Benchmarking Lightweight Techniques to Link E-Mails and Source Code

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

***Abstract*—During the evolution of a software system, a large amount of information, which is not always directly related to the source code, is produced. Several researchers have provided evidence that the contents of mailing lists represent a valuable source of information: Through e-mails, developers discuss design decisions, ideas, known problems and bugs, etc. which are otherwise not to be found in the system.**

A technical challenge in this context is how to establish the missing link between free-form e-mails and the system artifacts they refer to. Although the range of approaches is vast, establishing their accuracy remains a problem, as there is no benchmark against which to compare their performance.

To overcome this issue, we manually inspected a statistically significant number of e-mails pertaining to the ArgoUML system. Based on this benchmark, we present a variety of lightweight techniques to assign e-mails to software artifacts and measure their effectiveness in terms of precision and recall.

I. Introduction

The evolution of software systems leaves many traces. Some of these are structured data (e.g., SCM archives, analysis data), others are semi-structured information (e.g., test cases, bug reports), and the rest is completely unstructured. The "rest", made of documentation, wikis, forums, e-mails repositories, chat logs, etc, is also known as *semantic information* [12]. This kind of information –which is intended to be read by both the people involved in the evolution of the system and the people that use it– often references, explicitly or implicitly, other data sources such as source code and bug reports. However, actual linking to the referenced entities is "left as an exercise to the reader". Moreover, the links are one way: there is no link from source code to e-mails or bug reports to e-mails.

Connecting this *semantic information* to the source code can be helpful for various different tasks [1]:

- *Understanding software systems:* As systems are continuously growing in complexity and size, they help both bottom-up and top-down comprehension [1];
- *Recovering design rationales:* Often, developers discuss design decisions over free-form media channels, such as mailing lists or IRC (Internet Relay Chat) channels [3]. Establishing the link between software entities and those discussions permits to join design decisions and their implementation;
- *Performing impact analysis:* after a change is discussed and approved, it is implemented. Tracing such discussions with the subsequent code modifications gives useful hints about the impact of changes.
- *Identifying coupling*: software entities that are often mentioned at the same time are implicitly coupled.
- *Extracting developer behaviour:* provided the appropriate links, it is possible to verify how changes occur in the source code: if they are discussed before or after their implementation.

However, in order to gain such benefits –which are often critical to the success of software projects– the link between source code and user centric information must be present, up-to-date, and relevant. Achieving these features manually is arduous. They constitute a task done gradually, that forces the developer to interrupt his normal programming flow and that is tedious, error-prone and time-consuming. As an alternative to manual linking, in the last years, much work was devoted to automatically establish *traceability links* between semantic information and other artifacts [12].

In this research, we focus on e-mails, which are a special case of semantic information: They are used to discuss issues ranging from low-level decisions (e.g., software entities implementation, bug fixing) up to high-level considerations (e.g., design rationales, future planning), they can often be written and read by both software system developers and end-users, and they always come with additional information (e.g., time-stamp, thread, author) that can be taken into account. We present different lightweight approaches that, exploiting the specific characteristics of e-mails and the ones of the source code, are capable of establishing a bi-directional link between source code entities and e-mails. However, implementing these techniques is not enough: One needs to be sure they perform correctly. Thus, to assess the effectiveness of our approaches in terms of precision and recall, we manually created a statistically significant benchmark.

Structure of the paper. In Section II we present an overview of related work. In Section III we illustrate the benchmark we set up and the infrastructure supporting it. In Section IV we detail the different approaches we tested and in Section V we discuss how they perform with respect to our benchmark. Finally, Section VI concludes the work.

II. Related work

Various tools and techniques have been presented to deal with the problem of traceability between source code and user centric information. According to Zhao *et al.* [23], they can be classified in three categories: artifact traceability support tools, artifact traceability via intermediate abstraction, and artifact traceability via Information Retrieval.

A. Traceability support tools

These CASE tools help the developer to manually maintain links between source code and other artifacts. They provide support for recording, displaying and checking links.

TOOR (Traceability of Object-Oriented Requirements) [15] is a visual tool intended to be used during development. Through TOOR, the programmer can define any artifact (e.g., design charts, system manuals, interview transcripts) as an object of a certain class and can establish a relation connecting it to other artifacts. Such objects and relations can be later inspected graphically. For instance, all objects of a given class, or all objects participating in a given relation, can be shown at the same time. The tool, using properties such as transitivity, can also relate indirectly linked objects.

REMAP [16] and gIBIS [4] are based on IBIS (Issue Based Information System) [21], a method that provides a model for representing "argumentation" processes.

The Ophelia Traceability Layer [19] is a tool that considers all the artifacts of the software development lifecycle and helps the software engineers to create a graph of relationships. This graph is completely navigable and consistently maintains the traceability between artifacts.

B. Traceability via intermediate abstraction

The methods in this category create an intermediate abstraction, for both the source code and the documentation, that is used as a basis for the matching.

Fiutem and Antoniol [8] and Antoniol *et al.* [2] suggest an Abstract Object Language (AOL), Murphy *et al.* [14] use a reflexion model, and Sefika *et al.* [17] propose a model based on both static and dynamic information. These approaches require a high level of interaction from the user.

C. Traceability via Information Retrieval

These techniques make use of Information Retrieval (IR) technologies for automatically recovering traceability links.

Antoniol *et al.* introduce two different IR models to retrieve the links between code and documentation (in this case manual pages and functional requirements) [1]. The first one is a *probabilistic* IR model, which computes a ranking score based on the probability that a certain document is related to a specific source code component; the second one is a *vector space* IR model, which calculates the distance between the vocabulary of a document and a source code component. These approaches are tested on two small case studies (a C++ system, 95 KLOC, 208 classes, and 88 manual pages; and a Java software, 20 KLOC, 95 classes, and 16 functional requirements) and the precision and recall values obtained are shown. Interestingly, the author spot how these techniques, which are primarily used as stand-alone solutions, can be also exploited to aid a manual linking task, because they reduce the size of the documentation that must be read by one order of magnitude, at minimum.

Marcus *et al.* present a solution based on Latent Semantic Indexing (LSI) [13]. LSI is a method based on a vector space IR model, and it takes in consideration that a word always appears in a context. This additional information provides a set of mutual constraints that determines similarity in meaning of sets of words to each other. The authors test their approach on the same systems Antoniol *et al.* used in their case studies. De Lucia *et al.* started from the results of Marcus *et al.* to enhance the ADAMS tool [10].

Hayes *et al.* assert 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 [9]. They show how they used three IR algorithms (*Tf-Idf vector retrieval*, *vector retrieval with a simple thesaurus*, and LSI) to trace requirements-to-requirements, in order to aggregate candidate links to be evaluated by the software analyst. They proved the effectiveness of these algorithms in two case studies. The systems analyzed were approximately the same size of the ones used by Antoniol *et al.* (approximately 20 KLOC of C code and 58 documents for the first one, and 455 documents for the second one).

Baysal *et al.*, also considering the work of Antoniol *et al.* and Marcus *et al.*, try to correlate discussion archives (i.e. e-mails in mailing lists) and source code [3]. In particular, they search for a correlation between discussions and software releases. They first apply data mining techniques on the release history of a software system and discussion archives to recover information about them. Then they use Natural Language Processing (NLP) methods to search for a correlation. They present the correlation they find in two case studies: a visualization tool (a Java system of 144 java 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.* do not manually inspect the system of their case studies to verify the quality of their results.

Our work also aims at finding links between two software artifacts: source code and mailing lists. We want to verify whether it is possible to automatically find reliable traceability links between emails and software entities using lightweight approaches (namely string and regular expression matching) –which exploit intrinsic characteristics of source code elements– rather than expensive IR models or NLP.

III. Benchmarking the Linking Approaches

In this section we present the motivations and the techniques we used to develop the benchmark we created to assess the quality of our lightweight linking methods.

A. Motivation

Dekhtyar *et al.* put in evidence a crucial problem also shared by all the linking techniques presented in Section II-C: There is no specific, valid and standard benchmark for testing and comparing proposed linking solutions [6]. For example, IR techniques that are commonly employed in web searches are supported by a series of well-designed, robust and universally accepted benchmarks that are publicly available and distributed via the infrastructure of Text REtrieval Conference (TREC) series, sponsored by the National Institute of Standards and Technology (NIST) and the US Department of Defense (DARPA) [18]. These benchmarks are continuously evolving and they now include retrieval from many different kinds of information (e.g., spam, legal texts, genomic data, extremely large datasets, *etc.*).

However, software systems present unique characteristics that make them different from standard IR domains. It is not possible to assume that IR techniques would work with similar performances if applied to software system artifacts, which form document collections that are orders of magnitude smaller than the standard IR benchmarks. Further, the elements involved are composed of software artifacts, which are usually written in a specific and terse technical language. These characteristics mean that the common assumptions made for usual IR collections should be revalidated in the field of software mining. For example, the LSI technique, which was proposed by Marcus *et al.*, can be used in this field, but not in standard IR applications, because it is known not to scale to large document collections [6]. Specific benchmarks for software engineering need to be devised.

Our goal is to evaluate the true effectiveness of several lightweight methods for linking source code to e-mails. To assess the quality of our findings, we not only present and discuss a set of techniques, but also created a statistically significant benchmark against which to verify them.

In the following, we define a benchmark for linking source code entities to e-mails, which can be used in future work for further analysis of different techniques, ranging from other lightweight approaches to more sophisticate and expensive IR methods. The obtained results can then be compared to show the strengths and drawbacks of several approaches on the same data set. We designed this benchmark to be easily extensible with additional data.

B. Infrastructure

We analyzed ArgoUML[1], a UML modelling tool written in Java, developed over the course of approximately 9 years, and made available under the BSD Open Source License. We consider the release 0.28 (March 2009) that comprehends 2,197 classes. We employed the lightweight approaches to map such classes to the related e-mails in ArgoUML mailing lists.

ArgoUML e-mails are stored in six mailing lists (see Table I), for a total amount of 79,175 messages. We excluded the *issues* and *commits* mailing lists from our experiment, because they contain messages automatically generated respectively by the bug tracking system and the revision control system. We also excluded the small *announce* list, because it presents only public announces of ArgoUML releases, and does not contain technical discussions.

Mailing list	Messages	Inception
dev	24,347	Jan 30 2000
issues	44,208	Feb 03 2000
users	4,456	Oct 19 2000
module-dev	221	Oct 28 2001
announce	41	Dec 30 2001
commits	5,902	Sep 28 2006

Table I
ARGOUML MAILING LISTS BY CREATION DATE

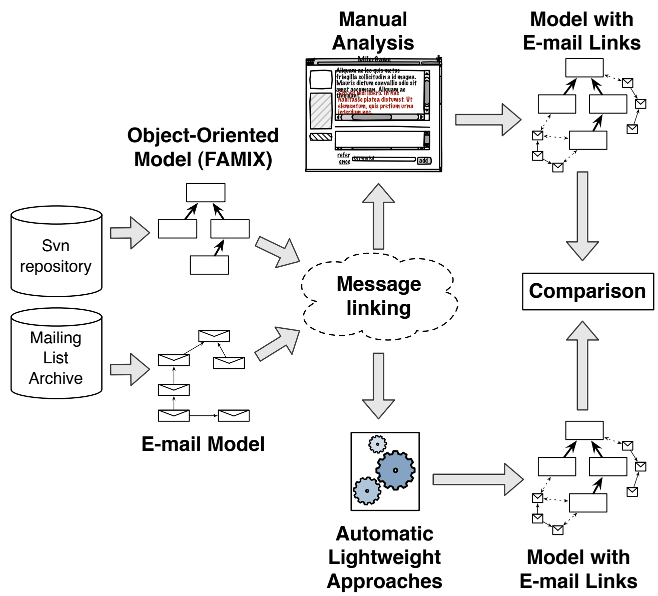


Figure 1. Infrastructure

Figure 1 details the infrastructure we set up for the experiment. First, using the tool iPlasma[2], we extracted the model of the ArgoUML release according to FAMIX, a language independent meta-model of object oriented code [7]. This model allowed us to easily extract all the information that is necessary for implementing the lightweight approaches (*i.e.,* class names and packages). Then, we extracted the data from the mailing list archives and structured it according to the e-mail model we built. Once the two models were ready, it was possible to conduct the message linking: This was done both by using the automatic lightweight approaches and by inspecting e-mails manually (*i.e.,* annotating e-mails using a web application developed for this task). This resulted in different object-oriented models of ArgoUML, annotated with e-mails. When both the manually created model (*i.e.,* the benchmark) and the automatically generated ones were ready, we compared them in order to validate the effectiveness of automatic methods.

[1]http://argouml.tigris.org/

[2]http://loose.upt.ro/iplasma/

E-Mails sample set size: To determine the size n of a sample set large enough to allow using the simple random sampling without replacement, we used the formula [20]:

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

Since previous work dealing with the same problem is unavailable, it is not possible to know *a-priori* the proportion ($\hat{p}$) of the e-mails referring a specific entity of the source code, thus we consider the worst case ($\hat{p} \cdot \hat{q} = 0.25$). As we are dealing with a relatively small population (*i.e.,* 29,024 messages in the mailing lists we used), the formula also considers its size (N). We took the standard confidence level of 95%, and an error (E) of 2%. This resulted in a value for the sample set size n of 2,218. However, in order to increase the significance of our benchmark, we enlarged this quantity to reach the final n value of 3,000, which corresponds to an error of 1.7%.

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 $f\% \pm 1.7\%$ of the population messages. This only validates the quality of this sample set as an exemplification of the population, and it is not directly related to the *precision* and *recall* values presented later.

Mailing list	Messages	Relative Size
dev	2,516	83.9%
users	467	15.5%
module-dev	17	0.6%

Table II
RANDOM E-MAILS BY MAILING LIST

Table II shows the number of random messages obtained from each mailing list. Although we decided not to use a stratified random sampling, but a simple random sampling, we note that the relative dimensions of the mailing lists are the same as in the population. This supports the hypothesis that the procedures used, both to determine the size of the sample set and to randomly extract elements from the mailing lists, led to a sample set that is as an accurate estimation of the entire population.

C. Benchmark building procedure

To evaluate the accuracy of automatic linking approaches, we manually built the benchmark (or *oracle*) by reading each e-mail and annotating them with the name of the classes they discuss. We developed a custom web application (depicted in Figure 2 and Figure 3) to assist this particular task.

The messages are presented in a randomly sorted list, with the sole distinction between analyzed and not yet analyzed messages (in Figure 2, point 2 and 1 respectively). After a message is chosen, its content and its header are displayed in a new page (Figure 3). If the e-mail is part of a larger discussion

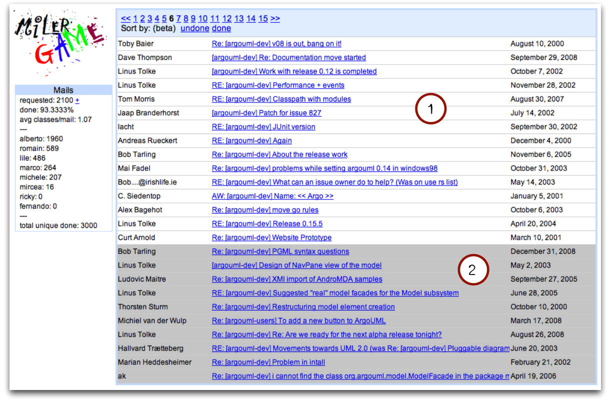


Figure 2. Web Application: List of e-mails

and there are quoted sentences from previous messages, these are coloured differently to improve the readability and, consequently, the quality of the analysis (Figure 3, point 1). In the same page, the analyzer finds the text-field that is used to add annotations. Entering a text into it triggers the auto-completion feature, which displays all the matching class names or packages of the ArgoUML system (Figure 3, point 2). The user can select the appropriate class/package and add it to the list of related entities for the displayed message (Figure 3, point 3). Completion ensures that users enter names of classes really existing in the system, also avoiding typos. When using this text-field, it is also possible to use regular expressions to retrieve a specific list of classes.

The readers annotated not only the classes that were explicitly mentioned in the e-mail body, but also those which were referred to using acronyms, abbreviations, or ArgoUML

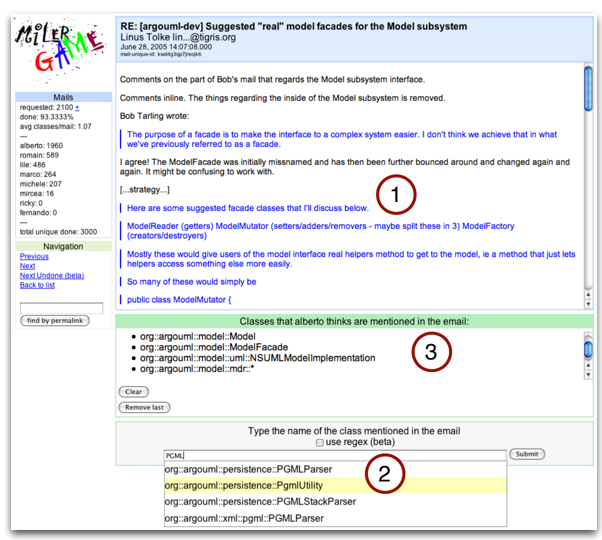


Figure 3. Web Application: E-Mail annotation

developers dialect (e.g., the infamous class NSUMLModelFacade [5] is often referenced using only the term "NSUML"). In addition, we let the reader annotate also the packages if they were mentioned in general. This allows the use of our benchmark for future approaches that would try to find the traceability links between e-mails and source code entities with a different granularity, or which aim to detect implicit references.

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 class names of the ArgoUML system. 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 real meaning of sentences.

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 database, from which they can be retrieved and exported.

Six members of the REVEAL research group, with several years of programming experience, inspected the sample set. The e-mails were randomly divided in overlapping sets, resulting in 17% of the messages analyzed by two people at least. A complete agreement was reached on 79% of these messages, with the remaining annotations featuring a slight degree of disagreement. Analyzing the conflicts, the most common difference was one of the two reviewer missing to annotate a link actually existing in the e-mail. We did not spot situations in which one of the reviewer erroneously annotated an unexisting link. For this reason, when two reviewers' annotations disagreed, we considered their union.

D. Evaluation

To compare the validity of our approaches against the benchmark, we used two well-known IR metrics: *precision* and *recall* [11], based on the following definitions:

- True Positives (*TP*): elements that are correctly retrieved by the approach under analysis (*i.e.,* links to source entities also present in the oracle)
- False Positives (*FP*): elements that are wrongly retrieved by the approach under analysis (*i.e.,* links to source entities not present in the oracle)
- False Negatives (*FN*): elements that are not retrieved by the approach under analysis (*i.e.,* links to source entities only present in the oracle)

Standard formulas for calculating *precision* and *recall* are:

$$Precision = \frac{|TP|}{|TP + FP|} \quad Recall = \frac{|TP|}{|TP + FN|}$$

The union of TP and FN constitutes the set of correct links present in the benchmark per e-mail, while the union of TP and FP constitutes the set of links retrieved by the used approach. In short, *precision* is the fraction of the retrieved links that are correct, while *recall* is the fraction of the correct links retrieved.

A number of e-mails in the benchmark have no references to source code entities, thus 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 automatic approaches not to find any link between an e-mail and source code. In this case, the *precision* value cannot be evaluated because the denominator in the corresponding formula is equals to zero. To overcome these issues, we first calculate the average of TP, FP, and FN, on the entire dataset. Then, we 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. A similar approach was used by Antoniol *et al.*, who encountered the same difficulty [1].

Precision (P) and recall (R) are two quantities that trade off against one another: Intuitively, it is possible to link each mail with all classes, reaching a recall value of 1, but a very low precision. For this reason, in order to measure such trade-off we added the *F measure*, which is the weighted harmonic mean of precision and recall:

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

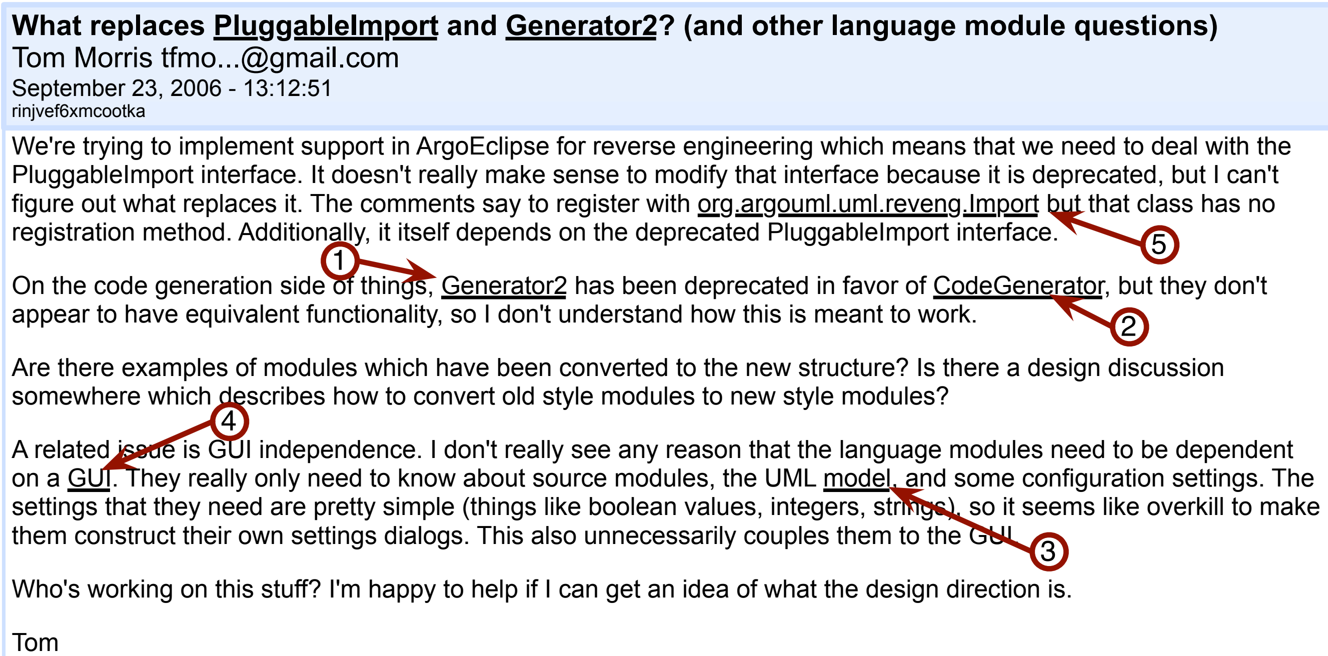
The weighting of precision and recall can be decided through the value of β. We decided to emphasize neither the recall nor the precision, because our approaches can be used in many different situations, and it is up to the engineer to select the most appropriate one. Thus, we prefer to give a general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

IV. Experiment

This section presents the results obtained applying different lightweight approaches. It begins illustrating techniques built on simple intuitions (*i.e.,* the match on the name of classes, not case sensitive), then it proceeds to others based on more sophisticated ideas (*i.e.,* mixed approaches based on regular expressions).

Each technique processes one class at a time: First it extracts the necessary information (*e.g.,* class name and package) from the FAMIX model of the system, then it prepares the matching procedure, and finally uses it against each e-mail content (*e.g.,* searching for the class name).

We consider an e-mail taken from the sample set (see Figure 4) to show how the different approaches work in practice.



What replaces PluggableImport and Generator2? (and other language module questions)
Tom Morris tfmo...@gmail.com
September 23, 2006 - 13:12:51
rinjvef6xmcootka

We're trying to implement support in ArgoEclipse for reverse engineering which means that we need to deal with the PluggableImport interface. It doesn't really make sense to modify that interface because it is deprecated, but I can't figure out what replaces it. The comments say to register with org.argouml.uml.reveng.Import but that class has no registration method. Additionally, it itself depends on the deprecated PluggableImport interface.

On the code generation side of things, Generator2 has been deprecated in favor of CodeGenerator, but they don't appear to have equivalent functionality, so I don't understand how this is meant to work.

Are there examples of modules which have been converted to the new structure? Is there a design discussion somewhere which describes how to convert old style modules to new style modules?

A related issue is GUI independence. I don't really see any reason that the language modules need to be dependent on a GUI. They really only need to know about source modules, the UML model, and some configuration settings. The settings that they need are pretty simple (things like boolean values, integers, strings), so it seems like overkill to make them construct their own settings dialogs. This also unnecessarily couples them to the GUI.

Who's working on this stuff? I'm happy to help if I can get an idea of what the design direction is.

Tom

Figure 4. An e-mail from the sample set

A. *Class name, case insensitive*

Intuition: The simplest way to reference a class from an e-mail is to mention it using its name. For example, in Figure 4 point 1, we see that the class *Generator2* is mentioned using only the name. Since it would be possible for participants of an electronic discussion not to use uppercase letters (as the context clarifies if they are discussing about a class), in this approach we decided to ignore the case.

In addition, the class name could be used inside quotation marks (*i.e.,* "classname"), parenthesis or other characters. In Figure 4, point 2, the name of the class *CodeGenerator* is followed by a comma. For this reason, we did not impose any restriction on the characters surrounding the class name.

Implementation: In this case, the implementation consists in extracting all the class names from the model of the system and verifying which class names appear in the e-mail content. When traversing the e-mail content looking for class names, the implementation takes into consideration neither the case of the class name nor characters that precede or follow it.

Results: Precision 0.09, Recall 0.70, F-Measure 0.16: The most interesting result of this simple match is the recall value: It reaches a significant value of 0.70. The trade-off is a low precision, due to the many false positives.

Considering Figure 4 point 3, we note that the word "Model" does not refer to the class *org.argouml.model.Model*, as this simple approach wrongly assumes. This is one of the many examples that make this approach generating false positives.

All the matching techniques that will follow use this simple class name match as the first step for their implementation: They all consider the class name (extracted from the FAMIX model) and require it to be present in one single word in the e-mail content. For this reason, the value reached with this first approach is the upper bound of the recall.

B. *Class name, case sensitive*

Intuition: In this case the match exploits the case sensitivity of modern object oriented languages. It is a widely used and accepted convention to define class names starting with a capital letter and using *media capitals* (*e.g.,* "ClassName"), also known as CamelCasing. From the usage of this approach, we expected a slight increase of the precision, without any major decrease of the recall value. For example, in Figure 4 point 3, the word "model", whose first letter is not capital, is not going to be considered as a valid reference to the class*org.argouml.model.Model*. At the same time, the references to *CodeGenerator* and *Generator2* are going to be preserved.

Implementation: After extracting the class names from the FAMIX model, the implementation searches for them in every e-mail and reports a link when all the characters of a class name are sequentially present in one single word, respecting the capital letters.

Results: Precision 0.33, Recall 0.69, F-Measure 0.46: As expected, the recall value did not decrease significantly. On the contrary, the simple additional case sensitivity check greatly increases precision (*i.e.,* by 24%). The number of false positives dropped, while the number of good links not retrieved was almost as high as in the previous approach.

This result points out that class names are mainly mentioned respecting camel casing. This simple check thus helps to separate common words of discussions from true references to source code entities.

One of the false positive created by this approach is the one marked with point 4 in Figure 4. The word "GUI" is not a reference to the class *org.argouml.ui.GUI*, on the contrary the author is writing about a component of the ArgoUML application, from a user point of view. Also, the class *Generator* is wrongly recognized as being referenced, because its name is part of the word "Generator2" or "CodeGenerator" (Figure 4 points 1 and 2, respectively).

C. Strict regular expression, case insensitive

Intuition: We also wanted to find a method that could give an indication of the upper bound for the precision. To achieve this, we took into consideration other intrinsic characteristics of class entities, in addition to the name.

First, java class entities are stored in files that have particular extensions: "java" for the source code, or "class" for the bytecode. For this reason, class names can be followed by those extensions. If extensions are not used, after the name there must be some empty space, otherwise the string can be part of another class name, resulting in a false positive (*e.g.,* the string "Model" matches a class named *Model*, but also a class named *ModelFacade*).

Java classes are also always part of a package (*e.g.,* "org.argouml") and, if two classes share the same name, they must be in two different packages. In many cases the package name is cited in the e-mail before the class name. We decided to take into account the package, however, to avoid amplifying the decrease in the recall value, we decided that only the last part of it has to be present before the name (*e.g.,* "argouml"). Before this part there can be either the rest of the package or some empty space. Possible packages separator are ".", "\" and "/".

Finally, as we imposed many other constraints, we decided not to use the case sensitivity, to preserve the recall value. In point 5 of Figure 4, we note that the only class that could be detected by this method in that specific case. However, it is not going to produce any false positives.

Implementation: To implement this approach, we make use of a simple regular expression. First, we extract the name and the package of every class from the FAMIX model; then, we use them inside the regular expression, setting all the constraints we want to impose. The resulting regular expression code, according the IEEE POSIX Basic Regular Expressions (BRE) standard, follows[3] (the parts in angle brackets are replaced with the class information retrieved from the FAMIX model):

[3]For convenience the text is in multiple lines.

```
(.*)
(\s*)
(<beginning of package>)?
(.|\\|/)
<last part of package>
<class name>
(.java|.class|\s+)
(.*)
```

Results: Precision 0.94, Recall 0.10, F-Measure 0.18: Using this strict match, the recall value radically changes and reaches a very low value. On the other hand, as we expected, the precision reaches a top value. Due to the small number of links that this approach retrieves, a very few false positives (*i.e.,* 11 for the whole data set) are sufficient to lower the precision from 1 to 0.94. Those false positives are caused by classes that are not in the ArgoUML model, but have the same name and last part of package, so they are recognized as references with this approach. The F-Measure value is as low as in the match based on the simple class name approach, not case sensitive. The results are identical whether using case sensitivity or not.

D. Loose regular expression, case sensitive

Intuition: Starting from the result of the previous approach, to raise the recall value again we tried to loosen the strictest constraint: We decided not to require the presence of the last part of the package, before the class name. At its place, there can be some empty space or the complete package. Moreover, we specified that it was also possible to have quotation marks, or a comma, after a class name, as well as the others constraints applied in the previous approach.

Since the requirements were less strict, we decided to use the case sensitivity to preserve the precision. From this we did not expect a serious decrease in the recall, which did not lower significantly, when case sensitivity was used in the second method we experimented. Considering the e-mail in Figure 4, this approach is going to correctly recognize classes marked by points 1, 2 and 5.

Implementation: As we did for the previous approach, we implement this technique using a regular expression and class names and packages extracted from the FAMIX model. The regular expression code follows:

```
(.*)
(\s*)
(<package>)?
(.|\\|/)
<class name>
(.java|.class|\s+|"|,)
(.*)
```

Results: Precision 0.45, Recall 0.54, F-Measure 0.49: Results show that the recall gained up 42%, while precision lost 49%. The F-Measure, which is slightly higher than the match on the class name case-sensitive, points out that this method is the best choice, up to now, if recall and precision are considered equally important.

One of the false positives, considering Figure 4, is the word marked by point 4. The string matches the regular expression, however the author is not discussing a class named “GUI”.

E. Mixed, using Dictionary, case sensitive

Intuition: We suppose that the high number of false positives in approaches which used the match on only the class name (case sensitive and not case sensitive) was caused by class names which were equal to common words that can be found in a dictionary, such as the classes *Dialog*, *Text*, and *Critic*. Under this assumption, we tried to mix the most optimistic methods with the strictest one, using a dictionary to select the right approach: For each class, we searched for its name in a common English dictionary of more than 2 millions words. If the string was present in the dictionary (*e.g.,* the word “Model”, as marked by point 3 in Figure 4), then we used the strict regular expression approach. Otherwise, if the string was not in the dictionary (*e.g.,* the word “CodeGenerator”, marked by point 2 in Figure 4), then we switched to the simple class name match.

Implementation: The first step of this approach is still to extract class names and packages from the model of the ArgoUML system. As usual, we consider one class at a time: When the class name is present in a common English dictionary, we search for the presence of links inside e-mails using the strict regular expression (see Section IV-C), otherwise we apply the simple matching on the class name, case sensitive (see Section IV-B).

Results: Precision 0.57, Recall 0.62, F-Measure 0.60: To obtain a stronger comprehension of the results obtained, we also tried to use the simple matching on the class name not case sensitive (see Section IV-A), in place of the case sensitive one. The results are: precision 0.20, recall 0.64. The difference in precision, related to the usage of case sensitivity, is evidence of the fact that the most cited classes are not part of the dictionary. In fact, we recall that the simple match on the class name is only used when the name is not on the dictionary, otherwise we apply the strict regular expression, which is not influenced by case sensitivity.

The F-measure shows that this approach, which makes use of a dictionary to select the severity of the match, is the most effective until now.

F. Mixed, using CamelCase, case sensitive

Intuition: Classes usually represent abstractions of real-world categories of objects. For this reason, it is common practice to give them names from common dictionary words. However, since empty spaces are not allowed in class names, whenever a class represents a concept which is clearer if named with two, or more, words, those are compounded using *media capitals*.

The intuition of this approach is that class names, which are formed by compounded words, are not part of a common dictionary word. For example, the class name *ExplorerTree* is formed by the two dictionary words “explorer” and “tree”, but their composition is not a dictionary word.

For this reason, we supposed that it was possible to exclude the dictionary to select most appropriate match. If the class name was a single word, then it was probably a dictionary word (*e.g.,* the class *Trash* has a name which is in the dictionary). On the other hand, a name made compounding words is probably not in the dictionary. To distinguish compounded and simple words, it is sufficient to check for the presence of *media capitals*.

Implementation: The implementation of this algorithm is similar to the previous one, but, since it does not require a dictionary, it is faster and requires less memory. For each class, when its name is a compounded word, we employ the simple match on the name, case sensitive (see Section IV-B), otherwise we use the strict regular expression matching (see Section IV-C). We keep the search for *media capitals* as simple as possible, and we define a word as compounded if it contains more than one capital letter. In this manner, cases like the string “PGMLParser” are correctly marked as compounded, even though the letters before the character “P”, of the word “Parser”, are all in capital case.

Results: Precision 0.63, Recall 0.62, F-Measure 0.62: The main goal of this approach was to lighten the previous one removing the dictionary search. However the results show that, in addition to performance, the precision value increased, without any loss in the recall value.

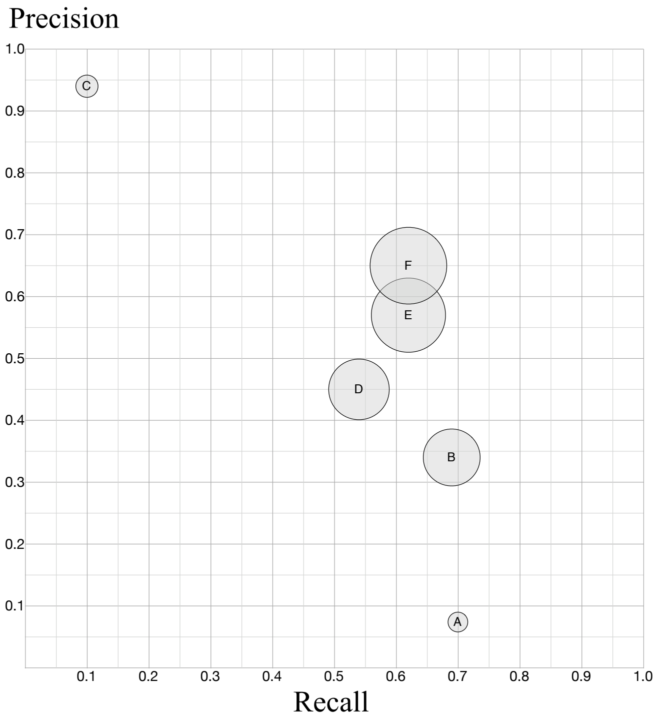
These results can be explained through a few examples. First, there are class names which are not in a simple English dictionary, but are common computer science terms: *Parser* is one of such terms. The previous approach did not find the string “Parser” in the dictionary, thus it wrongly treated it with the most optimistic match. On the contrary, this new approach found only one capital character and correctly switched to use the strictest match.

Second, there are class names that are part of dictionary words. For example, the class *Init* has a name that is part of many dictionary words (*e.g.,* “Initialization”, “Initial”), thus the most optimistic methods (which does not impose any restriction on characters surrounding the class name) finds wrong matches. On the contrary, this new method correctly treats such words, which have only one capital letter, using the strictest approach.

Finally, classes often have names that correspond to high level design concepts. For example, “Modeller” is a class name, and a concept in the ArgoUML dialect. It is possible to find it mentioned both as a high level design concept (not referring the class with the same name) and as a class of the system. Also in this case, this new approach takes the right decision of treating this case with the strictest matching.

The F-Measure shows that this method is the most effective among all the approaches we implemented.

V. DISCUSSION



	Approach	Recall	Precision	F-measure
A	Class name, case insensitive	0.70	0.09	0.16
B	Class name, case sensitive	0.69	0.33	0.46
C	Strict regular expression, case insensitive	0.10	0.94	0.18
D	Loose regular expression, case sensitive	0.54	0.45	0.49
E	Mixed, using Dictionary, case sensitive	0.62	0.57	0.60
F	Mixed, using CamelCase, case sensitive	0.62	0.63	0.62

Figure 5. Precision, Recall and F-measure of all approaches.

In Figure 5 we have summarized the results of all approaches. The case insensitive class name approach (A) represents the upper bound for the recall (*i.e.,* 0.70), as all the other approaches we used, as the first step, check for the presence of the class name. Then, they refine the precision using different techniques, which necessarily reduces the recall. We believe that this recall upper bound (*i.e.,* 70% of all the correct links are retrieved) was high enough for our needs. However, it is still possible to implement lightweight techniques to raise it. For example, during the benchmark creation, we noted that, sometimes, class names that are made of compounded words are mentioned in e-mails using those words separately. This aspect can be implemented through more refined regular expressions. Heavyweight approaches may find implicit references, thus increasing the recall.

On the opposite extreme, there is the case insensitive strict regular expression approach (C), which reaches top precision but retrieves few links. From the results of the two mixed approaches (E and F), the strict approach was useful to raise the precision value when used together with the case sensitive class name approach.

The two mixed approaches give the best results: They are not only capable of retrieving 62% of all the correct links between e-mails and source code entities, but they also completed this task with the same significant level of precision (*i.e.,* 63% of the links retrieved were true positives).

The lightweight approaches, apart from being simple to implement, are also fast: The slowest method (E), when operating on a laptop with a 2.4GHz Intel Core Duo processor and 4GB of RAM - using our implementation developed in Cincom Smalltalk - required ca. 3 seconds to verify the presence of a link to a class in all the mailing lists we used (*i.e.,* 29,024 e-mails).

Limitations: The main limitation of our experiment is caused by the fact that it was performed considering only one system: ArgoUML. The style of the discussions in different mailing lists can vary, potentially changing the results achieved. However, in our approaches we did not rely on the specific dialect of the ArgoUML developers to avoid reducing the generality of our results (*e.g.,* we did not match *NSUML* directly to the class "NSUMLModelFacade"). On the contrary, we exploited naming conventions that are common in all the Java systems. We plan to analyze systems that use different programming languages and naming conventions, include them in our benchmark, and then make it publicly available.

The REVEAL research group accurately analyzed the sample set of e-mails, which was used to create the benchmark. However, they are not ArgoUML developers, thus it is possible that in some cases they did not manage to understand implicit references to classes. Moreover, as they are human beings, the possibility that they made mistakes in the analysis should be taken into account. To avoid this last problem, we manually revised all the false positives generated by the most strict approaches, in order to see whether they were "true" false positive or if they were caused by human errors. Only two of them were "wrong" false positives, which we corrected before re-evaluating all the results. It is reasonable to think that such errors do not threaten the validity of the benchmark.

VI. CONCLUSION

We created a statistically significant and easily extensible benchmark to assess the quality of approaches to find traceability links between source code and e-mails. The benchmark can be used to test new approaches and verify if they are able to improve the results obtained in our experiments. The benchmark can also be exploited to train learning algorithms, such as the unsupervised word sense disambiguation algorithm proposed by Yarowsky [22], which is capable of distinguishing the sense of ambiguous words from their context. In our case, it could differentiate between words referring to class names and their other usages.

We devised and evaluated several lightweight methods to find traceability links between source code and e-mails.

Exploiting characteristics and naming conventions of software artifacts, we showed that such lightweight approaches can reach significant results in terms of accuracy. This indicates that heavyweight techniques may be not necessary to achieve good results in finding the traceability links between source code and e-mails. In addition, since our approaches are lightweight, they do not require pre-computation (which usually is time-expensive and compromises dynamism and interactivity of applications), but can be directly used at *run-time* (*e.g.,* to catalogue an e-mail with its references, or to check for the references to one class in all the e-mails archive). This allows the developer to choose every time the most convenient method.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063). We thank the members of the REVEAL research group for the effort they put in the creation of the benchmark, and Matteo Bertoni who provided advice on the creation of a statistically significant sample set.

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] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability for object-oriented systems. *Annals of Software Engineering*, 9(1-4):35–58, 2000.

[3] 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.

[4] J. Conklin and M. L. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. In *Proceedings of CSCW 1988 (3rd ACM conference on Computer-supported cooperative work*, pages 140–152. ACM, 1988.

[5] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *Transactions on Software Engineering (TSE)*, x(x):xxx–xxx, Feb. 2009.

[6] 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.

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

[8] R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *Proceedings of ICSM 1998 (14th IEEE International Conference on Software Maintenance)*, pages 94–102. IEEE Computer Society, 1998.

[9] 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.

[10] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of ICSM 2004 (20th IEEE International Conference on Software Maintenance)*, pages 306–315. IEEE Computer Society, 2004.

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

[12] A. Marcus, A. D. Lucia, J. H. Hayes, and D. Poshyvanyk. Working session: Information retrieval based approaches in software evolution. In *Proceedings of ICSM 2006 (22th IEEE International Conference on Software Maintenance)*, pages 197–209. IEEE CS Press, 2006.

[13] A. Marcus and J. I. 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 Computer Society, 2003.

[14] 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.

[15] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.

[16] B. Ramesh and V. Dhar. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, 1992.

[17] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of ICSE 1996 (18th international conference on Software engineering)*, pages 387–396. IEEE Computer Society, 1996.

[18] 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.

[19] M. Smith, D. Weiss, P. Wilcox, and R. Dewar. The OPHELIA traceability layer. In *Cooperative Methods and Tools for Distributed Software Processes (2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes)*, pages 150–161. FrancoAngeli, 2003.

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

[21] K. C. B. Yakemovic and E. J. Conklin. Report on a development project use of an issue-based information system. In *Proceedings of CSCW 1990 (5th ACM conference on Computer-supported cooperative work)*, pages 105–118. ACM, 1990.

[22] D. Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of ACL 1995 (33rd Annual Meeting of the Association for Computational Linguistics)*, pages 189–196. Association for Computational Linguistics, 1995.

[23] W. Zhao, L. Zhang, L. Yin, L. Jing, and J. Sun. Understanding how the requirements are implemented in source code. In *Proceedings of APSEC 2003 (10th Asia-Pacific Software Engineering Conference*, pages 68–77. IEEE Computer Society, 2003.