in terms of time and cost [26]. Therefore, if developers could communicate — through the main communication channel: mailing lists — more effectively, the projects they are participating would benefit from the increased productivity.

The messages in a mailing list, by their nature, are only normal emails. There are no built-in properties or components in an email to give itself semantic meanings. There is also no formal rules to categorize emails. Thus, some activities in the use of mailing lists could be less effective, such as identifying emails of a specific type, which the developers are interest in, or obtaining emails of a same context.

Nevertheless, participants in a mailing list are actually following various usage patterns, e.g. using certain keywords in an email's subject to highlight its purpose. One possible contribution is to find out and document these usage patterns, make them best practices and encourage developers to use them. This way, developers in mailing lists could communicate based on an acknowledged convention, thus the communication becomes more effective.

By analyzing the common usage patterns of mailing lists and following the development process of open source software, this thesis proposes a schema for categorizing the messages in a mailing list. More specifically, it is an annotation (tag) schema, in which various annotation tags can be used to flag certain message with various semantic meaning. Furthermore, this tag schema also serves as a codification for some of the best practices being used in open source collaboration.

Second. Unfortunately, most popular mailing list software today (e.g. GNU Mailman, Procmail SmartList, Ezmlm and Hypermail etc) serve a generic purpose, they are not specifically tuned to support open source software development. They may have powerful features of email distribution and subscription management, but most of them are lack of features like organizing emails by type, or performing certain actions based on email's semantic meaning, which, as mentioned above, could improve the efficiency of open source collaboration.

One possible contribution to this problem is that, based upon the above mentioned codification of the best practices, implement support for these best practices on top of mailing list software. For example, in the mailing list archive, one can filter emails by type, emails could be marked up with tags by their semantic meanings to draw developer's attention, and so on.

1.3 Thesis Outline

In the following chapters, the two problems stated in the above section will be addressed in details.

In chapter 2, the theory basis for modeling communications will be introduced, as well as the annotation tag schema and the categorization criteria for emails. In chapter 3, with respect of the tag schema, selected mailing lists will be analyzed to find out the actual usage patterns. Results analysis are also demonstrated. In chapter 4, the codification for the practices, or the tag schema, is proposed in details, with explanations and usage scenarios. The validation for the contribution will also be shown in form of a survey result. Then in chapter 5, an mailing list archiver application prototype, which supports the use of tags on emails, will be presented, from its requirement analysis to design and implementation, as well as its usage scenario. Besides conclusion, chapter 6 will also suggest some improvement possibilities / future works for the tag schema and the supporting tools.

1.4 Related Works

There are several contributions that are related to open source collaboration and mailing lists. Some of the works examine the collaboration characteristics of open source development by analyzing data (most of which are retrieved from mailing lists); Some works also propose categorization/annotation schema to identify different communication patterns in open source collaboration.

Madey et al hypothesize that open source software development can be modeled as self-organizing, collaboration, social networks. They've analyzed structural data on open source projects from SourceForge.net to find evidence in the presence of power-law relationships on project sizes, project membership and cluster sizes [18]. Toral et al model mailing list behavior in open source software projects, use a set of descriptors that could inform about their quality and evolution. They select the mailing list of ARM embedded Linux, analyze the messages to obtain the underlying patterns of behaviors based on several factors, e.g. number of messages, number of threads without an answer, etc [26]. Tang et al investigate the impact of global participation on communication on the developer mailing lists of PostgreSQL and GTK+ [25]. Ohira et al propose an analysis method for observing the time-lag of communications among developers in an OSS project and facilitating the communications effectively, they have conducted a case study based on the data from the mailing list of the Python project [21].

Yamauchi et al. have employed content analysis methods to find out the communication patterns in mailing lists, their findings suggest that spontaneous work coordinated afterward is effective, rational organizational culture helps achieve agreement among members and communications media moderately support spontaneous work [36]. Ankolekar et al. consider the application of semantic web technology to enhance the open source development environment [1]. Koivunen et al describe a metadata based annotation infrastructure and explain how it can be extended [16].

This thesis focuses on finding out practices that developers are already using. The data being analyzed are restricted on the subject text of the messages in mailing lists. The proposed annotation schema is made up by a number of recommended tags which also codify the collaboration practices in many open source projects.

2 Conceptual Model of Communication

In order to address a suitable categorization criteria for the messages in mailing lists, it is necessary to examine some of the existed conceptual models of communication first. In this chapter, the theory basis for modeling communication and collaboration will be introduced, the proposed annotation schema are based on the theory.

People communicate mainly by using languages, both in real life and in electronic media, which include mailing list. One perspective used to investigate the human cooperative activity, is that takes language as the primary dimension [35], this is called the "Language Action Perspective".

2.1 The Language Action Perspective

The language action perspective (LAP) is the basis of several approaches to business modeling and information systems modeling. This perspective emphasizes that communication is not limited to transfer of information, but is one kind of action [7]. The major source of inspiration for LAP approaches is the "Speech Act Theory" (SAT). In thses LAP approaches, different communicative actions are classified in accordance with the classification scheme defined by Searle (1979) — things one can do with an utterance: [32, 35]

Assertive/Representative Commit the speaker to the truth of the expressed proposition, e.g. reciting a creed.

Directive Attempt to get the hearer to do something, including both questions and commands.

Commissive Commit the speaker to some future course of action, e.g. promises.

Declaration Bring about the correspondence between the propositional content of the speech act and reality, e.g. pronouncing someone guilty or pronouncing a couple married.

Expressive Express a psychological state about a situation, e.g. apologizing and praising.

However, some LAP approaches go beyond single speech acts, there is a great interest for speech act patterns, i.e. how different acts are related to each other [7]. These approaches classify communicative actions using their own variant schemes. For example, in Action Workflow as well as DEMO [5] there is a pattern of four sequentially organized speech act types:

- Request
- Promise
- Statement
- Acceptance

One can conclude that the LAP approaches are built upon two theoretical basis: 1) Communication is action in accordance to generic speech act types; 2) Communicative acts are organized and framed in accordance with predefined patterns [7]. Another important example of the latter is Winograd and Flores' (1986) "Conversation for Action".

2.1.1 Categorization of Conversations

In conversation for action, one party (A) makes a request to another (B). The request is interpreted by each party as having certain conditions of satisfaction, which characterize a future course of actions by B.

Figure 2.1 shows the structure of this model. After the initial utterance (the request), B can accept (and thereby commit to satisfy the conditions), decline (and thereby end the conversation), or counter-offer with alternative conditions. Each of these in turn has its possible continuations (e.g. after a counter-offer, A can accept, cancel the request, or counter-offer back) [35].

While conversations for action form the central fabric of cooperative work, there are additional categories of conversations to be distinguished: conversation for clarification, conversation for possibilities, and conversation for orientation. A summary of those categories is listed below [27]:

- **Conversation for action** usually begins with a request or an offer, the intention of the conversation is that some actions need to be taken.
- In **conversation for clarification**, the intention of the participates is to obtain more information about something already said or a previous conversation.
- In **conversation for possibilities**, the intention of the participates is creating ideas or to settle on several existed ideas.
- In **conversation for orientation**, the intention of the participates is to exchange information.

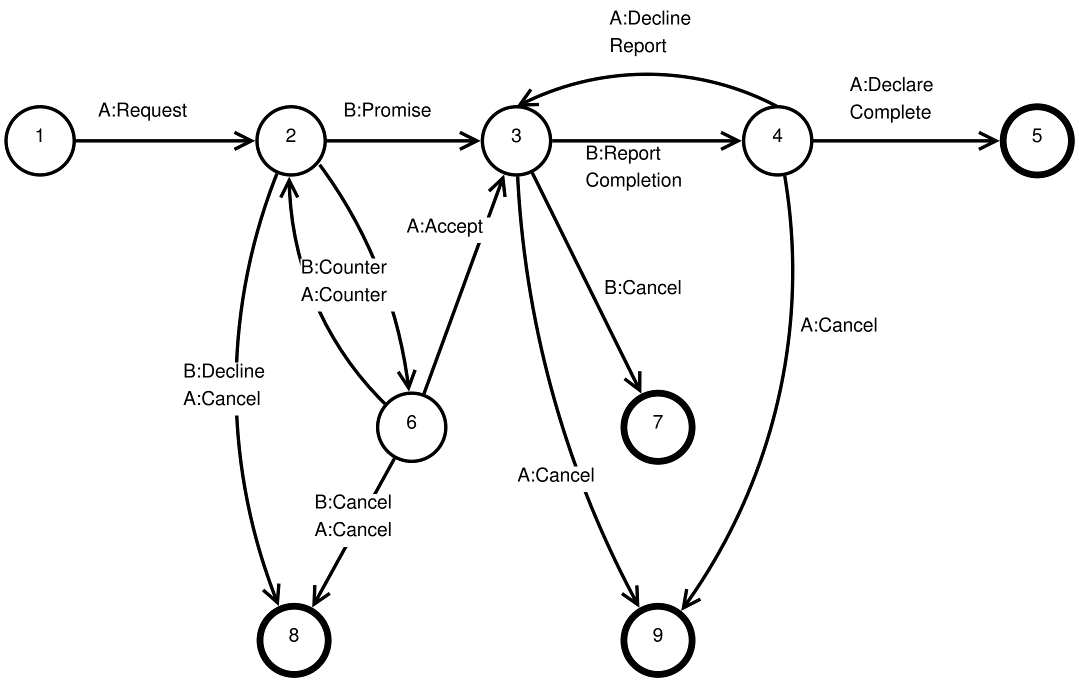


Figure 2.1: State transition diagram representing a conversation for action [35]

The typical activities in a mailing list could also be viewed as conversations. For example, asking and answering questions, discussing on a certain decision, commenting about certain topic, and so on. So Winograd's classification scheme also applies.

2.2 Conversations in Mailing Lists

Messages in a mailing list also have their intentions, or pragmatics. An email that reports a bug, has the intention of raising developer's attention so that the bug could be fixed. An email that starts a topic about a new idea of a project, has the intention of settling down ideas. These two cases are clearly to be classified into "conversation for action" and "conversation for possibilities", respectively. Yet, due to the broadcasting nature of mailing lists, every subscriber will get all the messages sent by others. In a busy mailing list, the amount of messages in one's inbox could be huge.

Another problem is that, as mentioned in chapter 1, a general email have no specific field that represents the (pragmatic-)semantic of itself. Custom headers could do the job but currently there is no standard for such headers. A good classification schema for emails might be useful for machines that can automatically parse an email to determine the suitable category for it, but for human developers, scanning lots of emails and quickly getting know each one's intention behind the plain subject text, is a hard task.

A practical solution is to include the semantic meaning of each email right in its content, such as subject or body text. A simple way of doing this is using annotations, e.g. adding one or more keywords in an email's subject line to indicate its main type: Is this email a bug report, or a idea proposal, or a support ticket? In this case, one could simply add keywords like "bug", "proposal" and "help" in each email's subject line. This makes the emails visually stand out, also reduces the effort to identify whether he or she is interested in this email — one does not need take time reading up the whole subject text to understand its possible intention. This "annotation" solution can be viewed as tagging and the keywords here are actually different tags.

2.3 Tags and Folksonomy

In online computer systems terminology, a tag is a non-hierarchical keyword or term assigned to a piece of information. It is a kind of metadata that helps describing an item, allowing it to be found by browsing or searching. Tagging was popularized by the "Web 2.0" trend and is an important feature of many Web 2.0 services, e.g. "Delicious" and "Flickr". Tags may be a "bottom-up" type of classification, compared to hierarchies, which are "top-down". In a tagging system, there are unlimited number of ways to classify an item. Instead of belonging to one category, an item may have several different tags [34]. Typically, users can freely choose tags, thus they create a folksonomy.

A folksonomy is a system of classification derived from the practice and method of collaboratively creating and managing tags to annotate and categorize content. This term itself is a portmanteau of *folks* and *taxnonomy* [29].

Using tags to annotate the messages in a mailing list is a simple and effective way to add semantic/pragmatic meanings to them, this in turn could help improving the efficient of collaboration. In fact, there is already such kind of practices being used by open source developers in various mailing lists. For example, in the Linux Kernel Mailing List (LKML), it is common to find emails containing a keyword in their subject line. The keyword is usually put in square brackets, like: "[PATCH] kernel/cpu.c: Fix many errors related to style.", "Re: [RFC] add pwmlib support" and "[ANNOUNCE] undertaker 1.0".

The challenge now is to find out a proper tag schema, which can be used to identify the most common and important activities and/or practices in mailing list collaboration. Also, in respect to folksonomy, these tags should already have their presence in developers' mind; Lastly, if the tags could fit into the four categories of conversation action too, it would lead to even better information retrieval result, because of the combination of tagging and hierarchy classification. In the next chapter, selected mailing lists will be analyzed to find out the existed tagging practices.

3 Analysis of Mailing Lists

In this chapter, four selected mailing lists of popular open source projects will be analyzed. The purpose is try to find out whether developers are actually using tags in email's subject line, and which tags are used most frequently.

3.1 Data Source and Analysis Method

The projects, whose mailing lists were used for analyzing, were selected according to their popularities ranked by the open source public directory Ohloh.net [22].

- **Linux Kernel**: Linux is a free software kernel, which combined with the GNU libraries, core utils and shell form the GNU/Linux operating system.
- **Apache HTTP Server**: The Apache HTTP Server Project is a collaborative software development effort aimed at creating a robust, commercial-grade, feature-rich, and freely-available source code implementation of an HTTP (Web) server. The project is jointly managed by a group of volunteers located around the world, using the Internet and the Web to communicate, plan, and develop the server and its related documentation. This project is part of the Apache Software Foundation.
- **X.Org**: X.Org provides an open source implementation of the network-transparent X Window System, as well as working on the standard itself. The development work is being done as part of the freedesktop.org community, sponsored by the X.Org Foundation.
- **Ubuntu**: Ubuntu is by far the most popular Linux distribution.

3.1.1 Retrieving Messages

X.Org and Ubuntu use "GNU Mailman" as their mailing list systems. All the messages are publicly available in the archives and can be downloaded in "mbox" format. Apache HTTP Server uses another mailing list system but the message archives are also downloadable in "mbox" format. The mbox format as defined in RFC 4155 [11] is actually a plain text file, in which email messages are concatenated in their original Internet Message (RFC 2822) format.

The Linux Kernel Mailing Lists, on the other hand, did not provide downloadable archives (There are 3rd party mbox packages, but are outdated). The message archives are, however, fully browsable on web (`https://lkml.org`). Messages are organized in days, each day has a corresponding URL. For example, the messages from April 1st, 2010 are listed on a page with the URL address "`https://lkml.org/lkml/2010/4/1`", so the pattern here is "`https://lkml.org/lkml/[yyyy]/[m]/[d]`". In addition, the message list page is a valid XML document, this makes it possible to use a "page crawler" to scan and fetch the messages, then use an XML Parser to extract the information needed, i.e. the subject line of each message.

A small python program was written, when executed, the code will create a list containing the URLs of the message list pages, Next step is visiting these URLs and fetching the page contents.

The Python library *urllib2* is used to fetch data from a remote location, the retrieved data is sent to another function called `process_page_content()`, which stores the data into files for further use. As mentioned above, the LKML's message list page is a valid XML document, so the information contained in the page can be extracted by parsing the XML DOM Structure, this is completed with the following function:

```
def parse_html_file(file_path):
  title_list = []
  f = open(file_path)

  xmldom = minidom.parseString(f.read())
  t_list = xmldom.getElementsByTagName('tr')
  for tl in t_list:
    if tl.attributes.has_key('class'):
      cl = tl.attributes['class'].value
      if cl == 'c0' or cl == 'c1':
        try:
          a_text = tl.childNodes[1].firstChild.firstChild.data
          if len(a_text.strip()) > 0:
            title_list.append(a_text + '\n')
        except AttributeError:
          print 'Error encountered while processing XML data!'
          continue

  f.close()
  return title_list
```

For the analysis, only the subject line of each message is needed. In the XML document, these lines are placed in the `tr` elements with the class "c0" and "c1", and they were displayed as HTML hyperlinks. The above function first builds an XML DOM Tree from the XML text, then selects all the `tr` elements, iterates them to locate those with the correct classes, and extracts the text from the hyperlinks, then pushes those text, which are the message subject lines, into a list.

Extracting the subject lines of messages from "mbox" files is easier. As said above, "mbox" are in fact plain text format. In an "mbox" file, the beginning of each email message is indicated by a line starts with five characters consist of "From" followed by a space and the return path email address [11]. The subject line of each message is the value of the "subject" header field: Header fields are lines composed of a field name, followed by a colon (":"), followed by a field body, and terminated by CRLF [10]. The basic algorithm to find the subject lines is to iterate the "mbox" file line by line, locate the lines star with "Subject: " — these are the "subject" header fields — and extract the text after the headers.

3.1.2 Analysis for Tag Frequency

Before getting started with the analysis, it should be clarified that, the "tags" being analyzed are defined as the words containing in *square brackets* in the subject lines of the messages. For example, given an email subject "[PATCH] kernel/cpu.c: Fix many errors related to style", the tag in this case is "PATCH". Of course, tags could also be written in any forms, such as in curly brackets or without any brackets just in upper case letters; And there could be simply no tags at all. So, besides searching for square brackets, it also makes sense to analyze the word frequency of the whole subject line as well.

The tool for this analysis is a Python program "histogram.py" [19]. This program can count the occurrence frequency of each words (except digits, punctuations and other words that need to be filtered). The main part of the program is as following:

```
from string import split, maketrans, translate, punctuation, digits
import sys
from types import *
import types

def word_histogram(source):
  """Create histogram of normalized words (no punct or digits)"""
  hist = {}
  trans = maketrans('', '')
  # words to be filtered
  ign_words = ['a', 'r', 'v', 'of', 'to', 're']
  for word in split(source):
    word = translate(word, trans, punctuation + digits).lower()
    if len(word) > 0:
      if not (word in ign_words):
        hist[word] = hist.get(word, 0) + 1
  return hist

def most_common(hist, num=1):
  pairs = []
  for pair in hist.items():
    pairs.append((pair[1], pair[0]))
  pairs.sort()
  pairs.reverse()
  return pairs[:num]
```

```
if __name__ == '__main__':
  if len(sys.argv) > 1:
    hist = word_histogram(open(sys.argv[1]))
  else:
    hist = word_histogram(sys.stdin)

  print "The most common words:"
  for pair in most_common(hist, 25):
    print str(pair[1]) + ',' + str(pair[0])
```

Basically, this program uses the Python dictionary to store each word and its occurrence, sorts the items by occurrence and outputs the result. The analysis is interested only in meaningful words, so some grammar components such as articles (“a”, “the”...) and prepositions (“of”, “to”, “in”...). These words can be added into the list `ign_words`. This program takes either a string or a file as input. As the email subject lines are retrieved and stored by the code in the previous section, the analysis for word frequency can be started already. What still missing is the tags, which must be extracted from the subjects. The simple way is using Regular Expression to find out square brackets as well as the words in them. Example code are:

```
def analyse_title_tags(input_file):
  all_tags = []
  f = open(input_file)
  bracket_pat = re.compile(r'\[(.*?)\]')
  for line in f.readlines():
    if len(line) == 0:
      continue
    found = bracket_pat.search(line)
    if found:
      grps = found.groups()
      for t in grps:
        all_tags.append(t)

  return all_tags
```

These code will iterate the file that contains all the message subjects line by line, in each line it will search if it contains square brackets, if so, extract the word inside each of the brackets and store them into a list. So, if a subject line is like “[ANNOUNCE] xf86-video-ati 6.12.5”, the word “ANNOUNCE” will be extracted; If there are more than one square bracket containing keywords, e.g. “[ANN][RFC] Plug-in XYZ”, then both “ANN” and “RFC” will be extracted.

3.2 Results and Interpretations

In this section, the results of the word/tag frequency are presented and examined. For each project, the result contains the top 20 words and top 15 tags with the most occurrence from the mailing lists.

3.2.1 Analysis Results

LKML

Table 3.1: Mailing List Analysis Result: LKML

Project:	Linux Kernel
Mailing List(s):	LKML
Data Source:	Archive emails from January 2007 to June 2010
Total Message Count:	503,229
Messages with Tags:	370,959 (73.7%)
Top 20 Words:	See Figure 3.3
Top 15 Tags:	See Figure 3.2

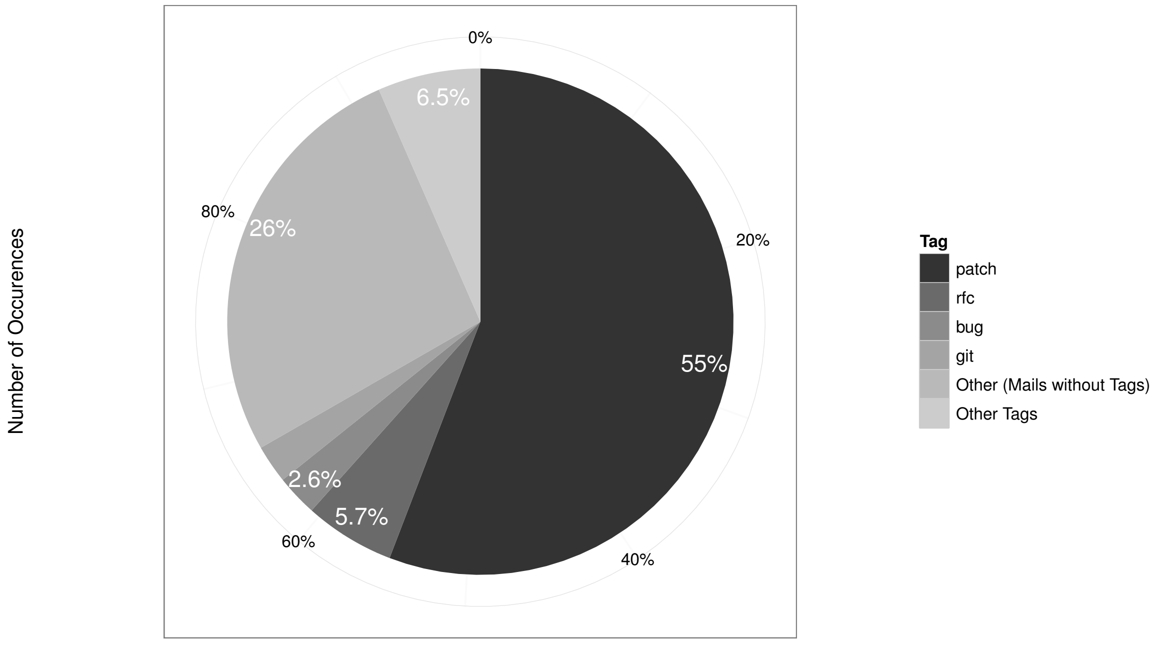


Figure 3.1: Percentage of tags in LKML

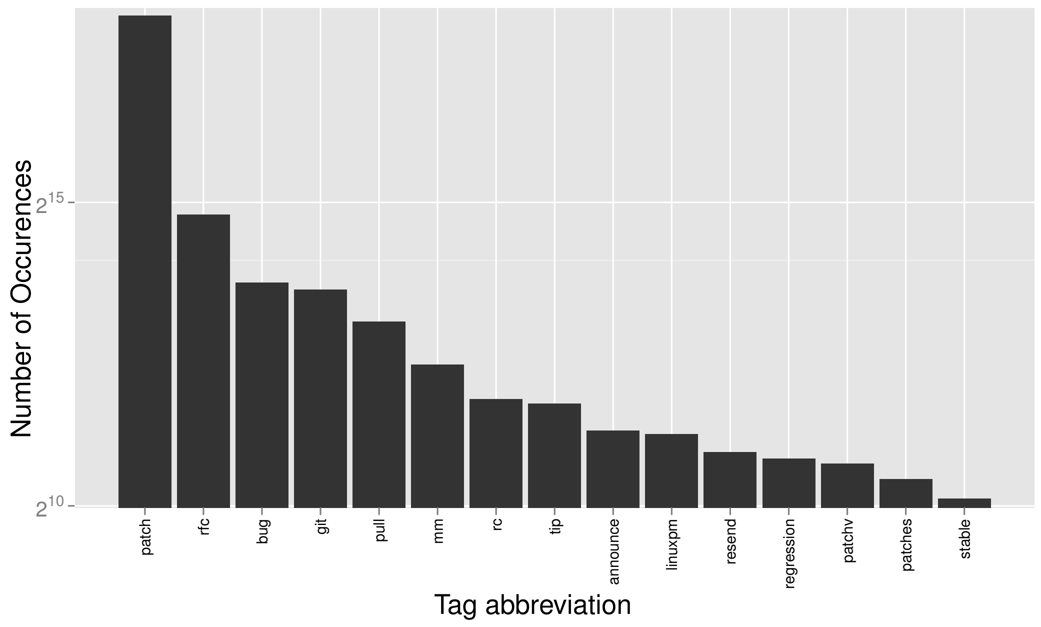


Figure 3.2: The top 15 most frequently used tags in LKML

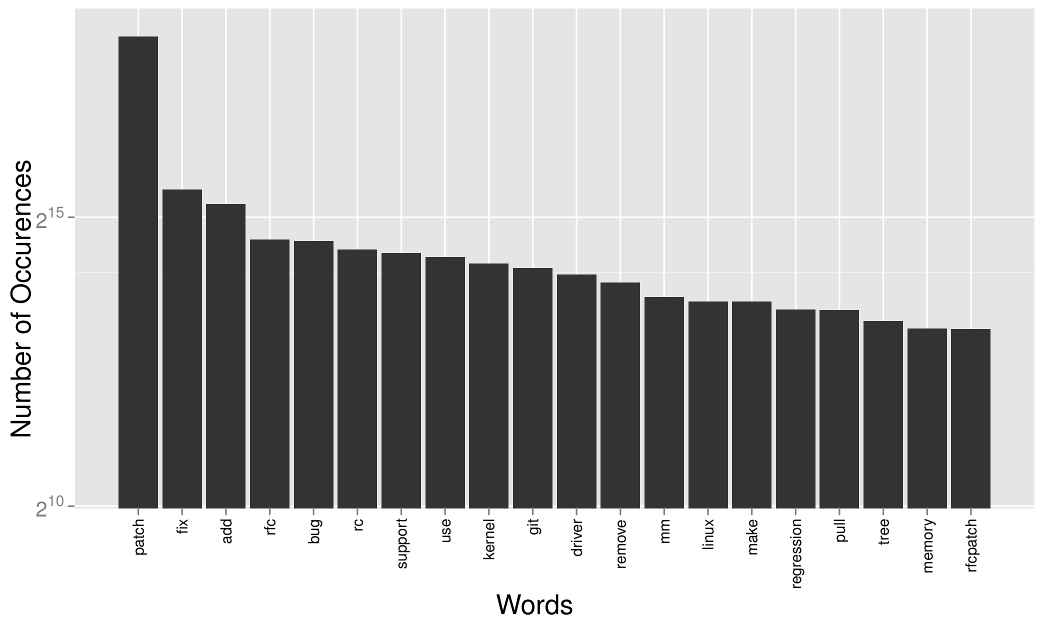


Figure 3.3: The top 20 most frequently used words in LKML

Apache HTTP Server

Table 3.2: Mailing List Analysis Result: Apache HTTPD

Project:	Apache HTTP Server
Mailing List(s):	Development Main Discussion List (dev@httpd.apache.org)
Data Source:	Archive emails from January 2007 to June 2010
Total Message Count:	13,293
Messages with Tags:	2,714 (20.4%)
Top 20 Words:	See Figure 3.6
Top 10 Tags:	See Figure 3.5

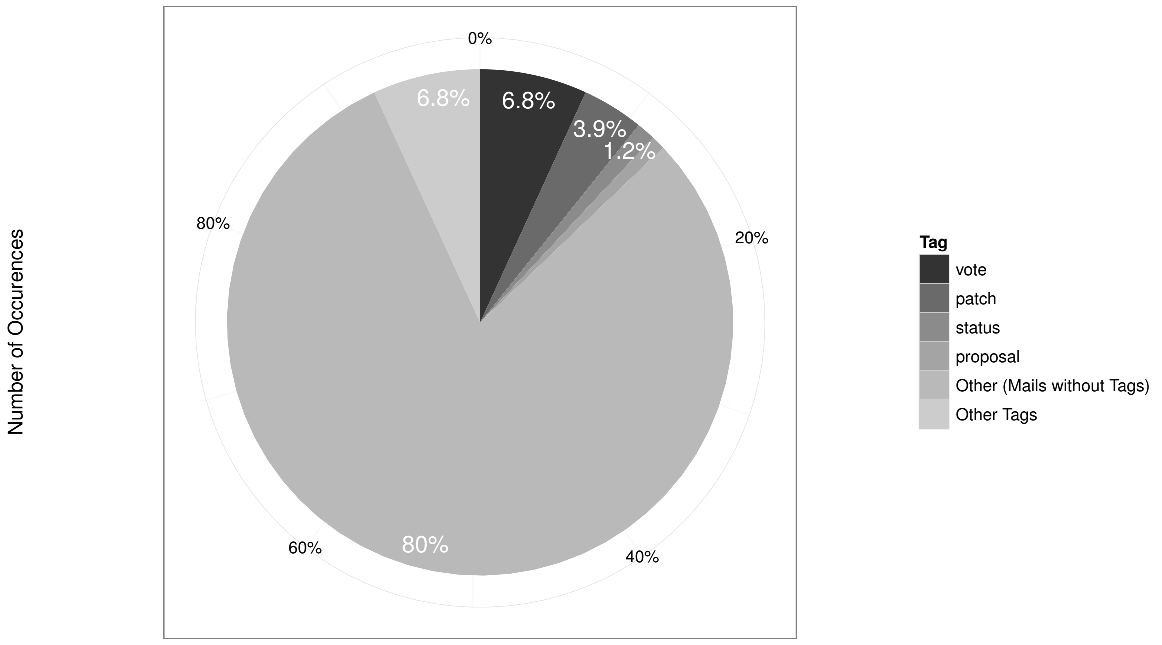


Figure 3.4: Percentage of tags in Apache HTTP Server development mailing list

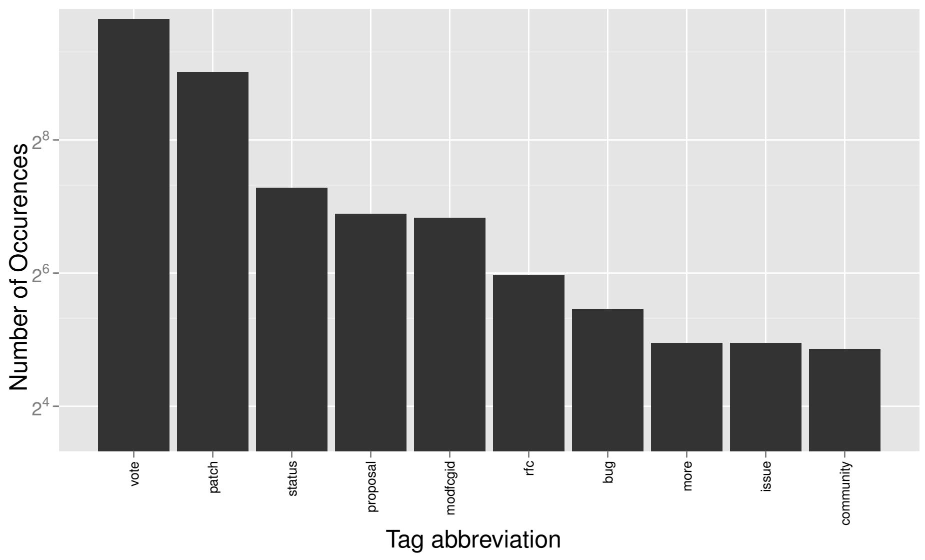


Figure 3.5: The top 10 most frequently used tags in Apache HTTP Server development mailing list

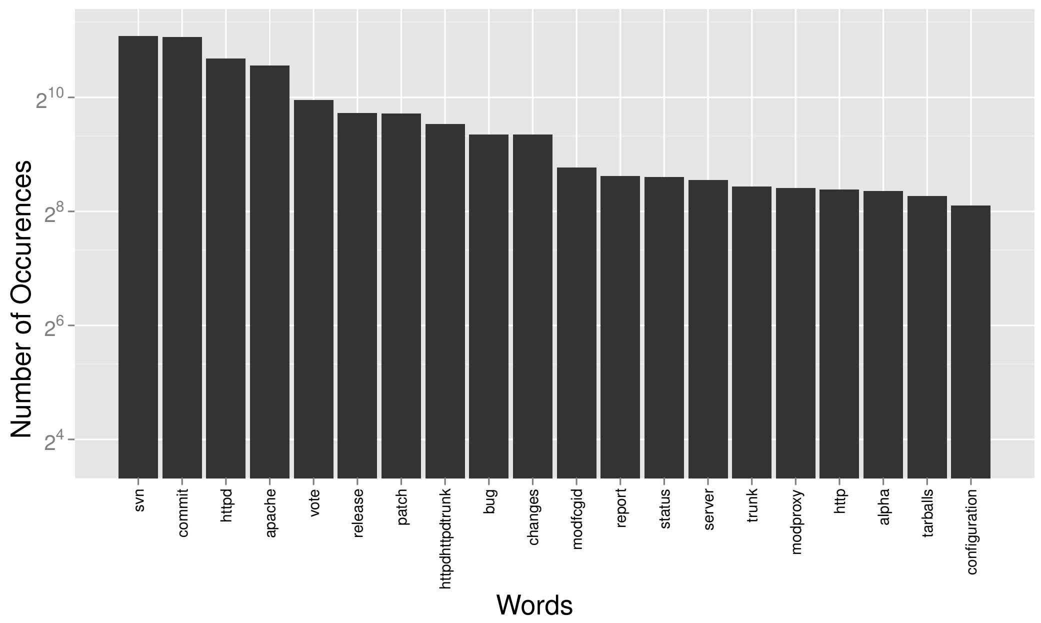


Figure 3.6: The top 20 most frequently used words in Apache HTTP Server development mailing list

X.Org

Table 3.3: Mailing List Analysis Result: X.Org

Project:	X.Org
Mailing List(s):	X.Org user support and discussion (xorg@lists.freedesktop.org)
Data Source:	Archive emails from January 2008 to June 2010
Total Message Count:	20,213
Messages with Tags:	4,856 (24%)
Top 20 Words:	See Figure 3.9
Top 15 Tags:	See Figure 3.8

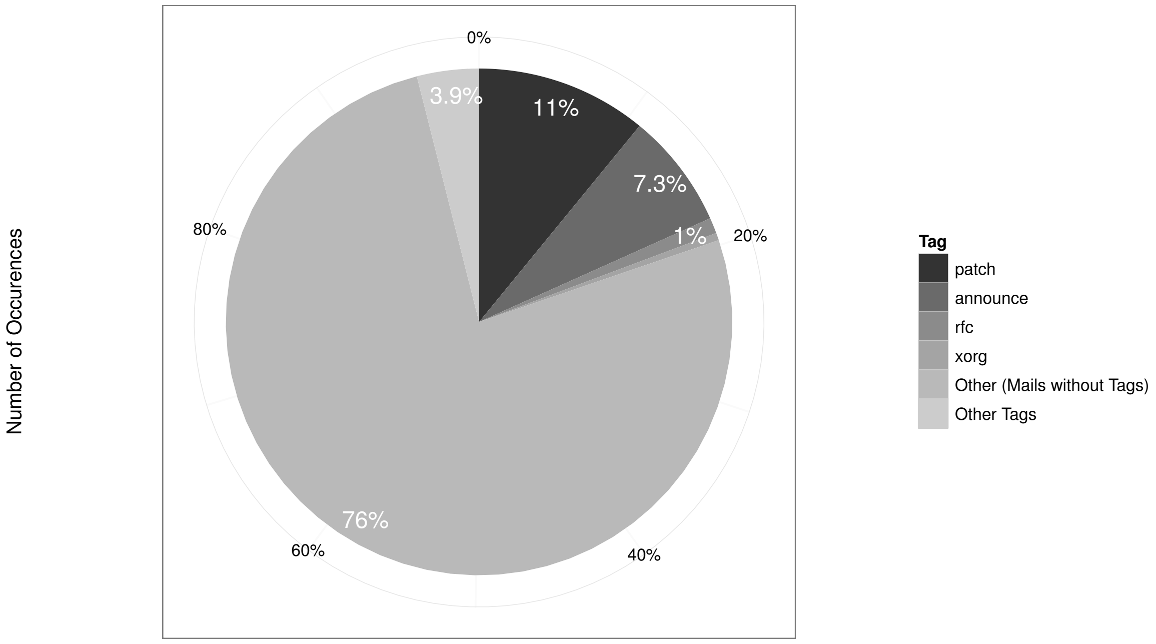


Figure 3.7: Percentage of tags in X.Org user support mailing list

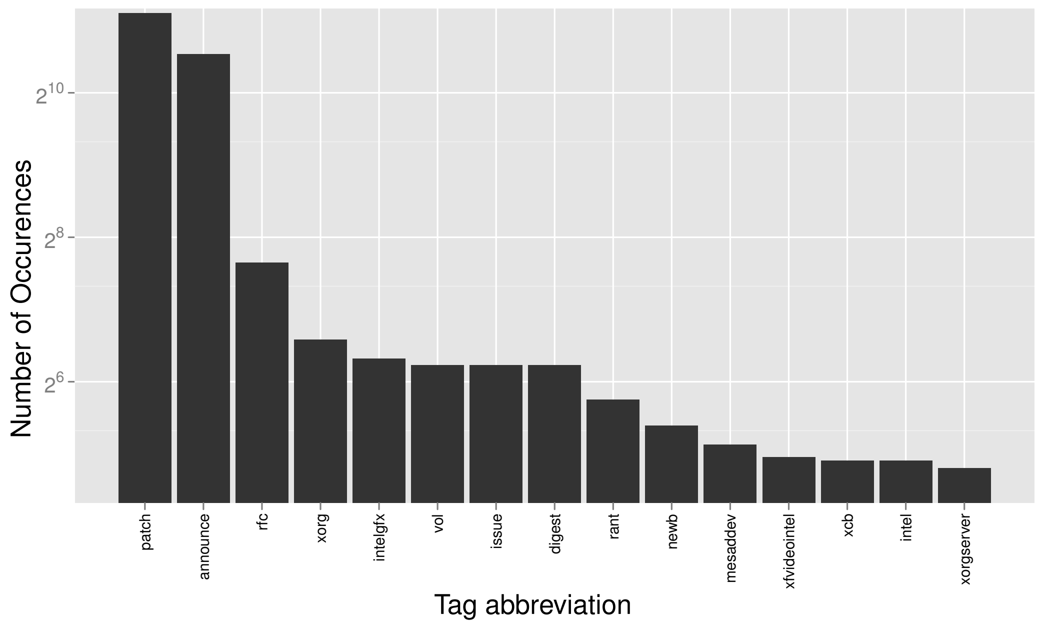


Figure 3.8: The top 15 most frequently used tags in X.Org user support mailing list

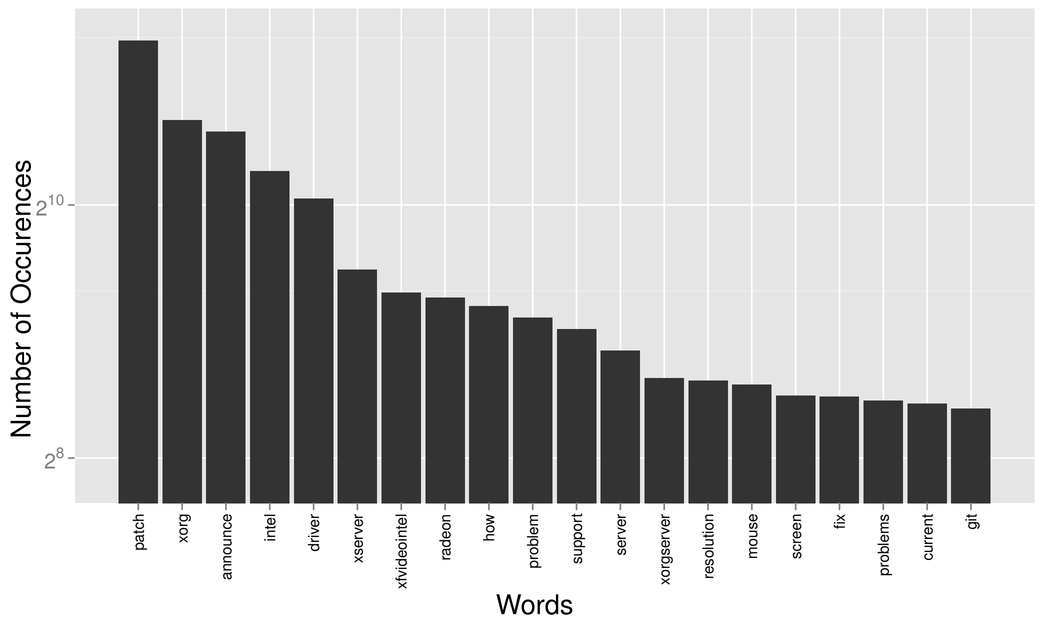


Figure 3.9: The top 20 most frequently used words in X.Org user support mailing list

Ubuntu

Table 3.4: Mailing List Analysis Result: Ubuntu

Project:	Ubuntu
Mailing List(s):	Ubuntu Development (ubuntu-devel@lists.ubuntu.com), Bazaar Discussion (bazaar@lists.canonical.com) and Kernel Team Discussions (kernel-team@lists.ubuntu.com)
Data Source:	Archive emails from the first message to June 2010
Total Message Count:	113,271
Messages with Tags:	42,405 (37.4%)
Top 20 Words:	See Figure 3.12
Top 15 Tags:	See Figure 3.11

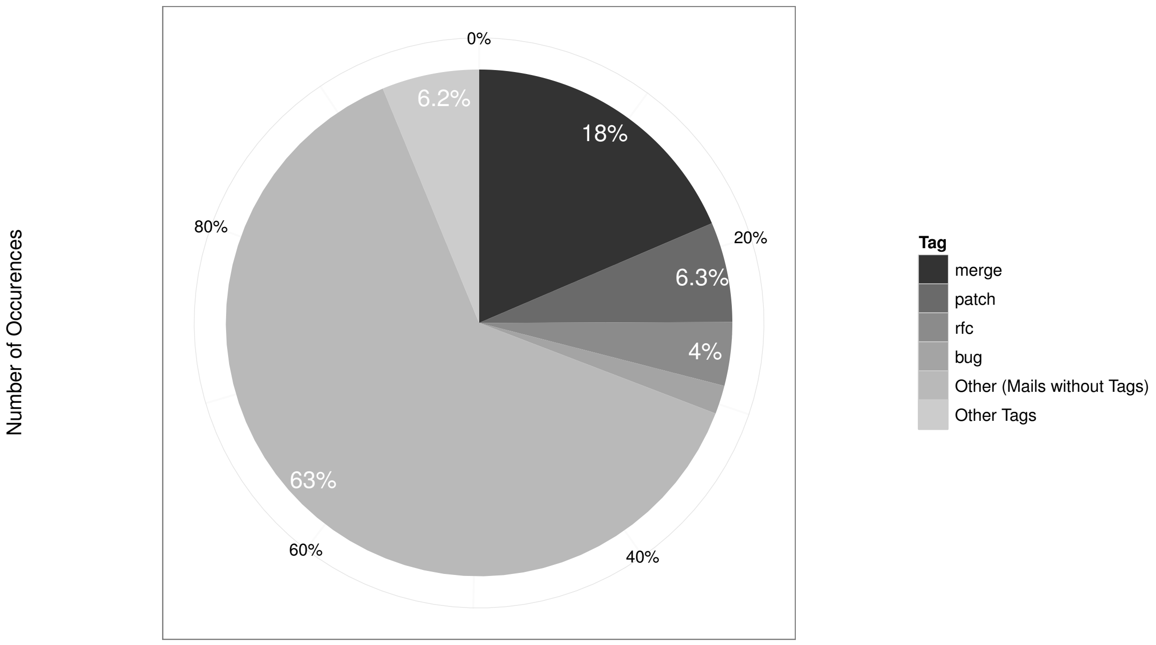


Figure 3.10: Percentage of tags in selected Ubuntu mailing list

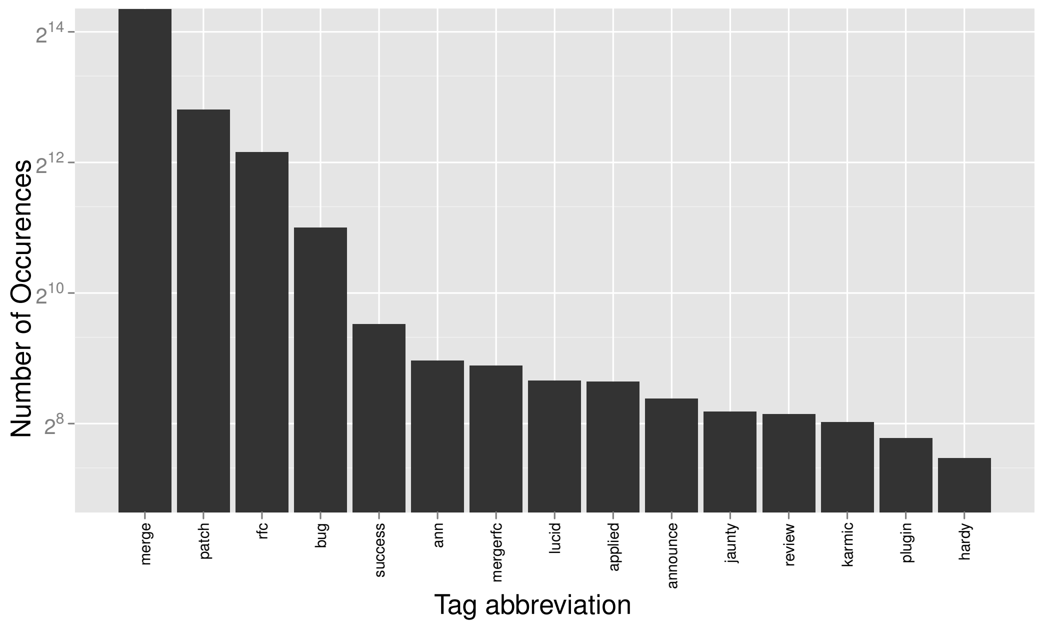


Figure 3.11: The top 15 most frequently used tags in selected Ubuntu mailing list

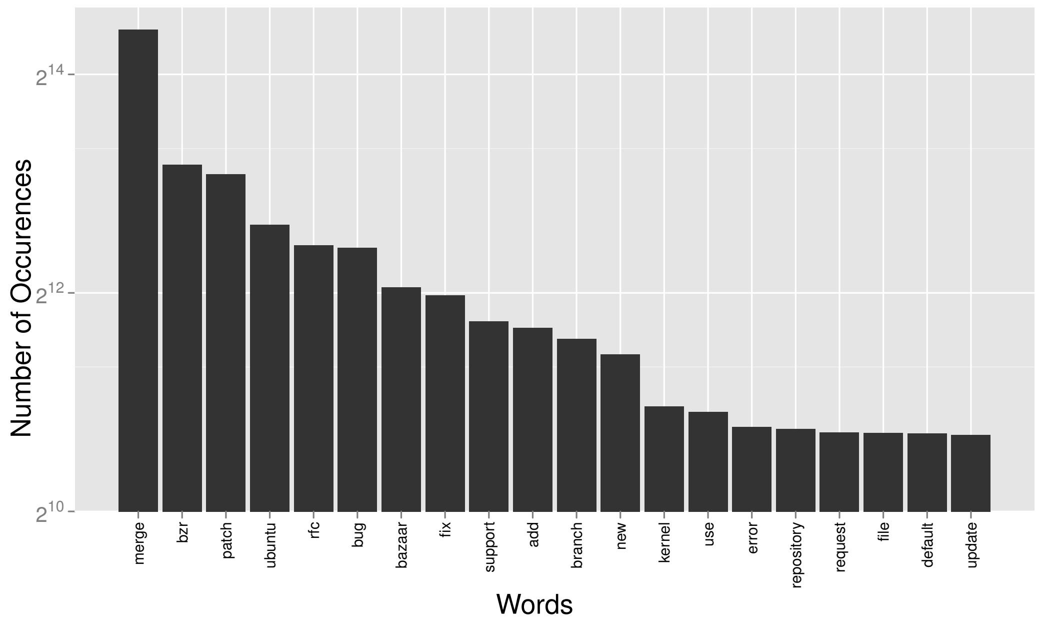


Figure 3.12: The top 20 most frequently used words in selected Ubuntu mailing lists

3.2.2 Representativity of Selected Mailing Lists

A survey has been conducted with support from the Open Source Research Group of the University of Erlangen-Nuremberg. The purpose of this survey is to validate the contributions in this thesis, based on the feedbacks from open source community.

There are several question groups in the survey. For this chapter, the most important questions are whether the selected mailing lists are representative. These questions include:

- "Do you think that the mailing lists of the following projects are good examples for best practices of collaboration in mailing lists?".

  The answer options contain the four projects, whose mailing lists were analyzed in the previous sections. Survey participants are required to give each project a score by choosing from "1" to "5", whereas score 1 means "No, there is not much of best practices shown in this mailing list", and score 5 means "Yes, one can find lots of best practices in it", If a user gives no answer, that means "I don't know enough about this mailing list".

- "Do you think that the mailing lists of the following projects are representative for all open source project's mailing lists?"

  Answer options are the same as the first question, while score 1 means "not representative" and score 5 means "definitely, it is the blueprint for open source projects".

The survey result was processed in R, and the results of the two questions above are shown in Table 3.5 and Table 3.6:

	List	Mean	Standard Error	Lower Bound	Upper Bound
1	LKML	4.50	0.22	4.08	4.92
2	X.Org	4.40	0.23	3.94	4.86
3	Ubuntu	3.60	0.38	2.86	4.34
4	Apache	4.80	0.19	4.43	5.00

Table 3.5: Best Practices shown in Mailing Lists

	List	Mean	Standard Error	Lower Bound	Upper Bound
1	LKML	4.50	0.22	4.08	4.92
2	X.Org	3.80	0.19	3.43	4.17
3	Ubuntu	4.20	0.19	3.83	4.57
4	Apache	4.40	0.38	3.66	5.00

Table 3.6: Representativity of Mailing Lists

Results show that these four selected open source projects are representative, their mailing lists contain good examples for best practices of collaboration as well. Since Linux Kernel, Ubuntu and Apache are all among the top 10 most popular open source projects ranked by Ohloh.net, this result is not surprising.

3.2.3 Comments to the Results

The analysis results of the selected mailing lists from the four open source projects show:

- Tags are widely used. For the analyzed mailing lists, more than 20% of their messages have used tags in the subjects. In LKML there are even as many as 73% of the messages using tags. This also indicates that the "annotation form" — keywords within square brackets — is widely accepted.
- Some tags (including their synonyms) not only have large number of usage, but also are adopted across different projects' mailing lists. Most notably: "Bug", "Patch", "Announce" and "RFC".
- The frequently used words of each project reflect the project's specific properties, such as features or components. For example, in LKML, the frequently used words "kernel", "git", "driver", "mm" and "linux" are all highly relevant to the development of Linux kernel. This may imply that the messages in the mailing lists share a central context – the project itself.
- Some frequently used tags are in accordance with the typical development process and artifacts of open source projects. Notably "Announce", "Proposal", "Bug" and "Patch". The tag "Patch" has a dominate majority in almost all these lists, this may be interpreted as when developers publish patch information in mailing lists, they tend to use tag to emphasize them.
- The top tag used in LKML — "Patch" — may indicate that the development of Linux kernel is rather code-driven. Developers post patches directly into mailing list and the discussions are also focused on patch information. While in the Apache mailing lists, the top tag is "Vote", this may be interpreted that a democratic decision making is so important in the Apache community that such messages are clearly flagged using tags. So the use of tags could also reflect the development culture of different projects.

So far, the actual usage practices of tags in mailing lists have been examined. The following chapters will try to bring up a proposal for a schema of tags as well as tools that provide further support.

4 Proposal for Collaboration Patterns Codification

Based on the “conversation for action” theory from chapter 2, as well as the analysis of actual tag usage practices in mailing lists of selected open source projects from chapter 3, this chapter will try to propose a schema of tags that codify open source collaboration patterns.

The identification criteria for the tags schema are:

- The tags should already be used by developers in mailing lists.
- The tags could be classified into the categories of “conversation for action”.
- The tags should conform with the process of open source software development.

4.1 Open Source Software Development Process

Open source software development can be divided into several phases [31]. Figure 4.1 shows the process-data structure of open source software development, including phases and the corresponding data elements. The process starts with a choice between the adopting of an existing project, or the starting of a new project. If a new project is started, the process goes to the “Initiation” phase. If an existing project is adopted, the process goes directly to the “Execution” phase.

4.2 Proposed Tags Schema

Currently, this thesis proposes a total of 10 tags: “Bug”, “Patch”, “Issue”, “RFC”, “Tip”, “Proposal”, “Vote”, “Announce”, “Solved” and the so called “Project Name”, which is actually the name or codename of a project.

4.2.1 Categorizations

7 of these 10 tags can be found in the most frequently used tags from the analysis results in the previous chapter. So they should already be familiar to developers. The tags “Tip” and “Solved” was inspired from many Internet forums: Posts with “Solved” in the title

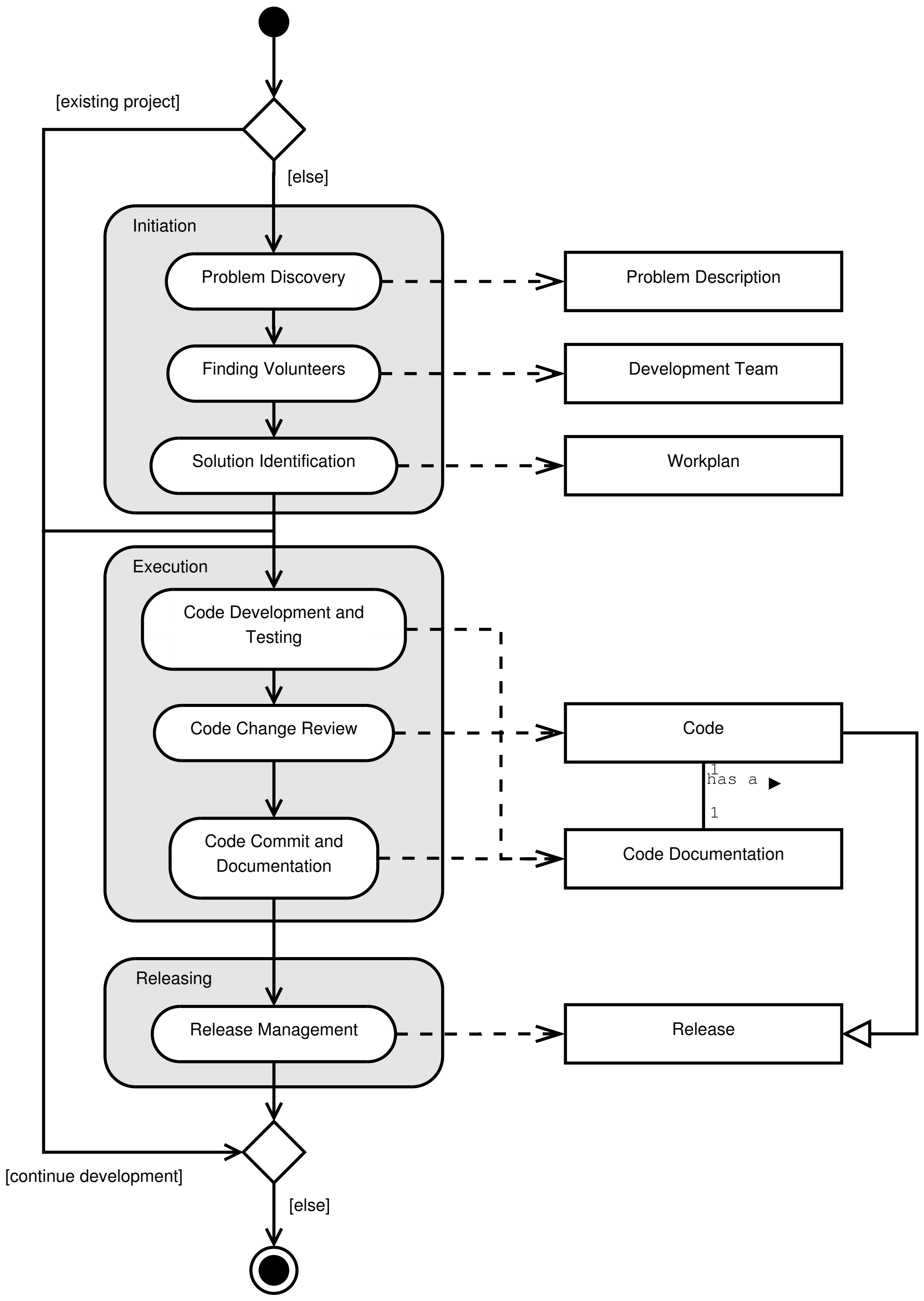


Figure 4.1: Process-Data Model for open source software development [31]

contain solutions, so that users who just seek for answers could save time by avoiding open questions. Posts with “Tip” in the title often provide reusable information, worth being collected.

A quick recall of Winograd’s “conversation for action” categories [35], along with the intention of each type of conversations:

- Conversation for **action**: some actions to be taken.
- Conversation for **clarification**: obtaining more information.
- Conversation for **possibilities**: creating ideas or settling on several ideas.
- Conversation for **orientation**: to exchange information.

An email with the tag “Bug” has the purpose of reporting bugs, its intention is therefore hoping the bug will be reviewed by developers and eventually be fixed. On the other hand, an email with the tag “Patch” usually provides fixes for a certain bug, this is also its intention. Judged by their intentions, the categorization of these 10 tags using Winograd’s schema is shown in Table 4.1.

Table 4.1: Categorization of Tags (Conversation for Action)

Category	Tags	Comment
Tags for “action”	Bug, Patch, Issue, RFC	Messages using these tags have the intention of taking some actions, e.g. reporting bugs and problems, providing fixes, requesting for comments.
Tags for “clarification”	Tip, Project Name	Messages with the tag “Tip” usually tend to provide extra (useful) information; Using project name as tags could help users distinguish messages that refer to specific projects.
Tags for “possibilities”	Proposal, Vote	Messages with the tag “Proposal” have the intention of raising new ideas or suggestions, while casting votes has the intention of making a decision, i.e. settling down ideas.
Tags for “orientation”	Announce, Solved	Messages with these tags declare certain events that cause users’ awareness.

4.2.2 Usage Pattern of Each Tag

In this section, the usage patterns of the 10 tags will be explained in details. This includes the usage context/scenario, tag's appearance variants and usage suggestions.

Bug

This tag is to be used when an email references a bug report or discusses a bug. One purpose of this tag is to allow bug tackers to automatically dispatch bug reports to mailing lists. Another purpose is that people can talk about a bug they might have found (Table 4.2).

Table 4.2: Usage Pattern of Tag "Bug"

Tag Name:	Bug
Variants:	[Bug], [Bug XXX] (XXX may be a number or string that refers to a bug-id)
Context:	Mailing lists dedicated to the development; Most subscribers are developers; The project is in the execution phase
Problem:	How to quickly identify emails that are related to certain bugs?
Solution:	Write "[Bug]" as prefix in the email's subject line, indicate that this message is about reporting a bug. All the discussions that are related to a bug should have this tag attached. Alternatively, there can also be a bug id specified.
Comment:	Bug tracker systems could be configured to automatically send emails with this tag — usually also with bug id — to mailing lists, when a bug was created. Although discussions about a bug can take place directly in the bug tracker, that is not a preferable way of communication [6].
Example:	Re: [BUG] khugepaged crashes on x86_32 [Bug 26922]USB: yurex: recognize GeneralKeys wireless presenter as generic HID

Patch

This tag is used on emails that reference a patch (Table 4.3).

Issue

This tag is to be used on emails that report general issues, such as software runtime errors, documentation errors etc. [Bug] is not used here because sometimes when a problem appears, one can not confirm if it is really a bug or just another broken feature (Table 4.4).

Table 4.3: Usage Pattern of Tag “Patch”

Tag Name:	Patch
Variants:	[Patch], [Patch XXX] (XXX could be a serial number or part number to identify this patch, or it can be an id number that is identical to the bug id, in this case, this patch is the response to that bug.)
Context:	Development discussions; Projects in the execution phase.
Problem:	How to quickly identify emails about patch information (and their related bugs)?
Solution:	Use “[Patch]” as prefix in the email’s subject line, indicate that this email provides patch/fix to a specific bug or issue. In order to locate the specific bug to which this patch applies, the bug id can be added in the tag. Alternatively, if a patch is divided into several emails, there can be a part number in the tag, e.g. [Patch 1/4], [Patch 7/7].
Comment:	The two tags “Bug” and “Patch” could act as adjacent pair [7]. Bugs and their corresponding patches (if available) should be “connected” by the bug id or other identity.
Example:	Re: [PATCH] perf: Cure task_oncpu_function_call() races

Table 4.4: Usage Pattern of Tag “Issue”

Tag Name:	Issue
Variants:	[Issue], [Error], [Problem]
Context:	Mailing lists of general discussions or user supports; Projects in the execution or releasing phase.
Problem:	For developers and supports, how can they quickly be notified for an issue or problem report (in contrast to general rants)?
Solution:	Add “[Issue]” tag as prefix in the email’s subject line.
Comment:	With this tag, developers and/or supports can pay less attention in reading and judging the whole email subject — not to mention in many cases subjects are not written descriptive enough. Usually for this type of emails, the sender are waiting for some answers (actions).
Example:	RE: [Issue] External links @ the wiki, aka pagechange wars

RFC

"RFC" is the acronym of "Request For Comment". This tag is used to ask other developers to give comments and feedbacks on certain features and/or functions (Table 4.5).

Table 4.5: Usage Pattern of Tag "RFC"

Tag Name:	RFC
Variants:	[RFC]
Context:	Development discussions
Problem:	How can developers quickly distinguish "request for comments" discussions from bug/patch and other topics.
Solution:	Use "[RFC]" as prefix in the email's subject line.
Comment:	The difference between "RFC" and "Proposal" is that emails tagged with "RFC" focus more on the development phase, e.g. to start discussion about a potential feature/function.
Example:	Re: [RFC] i2c-algo-bit: Disable interrupts while SCL is high

Tip

This tag marks the emails that provide useful information, but are not intended to start a discussion (Table 4.6).

Table 4.6: Usage Pattern of Tag "Tip"

Tag Name:	Tip
Variants:	[Tip], [Tips], [Hint]
Context:	Mailing lits of general discussions.
Problem:	How to share useful pieces of information better, so that users could identify and collect them more easily?
Solution:	Add "[Tip]" as prefix to the email's subject line.
Example:	[tip] some regedit tweaks to improve 3D performance in WINE

Name of Project

This tag is a bit special. Instead of a specific word, the name or codename of the project will appear in the square brackets, e.g. [gwt], [kubuntu], [mailman-dev] etc. Sometimes there are several projects or sub projects, which are discussed in the same mailing list, so each project's name or code-name is used as a tag, to group/distinguish them visually (Table 4.7).

Table 4.7: Usage Pattern of (Special) Tag “Project Name”

Tag Name:	Name of Project
Variants:	According to specific project
Context:	General discussions in a mailing list where several projects are involved; Multiple projects/sub projects share a same mailing list.
Problem:	If several projects are being discussed in one same mailing list, how can users easily distinguish topics of each project?
Solution:	Add project’s (code)name as prefix in the email’s subject line.
Comment:	The tag text should be short and unique, so using project’s codename is a good practice. Mailing list system such as Mailman has the configuration of adding a default prefix to the subject every message, this is the place that the tag fits best.

Proposal

This tag is used on emails that propose an idea, mostly at the initial phase of a project (Table 4.8).

Table 4.8: Usage Pattern of Tag “Proposal”

Tag Name:	Proposal
Variants:	[Proposal], [Idea], [Suggest]
Problem:	How to distinguish proposals from the mass? Because proposals generally bring up different attention.
Solution:	Use “[Proposal]” as prefix in the email’s subject line.
Comment:	Messages with this tag should focus on something creative, such as idea initiating or brainstorming.
Example:	[PROPOSAL] add a sslport option

Vote

This tag indicates that this email starts a vote to make some decisions. One can reply to the topic and include [+/-1] in the title to cast a quick vote (Table 4.9).

Announcement

This tag is to be used on ”press release”-like emails. To announce news, important changes, etc 4.10.

Table 4.9: Usage Pattern of Tag “Vote”

Tag Name:	Vote
Variants:	[Vote], [+1], [0], [-1]
Context:	General discussions; A decision needs to be made through voting.
Problem:	How to easily cast voting using emails?
Solution:	Add “[Vote]” as prefix in the email’s subject line. When reply the vote topic, include either “[+1]” for agree or “[-1]” for disagree in the subjects. One can also use “[0]” for no preference.
Comment:	The initial voting call uses “[Vote]” solely, participants cast vote by replying in the thread and adding “[+1/0/-1]” in the subjects. The result of the vote could be parsed by easily.
Example:	Re: [VOTE] Release httpd 2.3.6-alpha

Table 4.10: Usage Pattern of Tag “Announce”

Tag Name:	Announcement
Variants:	[Announce], [ANN], [Announcement]
Context:	General discussions; There are news to be published.
Problem:	How to quickly identify “press release”-like messages?
Solution:	Add “[Announce]” as prefix in the email’s subject line.
Example:	[ANN] Stable version 3.2 released!

Solved

This tag is inspired from QA-style forums. It reflects that a certain problem is solved. It allows readers that just want the answer to a question or problem, to skip most of the thread and read the answer right away without further exploring (Table 4.11).

Table 4.11: Usage Pattern of Tag "Solved"

Tag Name:	Solved
Variants:	[Solved]
Context:	In a discussion where certain problems get solved.
Problem:	How to quickly identify problems that are solved, i.e. closed questions?
Solution:	Add "[Solved]" as prefix in the email's subject line when replying.
Comment:	Because email cannot be edited once sent, marking a thread as solved does not works the same way like in forums, So use this tag in a reply message.

4.3 Survey Result for Proposal Acceptance

The survey mentioned in chapter 3 also contains questions regarding to these proposed tag schema. A description of each tag is given, survey participants can choose between "Yes, the description is correct" and "No, the description is not correct". When choosing "No", one can additionally give his or her own thoughts about the meaning of this tag.

The result is shown in Table 4.12.

	Tag	Acceptance	Margin of Error at Confidence Level 95%	Margin of Error at Confidence Level 99%
1	Bug	1.00	0.00	0.00
2	Patch	1.00	0.00	0.00
3	Issue	0.78	0.21	0.28
4	RFC	0.78	0.21	0.28
5	Tip	1.00	0.00	0.00
6	Proposal	1.00	0.00	0.00
7	Vote	0.89	0.16	0.21
8	Announce	0.67	0.24	0.31
9	Solved	0.89	0.16	0.21
10	Project Name	1.00	0.00	0.00

Table 4.12: Acceptance rate of tag schema

Result indicates that the proposed tag schema has high acceptance rate. Under the given margin of errors, more than 60% of the survey participants agree the definition of each tag. Half of the tags are even agreed by 100% of the participants. One exception is the tag “Announce”, its acceptance rate is relative low, however, the reason could be that in the survey questions, this tag has used “ANN” as the tag text, some may think these three letters have meanings other than “announcement”, or simply over-abbreviated.

4.4 Use Cases

The use cases of these tags are organized by the phases of open source development.

Project Initialization If a project is started from scratch, developers could use mailing list to exchange ideas or workplans about this project, they may also need to decide among several good ideas. In this case, the tags “Proposal” and “Vote” are suitable. Project organizers can look for messages tagged with “[Proposal]” if they want to focus on the project instead of other messages like self introductions.

Project execution Most development works happen in this phase. There are coding and testing tasks, code reviews, code commit and documentation. Tags that are especially suitable in this phase are “Bug”, “Patch”, “RFC”.

Project releasing A message with the tag “Announce” to declare the release of the project may be the best choice. After release, there will also be support/maintenance tasks: Issues reported by users need to be addressed, as well as answering questions and solving problems. In this case the tags “Issue” and “Solved” and “Tip” are suitable.

Generally, tags can be used as filter criteria, users could e.g. set their own filters to obtain customized message lists that they want. This function, however, requires tool supporting. Now that the tag schema has been validated, in chapter 5, the corresponding tool supporting for this tag schema will be presented.

5 Design and Implementation of Tool Supporting

The proposed tag schema has has the purpose of improving the efficiency of open source collaboration. However as mentioned in chapter 1, today's mailing list systems are mostly generic, features such as "tagging a message" and "filter messages by tag" are not supported. As the proposed codification schema has been validated, this chapter will present the design and implementation of tools that provide support for the use of tags in mailing lists: 1) A web-based form to enable users to add new tags as well as edit existed tags. One can set various properties of a tag, e.g. name, description and keywords that could be identified as this tag. This tool also support output of tag data in JSON format, so that tag data could be utilized by 3rd party applications. 2) A mailing list archiver, which, in addition to typical functions of a mailing list archiver (email aggregation), also provide support for the use of tags, e.g. highlighting messages with tags, filtering messages by tag, etc.

5.1 Requirement Analysis

For the web form, following requirements should be fulfilled:

- A set of properties that a tag should have needs to be defined, these properties also need to be exported in JSON format.
- A form that can let users add new tags and edit various properties of a tag.
- Clients should be able to access tag data (the exportable data in JSON format) via Internet.

This is a typical web application that performs data CRUD (Create/Retrieve/Update/Delete) operations, it also serve as a data source for tag information. Implementing this application based on a hosted platform (e.g. Google App Engine [8]) should be a proper choice.

Some basic requirements for the mailing list archiver include:

- Ability to access an email server to fetch messages, archive them (in a proper form of storage), and show them through a UI.

- The relationships of messages in a mailing list (e.g. topics and its replies) should be kept.
- Based on the tag data, it should be able to assign proper tags to a message by parsing its subject.
- Support of filtering messages by tags.

Most current popular mailing list systems have built-in archivers, they provide simple Web UI, where messages are organized by month or date and displayed in threads. There are also dedicated mailing list archives which aggregate messages from many mailing lists, make the messages browseable and searchable. Usually these archivers have a more sophisticated UI for better experiences. Examples of such archivers are "The Mail Archive [2]", "MarkMail [4]", "Gmane [14]" and "MARC [17]". The archiver application to be implemented is also a stand-alone, dedicated mailing list aggregator, web-based, plus features that support the use of tags.

5.2 System Design

The application infrastructure from Google — Google App Engine — is chosen for the simple form which do the tag editing, this application is thus quite simple to implement, the important task is designing the data model of tags.

The mailing list archiver, on the other hand, should be implemented as a typical multi-tier, data driven web application. Several design aspects need to be concerned, namely the data modeling for email messages, mailing lists and tags, the mechanism to access email inbox and fetch messages, the necessary UI logics to display email threads, and so on.

5.2.1 Data Modeling

Since the web form application is based on Google App Engine, the modeling of the "Tag" entity is directly shown in code:

```
class EmailTag(db.Model):
    name = db.StringProperty(required=True)
    description = db.TextProperty(default='')
    tag = db.StringProperty(default='')
    keywords = db.StringListProperty()
    update_time = db.DateTimeProperty(auto_now=True, auto_now_add=True)
    example = db.TextProperty(default='')
    author = db.UserProperty()
    version = db.IntegerProperty(default=1, required=True)
```

Most of the properties are text type and self-explained, worth to note is only the "keywords" property. Its value is a list of strings, which are the keywords that are used to identify this tag. For example, the tag “Announce” has keywords “ANN”, “Announce” and “Announcement”, if any one of these three keywords is present in an email’s subject line (more specifically, in the square brackets), then this message can be marked with the “Announce” tag. This is the simplest algorithm for tagging a message.

Data Modeling for the Archiver Application

An Entity-Relationship Diagram for the mailing list archiver application is shown in Figure 5.1. The main entity types in the archiver application are “Mailing List”, “Email”

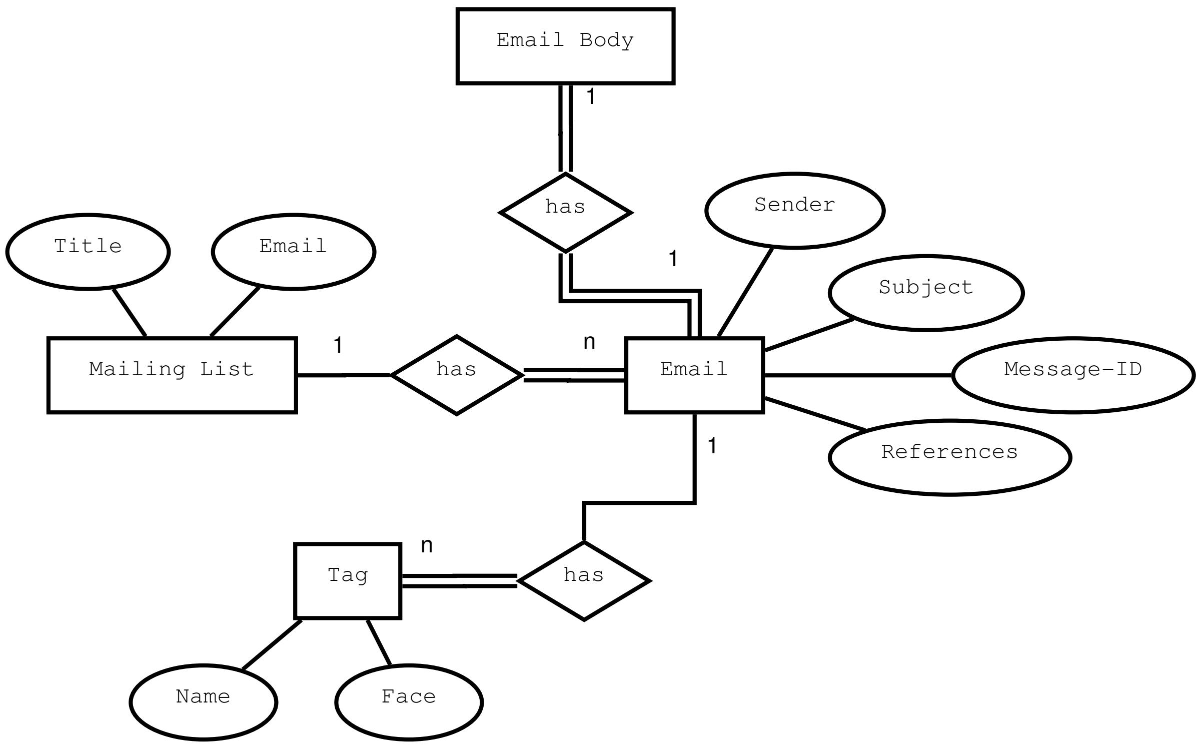


Figure 5.1: Entity-Relationship Diagram for the Mailing List Archiver

and “Tag”. The important attributes of “Mailing List” include its name and its email address for posting messages. “Email” has attributes that are in accordance with the header fields of a message defined in RFC 2822 [10], e.g. “Sender”, “Subject”, “Message-ID” and “References”, as shown in the diagram. The entity type “Email Body” is separately modeled because of the consideration about possible database performance issue. The body of an email may contain large amount of data, but normally, when a user browse or search the archive, the contents of messages will not be listed immediately, unless the user promptly requests i.e. clicks the subject to view the whole message.

5.2.2 Application Architecture

The mailing list archiver application, as described above, will be designed as a multi-tier data driven web application. Its main architecture is illustrated in Figure 5.2. All

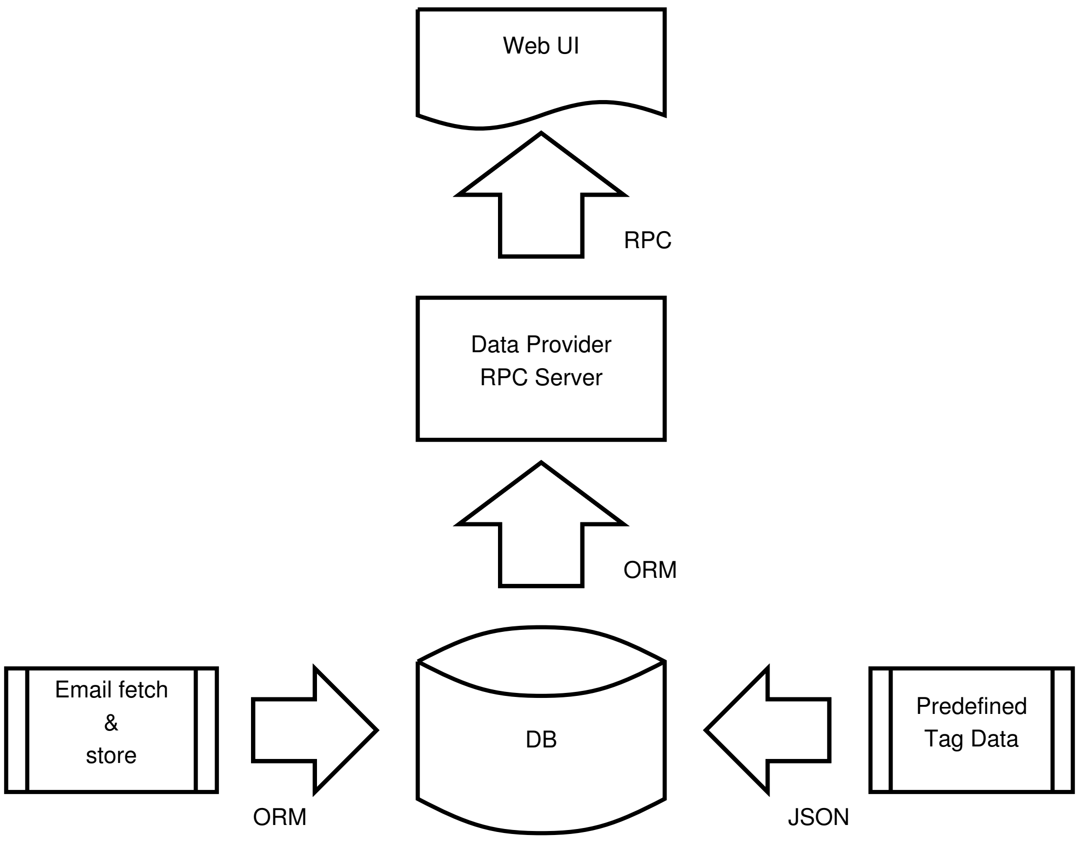


Figure 5.2: Main architecture of the mailing list archiver application

instances of the data model, i.e. entity set are stored in a database. The "Object-relationship mapping" (ORM) technique is employed, so that the data model could be represented in object, without concerning much about the underlying database-specific aspects.

The tags information is retrieved from the web form and stored in the database. The corresponded component will take care of tasks such as JSON format converting, web access and caching. Another component will communicate with an email inbox and fetch messages, then build instances of the email data model by using the header and body values of each message, finally persist those instances into database.

On the front-end, the users will interact with a panel-based UI. There will be list of mailing lists, list of messages (thread topics) that the current selected mailing list contains, then if a topic is selected, the whole thread will be displayed, which include subjects and contents.

The data that are presented in the front-end UI was provided by a middle tier. The necessary mechanism to get model instances out from database and logics for data manipulation are performed by this component.

5.2.3 Toolkits and Frameworks

For the concrete technologies that can be employed to implement this application, the following software, toolkits and frameworks are selected:

- PostgreSQL is chosen to implement the underlying database. It is a powerful, open source object-relational database system, runs on different platforms, also supports various enterprise level features [12].
- For the ORM functions, the Hibernate framework will be used. Hibernate is a collection of related projects enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping [13]. Hibernate has built-in SQL dialect that supports PostgreSQL.
- Because Hibernate is a Java-based framework, the programming language for the archiver application has the clear choice: Java. There are lots of frameworks that are specialized Java web development, among which, The Google Web Toolkit (GWT) is selected in this case. GWT is a development toolkit for building and optimizing complex browser-based applications. Its goal is to enable productive development of high-performance web applications without the developer having to be an expert in browser quirks, XMLHttpRequest, and JavaScript [9]. GWT has its own ways to implement the communication between client JavaScript code and the server-side code, one of them is "Remote Procedure Call" (PRC). GWT RPC is a mechanism for passing Java objects to and from a server over standard HTTP. You can use the GWT RPC framework to transparently make calls to Java servlets and let GWT take care of low-level details like object serialization. This is shown in the architecture figure above.
- The component that handles email fetching will be written based on the JavaMail API, which provides a platform-independent and protocol-independent framework to build mail and messaging applications [20].

5.3 System Implementation

The whole application is created upon the basis of a GWT Project. The basis project has already provided a skeleton of Java web application. All the 3rd party dependencies, e.g.

JDBC driver, Hibernate library files and JavaMail library can be referenced by placing them in the "WEBINF/lib" directory.

5.3.1 ORM and Persistence

Figure 5.3 shows the object-oriented data modeling of the main objects in the application.

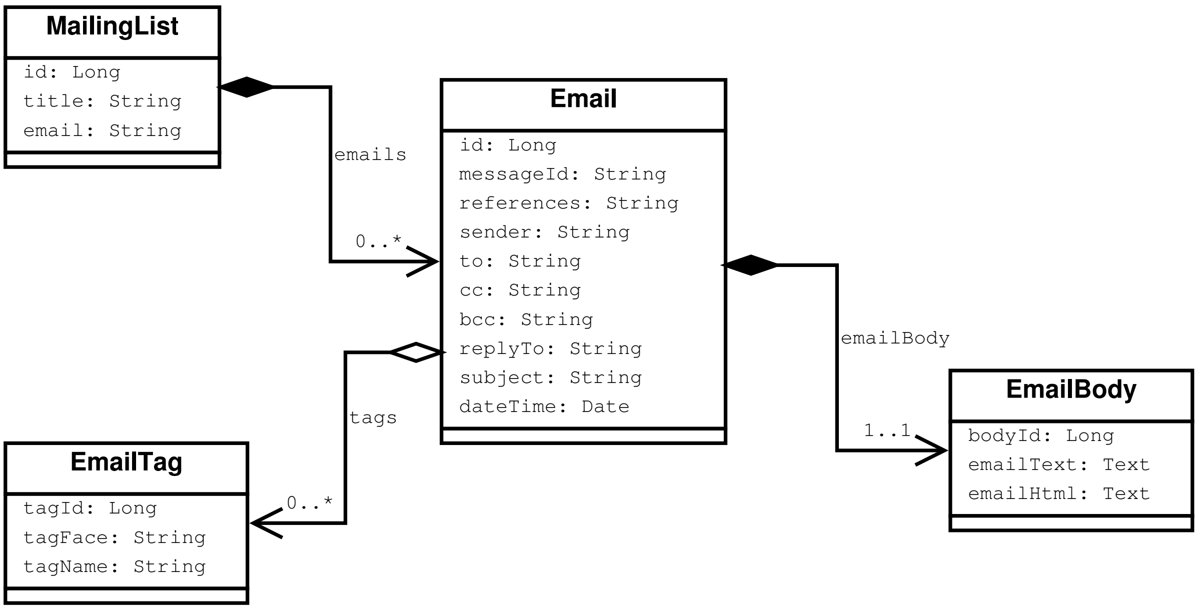


Figure 5.3: OO data modeling of the main objects in the archiver application

For Hibernate to automatically handle the ORM related tasks, such as creating database schema and the entity classes, corresponding Hibernate mapping file for each entity Java class must be created. Here shows the mapping file for the class "Email" as an example:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="de.fau.cs.osr.etikett.entity.Email" table="EMAILS">
    <id name="id" type="long" column="EMAIL_ID">
      <generator class="native" />
    </id>
    <property name="messageId" type="string" column="MESSAGE_ID" />
    <property name="references" type="text" column="EMAIL_REFERENCES" />
    <property name="sender" type="string" column="EMAIL_SENDER" />
    <property name="to" type="text" column="EMAIL_TO" />
    <property name="cc" type="text" column="EMAIL_CC" />
    <property name="bcc" type="text" column="EMAIL_BCC" />
    <property name="replyTo" type="string" column="EMAIL_REPLY_TO" />
    <property name="subject" type="string" column="EMAIL_SUBJECT" />
    <property name="dateTime" type="timestamp" column="EMAIL_DATE_TIME" />
    <many-to-one name="emailBody"
      class="de.fau.cs.osr.etikett.entity.EmailBody" column="BODY_ID"
      unique="true" cascade="all" />
    <set name="tags" table="EMAIL_TAGGING">
      <key column="EMAIL_ID" />
      <many-to-many column="TAG_ID"
        class="de.fau.cs.osr.etikett.entity.EmailTag" />
    </set>
  </class>
</hibernate-mapping>
```

An example of generated entity class “MailingList” is shown below:

```
...
public class MailingList implements java.io.Serializable {

  private long id;
  private String title;
  private String email;
  private Collection<Email> emails = new ArrayList<Email>(0);

  public MailingList() {
  }

  public MailingList(String title, String email) {
    this.title = title;
    this.email = email;
  }

  public MailingList(String title, String email, Collection<Email> emails) {
    this.title = title;
    this.email = email;
    this.emails = emails;
  }

  public long getId() {
    return this.id;
  }

  public void setId(long id) {
    this.id = id;
  }

  // Other getters and setters...
}
...
```

The persisting of entities also uses standard Hibernate methods, an example here is the persistence of a “Tag” instance.

```
...
// Helper class for SessionFactory
public class HibernateUtil {

  private static final SessionFactory sessionFactory;

  static {
    try {
      sessionFactory = (new Configuration()).configure()
        .buildSessionFactory();
    } catch(Throwable ex) {
      System.err.println("Initial SessionFactory creation failed: " + ex);
      throw new ExceptionInInitializerError(ex);
    }
  }

  public static SessionFactory getSessionFactory() {
    return sessionFactory;
  }

}

...

// initialize the EmailTag table
public Long addEmailTag(EmailTag tag) {
  Session sess = HibernateUtil.getSessionFactory().openSession();
  Transaction trans = null;
  Long tagId = null;
  try {
    trans = sess.beginTransaction();
    tagId = (Long)sess.save(tag);
    trans.commit();
  } catch(HibernateException e) {
    trans.rollback();
    e.printStackTrace();
  } finally {
    sess.close();
  }
  return tagId;
}
```

5.3.2 Fetching, Tagging and Storing of Emails

The component that fetches messages from a email inbox has the workflow as shown in Figure 5.4.

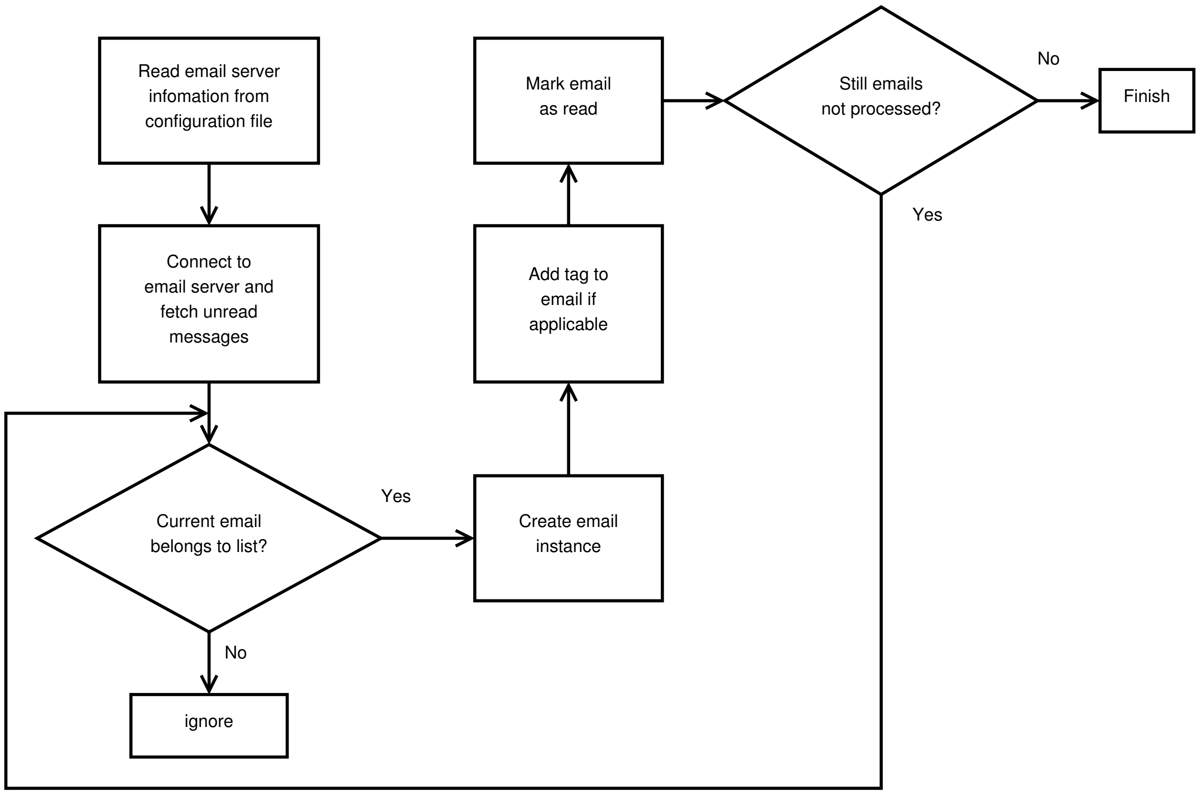

Read email server infomation from configuration file
Connect to email server and fetch unread messages
Current email belongs to list?
Yes
No
ignore
Create email instance
Add tag to email if applicable
Mark email as read
Still emails not processed?
No
Yes
Finish

Figure 5.4: Workflow of the email fetching component

One question here is how to obtain only unread emails. This can be accomplished by using the `search` method of the Folder object:

```
Message[] messages = folder.search(new FlagTerm(new Flags(Flags.Flag.SEEN),
  false)); // need only unread mails
```

The function of tagging a message is implemented in another component named "EmailTitleParser", it uses the "Strategy design pattern" [33], so that there can be more than one algorithm defined to parse the subject of a message to determine if it can be tagged. Currently only one simple algorithm is implemented — Using regular expression to search for square brackets in the subject, extracting the keywords inside, then see if they match any of the predefined tags.

Speak of predefined tags, they refer to the exported tags (in JSON format) from the web form. There is also a "EmailTagDataProvider" that access the web from to get the exported tags, cache them locally, and convert the JSON format into Java objects for other components to use.

The complete source code of these components are listed in Appendix A.

5.3.3 Data Transfer Objects

Google Web Toolkit has its own mechanism to make the client JavaScript code be able to call the server side methods (Figure 5.5).

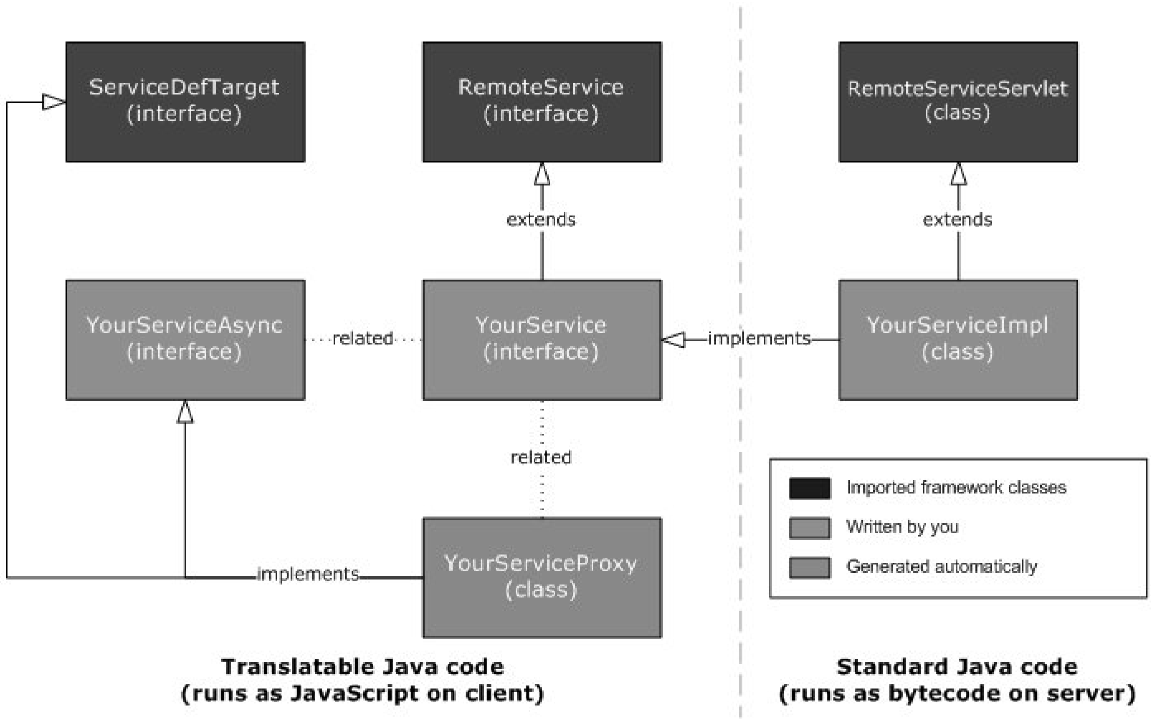

ServiceDefTarget (interface)
RemoteService (interface)
RemoteServiceServlet (class)
extends
extends
YourServiceAsync (interface)
related
YourService (interface)
implements
YourServiceImpl (class)
related
YourServiceProxy (class)
implements
Imported framework classes
Written by you
Generated automatically
Translatable Java code (runs as JavaScript on client)
Standard Java code (runs as bytecode on server)

Figure 5.5: The RPC mechanism of GWT [9]

Under the hood, GWT will serialize the server side objects, so that they could transfered to client side. But this serialization process cannot be applied to Hibernate objects [3]. Unfortunately in this case, the entities of the application are all Hibernate objects.

One solution is using "Data Transfer Objects" (DTO). this introduces a light object to go between the heavy Hibernate object and its data representation that the client side care about. The DTO is a simple POJO only containing simple data fields that the client side can access to display on the application page. The Hibernate objects can then be constructed from the data in the data transfer objects. The DTOs themselves will only contain the data that need to be persisted, but none of the lazy loading or persistence logic, which cause the GWT serialization failure.

Take the “MailingList” class as example. First, one more constructor needs to be added so that the Hibernate object can be created from the DTO:

```
public MailingList(MailingListDTO mailingListDTO) {
  this.id = mailingListDTO.getId();
  this.title = mailingListDTO.getTitle();
  this.email = mailingListDTO.getEmail();
  Collection<EmailDTO> emailDTOs = mailingListDTO.getEmails();
  if(emailDTOs != null) {
    Collection<Email> emails = new ArrayList<Email>(emailDTOs.size());
    for(EmailDTO emailDTO : emailDTOs) {
      emails.add(new Email(emailDTO));
    }
    this.emails = emails;
  }
}
```

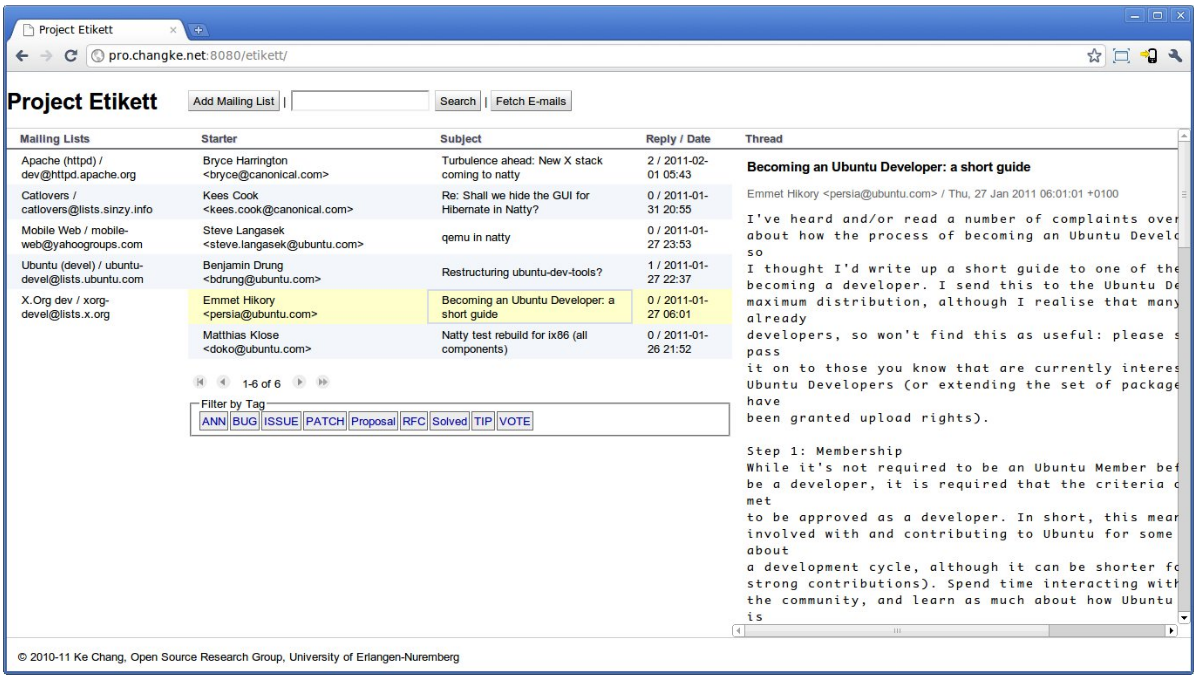
Then, in the logic where Hibernate objects are exposed to RPC, the DTOs are used instead:

```
// list all the mailing lists at front page
public List<MailingListDTO> getMailingLists() {
  Session sess = HibernateUtil.getSessionFactory().openSession();
  Transaction trans = null;
  List<MailingListDTO> listDTOs = null;
  try {
    trans = sess.beginTransaction();
    Query query = sess.createQuery(
      "from MailingList ml order by ml.title asc");
    List<MailingList> lists = new ArrayList<MailingList>(query.list());
    listDTOs = new ArrayList<MailingListDTO>(
      lists != null ? lists.size() : 0);
    if(lists != null) {
      for(MailingList ml : lists) {
        listDTOs.add(createMailingListDTO(ml));
      }
    }
    trans.commit();
  } catch(HibernateException e) {
    trans.rollback();
    e.printStackTrace();
  } finally {
    sess.close();
  }
  return listDTOs;
}
...
private MailingListDTO createMailingListDTO(MailingList mailingList) {
  return new MailingListDTO(mailingList.getId(), mailingList.getTitle(),
    mailingList.getEmail());
}
```

...

5.3.4 Web UI

GWT provides lots of built-in UI components, allow developers to create a web UI almost without the need of doing client HTML coding. The archiver application has mainly used the "LayoutPanels" to build the panel-based UI, and the "Data Presentation Widgets" to build the message lists (Figure 5.6).



Project Etikett

pro.changke.net:8080/etikett/

Project Etikett

Mailing Lists	Starter	Subject	Reply / Date
Apache (httpd) / dev@httpd.apache.org	Bryce Harrington <bryce@canonical.com>	Turbulence ahead: New X stack coming to natty	2 / 2011-02-01 05:43
Catlovers / catlovers@lists.sinzy.info	Kees Cook <kees.cook@canonical.com>	Re: Shall we hide the GUI for Hibernate in Natty?	0 / 2011-01-31 20:55
Mobile Web / mobile-web@yahoogroups.com	Steve Langasek <steve.langasek@ubuntu.com>	qemu in natty	0 / 2011-01-27 23:53
Ubuntu (devel) / ubuntu-devel@lists.ubuntu.com	Benjamin Drung <bdrung@ubuntu.com>	Restructuring ubuntu-dev-tools?	1 / 2011-01-27 22:37
X.Org dev / xorg-devel@lists.x.org	Emmet Hikory <persia@ubuntu.com>	Becoming an Ubuntu Developer: a short guide	0 / 2011-01-27 06:01
	Matthias Klose <doko@ubuntu.com>	Natty test rebuild for ix86 (all components)	0 / 2011-01-26 21:52

1-6 of 6

Filter by Tag: ANN BUG ISSUE PATCH Proposal RFC Solved TIP VOTE

Thread

Becoming an Ubuntu Developer: a short guide

Emmet Hikory <persia@ubuntu.com> / Thu, 27 Jan 2011 06:01:01 +0100

```
I've heard and/or read a number of complaints over
about how the process of becoming an Ubuntu Develo
so
I thought I'd write up a short guide to one of the
becoming a developer. I send this to the Ubuntu De
maximum distribution, although I realise that many
already
developers, so won't find this as useful: please s
pass
it on to those you know that are currently interes
Ubuntu Developers (or extending the set of package
have
been granted upload rights).

Step 1: Membership
While it's not required to be an Ubuntu Member bef
be a developer, it is required that the criteria o
met
to be approved as a developer. In short, this mear
involved with and contributing to Ubuntu for some
about
a development cycle, although it can be shorter fo
strong contributions). Spend time interacting with
the community, and learn as much about how Ubuntu
is
```

© 2010-11 Ke Chang, Open Source Research Group, University of Erlangen-Nuremberg

Figure 5.6: Web UI of the mailing list archiver application

As stated in the requirements analysis, this archiver should list messages in threads, much like a web forum, the middle panel shows only topic messages, as well as the number of replies. If a user select a topic, the whole thread will be displayed in the right panel. The GWT data presentation widget — in this case the "CellTable" — accepts a Java List object and will iterate it automatically to show its contents. This list object was obtained through RPC, so in the server side, a proper list of "topics" needs to be built. This needs some Hibernate query tricks:

```
...
public Collection<Topic> getTopics(Long listId, String tagFace) {
  Session sess = HibernateUtil.getSessionFactory().openSession();
  Transaction trans = null;
  Collection<Topic> topics = null;
  try {
    trans = sess.beginTransaction();
    MailingList list = (MailingList)sess.load(MailingList.class, listId);
    Collection<Email> topicEmails = null;

    if(tagFace.length() == 0) {
      // no tag specified, get all
      topicEmails = new ArrayList(sess.createFilter(list.getEmails(),
        "where this.references = ''").list());
    } else {
      EmailTag tag = getEmailTagByFace(tagFace);
      topicEmails = new ArrayList<Email>(
        sess.createFilter(list.getEmails(),
        "where this.references = '' and :tag in
        elements(this.tags)").setParameter("tag", tag).list());
    }
    topics = new ArrayList<Topic>(topicEmails != null ?
      topicEmails.size() : 0);
    if(topicEmails != null) {
      for(Email email : topicEmails) {
        // get reply count and last update time for each topic
        Topic topic = new Topic(createEmailDTO(email));
        Iterator results = sess.createQuery(
          "select count(em),
          max(em.dateTime)
          from Email as em
          where locate(:msg_id, em.references) > 0")
          .setString("msg_id",
          email.getMessageId()).list().iterator();
        if(results.hasNext()) {
          Object[] row = (Object[])results.next();
          Long replyCount = (Long)row[0];
          Date lastUpdateTime = (Date)row[1];
          if(replyCount > 0l) {
            topic.setReplyCount(replyCount);
            topic.setLastUpdateTime(lastUpdateTime);
          }
        }
        topics.add(topic);
      }
    }
    trans.commit();
  } catch(HibernateException e) {
    trans.rollback();
    e.printStackTrace();
  } finally {
    sess.close();
  }
  return topics;
}
```

This function also does the “filtering by tag” task. On the web UI the available tags are listed below the middle panel, If a user click a tag, then only topics with this tag will be shown.

To decide if a message is a “topic”, one can check for its “references” field. If the field value is empty, then this message should be a starting message, because it does not reference any other messages. Also by checking this field, one can find out which other message the current message replies to, so that it is easy to count the reply count of a topic, as shown in the code snippet above.

5.3.5 Deployment

A GWT project needs be compiled by the GWT SDK before deployment. A JSP container, e.g. Apache Tomcat is also needed. PostgreSQL server must be accessible by the application, and the corresponding parameters such as database username, password and database name need to be set in the Hibernate configuration files.

A email account should be setup before. This email will be used to subscribe mailing lists, and its account information needs to be written in the application’s own configuration file.

To enable emails being automatically checked after a certain period, a cronjob have to be added to the operating system. The application provides a URL entry address, the cronjob can use “wget” tool to initialize an local HTTP GET request and force the component to retrieve emails.

5.4 Survey Result for Feature Acceptance

The survey mentioned in chapter 3 also contains question regarding to the main features of the archiver application:

- While Mailman and The Mail Archive display emails of the same thread in a simple ”tree” form, our project and MarkMail display them in a ”flat” form, like a web forum, or the Gmail conversation view. What do you think about displaying email threads in a ”flat” form?

- I assume that, certain emails with tags in their titles can be identified more quickly because tags can attract your attention visually. I have included this feature in my project, so that if it finds a certain tag, it will show it separately. What do you think about this feature?

- Another good use is quick filtering by tags. Instead of typing search keywords, in my project one can list the emails with a certain tag in one click. What do you think about this feature?

The answers is a scoring from -2 to 2 (will be calculated as 1 to 5), means how much the participants dislike or like such each feature.

The result is shown in Table 5.1.

	List	Mean	Standard Error	Lower Bound	Upper Bound
1	Flat-Form Thread	4.56	0.24	4.08	5.00
2	Tags Highlight	4.44	0.34	3.78	5.00
3	Filter by Tag	4.67	0.33	4.01	5.00

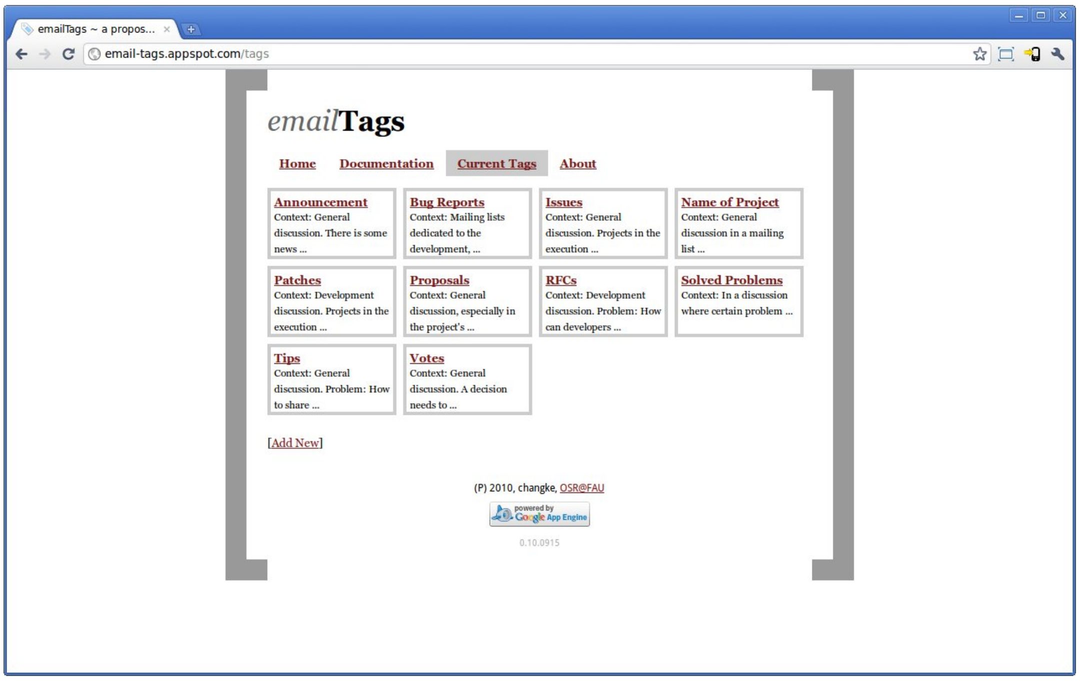
Table 5.1: Feature Preferences of the Mailing List Archiver Application

Result shows that all these features have got more than an average score of 4.0, which is high under the given standard errors. As was expected, these main features of the archiver application are clearly endorsed by the participants. The reason could be that the support of tags is unique, compared with other similar products. And the idea of message tagging is also widely accepted, as shown in other survey results in previous chapters.

5.5 Usage Scenario

The web form for tag editing provides a way of building folksonomy. Everyone can suggest new tags, or refine the keywords match set of current tags, so that these tags can be used more precisely. Some screenshots of the application is shown in Figure 5.7 and Figure 5.8.

Once a mailing list is subscribed, one can add its name and email address into the archiver application through a pop-up box. The application will update its message database regularly following the time interval configured in cronjob. Users choose mailing list to view the threads it contains, and then choose a thread to view all the messages with content. If a message contains one of the keywords of a predefined tag, this tag will be shown in the thread list to call the attention visually. Users can specify to show only messages with a certain tag by clicking in the tag list area (Figure 5.9).



Announcement Context: General discussion. There is some news ...

Bug Reports Context: Mailing lists dedicated to the development, ...

Issues Context: General discussion. Projects in the execution ...

Name of Project Context: General discussion in a mailing list ...

Patches Context: Development discussion. Projects in the execution ...

Proposals Context: General discussion, especially in the project's ...

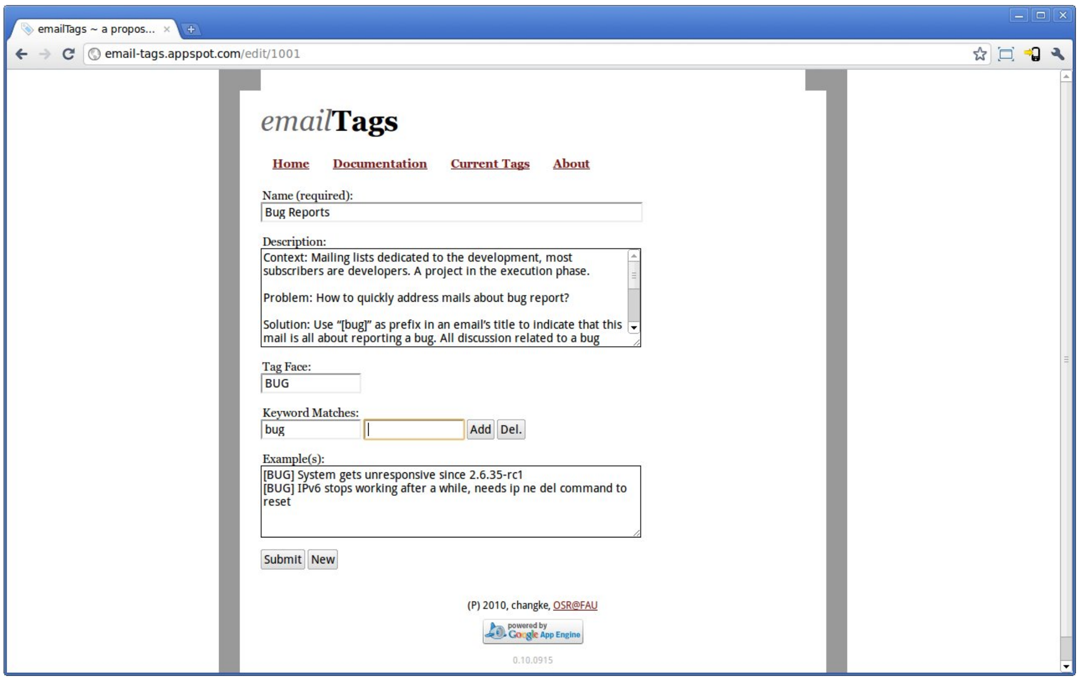
RFCs Context: Development discussion. Problem: How can developers ...

Solved Problems Context: In a discussion where certain problem ...

Tips Context: General discussion. Problem: How to share ...

Votes Context: General discussion. A decision needs to ...

Figure 5.7: Web UI listing current tags



Name (required):
Bug Reports

Description:
Context: Mailing lists dedicated to the development, most subscribers are developers. A project in the execution phase.

Problem: How to quickly address mails about bug report?

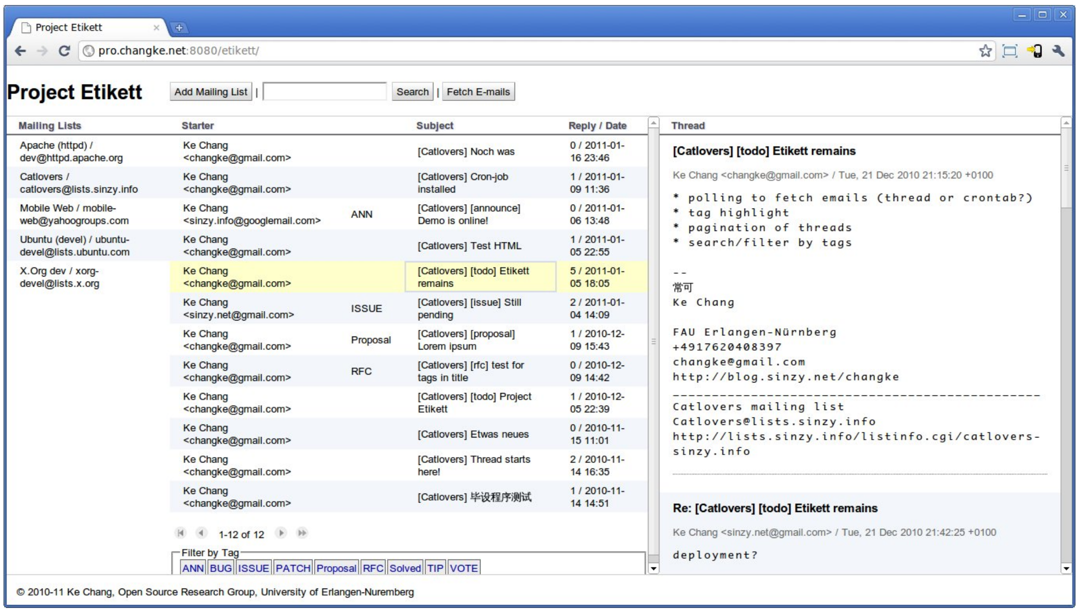
Solution: Use "[bug]" as prefix in an email's title to indicate that this mail is all about reporting a bug. All discussion related to a bug

Tag Face:
BUG

Keyword Matches:
bug

Example(s):
[BUG] System gets unresponsive since 2.6.35-rc1
[BUG] IPv6 stops working after a while, needs ip ne del command to reset

Figure 5.8: Web form for editing tag data



Project Etikett

Mailing Lists	Starter		Subject	Reply / Date
Apache (httpd) / dev@httpd.apache.org	Ke Chang <changke@gmail.com>		[Catlovers] Noch was	0 / 2011-01-16 23:46
Catlovers / catlovers@lists.sinzy.info	Ke Chang <changke@gmail.com>		[Catlovers] Cron-job installed	1 / 2011-01-09 11:36
Mobile Web / mobile-web@yahoogroups.com	Ke Chang <sinzy.info@googlemail.com>	ANN	[Catlovers] [announce] Demo is online!	0 / 2011-01-06 13:48
Ubuntu (devel) / ubuntu-devel@lists.ubuntu.com	Ke Chang <changke@gmail.com>		[Catlovers] Test HTML	1 / 2011-01-05 22:55
X.Org dev / xorg-devel@lists.x.org	Ke Chang <changke@gmail.com>		[Catlovers] [todo] Etikett remains	5 / 2011-01-05 18:05
	Ke Chang <sinzy.net@gmail.com>	ISSUE	[Catlovers] [issue] Still pending	2 / 2011-01-04 14:09
	Ke Chang <changke@gmail.com>	Proposal	[Catlovers] [proposal] Lorem ipsum	1 / 2010-12-09 15:43
	Ke Chang <changke@gmail.com>	RFC	[Catlovers] [rfc] test for tags in title	0 / 2010-12-09 14:42
	Ke Chang <changke@gmail.com>		[Catlovers] [todo] Project Etikett	1 / 2010-12-05 22:39
	Ke Chang <changke@gmail.com>		[Catlovers] Etwas neues	0 / 2010-11-15 11:01
	Ke Chang <changke@gmail.com>		[Catlovers] Thread starts here!	2 / 2010-11-14 16:35
	Ke Chang <changke@gmail.com>		[Catlovers] 毕设程序测试	1 / 2010-11-14 14:51

Thread

[Catlovers] [todo] Etikett remains

Ke Chang <changke@gmail.com> / Tue, 21 Dec 2010 21:15:20 +0100

```
* polling to fetch emails (thread or crontab?)
* tag highlight
* pagination of threads
* search/filter by tags


--
常可
Ke Chang

FAU Erlangen-Nürnberg
+4917620408397
changke@gmail.com
http://blog.sinzy.net/changke
______________________________________________
Catlovers mailing list
Catlovers@lists.sinzy.info
http://lists.sinzy.info/listinfo.cgi/catlovers-
sinzy.info
```

Re: [Catlovers] [todo] Etikett remains

Ke Chang <sinzy.net@gmail.com> / Tue, 21 Dec 2010 21:42:25 +0100

```
deployment?
```

© 2010-11 Ke Chang, Open Source Research Group, University of Erlangen-Nuremberg

Figure 5.9: Mailing List Archiver highlights Tags

6 Conclusion and Perspective

This chapter summarizes and concludes the works done by this thesis as well as potential improvements in the future.

6.1 Conclusion

The main concern of this thesis is the collaboration practices in open source software development. More specifically, the purpose of the works is to propose a way that can improve the efficiency of collaboration using mailing lists. The idea is to introduce an annotation schema that is used to flag the messages in a mailing list as different types, this in turn provides developers better control to the information flow they consumes, thus can make the collaboration run more effectively.

To do this, analysis have been conducted on selected mailing lists of several popular open source projects to find out the usage patterns of developers. For example, reporting bugs, submitting patches, announcing releases, casting votes and so on. These patterns are codified into several tags — most of which are already being used in mailing lists as prefixes to the subject lines of messages. In addition to the practical usage, the tag schema also conform with the "conversation for action" theory as well as the open source development process.

Besides the proposal of tag schema, this thesis also contributes tool supporting for it. With the web form, the tag schema can be continually developed and refined. The mailing list archiver application provides supports to the tags, e.g. highlights tags which a message may contain, organizes messages into threads and allows users to filter messages by tag.

A survey was conducted as a validation to the representativity of mailing list data source, the acceptance of tag schema and the feature set of the archiver application. Current result shows a positive response.

6.2 Perspective

The contributions of this thesis are still explorative, potential improvements could be found in following points:

- The tag schema could be extended, in order to represent the best practices of open source collaboration more accurately. Despite of the popularity of the projects, the number of mailing lists analyzed in this thesis is still limited.
- The tag schema are provided as JSON format data by the web form. One purpose of this design is aiming to provide data for 3rd party usage. There could be more tool that use these data to integrate the tag schema. For example, an email client software could possibly use the data to provide assistance features like tag auto-complete.
- The mailing list archiver application is still in prototype stage. It could implement more features such as full-text search, auto-actions based on message types, for example, it could automatically dispatch a bug report message to a project management system, or it could show the result of a vote by counting the "+1/-1" tags in the thread. Eventually it could evolve to a new type of information and collaboration hub for open source software development.

6.3 Acknowledgement

My thanks go to the Open Source Research Group [1] of University of Erlangen-Nuremberg, especially my thesis supervisor Prof. Dirk Riehle and Dipl.-Inf. Carsten Kolassa for their great ideas and supports.

[1]`http://osr.cs.fau.de/`

7 Appendix: Source Code Listing

List of important source code of the mailing list archiver application.

7.1 Hibernate Entity Mapping Files

7.1.1 Email.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="de.fau.cs.osr.etikett.entity.Email" table="EMAILS">
    <id name="id" type="long" column="EMAIL_ID">
      <generator class="native" />
    </id>
    <property name="messageId" type="string" column="MESSAGE_ID" />
    <property name="references" type="text" column="EMAIL_REFERENCES" />
    <property name="sender" type="string" column="EMAIL_SENDER" />
    <property name="to" type="text" column="EMAIL_TO" />
    <property name="cc" type="text" column="EMAIL_CC" />
    <property name="bcc" type="text" column="EMAIL_BCC" />
    <property name="replyTo" type="string" column="EMAIL_REPLY_TO" />
    <property name="subject" type="string" column="EMAIL_SUBJECT" />
    <property name="dateTime" type="timestamp" column="EMAIL_DATE_TIME" />
    <many-to-one name="emailBody"
      class="de.fau.cs.osr.etikett.entity.EmailBody" column="BODY_ID"
      unique="true" cascade="all" />
    <set name="tags" table="EMAIL_TAGGING">
      <key column="EMAIL_ID" />
      <many-to-many column="TAG_ID"
        class="de.fau.cs.osr.etikett.entity.EmailTag" />
    </set>
  </class>
</hibernate-mapping>
```

7.1.2 EmailBody.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="de.fau.cs.osr.etikett.entity.EmailBody" table="EMAIL_BODIES">
    <id name="bodyId" type="long" column="BODY_ID">
      <generator class="native" />
    </id>
    <property name="emailText" type="text" column="EMAIL_TEXT" />
    <property name="emailHtml" type="text" column="EMAIL_HTML" />
  </class>
</hibernate-mapping>
```

7.1.3 EmailTag.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="de.fau.cs.osr.etikett.entity.EmailTag" table="EMAIL_TAGS">
    <id name="tagId" type="long" column="TAG_ID">
      <generator class="native" />
    </id>
    <property name="tagFace" type="string" column="TAG_FACE" />
    <property name="tagName" type="string" column="TAG_NAME" />
  </class>
</hibernate-mapping>
```

7.1.4 MailingList.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="de.fau.cs.osr.etikett.entity.MailingList" table="MAILING_LISTS">
    <id name="id" type="long" column="LIST_ID">
      <generator class="native" />
    </id>
    <property name="title" type="string" column="LIST_TITLE" not-null="true"
      unique="true" />
    <property name="email" type="string" column="LIST_EMAIL" not-null="true"
      />
    <bag name="emails" cascade="all">
      <key column="LIST_ID" />
      <one-to-many class="de.fau.cs.osr.etikett.entity.Email" />
    </bag>
  </class>
</hibernate-mapping>
```

7.2 Server Side Code

7.2.1 EmailFetcher.java

```
package de.fau.cs.osr.etikett.server;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.Properties;

import javax.mail.AuthenticationFailedException;
import javax.mail.BodyPart;
import javax.mail.Flags;
import javax.mail.Folder;
import javax.mail.FolderClosedException;
import javax.mail.FolderNotFoundException;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.NoSuchProviderException;
import javax.mail.Part;
import javax.mail.Session;
import javax.mail.Store;
import javax.mail.StoreClosedException;
import javax.mail.Flags.Flag;
import javax.mail.internet.InternetAddress;
import javax.mail.search.FlagTerm;

import org.apache.commons.lang.StringUtils;

import de.fau.cs.osr.etikett.entity.Email;
import de.fau.cs.osr.etikett.entity.EmailBody;
import de.fau.cs.osr.etikett.entity.EmailTag;
import de.fau.cs.osr.etikett.server.titleparser.EmailTagEx;
import de.fau.cs.osr.etikett.server.titleparser.EmailTitleParser;
import de.fau.cs.osr.etikett.server.titleparser.EmailTitleParsingResult;
import de.fau.cs.osr.etikett.server.titleparser.EmailTitleParsingSimple;
import de.fau.cs.osr.etikett.util.SimpleConfigUtil;

public class EmailFetcher {

  private String imapHost = "";
  private String imapUserName = "";
  private String imapPassword = "";

  private EmailTitleParser titleParser = null;

  private EntityCruder cruder = null;
```

```
public EmailFetcher() {
  SimpleConfigUtil scu = new SimpleConfigUtil();
  this.imapHost = scu.getConfig("IMAP_HOST");
  this.imapUserName = scu.getConfig("IMAP_USERNAME");
  this.imapPassword = scu.getConfig("IMAP_PASSWORD");

  this.titleParser = new EmailTitleParser(new EmailTitleParsingSimple());
  this.cruder = new EntityCruder();
}

private boolean belongToList(Message message, String listEmail) {
  try {
    String allRecipients = InternetAddress
      .toString(message.getAllRecipients());
    if(StringUtils.contains(allRecipients, listEmail)) {
      return true;
    } else {
      return false;
    }
  } catch(MessagingException e) {
    e.printStackTrace();
    return false;
  }
}

public void fetchGmail(String listEmail, Long listId) {
  String host = this.imapHost;
  String userName = this.imapUserName;
  String password = this.imapPassword;

  Properties props = System.getProperties();
  props.setProperty("mail.store.protocol", "imaps");
  Session session = Session.getDefaultInstance(props, null);
  try {
    Store store = session.getStore("imaps");
    store.connect(host, userName, password);
    Folder folder = store.getDefaultFolder();
    folder = folder.getFolder("INBOX");
    folder.open(Folder.READ_WRITE);

    Message[] messages = folder.search(
      new FlagTerm(new Flags(Flags.Flag.SEEN), false));

    Collection<Email> emails = new ArrayList<Email>(0);

    for(Message message : messages) {
      if(belongToList(message, listEmail)) {
        emails.add(buildEmailObject(message));
        message.setFlag(Flag.SEEN, true); // mark email as read
        }
      }

      folder.close(true);
      store.close();

      this.cruder.batchAddEmailsToList(emails, listId);
    } catch(AuthenticationFailedException e) {
      e.printStackTrace();
    } catch(NoSuchProviderException e) {
      e.printStackTrace();
    } catch(FolderClosedException e) {
      e.printStackTrace();
    } catch(FolderNotFoundException e) {
      e.printStackTrace();
    } catch(StoreClosedException e) {
      e.printStackTrace();
    } catch(MessagingException e) {
      e.printStackTrace();
    }
  }

  private Email buildEmailObject(Message message) {
    Email email = new Email();
    try {
      // message id
      String messageId = StringUtils.join(
        message.getHeader("message-id"));
      email.setMessageId(messageId);
      // references
      String references = StringUtils.join(
        message.getHeader("references"));
      email.setReferences(references != null ? references : "");
      // from / sender
      String from = InternetAddress.toString(message.getFrom());
      email.setSender(from);
      // replyTo
      String replyTo = InternetAddress.toString(message.getReplyTo());
      email.setReplyTo(replyTo);
      // to
      String to = InternetAddress.toString(
        message.getRecipients(Message.RecipientType.TO));
      email.setTo(to);
      // cc
      String cc = InternetAddress.toString(
        message.getRecipients(Message.RecipientType.CC));
      email.setCc(cc);
      // bcc
      String bcc = InternetAddress.toString(
        message.getRecipients(Message.RecipientType.BCC));
      email.setBcc(bcc);
      // subject
      String subject = message.getSubject();
      email.setSubject(subject);
      // dateTime
      Date sentDateTime = message.getSentDate();
      email.setDateTime(sentDateTime);

      // email body
      Object content = message.getContent();
      String emailBodyText = "";
      if(content instanceof Multipart) {
        Multipart multipart = (Multipart)content;
        for(int i = 0; i < multipart.getCount(); i++) {
          BodyPart bodyPart = multipart.getBodyPart(i);
          String disposition = bodyPart.getDisposition();
          if(disposition != null
            && (disposition.equals(Part.ATTACHMENT))) {
            // how to handle attachment(s) here?
            continue;
          } else {
            emailBodyText = bodyPart.getContent().toString();
          }
        }
      } else {
        emailBodyText = content.toString();
      }
      EmailBody emailBody = new EmailBody();
      emailBody.setEmailText(emailBodyText);
      email.setEmailBody(emailBody);

      // time to parse title for tags
      EmailTitleParsingResult parseResult =
        this.titleParser.parse(subject);
      ArrayList<EmailTagEx> matchedTags = parseResult.getMatchedTags();
      if(matchedTags.size() > 0) {
        for(EmailTagEx tagEx : matchedTags) {
          EmailTag tag =
            this.cruder.getEmailTagByFace(tagEx.getTag());
          if(tag != null) {
            email.getTags().add(tag);
          }
        }
      }
    } catch(MessagingException e) {
      e.printStackTrace();
    } catch(IOException e) {
      e.printStackTrace();
    }
    return email;
```

```
  }

}
```

7.2.2 EntityCruder.java

```
package de.fau.cs.osr.etikett.server;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

import de.fau.cs.osr.etikett.client.dto.BugInfoDTO;
import de.fau.cs.osr.etikett.client.dto.EmailBodyDTO;
import de.fau.cs.osr.etikett.client.dto.EmailDTO;
import de.fau.cs.osr.etikett.client.dto.EmailTagDTO;
import de.fau.cs.osr.etikett.client.dto.MailingListDTO;
import de.fau.cs.osr.etikett.client.dto.PatchInfoDTO;
import de.fau.cs.osr.etikett.client.dto.Topic;
import de.fau.cs.osr.etikett.entity.BugInfo;
import de.fau.cs.osr.etikett.entity.Email;
import de.fau.cs.osr.etikett.entity.EmailBody;
import de.fau.cs.osr.etikett.entity.EmailTag;
import de.fau.cs.osr.etikett.entity.MailingList;
import de.fau.cs.osr.etikett.entity.PatchInfo;
import de.fau.cs.osr.etikett.util.HibernateUtil;

/* Everything CRUD */

public class EntityCruder {

  // create a mailing list
  public Long createMailingList(MailingListDTO mailingListDTO) {
    MailingList list = new MailingList(mailingListDTO);
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    Long listId = null;
    try {
      trans = sess.beginTransaction();
      listId = (Long)sess.save(list);
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return listId;
  }

  // list all the mailing lists at front page
  public List<MailingListDTO> getMailingLists() {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    List<MailingListDTO> listDTOs = null;
    try {
      trans = sess.beginTransaction();
      Query query = sess.createQuery(
        "from MailingList ml order by ml.title asc");
      List<MailingList> lists = new ArrayList<MailingList>(query.list());
      listDTOs = new ArrayList<MailingListDTO>(
        lists != null ? lists.size() : 0);
      if(lists != null) {
        for(MailingList ml : lists) {
          listDTOs.add(createMailingListDTO(ml));
        }
      }
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return listDTOs;
  }

  // when fetching emails we need all the ids of the lists
  public List<MailingList> getMailingListsForFetching() {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    List<MailingList> lists = null;
    try {
      trans = sess.beginTransaction();
      Query query = sess.createQuery(
        "from MailingList ml order by ml.title asc");
      lists = new ArrayList<MailingList>(query.list());
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return lists;
  }

  // add one email
  public void addEmailToList(Email email, Long mailingListId) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    try {
      trans = sess.beginTransaction();
      MailingList list = (MailingList)sess.load(
        MailingList.class, mailingListId);
      list.getEmails().add(email);
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
  }

  // add more emails
  public void batchAddEmailsToList(Collection<Email> emails,
    Long mailingListId) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    try {
      trans = sess.beginTransaction();
      MailingList list = (MailingList)sess.load(
        MailingList.class, mailingListId);
      list.getEmails().addAll(emails);
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
  }

  // when setting tags for emails, we need fetch the tag first
  public EmailTag getEmailTagByFace(String tagFace) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    EmailTag tag = null;
    try {
      trans = sess.beginTransaction();
      tag = (EmailTag)sess.createQuery(
        "from EmailTag as et where et.tagFace = :tag_face")
        .setString("tag_face", tagFace).uniqueResult();
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return tag;
  }

  // initialize the EmailTag table
  public Long addEmailTag(EmailTag tag) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    Long tagId = null;
    try {
      trans = sess.beginTransaction();
      tagId = (Long)sess.save(tag);
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return tagId;
  }

  // get all the emails (with bodies) from one list, not recommended
  public Collection<EmailDTO> getEmailsFromList(Long mailingListId) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    Collection<EmailDTO> emailDTOs = null;
    try {
      trans = sess.beginTransaction();
      MailingList list = (MailingList)sess.load(
        MailingList.class, mailingListId);
      Collection<Email> emails = list.getEmails();
      emailDTOs = new ArrayList<EmailDTO>(
        emails != null ? emails.size() : 0);
      if(emails != null) {
        for(Email email : emails) {
          emailDTOs.add(createEmailDTO(email));
        }
      }
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return emailDTOs;
  }

  public Collection<Topic> getTopics(Long listId, String tagFace) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    Collection<Topic> topics = null;
    try {
      trans = sess.beginTransaction();
      MailingList list = (MailingList)sess.load(
        MailingList.class, listId);
      Collection<Email> topicEmails = null;

      if(tagFace.length() == 0) {
        // no tag specified, get all
        topicEmails = new ArrayList(
          sess.createFilter(list.getEmails(),
          "where this.references = ''").list());
      } else {
        EmailTag tag = getEmailTagByFace(tagFace);
        topicEmails = new ArrayList<Email>(
          sess.createFilter(list.getEmails(),
          "where this.references = '' and :tag in
          elements(this.tags)").setParameter("tag", tag).list());
      }
      topics = new ArrayList<Topic>(
        topicEmails != null ? topicEmails.size() : 0);
      if(topicEmails != null) {
        for(Email email : topicEmails) {
          // get reply count and last update time for each topic
          Topic topic = new Topic(createEmailDTO(email));
          Iterator results = sess.createQuery(
            "select count(em), max(em.dateTime) from Email as em
            where locate(:msg_id, em.references) > 0")
            .setString("msg_id",
            email.getMessageId()).list().iterator();
          if(results.hasNext()) {
            Object[] row = (Object[])results.next();
            Long replyCount = (Long)row[0];
            Date lastUpdateTime = (Date)row[1];
            if(replyCount > 0l) {
              topic.setReplyCount(replyCount);
              topic.setLastUpdateTime(lastUpdateTime);
            }
          }
          topics.add(topic);
        }
      }
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return topics;
  }

  public Collection<EmailDTO> getThreadEmails(String topicMessageId,
    Long topicId) {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    Collection<EmailDTO> threadEmailDTOs = null;
    try {
      trans = sess.beginTransaction();
      Query qry = sess.createQuery(
        "from Email as em where locate(?, em.references) > 0
        order by em.dateTime");
      qry.setString(0, topicMessageId);
      Collection<Email> threadEmails = qry.list();
      threadEmailDTOs = new ArrayList<EmailDTO>(
        threadEmails != null ? threadEmails.size() : 0);
      // fetch topic
      Email topicEmail = (Email)sess.load(Email.class, topicId);
      threadEmailDTOs.add(createEmailDTO(topicEmail));
      if(threadEmails != null) {
        for(Email email : threadEmails) {
          threadEmailDTOs.add(createEmailDTO(email));
        }
      }
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }

    return threadEmailDTOs;
  }

  // return all tags, for showing on front page to filter emails
  public List<EmailTagDTO> getAllTags() {
    Session sess = HibernateUtil.getSessionFactory().openSession();
    Transaction trans = null;
    List<EmailTagDTO> tagDTOs = null;
    try {
      trans = sess.beginTransaction();
      Query query = sess.createQuery(
        "from EmailTag where tagFace != '' order by tagFace");
      List<EmailTag> tags = new ArrayList<EmailTag>(query.list());
      tagDTOs = new ArrayList<EmailTagDTO>(
        tags != null ? tags.size() : 0);
      if(tags != null) {
        for(EmailTag tag : tags) {
          tagDTOs.add(createEmailTagDTO(tag));
        }
      }
      trans.commit();
    } catch(HibernateException e) {
      trans.rollback();
      e.printStackTrace();
    } finally {
      sess.close();
    }
    return tagDTOs;
  }

  // Hibernate objects to DTO transformation
  private MailingListDTO createMailingListDTO(MailingList mailingList) {
    return new MailingListDTO(mailingList.getId(),
      mailingList.getTitle(), mailingList.getEmail());
  }

  private EmailDTO createEmailDTO(Email email) {
    Set<EmailTag> tags = email.getTags();
    Set<EmailTagDTO> tagDTOs = new HashSet<EmailTagDTO>(
      tags != null ? tags.size() : 0);
    if(tags != null) {
      for(EmailTag tag : tags) {
        tagDTOs.add(createEmailTagDTO(tag));
      }
    }
    EmailBodyDTO bodyDTO = createEmailBodyDTO(email.getEmailBody());
    return new EmailDTO(email.getId(), email.getMessageId(), email.getReferences(),
        email.getSender(), email.getTo(), email.getCc(),
        email.getBcc(), email.getReplyTo(), email.getSubject(),
        email.getDateTime(), bodyDTO, tagDTOs);
  }

  private EmailTagDTO createEmailTagDTO(EmailTag tag) {
    return new EmailTagDTO(tag.getTagId(), tag.getTagFace(),
      tag.getTagName());
  }
  private EmailBodyDTO createEmailBodyDTO(EmailBody body) {
    EmailBodyDTO bodyDTO = new EmailBodyDTO();
    bodyDTO.setBodyId(body.getBodyId());
    bodyDTO.setEmailText(body.getEmailText());
    bodyDTO.setEmailHtml(body.getEmailHtml());
    return bodyDTO;
  }

  private BugInfoDTO createBugInfoDTO(BugInfo bugInfo) {
    // TODO
    return null;
  }

  private PatchInfoDTO createPatchInfoDTO(PatchInfo patchInfo) {
    // TODO
    return null;
  }

}
```

7.2.3 EmailTagDataProvider.java

```
package de.fau.cs.osr.etikett.server.titleparser;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class EmailTagDataProvider {

  private String cacheFileContent;

  public String getCacheFileContents() {
    return this.cacheFileContent;
  }

  public EmailTagDataProvider() {
    // check if cache exists
```

```
File cacheFile = new File("/tmp/etikett/email-tags.json");
boolean cacheExists = cacheFile.exists();
if(cacheExists) {
  // read cache file into a string
  try {
    BufferedReader in = new BufferedReader(
      new FileReader(cacheFile));
    StringBuilder sb = new StringBuilder();
    String line;
    while((line = in.readLine()) != null) {
      sb.append(line);
    }
    in.close();

    this.cacheFileContent = sb.toString();
  } catch(IOException e) {
    e.printStackTrace();
  }
} else {
  // check if dir exist
  File cacheFolder = new File("/tmp/etikett");
  if(!cacheFolder.exists()) {
    cacheFolder.mkdir();
  }

  // download json file for caching
  URL url;
  InputStream is = null;
  BufferedReader br;
  String line;
  BufferedWriter bw = null;
  try {
    url = new URL("http://email-tags.appspot.com/json");
    is = url.openStream();
    br = new BufferedReader(new InputStreamReader(is));
    StringBuilder sb = new StringBuilder();
    while((line = br.readLine()) != null) {
      sb.append(line);
    }
    this.cacheFileContent = sb.toString();

    // save json file
    bw = new BufferedWriter(new FileWriter(cacheFile));
    bw.write(this.cacheFileContent);
  } catch(MalformedURLException e) {
    e.printStackTrace();
  } catch(IOException e) {
    e.printStackTrace();
  } finally {
    try {
          is.close();
          bw.close();
        } catch(IOException e) {
          e.printStackTrace();
        }
      }
    }
  }

  public ArrayList<EmailTagEx> getEmailTags() {
    // create json object
    ArrayList<EmailTagEx> emailTags = new ArrayList<EmailTagEx>();
    try {
      JSONArray tags = new JSONArray(this.cacheFileContent);
      for(int i = 0; i < tags.length(); i++) {
        JSONObject tag = tags.getJSONObject(i);
        EmailTagEx emailTag = new EmailTagEx();
        emailTag.setName(tag.getString("name"));
        emailTag.setDescription(tag.getString("description"));
        emailTag.setTag(tag.getString("tag"));
        JSONArray tagKeywords = tag.getJSONArray("keywords");
        for(int j = 0; j < tagKeywords.length(); j++) {
          emailTag.getKeywords().add(tagKeywords.getString(j));
        }
        emailTags.add(emailTag);
      }
    } catch (JSONException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    return emailTags;
  }

}
```

7.2.4 EmailTitleParsingSimple.java

```
package de.fau.cs.osr.etikett.server.titleparser;

import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EmailTitleParsingSimple implements EmailTitleParsingStrategy {

  /*
   * (non-Javadoc)
   * @see de.fau.cs.osr.EmailTitleParsingStrategy#parse(java.lang.String)
   *
   * Simple tag parsing strategy:
```

```
 * 1. locate brackets
 * 2. split by " ", iterate them for keyword matches
 * 3. extra info for [bug] and [patch] TODO
 *
 */

private ArrayList<EmailTagEx> availEmailTags;
private ArrayList<EmailTagEx> foundEmailTags;

public EmailTitleParsingSimple() {
  this.availEmailTags = (new EmailTagDataProvider()).getEmailTags();
  this.foundEmailTags = new ArrayList<EmailTagEx>(0);
}

private void checkBracket(String bracket) {
  String[] bracketSegs = bracket.split(" ");

  for(String seg : bracketSegs) {
    for(EmailTagEx tag : this.availEmailTags) {
      for(String keyword : tag.getKeywords()) {
        if(seg.equalsIgnoreCase(keyword)) {
          this.foundEmailTags.add(tag);
          break; // just one keyword match is enough!
        }
      }
    }
  }
}

@Override
public EmailTitleParsingResult parse(String emailTitle) {
  this.foundEmailTags.clear();

  EmailTitleParsingResult result = new EmailTitleParsingResult();
  result.setEmailTitle(emailTitle);

  String bracketPattern = "\\[(.*?)\\]";

  Pattern pattern = Pattern.compile(bracketPattern,
    Pattern.CASE_INSENSITIVE);
  Matcher matcher = pattern.matcher(emailTitle);

  while(matcher.find()) {
    String bracketContent = matcher.group(1);
    checkBracket(bracketContent);
  }

  result.setMatchedTags(foundEmailTags);

  return result;
  }

}
```

7.3 Client Side Code

7.3.1 Etikett.java

```
package de.fau.cs.osr.etikett.client;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Set;

import com.google.gwt.cell.client.AbstractCell;
import com.google.gwt.cell.client.ClickableTextCell;
import com.google.gwt.cell.client.DateCell;
import com.google.gwt.cell.client.FieldUpdater;
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.i18n.client.DateTimeFormat;
import com.google.gwt.i18n.client.DateTimeFormat.PredefinedFormat;
import com.google.gwt.safehtml.shared.SafeHtmlBuilder;
import com.google.gwt.safehtml.shared.SafeHtmlUtils;
import com.google.gwt.user.cellview.client.CellTable;
import com.google.gwt.user.cellview.client.Column;
import com.google.gwt.user.cellview.client.SimplePager;
import com.google.gwt.user.cellview.client.TextColumn;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Anchor;
import com.google.gwt.user.client.ui.RootLayoutPanel;
import com.google.gwt.view.client.ListDataProvider;

import de.fau.cs.osr.etikett.client.dto.EmailDTO;
import de.fau.cs.osr.etikett.client.dto.EmailTagDTO;
import de.fau.cs.osr.etikett.client.dto.MailingListDTO;
import de.fau.cs.osr.etikett.client.dto.Topic;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class Etikett implements EntryPoint {

  private EtikettServiceAsync etikettService = GWT
```

```
      .create(EtikettService.class);

  private final ListAdminWidget listAdminWidget = new ListAdminWidget();
  private final MainLayout mainLayout = new MainLayout();

  private CellTable<MailingListDTO> tblMailingLists;
  private CellTable<Topic> tblTopics;
  private ListDataProvider<Topic> topicDataProvider;
  private SimplePager pgrTopicList;
  private CellTable<EmailDTO> tblThreadEmails;

  private Long currentListId;

  // custom cell for displaying thread emails
  static class ThreadEmailCell extends AbstractCell<EmailDTO> {

    @Override
    public void render(EmailDTO value, Object key, SafeHtmlBuilder sb) {
      if(value == null) return;

      sb.appendHtmlConstant("<div class='my-threademail'>");

      sb.appendHtmlConstant("<div class='my-threademail-subject'><h3>");
      sb.append(SafeHtmlUtils.fromString(value.getSubject()));
      sb.appendHtmlConstant("</h3></div>");

      sb.appendHtmlConstant("<div class='my-threademail-info'>");
      sb.append(SafeHtmlUtils.fromString(
        value.getSender() + " / " + DateTimeFormat.getFormat(
        PredefinedFormat.RFC_2822).format(value.getDateTime())));
      sb.appendHtmlConstant("</div>");

      sb.appendHtmlConstant("<div class='my-threademail-body'>");
      sb.appendEscapedLines(value.getEmailBody().getEmailText());
      sb.appendHtmlConstant("</div>");

      sb.appendHtmlConstant("</div>");
    }

  }

  /**
   * This is the entry point method.
   */
  public void onModuleLoad() {
    // first run will fill the email tags to database
    initTags();

    this.currentListId = 0l;
```

```
RootLayoutPanel.get().add(mainLayout);
mainLayout.pnlAdmin.add(listAdminWidget);

// events for the (temp-)admin panel
listAdminWidget.btnSubmit.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    String title = listAdminWidget.txtTitle.getText();
    String email = listAdminWidget.txtEmail.getText();
    if(title.length() == 0 || email.length() == 0) {
      Window.alert("Fields should not be empty!");
      return;
    } else {
      addList(title, email);
    }
  }
});

listAdminWidget.btnFetchEmails.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    testFetchEmails();
  }
});

// build table for mailing lists
tblMailingLists = new CellTable<MailingListDTO>();
Column<MailingListDTO, String> colListTitle = new
  Column<MailingListDTO, String>(new ClickableTextCell()) {
  @Override
  public String getValue(MailingListDTO object) {
    return object.getTitle() + " / " + object.getEmail();
  }
};
colListTitle.setFieldUpdater(new FieldUpdater<MailingListDTO,
  String>() {
  @Override
  public void update(int index, MailingListDTO object, String value) {
    refreshEmailsTable(object.getId());
  }
});

tblMailingLists.addColumn(colListTitle, "Mailing Lists");

mainLayout.pnlNav.add(tblMailingLists);

// build table for Topics
tblTopics = new CellTable<Topic>();
TextColumn<Topic> colTopicFrom = new TextColumn<Topic>() {
  @Override
```

```
    public String getValue(Topic object) {
      return object.getEmail().getSender();
    }
  };
  Column<Topic, String> colTopicSubject = new Column<Topic, String>(
    new ClickableTextCell()) {
    @Override
    public String getValue(Topic object) {
      return object.getEmail().getSubject();
    }
  };
  colTopicSubject.setFieldUpdater(new FieldUpdater<Topic, String>() {
    @Override
    public void update(int index, Topic object, String value) {
      showThread(object.getEmail().getMessageId(),
        object.getEmail().getId());
    }
  });
  TextColumn<Topic> colTopicTags = new TextColumn<Topic>() {
    @Override
    public String getValue(Topic object) {
      Set<EmailTagDTO> tags = object.getEmail().getTags();
      if(tags.size() == 0) {
        return " ";
      } else {
        StringBuilder sb = new StringBuilder();
        for(EmailTagDTO tag : tags) {
          sb.append(tag.getTagFace() + " ");
        }
        return sb.toString();
      }
    }
  };
  TextColumn<Topic> colTopicRepUpdate = new TextColumn<Topic>() {
    @Override
    public String getValue(Topic object) {
      return Long.toString(object.getReplyCount()) + " / " +
        DateTimeFormat.getFormat(PredefinedFormat.DATE_TIME_SHORT)
        .format(object.getLastUpdateTime());
    }
  };
  tblTopics.addColumn(colTopicFrom, "Starter");
  tblTopics.addColumn(colTopicTags);
  tblTopics.addColumn(colTopicSubject, "Subject");
  tblTopics.addColumn(colTopicRepUpdate, "Reply / Date");

  tblTopics.setPageSize(15); // how many topics per page?

  // data provider and pager
  topicDataProvider = new ListDataProvider<Topic>();
```

```
    topicDataProvider.addDataDisplay(tblTopics);
    pgrTopicList = new SimplePager();
    pgrTopicList.setDisplay(tblTopics);

    mainLayout.pnlTopicList.add(tblTopics);
    mainLayout.pnlPager.add(pgrTopicList);

    // email thread
    tblThreadEmails = new CellTable<EmailDTO>();
    Column<EmailDTO, EmailDTO> colThreadEmail = new Column<EmailDTO,
      EmailDTO>(new ThreadEmailCell()) {

      @Override
      public EmailDTO getValue(EmailDTO object) {
        return object;
      }
    };
    tblThreadEmails.addColumn(colThreadEmail, "Thread");
    mainLayout.pnlThread.add(tblThreadEmails);

    // display mailing lists at startup
    refreshMailingListTable();

    mainLayout.pnlTagList.setVisible(false);
  }

  private void listAllTags() {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    etikettService.getAllTags(new AsyncCallback<List<EmailTagDTO>>() {
      @Override
      public void onSuccess(List<EmailTagDTO> result) {
        for(EmailTagDTO tagDTO : result) {
          Anchor tagAnchor = new Anchor(tagDTO.getTagFace());
          final String tagFace = tagDTO.getTagFace();
          tagAnchor.addClickHandler(new ClickHandler() {

            @Override
            public void onClick(ClickEvent event) {
              refreshEmailsTableByTag(tagFace);
            }
          });
          mainLayout.pnlTagList.add(tagAnchor);
        }
      }

      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }
    });
  }

  private void showThread(String topicMessageId, Long topicId) {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    etikettService.getThreadEmails(topicMessageId, topicId,
      new AsyncCallback<Collection<EmailDTO>>() {
      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }

      @Override
      public void onSuccess(Collection<EmailDTO> result) {
        tblThreadEmails.setRowCount(result.size());
        tblThreadEmails.setRowData(0, new ArrayList<EmailDTO>(result));
      }
    });
  }

  private void refreshEmailsTable(Long listId) {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    currentListId = listId;

    // get all topics
    etikettService.getTopicsFromList(listId, "",
      new AsyncCallback<Collection<Topic>>() {
      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }

      @Override
      public void onSuccess(Collection<Topic> result) {
        mainLayout.pnlTagList.setVisible(true);
        tblTopics.setRowCount(result.size());
        topicDataProvider.setList(new ArrayList<Topic>(result));
      }
    });
  }

  private void refreshEmailsTableByTag(String tagFace) {
    if(currentListId == 0l) return;

    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    etikettService.getTopicsFromList(currentListId, tagFace,
      new AsyncCallback<Collection<Topic>>() {
      @Override
      public void onSuccess(Collection<Topic> result) {
        mainLayout.pnlTagList.setVisible(true);
        tblTopics.setRowCount(result.size());
        topicDataProvider.setList(new ArrayList<Topic>(result));
      }

      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }
    });
  }

  private void addList(String title, String email) {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }
    etikettService.addMailingList(title, email, new AsyncCallback<Void>() {
      @Override
      public void onSuccess(Void result) {
        refreshMailingListTable();
        listAdminWidget.txtTitle.setText("");
        listAdminWidget.txtEmail.setText("");
        listAdminWidget.dlgAddList.hide();
      }

      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }
    });
  }

  private void testFetchEmails() {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    listAdminWidget.imgAjaxLoading.setVisible(true);
    etikettService.fetchAllEmails(new AsyncCallback<Void>() {
      @Override
      public void onSuccess(Void result) {
        Window.alert("Email fetching done!");
        listAdminWidget.imgAjaxLoading.setVisible(false);
      }

      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
        listAdminWidget.imgAjaxLoading.setVisible(false);
      }
    });
  }

  private void refreshMailingListTable() {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    etikettService.getMailingLists(
      new AsyncCallback<List<MailingListDTO>>() {

      @Override
      public void onSuccess(List<MailingListDTO> result) {
        tblMailingLists.setRowCount(result.size(), true);
        tblMailingLists.setRowData(0, result);
      }

      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }
    });

  }

  private void initTags() {
    if(etikettService == null) {
      etikettService = GWT.create(EtikettService.class);
    }

    etikettService.initTags(new AsyncCallback<Void>() {

      @Override
      public void onFailure(Throwable caught) {
        Window.alert(caught.getMessage());
      }

      @Override
      public void onSuccess(Void result) {
        // list all tags for filtering
```

```
            listAllTags();
        }

    });
  }
}
```

List of Figures

List of Tables

Bibliography

[1] Anupriya Ankolekar, James D. Herbsleb, and Katia Sycara. Addressing Challenges to Open Source Collaboration With the Semantic Web. 2003.

[2] Jeff Breidenbach. The Mail Archive. `http://www.mail-archive.com/`.

[3] Sumit Chandel. Using GWT with Hibernate. `http://code.google.com/webtoolkit/articles/using_gwt_with_hibernate.html`, 2009.

[4] Mark Logic Corporation. MarkMail. `http://markmail.org/`.

[5] Jan Dietz. Understanding and Modelling Business Processes with DEMO. 1728:767–767, 1999.

[6] Karl Fogel. *Producing Open Source Software*. O'Reilly Media, 2005.

[7] Göran Goldkuhl. Conversational Analysis as a Theoretical Foundation for Language Action Approaches? 2003.

[8] Google. Google App Engine. `http://code.google.com/appengine/`.

[9] Google. Google Web Toolkit. `http://code.google.com/webtoolkit/`.

[10] Network Working Group. RFC 2822 - Internet Message Format. `http://tools.ietf.org/html/rfc2822`.

[11] Network Working Group. RFC 4155 - The application/mbox Media Type. `http://tools.ietf.org/html/rfc4155`.

[12] PostgreSQL Global Development Group. Postgresql. `http://www.postgresql.org/about/`.

[13] Red Hat Inc. Hibernate Community Documentation. `http://www.hibernate.org/docs`.

[14] Lars Magne Ingebrigtsen. Gmane. `http://gmane.org/`.

[15] Open Source Initiative. opensource.org. `http://www.opensource.org/`.

[16] Marja-Riitta Koivunen and Ralph Swick. Metadata Based Annotation Infrastructure offers Flexibility and Extensibility for Collaborative Application and Beyond.

[17] Hank Leininger. Marc: Mailing list archives. `http://marc.info/`.

[18] Greg Madey, Vincent Freeh, and Renee Tynan. The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory. 2002.

[19] David Mertz. *Text Processing in Python*. Addison-Wesley Professional, 2003.

[20] Oracle Technology Network. Javamail. `http://www.oracle.com/technetwork/java/javamail/index.html`.

[21] Masao Ohira, Kiwako Koyama, Akinori Ihara, Shinsuke Matsumoto, Yasutaka Kamei, and Ken ichi Matsumoto. A Time-Lag Analysis towards Improving the Efficiency of Communications among OSS Developers. 2009.

[22] Ohloh. Open Source Projects. `http://www.ohloh.net/p`. Stand: Jan. 2011.

[23] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly Media, 2001.

[24] Gregorio Robles. A Software Engineering Approach to Libre Software. *Open Source Jahrbuch 2004*, 2004.

[25] Ran Tang, Ahmed E. Hassan, and Ying Zhou. A Case Study on the Impact of Global Participation on Mailing Lists Communications of Open Source Projects. 2009.

[26] Sergio L. Toral, Rocío Martínez Torres, and Federico Barrero. Modelling Mailing List Behaviour in Open Source Projects: the Case of ARM Embedded Linux. *Journal of Universal Computer Science*, 15(3):648–664, 2009.

[27] Douglas P. Twitchell, Mark Adkins, Jay F. Nunamaker Jr., and Judee K. Burgoon. Using Speech Act Theory to Model Conversations for Automated Classification and Retrieval. 2004.

[28] Wikipedia. Electronic mailing list. `http://en.wikipedia.org/wiki/Electronic_mailing_list`.

[29] Wikipedia. Folksonomy. `http://en.wikipedia.org/wiki/Folksonomy`.

[30] Wikipedia. Open-source software. `http://en.wikipedia.org/wiki/Open-source_software`.

[31] Wikipedia. Open source software development. `http://en.wikipedia.org/wiki/Open_source_software_development`.

[32] Wikipedia. Speech act. `http://en.wikipedia.org/wiki/Speech_act_theory`.

[33] Wikipedia. Strategy pattern. http://en.wikipedia.org/wiki/Strategy_pattern.

[34] Wikipedia. Tag (metadata). http://en.wikipedia.org/wiki/Tag_%28metadata%29.

[35] Terry Winograd. A Language/Action Perspective on the Design of Cooperative Work. *Human-Computer Interaction*, 3:3–30, 1987.

[36] Yutaka Yamauchi, Makoto Yokozawa, Takeshi Shinohara, and Toru Ishida. Collaboration with lean media: how open-source software succeeds. pages 329–338, 2000.