Extracting Source Code from E-Mails

Alberto Bacchelli, Marco D'Ambros, Michele Lanza
REVEAL@ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—E-mails, used by developers and system users to communicate over a broad range of topics, offer a valuable source of information. If archived, e-mails can be mined to support program comprehension activities and to provide views of a software system that are alternative and complementary to those offered by the source code.**

However, e-mails are written in natural language, and therefore contain noise that makes it difficult to retrieve the important data. Thus, before conducting an effective system analysis and extracting data for program comprehension, it is necessary to select the relevant messages, and to expose only the meaningful information.

In this work we focus both on classifying e-mails that hold fragments of the source code of a system, and on extracting the source code pieces inside the e-mail. We devised and analyzed a number of lightweight techniques to accomplish these tasks. To assess the validity of our techniques, we manually inspected and annotated a statistically significant number of e-mails from five unrelated open source software systems written in Java. With such a benchmark in place, we measured the effectiveness of each technique in terms of precision and recall.

I. Introduction

The evolution of a software system involves the creation and the growth of a number of different artifacts that revolve around the source code and the development process, such as system documentation, design documents, bug reports, e-mail messages, wikis, forums, chat logs *etc.* The information stored in these artifacts is valuable for various tasks related to software evolution analysis and program comprehension, as different types of artifacts emphasize different aspects of the system's evolution. Even though some artifacts are logically related, *e.g.,* source code and documentation, they tend to drift apart over time increasing the inconsistencies between the information they contain [18].

LaToza *et al.* conducted a study at Microsoft, finding that developers cannot find up-to-date models of the system in artifacts that form the official documentation of the software they build [14]. Programmers who need to know the design rationale behind an implementation (which is the most common information need for a developer [13]) have to directly communicate with other developers. This induces developers to form their own implicit mental models of the system, which do not end up stored in appropriate documents. Indeed, developers prefer face-to-face meetings to communicate on a system [13], but when such meetings cannot be held (*e.g.,* in distributed development teams), they favor e-mail communication. Consequently, e-mail archives offer an up-to-date and significant source of information about a system. E-mails are used to discuss issues ranging from low-level decisions (*e.g.,* implementation strategies, bug fixing) up to high-level considerations (*e.g.,* design rationale, future planning). Our goal is to use this kind of artifacts to enhance program comprehension and analysis.

We already devised techniques to recover the traceability links between source code entities and e-mails [4]. However, such ties can only be retrieved by knowing the entities of the system in advance, *i.e.,* by having the model of the system generated from the source code. In this paper we tackle a different problem: We want to extract source code fragments from e-mail messages. To do this, we first need to select e-mails that contain source code fragments, and then we extract such fragments from the content in which they are enclosed. Separating source code from natural language in e-mail messages brings several benefits: The access to structured data (1) facilitates the reconstruction of the system model and (2) can be employed to improve e-mail recommender systems based on machine learning (*e.g.,* [12]), and (3) source code fragments provide examples on the usage of entities, complementing the information expressed in natural language.

However, reliably extracting system models from e-mail archives is a non-trivial task: Not only are e-mails written in free-form natural language, but they can also contain noise (*e.g.,* discussions not related to the development) that makes it difficult to retrieve the relevant data. As a consequence, the extraction of the information from e-mail archives requires more refinements: (1) E-mails not containing source code snippets must be filtered out, (2) lines that include source code must be detected in the remaining e-mails, and (3) the specific source code fragments must be separated from natural language. For this reason, we conduct our analysis at three different levels, and evaluate the effectiveness of our methods for each of them.

To evaluate the accuracy of our techniques, we manually build a statistically significant benchmark taking sample e-mails from five unrelated open source Java software systems.

Structure of the paper. In Section II we discuss the related work. In Section III we explain how we set up our benchmark and present the infrastructure we devised to support its creation. In Section IV we detail the different approaches we tested. In Section V we discuss how they perform with respect to our benchmark. Section VI concludes the work by summarizing our findings and discussing possible extensions of our approach.

978-0-7695-4113-6/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPC.2010.47

IEEE computer society

II. Related work

Our research is influenced by the seminal work of Murphy and Notkin [17]. They proposed a lightweight lexical approach, based on regular expressions, to extract models of a software system from different software artifacts. Software engineers can obtain consistent models from any kind of textual artifacts concerning software with such approach. The engineer must follow three steps: (1) define patterns (using regular expressions) that describe source code constructs of interest in a software artifact, *e.g.,* function calls or definitions; (2) establish the operations to be executed whenever a pattern is matched in an artifact being scanned; and (3) implement post-processing operations for combining information extracted from individual files into a global model. Although this approach is lightweight, flexible, and tolerant, the first step is non-trivial, especially when dealing with unstructured artifacts written in natural language, such as e-mails. Choosing the best approach to expose source code fragments of interest requires an accurate analysis of the advantages and drawbacks that the different regular expressions offer. Practitioners could find it difficult to perform this task through the iterative trial and error process proposed by Murphy and Notkin.

Bettenburg *et al.* proposed four filtering techniques to extract patches, stack traces, source code snippets, and enumerations from the textual descriptions that accompany bug reports [7]. The authors report results only on the effectiveness of their techniques in *differentiating* documents that contain source code snippets from those that do not contain source code. In this task, using island parsing [16], they reached almost perfect results, *i.e.,* a precision value of 0.98 and recall of 0.99. Since using a parser requires a high computational effort and scaling up to archives containing tens of thousands of documents can be problematic, we propose fast and lightweight techniques, based on regular expressions and pattern matching. In addition, we consider development mailing list archives as our natural language documents, which are more prone to noise not related to system development, if compared to bug reports. Finally, we evaluate the effectiveness of our techniques in detecting not only e-mails that contain source code, but also lines, and in extracting the fragments.

Bird *et al.* proposed an approach to measure the acceptance rate of patches submitted via e-mail in open source software projects [8]. They have been able to classify e-mails containing source code patches. However, since e-mail classification was not the focus of the work, the authors provided little information about their extraction techniques and no details on the benchmark they used to assess their effectiveness.

Dekhtyar *et al.* discussed the opportunities and challenges for text mining applied to software artifacts written in natural language [10].

A number of approaches in the information retrieval field are related to our work. Tang *et al.* addressed the issue of cleaning the e-mail data for subsequent text mining [20]. They propose a cascaded approach to clean e-mails in four passes: (1) non-text filtering, (2) paragraph, (3) sentence, and (4) word normalization. In the first pass, what they consider non-text are actually e-mail headers, signatures, and source code snippets. They randomly chose a total of 5,459 emails from 14 unrelated sources (*e.g.,* development newsgroups at Google) and created 14 data sets in which they manually labeled headers, signatures, quotations, and program codes. They used an approach based on Support Vector Machines (SVM) to detect source code fragments. They evaluated the effectiveness of the method at line level, and achieved reasonable results, *i.e.,* 0.93 in precision, and 0.72 in recall. These findings are promising, and in the data mining field, much research is devoted to extract information that has specific patterns using methods based on probabilistic and machine learning models (*e.g.,* Maximum Entropy Models [6] or Hidden Markov Models [5]). However, such approaches require extensive expertise and effort, also for data collection, that could discourage an utilization by practitioners.

For this reason, we devised lightweight approaches based on simple methods to extract source code fragments from e-mails. Our approaches use lightweight and easy to implement techniques based on regular expressions that exploit intrinsic characteristics of source code elements.

III. Evaluation Benchmark

While surveying the related work reported in the previous section, we noted that there is a gap between the validation approaches used by the software engineering researchers and the ones used in information retrieval: For instance, the work by Tang *et al.* is supported by an extensive and statistically significant benchmark against which their techniques were evaluated. This does not hold for the software engineering approaches.

This faulty trend in software engineering was first reported by Sim *et al.* [19]. They stressed the importance of widely used and reliable benchmarks to assess the quality of research findings, and challenged the software engineering research community to define appropriate benchmarks. We share the concern raised by Sim *et al.*. Since no previous benchmark has been devised for the issues we tackle with our work, besides presenting and discussing a set of techniques, we also create a statistically significant, and publicly available benchmark, against which to verify them.

Our benchmark for extracting source code fragments can be employed for further analysis of different techniques. The obtained results can be compared to show the strengths and drawbacks of several approaches on the same data set. We designed this benchmark to require no special infrastructure to be used and to be easily extensible with additional data.

System URL	Description	Mailing Lists				Sample E-mails	
		Name	Inception	E-Mails	Filtered	Number	with Code
ArgoUML argouml.tigris.org	A UML modeling tool developed over the course of approximately 9 years.	org.tigris.argouml.dev	Jan 2000	24,876	24,876	379	48
Freenet freenetproject.org	A peer-to-peer software for anonymous file sharing, and for browsing and publishing "freesites" (web sites accessible only through Freenet).	org.freenetproject.devl	Apr 2000	22,095	22,095	378	35
JMeter jakarta.apache.org/jmeter	A desktop application designed for load and stress testing of Web Applications. The first release was done in 1999.	org.apache.jakarta.jmeter-dev	Feb 2001	21,637	9,810	370	105
Mina org.apache.mina.dev	A realtime 3D engine for Flash, written in ActionScript, an object-oriented programming language compliant with the ECMAScript Language Specification.	org.apache.mina.dev	May 2007	18,565	12,869	374	101
OpenJPA org.apache.openjpa.dev	A network application framework which provides an abstract event-driven asynchronous API over various transports such as TCP/IP and UDP/IP via Java NIO.	org.apache.openjpa.dev	Oct 2006	14,992	6,328	363	97

Table I
THE SOFTWARE SYSTEMS CONSIDERED FOR THE BENCHMARK

A. Subjects of the experiment

Table I shows the five open source Java projects we considered to create our benchmark. We selected unrelated software systems emerging from the context of different free software communities, *i.e.,* Apache, ArgoUML, and Freenet. The development environment, the usage of the mailing lists, and the development paradigms are all likely to differ among the systems, providing a good test for the adaptability of our lightweight approaches to a wide variety of systems, and helping to assess their effectiveness.

Even though all the systems offer multiple mailing lists that can be analyzed, we focus on the development mailing lists, as they provide the highest density of information related to software development. We excluded from our benchmark messages automatically generated by the bug tracking system and the revision control system, since they contain only a reduced amount of natural language text. In the exceptional case of JMeter, we decided to also include part of the messages generated by the revision control system, to see their effect in the outcome of our experiments.

The section "Mailing Lists" of Table I provides details on the mailing list name, the total number of e-mails imported, and the number of e-mails in the set after filtering out the automatically generated ones.

E-Mails sample set size: Since we could not afford to manually annotated the entire set of nearly 75,000 emails, we extracted a sample. Due to the lack of any knowledge about the considered mailing lists, we employ simple random sampling [21] to extract the e-mails to be included in our benchmark.

We used the following formula to determine the number n of e-mails that must be sampled from the populations [21]:

$$n = \frac{N \cdot \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}{(N-1)\,E^2 + \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}$$

In the formula, $\hat{p}$ is a value between 0 and 1 that represents the proportion of e-mails containing a source code fragment, while $\hat{q}$ is the proportion of e-mails not containing source code, *i.e.,* $\hat{q} = 1 - \hat{p}$. Since previous work dealing with the same data is unavailable, it is not possible to know *a-priori* the proportion ($\hat{p}$) of the e-mails including source code, thus we consider the worst scenario –in which $\hat{p}$ and $\hat{q}$ have the same value, so $\hat{p} \cdot \hat{q} = 0.25$. As we are dealing with small populations from a statistical point of view (*i.e.,* 75,000 e-mails), the formula also considers their size (N). We took the standard confidence level of 95%, and error (E) of 5%. This resulted in the values for the sample sets reported in Table I. If source code fragments are present in the $f\%$ of the sample set e-mails, we are 95% confident they will be present in the $f\% \pm 5\%$ of the population messages. This only validates the quality of this sample set as an exemplification of the populations, and is not related to the *precision* and *recall* values presented later.

B. Benchmark creation

To evaluate our source code extraction methods, we manually built the benchmark by reading all the e-mails and annotating them with the source code fragments they contain. We inspected the entire sample set, and then randomly selected and re-inspected 10% of the e-mails to verify the quality of the annotations. Figure 1 shows the excerpt of an e-mail that contains source code.

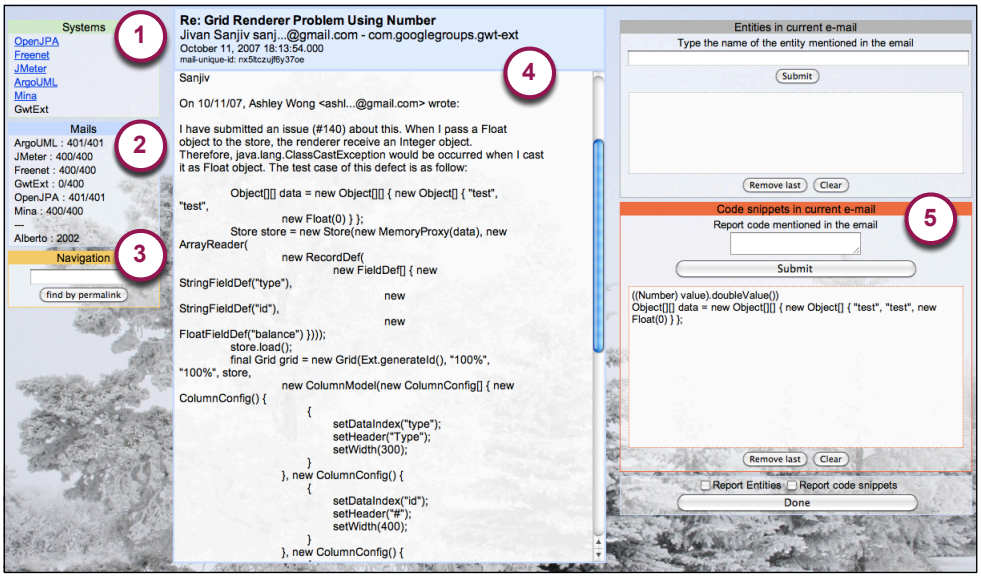

Figure 2. Web Application: E-Mail annotation of source code fragments

```
(1)  Hi Bob,

(2)  I have used swidget version add(LabelledLayout.getSeperator());  from
(3)  org.argouml.uml.ui.LabelledLayout earlier and it worked fine.
(4)  There is another class LabelledLayout in org.tigris.swidgets that has method
(5)  getSeperator(), but it also does not work.
(6)  However, after transfer to new ArgoUML version there was no error
(7)  in code, but elements were not arranged in two columns any more.
(8)  Here is the code I have implemented:

(9)  import javax.swing.ImageIcon;
[...]
(10) private static String orientation =
(11)  Configuration.getString(Configuration
(12)    .makeKey("layout", "tabdocumentation"));

(13) //make new column with LabelledLayout
(14)  add(LabelledLayout.getSeperator());

(15)  consequences = new UMLTextArea2(
(16)      new
(17) UMLModelElementValue(DepthsArgo.CONSEQUENCES_TAG));

(18) Could you help me, please?

(19) Thanks,
(20) Zoran
```

Figure 1. An e-mail excerpt containing source code fragments

The red parts in a monospaced font are those we would have marked as source fragments in our benchmark. We consider the method call in line 2 as valid (it is actual source code that can be part of a method), but we do not consider the class, package, and method names, written in blue, in lines from 3 to 5, as they are part of the discussion and are simply used as names. We exclude commented lines of code (Line 13).

We developed a custom web application, named *Miler*, to assist this particular task. A detailed description of Miler, the infrastructure we created to analyze mailing lists, is described in [2]. Here, we limit ourselves to summarize the most important aspects and the extensions we implemented for this benchmark[1].

Figure 2 shows the main page of the web application presented after the user log-in. It is composed by a number of panels: The *Systems* (1) shows the list of the software systems loaded in the application that must be analyzed for creating the benchmark; the *Mails* (2) keeps the user updated on the number of e-mails for each system that have been read over the total number of e-mails to analyze; the *Navigation* (3) lets the user retrieve any e-mail by its unique ID (displayed in the e-mail header); the *Main* (4) contains the e-mail header (*i.e.,* subject, author, date, mailing list) and its body. Sentences quoted from other e-mails are colored according to the quotation level: This increases the e-mail readability and the quality of the analysis; the *Code Snippets* (5) contains the annotated code fragments. The user can add detected source code fragments either by copying and paste them in the appropriate text area in the *Code Snippets* panel, or simply selecting them with the mouse and using a keyboard shortcut to add them in the annotations.

Despite the repetitiveness of the annotating task, we decided not to ease it adding features that could have influenced the results. For example, it would have been possible to highlight pieces of text containing Java keywords. However, this could have influenced the reader of the mail, who could have only skimmed the e-mail content in search of highlighted text, without paying attention to the meaning.

As we wanted our benchmark data to be easily accessible without the need of a specific framework, we stored the e-mails and the corresponding annotations in a PostgreSQL

[1] A demo and the entire dataset are available at http://miler.inf.usi.ch/code.

database, from which they can be retrieved and exported.

C. Evaluation

To evaluate the techniques to detect documents and lines containing source code fragments, we use the two well-known IR metrics *precision* ($Precision = \frac{|TP|}{|TP+FP|}$) and *recall* ($Recall = \frac{|TP|}{|TP+FN|}$) [15]. *True Positives (TP)* are elements that are correctly retrieved by the approach under analysis (*e.g.,* e-mails reported as containing source code fragments by the evaluated methods, that are also present in the benchmark). *False Positives (FP)* are elements that are wrongly retrieved by the approach under analysis (*i.e.,* lines in the e-mails reported as containing source code fragments by the methods, but not classified as containing source code in the benchmark). *False Negatives (FN)* are elements that are not retrieved by the approach under analysis (*i.e.,* e-mails with source code fragments only classified in the benchmark).

Considering an e-mail containing source code fragments among its lines, the union of TP and FN constitutes the set of lines labeled as containing code in the benchmark, while the union of TP and FP constitutes the set of lines labeled as containing code by our approach. In short, *precision* is the fraction of the retrieved lines that contain code, while *recall* is the fraction of the correct lines retrieved.

There is a number of e-mails in the benchmark that do not contain source code, therefore the union of TP and FN is empty. In these cases, the denominator in the recall formula is zero and the *recall* value cannot be calculated. Analogously, it is possible for our approach to find no source code fragments inside an e-mail: In this case, the *precision* value cannot be evaluated because the denominator in the corresponding formula is equal to zero. To overcome these issues, we calculate the average of TP, FP, and FN on the entire dataset, and we measure average *precision* and *recall* from those values. This solution also considers the impact of false positives on *precision*, when the set of benchmark lines is empty. Antoniol *et al.* used a similar approach [1].

Precision (P) and recall (R) trade off against one another: It is possible to link each mail with all classes, reaching a recall value of 1, at a cost of a very low precision. We also compute the *F-measure* –the weighted harmonic mean of precision and recall– with the following formula:

$$F = \frac{1}{\alpha \frac{1}{P} + (1-\alpha)\frac{1}{R}}, \beta^2 = \frac{1-\alpha}{\alpha} \longrightarrow F = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The weighting of precision and recall can be set through the value of β. We decided to emphasize neither the recall nor the precision, because our approaches can be used in many different situations, and it is up to the engineer to select the most appropriate one. Thus, we use a β value of 1 to obtain the default *balanced F-measure*.

To assess the effectiveness of our approach in extracting fragments, we computed the *Levenshtein distance* [15] line by line, between the text labeled as source code in the benchmark and the extracted fragments. This function, also called *edit distance function*, outputs the minimum number of changes needed to transform one string into another. The allowed transformation operations are deletion, insertion and substitution, and they are given the same unitary cost. As an example we consider the first source code fragment labeled in Figure 1: *add(LabelledLayout.getSeperator());*. If we evaluate Levenshtein distance between this fragment and another candidate we obtain 0 if the two strings are identical, otherwise a positive number that increases linearly with the number of operations required to transform the candidate in the correct string. For example, the distance between this string and *version add(LabelledLayout.getSeperator* is 12.

Other edit distance functions, such as the *Damerau-Levenshtein distance* [9] and the *Hamming distance* [11], are not appropriate for our task as they add unnecessary operations (such as transposition of two adjacent characters) or do not provide enough flexibility, *e.g.,* the Hamming distance only applies to strings of the same size.

IV. Experiments

We first tackle the problem of classifying e-mails that contain source code, then we move to the line level, and we conclude by showing how our methods can extract source code fragments. We begin from the techniques based on the simplest intuitions and we proceed to others based on more refined concepts.

A. Classification of e-mails including source code fragments

1) Special characters and keywords: Special characters (*e.g.,* semicolon, curly brackets) and reserved keywords (*e.g., public*, *static*) are fundamental tokens with special meanings to the programming language, and are necessary to write the source code of any Java system. Even though some keywords are common dictionary words (*e.g., for*) the presence of a high number of occurrences of keywords or special characters in a natural language text can be an evidence of an enclosed source code fragment. However, the length of the e-mail content could influence the necessary number of occurrences to distinguish e-mails with source code fragments from those without them. Thus, we devised two different approaches to classify e-mails:

1) According to the number of occurrences of either Java keywords or Java special characters, and
2) according to keywords or special characters frequencies.

If the number of occurrences of keywords, or their frequencies, is above a certain threshold, we classify an e-mail as containing source code fragments. We evaluate the results using several thresholds, to verify whether an optimal value can be defined.

Implementation. The occurrences of java keywords in an e-mail can be counted by dividing its content in words

through any space separator (*e.g.,* end of lines, blanks) and summing one for each keyword occurrence. For the special characters, we do not divide the text in words, but we only count the occurrences of characters. To compute the frequencies we divide the number of occurrences by the total number of words or characters.

Results. Figure 3 shows the F-Measure results among all the systems, when considering occurrences of keywords.

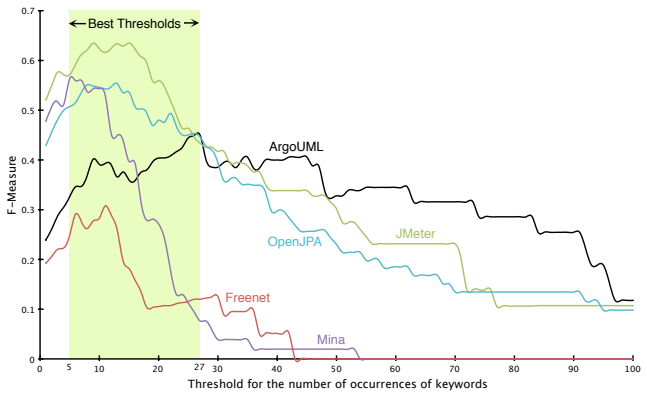

Figure 3. E-Mail classification on *occurrences* of keywords

The maximum values obtained vary significantly between systems: from 0.31 for Freenet up to 0.63 for JMeter. The best threshold spans between distant values: 5 and 27. As an example, Figure 4 details OpenJPA showing precision, recall, and F-Measure.

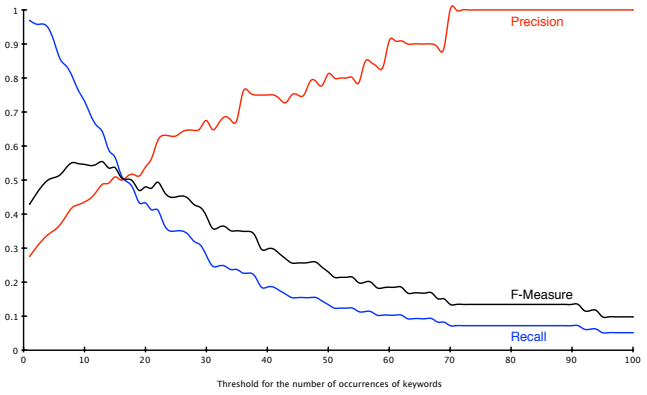

Figure 4. OpenJPA: E-Mail classification based on *occurrences* of keywords

Precision and recall trade off one against the other, but a very low value in one of them does not automatically guarantee a extremely high value for the other: Both of them can be low. The relevant feature of this simple approach is that, by varying the threshold, we can obtain either almost perfect recall or perfect precision, according to our needs.

We obtained similar but more consistent results using special characters occurrences as our discriminator: the best F-Measure values vary between 0.50, for Freenet, and 0.63, for JMeter. The best threshold spans between less distant values: 200 and 600 occurrences of special characters. The curves have trends equivalent to those in Figure 3 and Figure 4.

Figure 5 shows the F-Measure values when considering frequencies of characters.

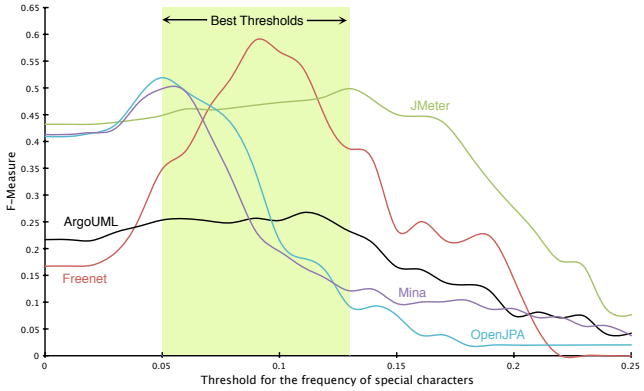

Figure 5. E-Mail classification based on *frequencies* of special characters

In this case, the best thresholds span between more distant values and the F-Measure values do not show significant improvements compared to the simple occurrence count. For the single system, the curves still show a trend near to those in Figure 4.

2) End of lines: Taking the lines of code in Figure 1 as an example, we note a peculiarity present in many programming languages (*e.g.,* Java, C, C#, Perl): The developer must end each statement with a semicolon. In the e-mail, we note that this happens in lines 9,12,14,17, and also in the code enclosed in line 2. Based on this intuition, this approach verifies whether the text contains lines whose last character is a *semicolon*. Since a natural language text line does not often end with such a character, it can be a significant hint on the presence of source code fragments. To build a more comprehensive approach, we also consider that a source code line can end with a curly bracket, mainly used to open or close a block. This approach can be parametrized on a threshold that represents the number of lines that must end with the special convention.

Implementation. Even though this approach still classifies e-mails and not lines of code, we consider the presence of peculiar lines in the text. The implementation consists in analyzing the e-mail content line by line and verifying if the last character is a semicolon or a curly bracket.

Results. Figure 6 shows the F-Measure for each system: the best values are significantly better than the previous approach: They vary from 0.74 to 0.92. Moreover, the best threshold is a single value (*i.e.,* one) that is the same for all the systems. Two lines ending with a semicolon are always

the best indicator of the presence of source code fragments with this approach.

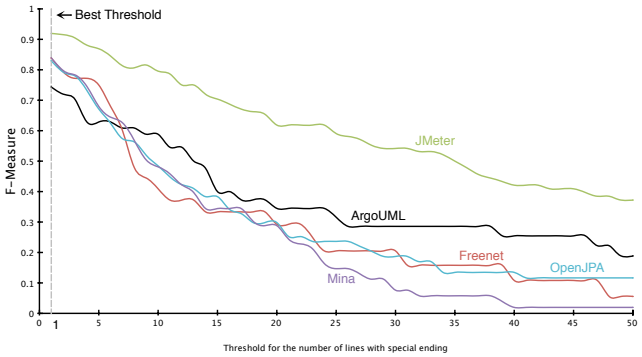

Figure 6. E-Mail classification based on end of lines

Taking Freenet as a sample system (Figure 7), we show that the approach can quickly achieve the maximum precision, while maintaining the recall value higher than 0.60. All the systems show similar results. Even in the worst case, ArgoUML, at the second step of the threshold the precision is 1 and the recall is higher than 0.50.

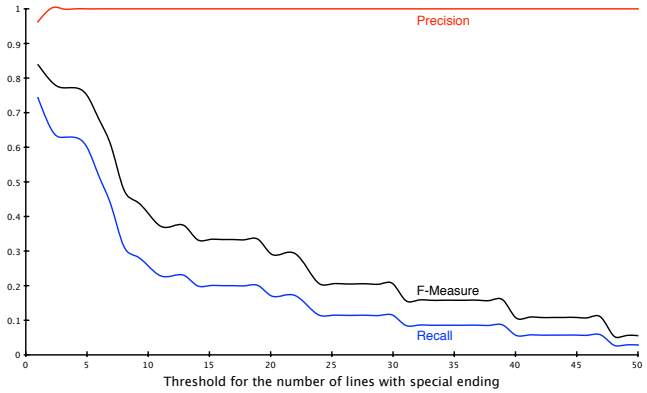

Figure 7. Freenet: E-Mail classification based on end of lines

3) End of lines and regular expression: Even though the previous approach reaches significantly high values, the recall can still be improved. Analyzing the false negatives, we noted that the method does not consider a common pattern in the Java programming language: the method call pattern. This pattern usually ends with a semicolon, but a method call can be split in multiple lines (as can be seen in line 11 of Figure 1), or used without semicolon in a stack trace. Our intuition is to raise the recall by checking this pattern in lines without a final semicolon.

Implementation. This approach extends the previous one, adding the check on the method call pattern. If a line does not end with any character specified in the previous method, it checks if it matches the method call pattern. The most effective technique to match such a regular pattern is using regular expressions. We implemented our regular expression, according the IEEE POSIX Basic Regular Expressions (BRE) standard, as in the following code (in multiple lines for convenience):

```
(1)   (.*)
(2)   ([[:alnum:]]+\.)+
(3)   ([[:alnum:]]|<[[:alnum:]]+>)+
(4)   \(
(5)   (.*)
```

Lines 1 and 5 allow the match to be found without requirements on characters that follow or precede it. Lines 2 to 3 require one or more occurrences of a character followed by a single dot, *e.g., foo.*, *boo.*, *boo.foo.* are valid sequences. After this pattern, there must be another alphanumeric string enclosed, or not, in angle brackets, *e.g., foo*, or *<foo>*. Finally, there must be an open parenthesis (line 4). This regular expression correctly matches the example code in line 11, Figure 1. Such code would have been matched, even though the part after the open parenthesis was spread out over multiple lines.

Results. Table II shows the results for each system.

System	Precision	Recall	F-Measure
ArgoUML	0.92	0.71	0.80
Freenet	0.97	0.91	0.94
JMeter	0.91	0.96	0.94
Mina	1.00	0.80	0.89
OpenJPA	0.89	0.88	0.88
Average	0.94	0.85	0.89

Table II
END OF LINE AND REGULAR EXPRESSION APPROACH RESULTS

We report the results obtained with the threshold of one, which is still the most effective, confirming the previous results. With this addition we included cases that were not correctly retrieved by using only the checking on the end of line. We significantly increased the recall, at the cost of a very small decrement in the precision. Overall, the results are promising: On average our lightweight approach retrieves 85% of the e-mails containing source code, and provides a correct classification in 94% of the cases.

B. Classification of text lines including source code fragments

1) Special characters and keywords: Our first method, which uses occurrence or frequency of special characters and keywords on the complete e-mail content, has the advantage of retrieving *all* the e-mails with source code fragments, and still providing results with reasonable precision. This approach uses the same idea on the finer granularity of lines: a line having many occurrences, or a high frequency, of keywords or special characters probably contains a code fragment. We apply the classification on different thresholds to find the optimal one.

Implementation. This approach has a similar implementation to its equivalent for e-mail classification, except that we split the text into lines and we count keyword and special character occurrences *per line*. We divide the occurrences by the number of words in the line to obtain frequencies. If a value passes the threshold, we classify the line positively.

System	Threshold	Precision	Recall	F-Measure
ArgoUML	1	0.11	1.00	0.20
	4	0.19	0.62	0.29
Freenet	1	0.05	0.99	0.09
	5	0.17	0.40	0.24
JMeter	1	0.22	0.98	0.35
Mina	1	0.14	0.98	0.25
	4	0.30	0.58	0.40
OpenJPA	1	0.16	0.95	0.28
	4	0.25	0.66	0.36
Average	1	0.14	0.98	0.24
	best	0.22	0.65	0.33

Table III
LINE CLASSIFICATION BY OCCURRENCES OF CHARACTERS

Results. Table III shows the result obtained using the occurrence of characters for the line classification. It reports both the value with a threshold of one, which provides the best recall, and the threshold for the best F-Measure (if different from one). We note that the best threshold values vary less than in the approach for the e-mail classification: between 4 or 5 for all the systems, except for JMeter. The results obtained using the frequency of special characters are similar but have lower values. Interestingly, for the line classification, both the occurrence and the frequency of keywords produce significantly lower results. In Figure 1 we see the reason: Eight lines out of the nine with source code fragments include, at least, one special character, while only four lines contain a keyword.

2) End of line and regular expression: We use the approach for text classification for line classification: Each line that ends with a semicolon, a open -or closed- curly bracket, or that matches the regular expression described previously, is classified positively. The presence of code comments in a complete text can be evidence of a larger source code fragment, for this reason, we did not remove comments in the e-mail classification approach. However, we do not consider comments as actual source code fragments, thus we remove them from lines during the classification. A threshold is not necessary: either a line does or does not respect the conditions. For the OpenJPA system, we also consider annotations as code fragments, since they are a relevant aspect of that Java persistence library.

Implementation. The implementation is similar to the complete text approach, but it classifies line by line, instead of the whole content. As a first pass, from each line, we remove the leading and trailing whitespace and the comments, We detect OpenJPA annotations by checking whether the first character of a line is a "@".

System	Precision	Recall	F-Measure
ArgoUML	0.93	0.89	0.91
Freenet	0.94	0.88	0.91
JMeter	0.97	0.86	0.91
Mina	0.90	0.84	0.87
OpenJPA	0.93	0.73	0.82
Average	0.93	0.84	0.88

Table IV
LINE CLASSIFICATION BY END OF LINE AND REGULAR EXPRESSION

Results. Table IV shows the results obtained, by system. The approach provides a very high precision (0.93 on average), keeping a substantial recall. For OpenJPA, the recall below the average is caused by the fragmentation of e-mail lines: Many of the words are truncated and split over different lines. A text normalization [20] would alleviates this problem, probably providing the approach with results obtained for the other systems.

3) Beginning of block: Even though the previous approach results meet a high target, we manually inspected all the false negatives to increase the recall. Lines with a class or method declaration are the most common problem: Methods and classes are usually defined in a single line, *e.g., public class Foo()*, however, the curly bracket that starts the subsequent block is in a new line. Since our previous approach does not consider these cases, we also check them. The simple intuition is to check whether a line begins with a keyword: We see a partial example of this in Figure 1, line 10.

Implementation. The implementation, similar to the previous one, now also verifies if the first word is a keyword.

Results. As expected, this approach increases the recall value of all the systems, of at least two points (Table V). The high values already obtained with the previous approach, make it interesting for the practitioner who wants more matches, with a lower precision.

System	Precision	Recall	F-Measure
ArgoUML	0.74	0.91	0.81
Freenet	0.60	0.90	0.72
JMeter	0.94	0.91	0.93
Mina	0.80	0.89	0.85
OpenJPA	0.77	0.74	0.75

Table V
LINE CLASSIFICATION BY MIXED APPROACH

C. Source code extraction

Considering the high results that our lightweight approaches achieved on line classification, we conducted a statistical analysis on the benchmark to determine whether

devising additional methods for a finer code extraction is necessary in practice.

Considering all the lines labelled as containing source code fragments in the benchmark (*i.e.,* 11,978), we evaluated the Levenshtein distance of the complete content from the source code part. More than 7,664 lines had a distance larger than three (*i.e.,* more than three operations are necessary to transform the content in the source code). Analyzing these lines, we noted that they are mainly composed of lines similar to those reported in Figure 8.

```
(1)     remove(LabelledLayout.getSeperator());

(2) at org.apache.maven.Maven.doExecute(DefaultMaven.java:336)

(3) +   add(LabelledLayout.getSeperator());
(4) -   remove(LabelledLayout.getSeperator());
```

Figure 8. Common lines with source code distant from the content

In line 1, the source code does not include leading and trailing whitespaces, which is present in the complete content. Line 2 starts with the "at" word used in stack traces, which is not source code. Lines 3 and 4 are part of a patch and start with "+" or "-" to mark added or deleted lines.

Removing these special cases and recomputing the Levenshtein distance, only 378 lines remained: the 3% on the total number of lines with code fragments. Since we considered statistically significant sample sets for the creation of our benchmark, this 3% ratio is a value that is to be found also in the whole mailing list populations.

Simply removing the special starting characters in Figure 8, and trailing and leading whitespace, we can use the same approaches explained for the line classification and obtain the perfect extraction of the source code for all the lines, except -at maximum- for those 3% that also include other not relevant characters. Since we want to maintain our methods fast and simple, we deem this result as acceptable.

V. Discussion

We presented a number of approaches to detect e-mails and lines that contain source code fragments. Table VI summarizes the average effectiveness of the methods over the 5 systems, in terms of precision, recall and F-Measure.

Method	Precision	Recall	F-Measure
E-mail classification			
Special characters and keywords	0.24	0.97	0.37
End of lines and reg. exp.	0.94	0.85	0.89
Line classification			
Special characters and keywords	0.14	0.98	0.24
End of lines and reg. exp.	0.93	0.84	0.88
Beginning of block	0.77	0.87	0.81

Table VI
Average effectiveness of detection methods

Since the methods vary with respect to the precision and recall they achieve, choosing the most appropriate method depends on the relative weight of precision and recall, for a given task. If a practitioner wants to retrieve most of the source code, at the cost of many e-mail lines without source code (*i.e.,* he wants a very high recall at the price of a lower precision) the best method is the one based on the occurrences of special character and keywords. On the other hand, if one is interested in correctly retrieving only documents with source code (*e.g.,* for time reasons), at the price of not retrieving all of the documents (*i.e.,* favoring precision over recall), then "end of lines with regular expression" is the most appropriate method. It should be also selected when precision and recall have the same importance, since it offers the highest F-Measure. When good overall effectiveness (indicated by the F-Measure) are required, the mixed method based on end of lines, regular expression and keywords at the beginning of the line, is to best choice. While the method based on occurrences of special character and keywords does not represent a good option, due to the low average F-Measure.

1) Comparison with similar approaches: In Section II we reported two approaches tackling a similar problem, one by Bettenburg *et al.* [7] and one by Tang *et al.* [20]. Bettenburg *et al.* detected bug reports containing source code fragments using an approach based on island parsing. Their technique reached results better than ours, with a precision of 0.98 and a recall of 0.99, an almost perfect classification. However, it is difficult to compare the results of the approaches, as they are applied on different data sets, with different characteristics (bug archive and development mailing list). Moreover, our methods have two main benefits over their approach: (1) they are faster and more scalable than island parsing and (2) they work correctly also at the line level.

Tang *et al.* applied a technique based on machine learning to detect source code fragments in a data set similar to ours but larger. In terms of precision, we reported similar results: they reached a precision of 0.93 and our best is 0.94 at the e-mail level and 0.93 at the line level. We achieved better results in the recall value: 0.84 against 0.72.

2) Limitations: Due to strong developer cultures, the style of e-mails in discussion lists can vary greatly. To alleviate this, we considered five different open source software systems emerging from different communities, in which the usage, the participants, and the age of the mailing lists vary.

The main limitation of our experiment consists in considering only the Java programming language. However, we devised methods that are based on characteristics available in many different programming languages, *e.g.,* keywords, special characters, and peculiar end of lines. We plan to analyze systems that use different programming languages and conventions, to include them in our public benchmark.

We inspected accurately all the e-mails in the sample sets. However, since human beings are involved, there is the possibility that they made mistakes in the analysis. To avoid

this problem we devised a web application to ease the task, and we manually re-inspected a relevant number of false negatives and false positives generated by our approaches during their construction, without finding errors.

VI. Conclusion

In this work we tackled the issue of detecting and extracting source code fragments in development e-mails, at document level and at line level. We devised lightweight techniques which, on the basis of simple text inspections, exploiting characteristics of source code text, can detect source code fragments in e-mails, fast and with a high accuracy. A practitioner can precisely classify thousands of e-mails, even at run-time. We also proposed novel methods for classifying lines that enclose source code. Using refined approaches, based on those used for the complete document classification, our methods achieve performance higher than the ones previously obtained through complex machine learning techniques. Moreover, almost all methods we developed can be configured with a threshold parameter that allows choosing the best trade-off between precision and recall, according to the user's needs.

To assess our techniques, we created a statistically significant and easily extensible and publicly available[2] benchmark: It features sets of sample e-mails, randomly extracted from five unrelated Java software systems, which we analyzed to label source code fragments. Using our benchmark, we conducted a statistical analysis of the e-mail content and assessed that the vast majority of source code fragments are mentioned as lines separated from the natural language text. Our work indicates that lightweight methods are to be preferred to heavyweight techniques in detecting and extracting source code fragments from development e-mails.

As a future work, we plan to add our lightweight techniques in *Remail* [3], a plug-in we are building to integrate e-mail communication, linking, and searching in the Eclipse IDE.

Acknowledgments We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] A. Bacchelli, M. Lanza, and M. D'Ambros. Miler - a tool infrastructure to analyze mailing lists. In *Proceedings of FAMOOSr 2009 (3rd International Workshop on FAMIX and Moose in Reengineering)*, 2009.

[3] A. Bacchelli, M. Lanza, and V. Humpa. Towards integrating e-mail communication in the IDE. In *Proceedings of SUITE 2010 (2nd International Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation)*, pages xxx–xxx, 2010.

[4] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages xxx–xxx, 2010.

[5] M. J. Beal, Z. Ghahramani, and C. E. Rasmussen. Factorial hidden markov models. In *Machine Learning*, pages 29–245. MIT Press, 1997.

[6] A. L. Berger, V. J. D. Pietra, and S. A. D. Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of MSR 2008 (5th IEEE Working Conference on Mining Software Repositories)*, pages 27–30. ACM, 2008.

[8] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in OSS projects. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, pages 26–29. IEEE Computer Society, 2007.

[9] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[10] A. Dekhtyar, J. H. Hayes, and T. Menzies. Text is software too. In *Proceedings of MSR 2004 (1st International Workshop on Mining Software Repositories)*, pages 22–26, 2004.

[11] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[12] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan. Should i contribute to this discussion? In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 181–190. IEEE CS, 2008.

[13] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 344–353. IEEE Computer Society, 2007.

[14] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, pages 492–501. ACM, 2006.

[15] C. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[16] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22. IEEE CS, 2001.

[17] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[18] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[19] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of ICSE 2003 (25th International Conference on Software Engineering)*, pages 74–83. IEEE Computer Society, 2003.

[20] J. Tang, H. Li, Y. Cao, and Z. Tang. Email data cleaning. In *Proceedings of KDD 2005 (11th ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 489–498. ACM, 2005.

[21] M. Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[2] http://miler.inf.usi.ch/code