Linking E-Mails and Source Code Artifacts

Alberto Bacchelli, Michele Lanza
REVEAL @ Faculty of Informatics
University of Lugano
{alberto.bacchelli,michele.lanza}@usi.ch

Romain Robbes
PLEIAD @ DCC & REVEAL
University of Chile & University of Lugano
rrobbes@dcc.uchile.cl

ABSTRACT

E-mails concerning the development issues of a system constitute an important source of information about high-level design decisions, low-level implementation concerns, and the social structure of developers.

Establishing links between e-mails and the software artifacts they discuss is a non-trivial problem, due to the inherently informal nature of human communication. Different approaches can be brought into play to tackle this traceability issue, but the question of how they can be evaluated remains unaddressed, as there is no recognized benchmark against which they can be compared.

In this article we present such a benchmark, which we created through the manual inspection of a statistically significant number of e-mails pertaining to six unrelated software systems. We then use our benchmark to measure the effectiveness of a number of approaches, ranging from lightweight approaches based on regular expressions to full-fledged information retrieval approaches.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

1. INTRODUCTION

It is estimated that up to 60 percent of software maintenance is spent on program comprehension [21]. This is because the knowledge about a system is often expressed implicitly and thus is difficult to retrieve [16]. For this reason, many approaches extracting information from the source code and the program structure have been presented in a process known as reverse engineering [9]. However, the development of any software system sees the creation of artifacts beyond the actual source code, such as requirements and design documents, system documentation, mailing list discussions, bug reports, *etc.* While such non-source artifacts enclose valuable information about the system they document, they are often informal, free text, natural language documents, and therefore non-trivial to process.

Free-form natural language artifacts (*e.g.,* documentation, wikis, forums, e-mails), intended to be read both by system developers and by users, often reference -implicitly or explicitly- other artifacts, such as source code or bug reports. However, reliably finding the traceability link between artifacts is an issue that the software engineering research community is still trying to address. Moreover, it is difficult to compare the effectiveness of such approaches, as often there is no established benchmark.

We focus on recovering the traceability link between e-mails and source code artifacts. Mailing lists are employed by developers and users to discuss various topics, ranging from low-level concerns (*e.g.,* bug fixes, refactoring) to high-level resolutions (*e.g.,* future planning, design decisions). E-mails provide additional metadata (author, date and time, threading, etc.) that enable further analyses, such as the social interaction between participants [8], geographic analysis [23], the behavior of developers and users [20], the correlation between mailing lists development activity [7].

In a preliminary work [3], we devised a set of lightweight methods, based on regular expressions, to establish the link between e-mails and software artifacts. We evaluated them in terms of precision and recall considering one single Java system. In this paper we overcome a number of limitations of our previous work, resulting in the following contributions:

- *An extensive and publicly available*[1] *benchmark and toolset for recovering traceability links between e-mails and source code artifacts.* We created our benchmark by analyzing the mailing lists of six different software systems written in four different programming languages. For each system we manually annotated a statistically significant number of e-mails.
- *A comprehensive evaluation of linking techniques.* We evaluated and compared, in terms of precision and recall, different linking methods, ranging from lightweight grep-style approaches to more complex approaches from the information retrieval (IR) field.

Structure of the paper. In Section 2, we review related work. In Section 3, we present our benchmark, how we created it, and Miler, its supporting tool infrastructure. In Section 4 we illustrate the lightweight linking approaches, while in Section 5 we present the information retrieval techniques. In Section 6 we show and discuss the results achieved by each technique. We draw our conclusion in Section 7.

[1]See `http://miler.inf.usi.ch/`

ICSE '10, May 2-8 2010, Cape Town, South Africa

2. RELATED WORK

Researchers proposed several techniques to deal with the issue of traceability between source code and non-source artifacts. We focus on methods that use information retrieval (IR) techniques to automatically retrieve the missing links.

Probabilistic and Vector Space Model (VSM). Antoniol *et al.* experimented with two different IR models to retrieve the links between code and documentation [1]: a *Probabilistic IR Model* and a *Vector Space IR Model*. The former computes a ranking score based on the probability that a document is related to a specific source code component, while the latter calculates the distance between the vocabulary of all the documents and a code component.

They analyzed two small software systems: The first is LEDA (Library of Efficient Data Types and Algorithms), a C++ library of 208 classes, totaling 95 kLOC. The documents to link were 88 manual pages automatically generated through scripts that extract comments from the source files: names of functions, parameters, and data types present in those files were also present in the manual pages. Each class was described by at most one manual page, while 53% (110) of the classes were not described at all. 78 out of 88 manual pages (89%) were relevant, the remaining ones contained basic concepts and algorithms. There were 98 links overall. The second system was implemented by students, and had 95 classes, of which 60 were considered in the study. The classes were matched with functional requirements written beforehand. There were 58 links overall. They used the collection of the documents as the corpus in which to find the missing links, and every single source code file as the "query". Each document was pre-processed to ease the linking task: they converted any letter to lowercase, removed stop-words (*i.e.,* articles, punctuation, numbers, common words, *etc.*), and performed stemming on the result (*i.e.,* converting plurals into singulars, transforming verbs into their infinitive form, *etc.*). In addition to this, they also extracted the list of identifiers from the source code, removed the language keywords, and split compounded words (*e.g., ClassName* into *class name*). They chose to disregard comments.

In both case studies, results revealed a better performance overall of VSM over the probabilistic approach.

Latent Semantic Indexing (LSI). Marcus *et al.* proposed a solution based on LSI [19]. LSI is based on a Vector Space IR Model that takes into consideration that a word always appears in a context. This additional information provides a set of mutual constraints that determines meaning similarity in sets of words.

They evaluated the effectiveness of their technique on the same systems considered by Antoniol *et al.*, adopting an inverse approach: they used the collection of the source code files as the corpus in which to find the link, and each document as the "query". As text pre-processing, they removed non-textual tokens from documents, converted any letter to lowercase, and split compounded words in the source files while also keeping the original form (*i.e., ClassName* becomes *classname class name*). They considered source code comments inside as relevant. Finally, as LSI does not use a predefined vocabulary or a predefined grammar, it was not deemed necessary to perform the stemming process, i.e., there was no morphological analysis.

The results were slightly improved with respect to the approach by Antoniol *et al.*, especially for LEDA: There were more documents in the corpus and the same entity identifiers were used in both the source code and the documents.

Hayes *et al.* asserted that IR techniques must not substitute the human decision-maker in the linking process, but should be used to generate an appropriate list of candidate links [13]. They show how they used the three IR algorithms proposed by Antoniol *et al.* and Marcus *et al.* (Vector Space Model, Vector Space Model with a simple thesaurus, and LSI) to trace requirements-to-requirements and aggregate candidate links to be evaluated by software analysts. They validated the algorithms on two systems of similar size to the ones used by Antoniol *et al.* (one of circa 20 KLOC of C code and 455 documents, and the other with 58 documents).

Lormans *et al.* used LSI to find traceability relations between requirements, design documents, and test cases [17]. They evaluated the effectiveness of LSI in terms of precision and recall on two small case studies.

Natural Language Processing (NLP). Baysal *et al.* tried to correlate discussion archives (*i.e.,* the e-mails in mailing lists) and source code [4]. They looked for a correlation between discussions and software releases. First, they recovered information about the system applying data mining techniques on its release history and the discussion archives. Then, they used Natural Language Processing methods to search for a correlation. They presented two significant case studies: a visualization tool (a Java system with 144 files and an archive of 495 e-mails) and Apache Ant (a Java system with 667 java files and an archive of 67,377 e-mails). Baysal *et al.* did not manually inspect the systems of their case studies to verify the quality of their results.

Reflections. While all approaches led to relevant results, they incur an inevitable bias because of the small number of analyzed systems, the small size of the systems, and the low number of artifacts. The only exception is the work by Baysal *et al.*, where however there was no manual inspection of the results to verify their quality. We argue that for any approach to have a solid validation, a benchmark is needed, against which any approach can be tested.

3. BENCHMARK

The areas in which IR techniques have proven useful (*e.g.,* management of scientific and legal literature, web searches) are supported by a set of well-designed, robust, and universally accepted benchmarks. Such benchmarks are publicly available and distributed via the infrastructure of the Text REtrieval Conference series (TREC), sponsored by the National Institute of Standards and Technology (NIST) and the US Department of Defense (DARPA) [22]. They keep evolving and now include retrieval tasks for many different kinds of information (e.g., spam, genomic data).

However, software systems have traits that distinguish them from the standard IR domains. For example, software artifacts form document collections orders of magnitude smaller than standard IR corpuses. Also, although the developers' knowledge is contained in identifier names and code comments, developers write them in a terse technical language. As a consequence, we cannot assume, without prior verification, that IR techniques would work with the same performances in the software engineering field. Specific benchmarks for software engineering need to be devised.

System URL	Description	Language	Mailing Lists		Sample		
			Creation	E-Mails	Size	E-Mails with a Link	Total Links
ArgoUML argouml.tigris.org	A UML modeling tool developed over the course of approximately 9 years.	Java	Jan 2000	29,112	355	108	290
Freenet freenetproject.org	A peer-to-peer software for anonymous file sharing, and for browsing and publishing ``freesites'' (web sites accessible only through Freenet).	Java	Apr 2000	26,412	379	148	570
JMeter jakarta.apache.org/jmeter	A desktop application designed for load and stress testing of Web Applications. The first release was done in 1999.	Java	Feb 2001	20,554	380	207	617
Away3D away3d.com	A realtime 3D engine for Flash, written in ActionScript, an object-oriented programming language compliant with the ECMAScript Language Specification.	ActionScript 3	May 2007	9,757	370	243	747
Habari habariproject.org	A blogging platform, written in object-oriented PHP 5.	PHP 5	Oct 2006	13,095	374	135	252
Augeas augeas.net	A configuration file editing tool, which parses configuration files and store them into a tree for successive modifications.	C	Feb 2008	2,219	281	140	273

Table 1: The software systems considered for the benchmark

3.1 Data set

To create our benchmark to validate the effectiveness of automatic linking techniques, we analyzed six unrelated software systems written in four different languages. They are all open-source and both the source code and the mailing lists are freely accessible. Table 1 details the systems.

We release this benchmark, so that other researchers can benefit from it to analyze new techniques. It does not require any special infrastructure to be used, it can be improved, and is easily extensible with additional data.

E-mail archives. All the projects have active mailing lists discussing different topics. We focus on development mailing lists, because they have the highest density of information about code entities and thus of the links we strive to find. As we had no prior details about the distribution of traceability links in e-mail archives, we employ random sampling without replacement (as opposed to other techniques, *e.g.,* stratified random sampling) to extract reliable sample sets from the populations of the e-mails. We establish the size (n) of such sets with the following formula [24]:

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

Since the proportion ($\hat{p}$) of the e-mails referring a specific entity of the source code is not known *a priori*, we consider the worst case scenario (*i.e.,* $\hat{p} \cdot \hat{q} = 0.25$). We have populations that –from a statistical point of view– are relatively small, so we included the finite population correction factor in the formula: It allows us to take the population size (N) into account (*e.g.,* 20,554 e-mails for JMeter). We keep the standard confidence level of 95% and error (E) of 5%, *i.e.,* if a specific source code entity is cited in the f% of the sample set e-mails, we are 95% confident it will be cited in the $f\% \pm 5\%$ of the population messages. This only validates the quality of this sample set as an exemplification of the population; it is not directly related to the *precision* and *recall* values presented later, which are actual values based on manually analyzed elements.

This resulted in the sample sizes (n) shown in Table 1. The column "E-Mails with a Link" counts the number of e-mail with at least one reference to a code entity. "Total Links" sums all the links retrieved from these e-mails.

Source code. The other ingredient of our benchmark is the source code of the systems. We consider all the e-mails since the inception of the mailing lists, so we also consider any system release throughout the system's history. When available, we took official releases as our milestones (*e.g.,* for JMeter or ArgoUML), otherwise we used the "checkout by date" feature of the version control system (*i.e.,* we retrieved the committed source code in 3 months intervals, starting 3 months after the repository creation).

We are interested in linking e-mails with source code entities (*i.e., classes* in object-oriented systems, *functions* and *structures* in procedural language systems), so we do not consider source files as the unit for documents (as opposed to Antoniol *et al.* and Marcus *et al.*), but we strive for a finer granularity.

We parse the source code, extract the model, and find the links between model entities and e-mails.

System	Releases	Number of Entities			
		First Release	Last Release	Total	Linked
ArgoUML	11	906	2,396	18,252	192
Freenet	30	822	2,026	37,878	387
JMeter	20	16	906	11,105	271
Away3D	9	132	465	2,351	209
Habari	12	20	124	1,105	74
Augeas	17	60	675	8,042	53

Table 2: Source Code Entities per Software System

Table 2 lists the collected data.

3.2 Tool Infrastructure: Miler

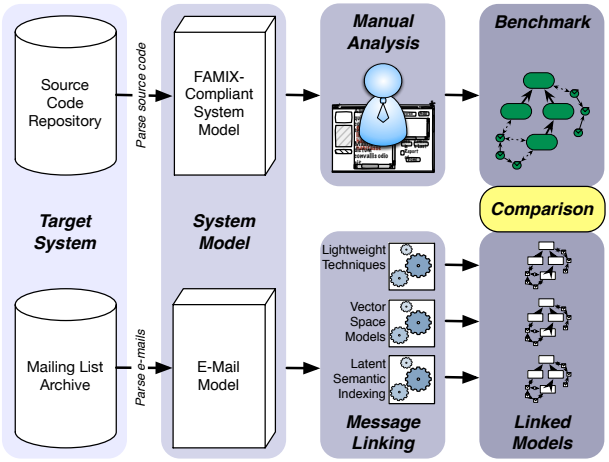

Figure 1: Infrastructure

For this experiment we created Miler, the infrastructure shown in Figure 1. For each system, we extract the e-mails from the mailing list archive in which they reside and import them in our infrastructure according to the e-mail meta-model we implemented. We parse the source code of each release and create a system model complying to FAMIX, a language independent meta-model for procedural and object-oriented code [12].

Once the models are ready, we link the messages and the source code. Message linking produces FAMIX models in which each entity is annotated with the reference to any e-mail treating it. The model containing manually inferred links is our oracle; it is compared to the models produced by the automatic techniques.

3.3 Creation of the benchmark

The creation of the benchmark consisted in reading all the e-mails in the sample set and annotating them with the source code entities they treat. We built a web application in Miler to assist this task. Figure 2 shows its main page, after a user logs in.

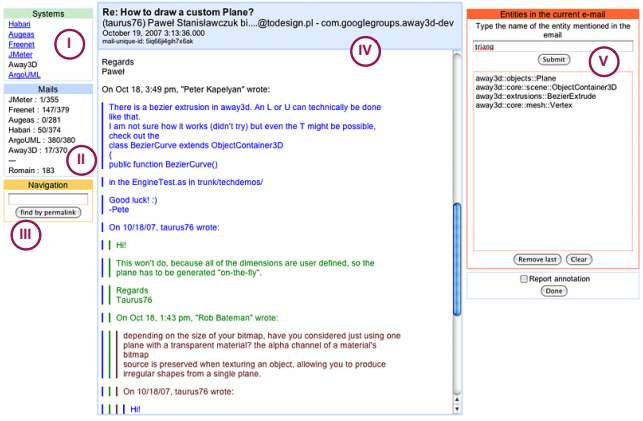

Figure 2: Benchmark Creation Web Application

It has the following components:

- The *Systems* panel (I) shows the list of the software systems that are loaded in the application and must be analyzed for creating the benchmark.
- The *Mails* panel (II) 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* panel (III) lets the user retrieve any e-mail by its permalink (displayed in the e-mail header).
- The main panel (IV) 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 *Annotation* panel (V) contains the list of already related entities and an autocompletion field (Figure 3).

The autocompletion field helps the user when annotating the e-mail: The user can see any entity whose name includes the letters she inserted, and the autocompletion avoids typos since only entities actually in the system can be related. The panel shows how entity names are colored following a special convention: Entities are black if belonging to the last release before the e-mail date; light-grey if belonging to an older release; blue if implemented in the first release after the e-mail date; and light blue if released later. For example, consider the user typing "ObjectContainer3D (Figure 3). The autocompletion menu shows the homonymous entities in three colors: "[..]proto::ObjectContainer3D" is displayed lightgray, because it is older than the current release; "[..]containers::ObjectContainer3D" is blue: it will be released in the next version; "[..]scene::ObjectContainer3D", is black, as in the current release. This helps the reader in the choice of the most appropriate entity.

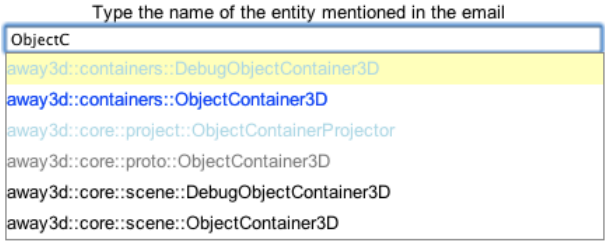

Figure 3: Web Application: Autocompletion menu

Six members of our research group, with several years of programming experience, inspected the sample set. The e-mails were randomly divided in overlapping sets, resulting in 51% of the messages analyzed by two people. A complete agreement was reached on 92% of these messages, with the remaining annotations featuring small differences: Almost all the divergences were caused by one of the two reviewers missing to annotate a link that was actually present in the e-mail. All the errors were corrected.

Annotators did not differentiate between links only present in text quoted from previous messages and present in the new content of the e-mail. This allows the usage of this benchmark as a general case of textual information containing source code identifiers and discussions.

3.4 Evaluation

To compare the effectiveness of all the approaches, we measure two well known IR metrics for the quality of the results [18], namely *precision* ($Precision = \frac{|TP|}{|TP+FP|}$) and *recall* ($Recall = \frac{|TP|}{|TP+FN|}$). TP (true positives) are elements correctly retrieved, FN (false negatives) correct elements not retrieved, and FP (false positives) elements incorrectly presented as correct. We can describe *precision* as the fraction of the retrieved links that are correct, and *recall* as the fraction on the total number of correct links. The union of TP and FN is empty in those benchmark e-mails that have no reference to source code entities. In these cases, the recall value cannot be calculated. Likewise, automatic approaches can find no link between an e-mail and source code, so the *precision* value cannot be evaluated. To overcome these issues, we calculate the average of TP, FP, and FN, on the entire dataset, and measure the average *precision* and *recall* from those values. This solution also takes into account the impact of false positives on *precision*, when the set of benchmark references is empty. Precision (P) and recall (R) trade off against one another, so we also use *F-measure*, their weighted harmonic mean:

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

In this formula, β decides the weighting of precision and recall. We chose to emphasize neither recall nor precision by using a β value of 1 to obtain the *balanced F measure*.

4. LIGHTWEIGHT LINKING

In this section we show how we apply the best performing lightweight techniques presented in our preliminary work [3]. We investigate whether the results we obtained are still valid in this extended case study, and we show a new lightweight technique for the other programming languages we consider.

Entity name, case sensitive. The simplest way to reference entities from an e-mail is using their names. In object-oriented programming languages (such as Java, ActionScript, and PHP5), developers use *CamelCasing* as a widely accepted behavior to define class names. It consists in capitalizing the first letter of a class name and adding additional words to the first putting the initial letter capitalized within the compound.

System	Precision	Recall	F-Measure
ArgoUML	0.27	0.68	0.38
Freenet	0.17	0.70	0.27
JMeter	0.15	0.73	0.25
Away3D	0.32	0.74	0.44
Habari	0.40	0.41	0.41
Augeas	0.09	0.72	0.15

Table 3: Entity name, case sensitive

The results achieved for all the object-oriented systems are similar, as shown in Table 3. We obtained a lower precision with Freenet and JMeter, because they have a higher number of class names that are dictionary words (*e.g., Node* or *Client* for Freenet, *Cut* or *Copy* for JMeter). We expected low performances for Augeas: since it is written in C, identifier names are lowercase both for functions and structures, thus the performances are consistent with those achieved for the other systems when not using case sensitivity.

Mixed approach. As seen in JMeter and Freenet results, classes whose name is a single dictionary word are the most problematic for the linking process. In our previous work we achieved the best results using a mixed approach based on the following intuition: Since class names usually represent abstractions of real-world objects, it is common practice to give them common dictionary names, such as *Node* or *Client*. However, programmers often need to use multiple words to name an abstraction, and since empty spaces are not allowed, they compound words through camel casing (*e.g.,* the entity name *ObjectContainer* is formed by the dictionary words "object" and "container"). However, compounded words are not part of a common dictionary, so we use this trait to distinguish cases in which a stricter matching is required. Programmatically, we distinguish compounded words from single words simply by counting the number of capital lettersr. Two or more capitals constitute a compounded words, so we use the case sensitive match on the entity name, otherwise the following regular expression.[2]

```
(.*) (\s|\.|\\|/) <packageTail> (\.|\\|/)
< EntityName > ((\.(java|class|as|php|h|c))|(\s))+ (.*)
```

This regular expression exploits the characteristic in the naming convention of source code files. It requires the presence of the last part of the package in which the entity reside (such information is easily retrievable once having the FAMIX model of the system) before the entity name itself, which must be then followed by a source code extension or any kind of separator. When used alone, this regular expression results in a high precision but a very low recall.

System	Precision	Recall	F-Measure
ArgoUML	0.64	0.61	0.63
Freenet	0.59	0.59	0.59
JMeter	0.59	0.65	0.62
Away3D	0.40	0.54	0.46
Habari	0.83	0.09	0.17
Augeas	0.14	0.02	0.04

Table 4: Mixed with regular expression, c.s.

Table 4 shows that the results we previously obtained on a single Java system are still valid in other unrelated Java systems: both Freenet and JMeter reach a precision of 0.59 and a recall which is the same or higher.

The algorithm performs well with Away3D, reaching a F-Measure value of 0.46. However, characteristics of this system make the results for the precision lower that in it is for the Java systems: (1) due to its rapid evolution many classes are often moved between packages, however our algorithm uses the package information only for non-compounded words. In the other cases, all the possible classes with the same name, but on different packages, are returned by the algorithm; (2) in ActionScript, the programming language in which Away3D is written, it is less common to mention a class using its package.

[2]Names with capitals letters only, *e.g., XML*, are treated using the strict regular expression.

Our algorithm applied on Augeas offers poor performances, because the C language does not follow the camel casing convention and does not have packages. The results on Habari are also low, this is due to two intrinsic characteristics of the system: (1) The majority of class names are not compounded words, so the algorithm switches always to the strict matching (which lowers the recall); (2) Namespaces (*i.e.,* packages) where only introduced in PHP 5.3, and Habari developers do not use them in the releases considered, making the first part of the regular expression useless.

Punctuation. We reach better results for non-Java systems using the following regular expression[3]:

```
(.*) (:punct:|\s)+ <EntityName> (:punct:|\s)+ (.*)
```

The intuition behind it is that an entity name is written as a single word separated from others by empty space or connected to them through source code tokens (*i.e.,* punctuation). For example, let us consider the following text:

```
1. var data:Plane
2. Casting is not necessary in SmallTalk
```

Line 1 shows the declaration of the variable "data" in Away3D. Although *Data* is one of the entities available, this approach does not report a link, because of case sensitivity. *Plane* however is correctly matched. We: check case sensitivity; count the capital letters (1); and use the above regular expression, which reports *Plane* as a matching entity, since it is surrounded only by punctuation and spaces. In line 2 the name of the entity *Cast* partially matches the word "Casting". Due to the single capital letter, the regular expression is used, and it refuses the match because of the letter "i" after the last letter "t".

System	Precision	Recall	F-Measure
ArgoUML	0.35	0.68	0.46
Freenet	0.27	0.69	0.39
JMeter	0.30	0.72	0.42
Away3D	0.41	0.72	0.52
Habari	0.49	0.38	0.43
Augeas	0.15	0.64	0.24

Table 5: Mixed with new regular expression, c.s.

This simple variation in our approach gives higher outcomes for all non-Java systems, as shown in Table 5. Augeas increases both recall and precision, while Away3D and Habari have a higher F-Measure due to increasing recall.

Reflections. The "grep-like" lightweight methods offer speed, accuracy, and simplicity, but with some drawbacks:

- *Strict matching.* As the similarity between software artifacts decreases, so does the performance of "grep-like" methods. We reach interesting results in finding the link between source code and e-mails because the entities are mainly mentioned using their full name. When this is not the case these methods are unusable.
- *Bounded recall.* The simple matching on the entity name, not case sensitive, sets the upper bound for the recall value. We could probably increase it, if we also consider compounded names as written separated by a space. However, since e-mails can discuss about entities without ever mentioning their names, these methods cannot reach a recall value of 1.
- *No ranking.* "Grep-like" methods return documents without any ranking: A document either matches or not the regular expression (or query). As a consequence, developers who want to keep only relevant results must read all the returned documents.
- *Behavior on large systems.* Lightweight methods do not require a pre-processing of the corpus in which to find the links. This means that they are fast and easily implementable, however their efficiency is inversely proportional to the number of documents. To make them more efficient, one would need to build a specialized index on the corpus.

[3]":punct:" is the POSIX standard matching all the punctuation characters.

5. IR TECHNIQUES

The two Information Retrieval techniques we introduce promise to overcome the limitations of grep-style approaches. We briefly present them and then we test their effectiveness and compare the results they reach to the ones we obtained with lightweight approaches.

5.1 Vector Space Model

The vector space information retrieval models (VSM) represent the query and the documents in the corpus as *term vectors*. The size of such vectors is the number of terms present in the corpus vocabulary. If we consider a document (d), the cardinality of the vocabulary ($|C|$), and the number of times each term (t_i) occurs in the document, we can define the vector as: $v_d = [t_{1(d)}, t_{2(d)}, \ldots, t_{C(d)}]$.

Term vectors are aggregated and transposed to form the *Term Document Matrix*, (*tdm*):

$$tdm = \begin{array}{c|cccc} & D_1 & D_2 & \cdots & D_N \\ \hline t_1 & 0 & 1 & \cdots & 0 \\ t_2 & 0 & 0 & \cdots & 4 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ t_C & 1 & 2 & \cdots & 0 \end{array}$$

Researchers have proposed many forms of weighting to take into account the relevance of terms in each document (*local weighting*) and in all the corpus (*global weighting*). In our experiment, we use a widely recognized IR weighting method called *tf-idf*. *Tf* (term frequency) is used for the local weighting: each cell contains the number of occurrences for the document D_n divided by the number of terms in the entire document. *Idf* (inverse document frequency) is the global weighting: the more a term is common among all the documents, the less it is weighted.

$$tf_{c,n} = \frac{d_{c,n}}{\sum_{i=1}^{C} d_{i,n}} \qquad idf_c = \log|D| - \log|\{d : t_c \in d\}| + 1^4$$

$$tf\text{-}idf_{c,n} = tf_{c,n} \cdot idf_c$$

[4]Since the *idf* value can be zero (*i.e.,* a term is present in all the documents), we add 1 to the formula.

In our experiment, the e-mails form the corpus and the source code entities are the queries. First we take the e-mails and normalize their text by removing punctuation and stopwords –very common words that are not useful to distinguish documents, such as conjunctions or prepositions. Contrary to Antoniol *et al.*, we achieve better precision results *avoiding* the stemming step. The stemming clusters words with the same root (*e.g.,* the terms "model" and "models" are reduced to the root "model") augmenting the recall of the query, but also severely decreasing the precision. For example, considering *Plugin* and *Plugins*, two classes in Habari, if stemming was performed then the same links would be inferred for both classes. This would have highly increased the false positives.

Once the matrix is created, it is possible to evaluate a free-form text query on it. Any query is handled as a new document that is compared to the documents in the matrix. The "closest" documents to the query are returned as results.

First, a new document vector must be created with cells populated by the terms in the query (only words that are already in the corpus are considered). Once the query vector (q) is obtained, we evaluate the *similarity* between it and the vectors (d) of the documents in the corpus. We compute it evaluating the cosine of the angle between the vectors [5]:

$$cos\theta_n = \frac{d_n^T q}{||d||_2 \, ||q||_2} = \frac{\sum_{i=1}^{C} d_{c,n} q_c}{\sqrt{\sum_{i=1}^{C} d_{c,n}^2} \sqrt{\sum_{i=1}^{C} q_c^2}}$$

Any document for which the distance is less than the required threshold is returned as a candidate match.

5.2 Latent Semantic Indexing (LSI)

Synonymy and *polysemy* are the two main problems shared by all the previous techniques. Because of synonymy, a document can reference an entity in many different ways, beyond its formal and common identifier. For example, ArgoUML developers often use the name "NSUML" when referring to the class *NSUMLModelFacade*. Because of polysemy, when common dictionary words are used as names for source code entities, it is more difficult to distinguish them.

The aim of LSI is overcoming these two issues by spotting the relationships between terms, in order to disclose the *latent semantic structure* of the corpus and recognize its *topics* [10]. LSI builds these underlying relationships by considering words co-occuring in multiple documents. Instead of depending on individual words to locate documents, it uses such topics to find relevant documents. Documents are no longer represented by vectors of term frequencies but by vectors of topics inferred from the co-occurences of these terms (*e.g.,* if the terms "NSUML" and "ModelFacade" occur together often, a document containing "NSUML" only might still be returned if the query is "ModelFacade").

LSI starts where VSM approaches stop: given a term-document matrix, it outputs a reduction through Singular Value Decomposition (SVD). SVD is a technique originally used in signal processing to reduce noise while preserving the original signal [15]. LSI assumes that the original term-document matrix is filled with the noise generated by synonymy and polysemy, so the reduction is a model of the corpus with such noise lowered.

As for the previous technique, the e-mails form the corpus and LSI uses a term-document matrix that is generated through an identical pre-processing step (*i.e.,* we remove punctuation and extremely common words, and we do *not* perform any stemming on words). SVD reduces the vector space model in less dimensions, while at the same time preserving as much information as possible about the relationships between terms. The dimension of the resulting matrix is equal to the number (k) of topics to be considered. Finding the right value of k is crucial to obtain the appropriate results from LSI usage: In search engines it mainly ranges between 200 and 500, while in the analysis of source code topics for clustering, it usually ranges between 20 and 50 [15]. Researchers are still investigating how to determine it [14]. After the resulting reduced matrix is computed, we can query it using any external document. As for the previous approach, we consider the query as a vector and evaluate its similarity to other documents using the cosine of their angles. To index the query, a naïve, but slow, approach is recomputing the entire SVD matrix with the query document added to the matrix and extract its vector. Instead, we use topic inference techniques to recover the topic composition of the query document based on its term frequency [6].

5.3 Results

We explore the impact of different settings for the parameters of VSM and LSI: (1) discussing the impact of topics for LSI; (2) exploring the various query types for both approaches; (3) measuring the best distance thresholds; and (4) investigating the role of corpus size. We performed an exhaustive comparison of all parameter combinations, but we outline here general trends only.

Impact of the number of topics. The number of topics impacts both the results of the approach and the time for corpus indexing and query comparison. The quality of the results and the computation time increase with the number of topics. However, the quality decreases when the number of topics overcomes a certain threshold. We are interested in finding the minimal but still effective number of topics. Figure 4 plots the best F-Measure values obtained with LSI, by number of topics and query type. We see a performance plateau after 200 topics, and a maximum around 250 topics.

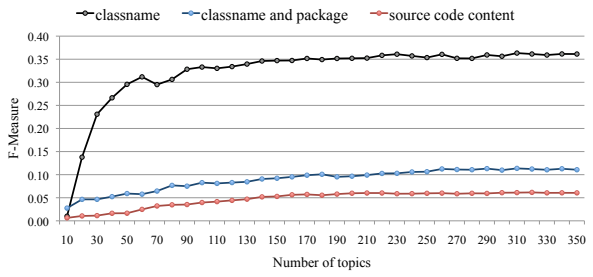

Figure 4: LSI, F-Measure by topics and query type

Most effective query type. We considered three different kind of queries generated from the entity to search links for: (1) the entity name; (2) the entity name and the package in which it resides; (3) the whole source code of the entity. The entity name query is the best performing, while the others provide too much noise (Figure 4). We obtained

consistent results using VSM with tf-idf.

Optimal distances. Considering the F-Measure as the indicator of overall performance, in Figure 5 we see that the best distance threshold for VSM with tf-idf is in the 0.85–0.91 range.

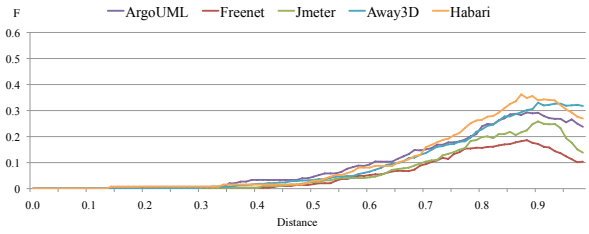

Figure 5: VSM, tf-idf: F-Measure by distance

For LSI the results are less conclusive (Figure 6): The optimal distance is heavily dependent on the system (*e.g.,* for Freenet is in the 0.25–0.35 range, while 0.6–0.8 for Habari and 0.9–1 for Augeas), thus one must discover the optimal distance for each case.

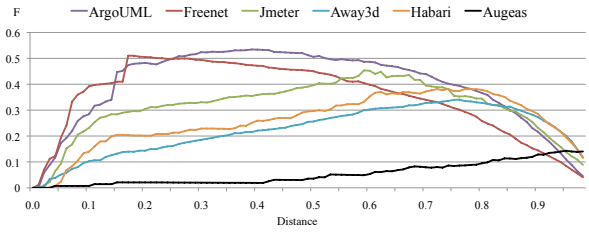

Figure 6: LSI: F-Measure by distance

Use of full/partial corpus. We computed the results detailed above using only the manually annotated emails included in the benchmark as the corpus. However, unlike regular expressions, Information Retrieval techniques are affected by the corpus used. A larger corpus might significantly change weightings of VSM with tf-idf, and alter topics inferred by LSI. On the one hand, the topic descriptions and weightings could be more accurate, on the other hand there might be much more noise. To test this, we used VSM and LSI to index the entire mailing list content and have a full corpus. Then, when running the benchmark, we kept only the documents also in the benchmark as valid results, discarding the others. Our tests show that using a full corpus has a harmful effect for both approaches: VSM's results are much lower, while LSI's performance seems to improve only with a very large number of topics. Unfortunately, large numbers of topics (3,000 or more) are very expensive to compute when generating the approximate matrix and also increase the time needed for the distance computation, i.e., it took more than 24 hours[5] to obtain the approximate matrix from a complete mailing list using a fast C implementation of SVD[6]. Linking a single class to the same complete mailing list using lightweight approaches takes seconds. Moreover, results are worse than with a restricted corpus: with 100 topics the maximum F-Measure value reached is 0.06, 0.19 with 1,000 topics, and 0.24 with 3,000 topics.

Overall results. Table 6 shows the overall results with optimal parameters (entity name as a query type, best overall distance, restricted corpus, 200-300 topics for LSI).. As expected, IR methods achieve higher recall values than lightweight methods, but at a significant cost in precision. F-Measure values for LSI outperform VSM with tf-idf results, but are still far from the performance of the lightweight methods.

System	Vector Space Model, tf-idf			Latent Semantic Indexing		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure
ArgoUML	0.25	0.34	0.29	0.60	0.48	0.53
Freenet	0.15	0.25	0.19	0.62	0.43	0.51
JMeter	0.21	0.34	0.26	0.52	0.40	0.45
Away3D	0.35	0.31	0.33	0.35	0.33	0.34
Habari	0.34	0.39	0.36	0.36	0.41	0.38
Augeas	0.10	0.20	0.14	0.10	0.28	0.14

Table 6: VSM and LSI results, optimal parameters

Both the techniques applied on Augeas still offer poor results. Before applying the techniques on all the systems we pre-processed the text –converting it to lowercase–, thus case sensitivity cannot be the cause. However, many components (such as executables or configuration settings) have names identical to source code entities (functions and structures); they can be distinguished only by understanding the context in which they are mentioned.

6. DISCUSSION

Figure 7 summarizes the bests results obtained by all approaches. The crosses plotted on the graph represent precision and recall of each approach, while the areas of the bubbles are proportional to the F-Measure. Bubble borders differentiate the approaches: Full for the lightweight approach, thick dashes for VSM with tf-idf, and thin dashes for LSI. F-measure ranges are: 0.24–0.63 for regular expressions (choosing the regular expression according to the language ofthe system); 0.14–0.33 for VSM with tf-idf; and 0.14–0.53 for LSI. As it emerges from the graph, the lightweight methods based on regular expressions outperform information retrieval approaches consistently. Indeed, authors of e-mails often mention source code entities by name, hence the benefit of accounting for indirect references (the higher recall of LSI and VSM), is offset by their sensibility to noise (much lower precision). The ranking of the approaches is stable between different projects: for example, if we consider Augeas, which has low values for all the rankings, the lightweight approach is still the best performer, followed by LSI, and finally by VSM. This order is preserved when considering all the systems.

However, several additional aspects must be taken into account before drawing conclusions. We discuss each approach individually, before issuing our overall recommendation.

[5]on a dual quad-core Intel Xeon server with 42GB of RAM
[6]http://tedlab.mit.edu/ dr/SVDLIBC/

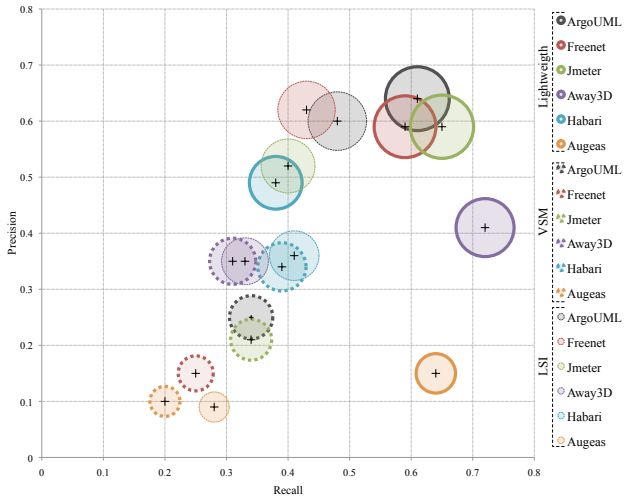

Figure 7: Overall Precision, Recall and F-Measure

Lightweight approaches. Our lightweight approaches are more language-dependent (with respect to other techniques): Our first mixed approach reached equivalent results on all the three Java systems, however the results for other systems are relatively more distant. This is caused by the "strict part" of the lightweight approach, *i.e.,* the regular expression, since it heavily relies on common conventions and intrinsic syntactical characteristics of the programming language. To overcome this issue, it is necessary to devise appropriate regular expressions capable of taking into account the syntactic features of the programming language of the system for which links must be found. We showed how, with a simple change, it is possible to achieve good results also in non-Java systems. However, when naming conventions are not followed and entities are not mentioned by name, this technique offer poor results, as shown by the outcome on the Augeas system, developed in C.

VSM. The VSM with tf-idf approach does not reach high values even considering the best outcome for each system. It also suffers of serious performance issues when used in very large corpuses, since it must store all the vocabulary of the corpus in the term-document matrix. For example, performance seriously decreased when we used it on the entire e-mail population of JMeter (20,554 e-mails) both to build the *tdm* and to compute distances between vectors.

LSI. Theoretically, LSI should not suffer from performance issues when used in large corpuses as it reduces the size of the matrix to the approximate one, which has a lower rank. However, in practice, computing the approximate matrix of a very large corpus is very expensive. If using the same number of topics used in a small corpus, the results are not maintained. We measured how, for source code to e-mails linking, it is necessary to increase the number of topics when having a larger corpus to improve results. Even impractical number of topics (computing 3,000 topics took more than 24 hours) did not provide good results when the entire mailing list was indexed. For this reason, LSI suffers from the same scalability problems as VSM. This issue was also reported in other applications of LSI to standard IR corpuses [11].

Hence, our final recommendation is that the IR approaches we tried are too heavyweight and still not accurate enough to be worth the investment. In addition, they do not help to solve the problem that code entities are often referred to in ways other than their actual names. The best approach to link email and source code is using regular expressions, while being careful that these are tailored to the programming language in use.

6.1 On The Threats to Validity

Construct Validity. Threats to construct validity are concerned with whether what one measures is what one intends to measure. In our case, there could be several reasons why the links established between the emails and the source code as part of the benchmark are incorrect. We rely on human judgment to annotate the emails, which is a potentially error-prone process. To alleviate this issue, two different judges annotated 50% of the emails we inspected. When measuring the agreement between them, we found an overlap of 92%, where the 8% of disagreement was due to one judge missing one link in some emails. We corrected these errors in the set of email that was inspected twice. We expect the same low proportion of missing links in the other half of the sample, which may affect the accuracy of the results. To address this issue further, we plan to have additional judges review the emails inspected only once.

Another issue is the domain knowledge of the judges. Being unfamiliar with the reviewed systems, they may miss some implicit references (*e.g.,* abbreviations) to entities that a seasoned developer of the system might understand. A qualitative evaluation of our benchmark that involves system developers could measure the effect of this threat.

Statistical Conclusion. Threats to statistical conclusion are concerned with whether we have enough data to support our claims with a reasonable confidence. We took samples of e-mail populations that were representative with a 95% confidence and a 5% error level, which are standard values. On the number of links, our corpus has 2,749 mail-to-code links, about 20 times as many as in Antoniol's study [1].

External Validity. Threats to external validity are concerned with the generalizability of the results. The approaches we tried may show different results when applied to other software systems. To alleviate this, we chose 6 systems with unrelated characteristics. The systems are developed from separate communities and are implemented in 4 different programming languages in two paradigms, object-oriented and procedural. The sizes of the systems, and of their mailing lists both varied by one order of magnitude. In general, we found that if approach A performs better than approach B on a system, it tends to perform similarly on all the systems. There is one caveat: Lightweight approaches based on regular expressions are language-specific.

There are however some aspects in which our selection is not representative: We only consider open-source systems. Usage patterns may vary in the industry. In particular, mailing lists often occupy a central role in the development of open-source systems, which may not be the case in systems developed in a more centralized and confidential fashion. Finally, we have not analyzed truly large-scale systems (our largest system has around 2,000 classes): we cannot confirm that our results are similar in these cases. In particular, we expect the VSM and LSI approaches to become more resource-intensive as systems and email sets grow in size.

7. CONCLUSION

E-Mail archives enclose significant information on the software system they discuss. We dealt with the problem of recovering traceability links between e-mails and source code. We showed the need for a benchmark to assess linking approaches against and presented Miler, the tool infrastructure we created to build a statistically significant benchmark of links between e-mail and source code over six software systems.

We evaluated different automated approaches to retrieve these links: Lightweight methods based on capturing programming languages elements with regular expressions, and two Information Retrieval approaches. We tested all approaches against the benchmark we created and measured their effectiveness in terms of precision, recall and F-measure.

The results we obtained show how, for this task, "less is more": The lightweight methods consistently and significantly outperform the IR approaches in all six systems. The reason is that in e-mails entities are often referred to by name, not synonyms, and source code is rare.

Our future work is twofold: (1) since naming conventions greatly improve the linking, easing their usage when writing emails is critical, and (2) we will exploit these links; we have already shown their usefulness for bug prediction [2].

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

8. 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. D'Ambros, and M. Lanza. Are popular classes more defect prone? In *Proceedings of FASE 2010 (13th International Conference on Fundamental Approaches to Software Engineering)*, pages xxx–xxx, 2010.

[3] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering)*, pages 205–214. IEEE CS Press, 2009.

[4] O. Baysal and A. J. Malton. Correlating social interactions to release history during software evolution. In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, page 7. IEEE Computer Society, 2007.

[5] M. Berry and M. Browne. *Understanding Search Engines - Mathematical Modeling and Text Retrieval*. SIAM, 2nd edition, 2005.

[6] M. W. Berry, S. T. Dumais, and T. A. Letsche. Computational methods for intelligent information access. In *Proceedings of SC 1995 (ACM/IEEE Conference on Supercomputing)*, 1995.

[7] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of MSR 2006 (3th International Workshop on Mining Software Repositories)*, page 137, 2006.

[8] C. Bird, D. S. Pattison, R. M. D'Souza, V. Filkov, and P. T. Devanbu. Latent social structure in open source projects. In *SIGSOFT FSE*, pages 24–35, 2008.

[9] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[10] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[11] A. Dekhtyar and J. Hayes. Good benchmarks are hard to find: Toward the benchmark for information retrieval applications in software engineering. In *ICSM 2006 Working Session: Information Retrieval Based Approaches in Software Evolution*, 2007.

[12] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[13] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[14] A. Kontostathis. Essential dimensions of latent semantic indexing (LSI). In *Proceedings of HICSS 2007 (40th Annual Hawaii International Conference on System Sciences)*, pages 73–80. IEEE CS Press, 2007.

[15] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[16] 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.

[17] M. Lormans and A. van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 47–56, 2006.

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

[19] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of ICSE 2003 (25th International Conference on Software Engineering)*, pages 125–135. IEEE CS Press, 2003.

[20] D. S. Pattison, C. Bird, and P. T. Devanbu. Talk and work: a preliminary report. In *MSR*, pages 113–116, 2008.

[21] S. Pfleeger and J. Atlee. *Software Engineering - Theory and Practice*. Pearson, 2006.

[22] 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 CS Press, 2003.

[23] R. Tang, A. E. Hassan, and Y. Zou. Techniques for identifying the country origin of mailing list participants. In *Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering)*, pages 36–40. IEEE CS Press, 2009.

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