Text is Software Too

Alexander Dekhtyar
Dept. Computer Science
University of Kentucky
dekhtyar@cs.uky.edu

Jane Huffman Hayes
Dept. Computer Science
University of Kentucky
hayes@cs.uky.edu

Tim Menzies
Dept. Computer Science,
Portland State University,
tim@menzies.us

Abstract

Software compiles and therefore is characterized by a parseable grammar. Natural language text rarely conforms to prescriptive grammars and therefore is much harder to parse. Mining parseable structures is easier than mining less structured entities. Therefore, most work on mining repositories focuses on software, not natural language text. Here, we report experiments with mining natural language text (requirements documents) suggesting that: (a) mining natural language is not too difficult, so (b) software repositories should routinely be augmented with all the natural language text used to develop that software.

1 Introduction

"I have seen the future of software engineering, and it is.....Text?"

Much of the work done in the past has focused on the mining of software repositories that contain structured, easily parseable artifacts. Even when non-structured artifacts existed (or portions of structured artifacts that were non-structured), researchers ignored them. These items tended to be "exclusions from consideration" in research papers.

We argue that these non-structured artifacts are rich in semantic information that cannot be extracted from the nice-to-parse syntactic structures such as source code. Much useful information can be obtained by treating text as software, or at least, as part of the software repository, and by developing techniques for its efficient mining.

To date, we have found that information retrieval (IR) methods can be used to support the processing of textual software artifacts. Specifically, these methods can be used to facilitate the tracing of software artifacts to each other (such as tracing design elements to requirements). We have found that we can generate candidate links in an automated fashion faster than humans; we can retrieve more true links than humans; and we can allow the analyst to participate in the process in a limited way and realize vast results improvements [10, 11].

In this paper, we discuss:

- The kinds of text seen in software;
- Problems with using non-textual methods;
- The importance of early life cycle artifacts;
- The mining of software repositories with an emphasis on natural language text; and
- Results from work that we have performed thus far on mining of textual artifacts.

2 Text in Software Engineering

Textual artifacts associated with software can roughly be partitioned into two large categories:

1. Text produced during the initial development and then maintained, such as requirements, design specifications, user manuals and comments in the code;
2. Text produced after the software is fielded, such as problem reports, reviews, messages posted to on-line software user group forums, modification requests, etc.

Both categories of artifacts can help us analyze software itself, although different approaches may be employed. In this paper, we discuss how lifecycle development documents can be used to mine traceability information for Independent Validation & Verification (IV&V) analysts and how artifacts (e.g., textual interface requirements) can be used to study and predict software faults.

3 If not text..

One way to assess our proposal would be to assess what can be learned from alternative representations. In the software verification world, reasoning about two represenations are common: formal models and static code measures.

A formal model has two parts: a *system model* and a *properties model*. The system model describes how the program can change the values of variables while the properties model describes global invariants that must be maintained when the system executes. Often, a temporal logic[1] is used

[1] Temporal logic is classical logic augmented with some temporal operators such as $\Box X$ (always X is true); $\Diamond X$ (eventually X is true); $\bigcirc X$ (X is true at the next time point); $X \bigcup Y$ (X is true until Y is true).

to express the properties model. Modern model checkers such as SPIN [15] search the systems model for a method of proving the negation of the properties model. The cost of formal modeling includes the *writing cost*, the *running cost*, and the *rewriting costs*. The writing cost has two components. Firstly, there is a short supply of analysts skilled in creating temporal logic models. Secondly, even when analysts with the right skills are available, the writing process is time-consuming.

Another significant cost of formal modeling is the *running cost* of model checking. A rigorous analysis of formal properties implies a full-scale search through the systems model. This space can be too large to explore, even on today's fast machines. Much of the research into formal modeling focuses on how to reduce this running cost of model checking. Various techniques have been explored but none are panaceas. For example, optimisations based on clustering (e.g. [3]) generally fail for tightly connected models. Consequently, in the general case, classic formal methods do not reduce the effort of testing a system. However, for the kernel of mission-critical or safety-critical systems, the large cost of formal methods is often justified.

At the other end of the spectrum from model-rich formal modeling are defect measures based on syntactic static code measures such as the Halstead [7] or Mccabe [18] metrics. Such static code measures are a weak *primary* method of finding errors. Such metrics are collected on a module-by-module basis. Hence, they know neither: (a)how often that module will be called, nor (b)the severity of the problem resulting from the module failing, nor (c)the connections *from* this module *to* other modules.

However, static code measures are adequate *secondary* detectors that can audit the results of primary methods. Elsewhere, we have shown that such detectors are stable across multiple projects and can be selected such that they have a very low probability of false alarms [19].

Nevertheless, the current situation is as follows. Complex comprehension methods such as formal modeling can be too complex for many applications. Simpler methods such as static code measures may only be suitable for *augmenting* other methods. In neither case do we possess methods that are both very insightful and widely applicable. We are hence motivated to work on other methods.

This understanding has been demonstrated by several other researchers including Di Lucca, Di Penta and Gradara [5], who have examined the problem of classifying, via a variety of different algorithms, textual maintenance requests into eight categories. Their best result, using support vector machines (SVM) [16], was 84% accuracy. Lee and Bryant [17] examined the problem of formalizing natural language requirements specifications unsing Natural Language Processing (NLP) techniques. Thus, we observe that in recent years, information retrieval and text mining methods are starting to be applied to adress Software Engineering problems.

4 Possibility

So, what can we do if we add text to software repositories? Here, we discuss two different problems that can be addressed in such a manner: fault analysis and requirements tracing.

4.1 Fault Analysis

Barry Boehm's seminal work in software engineering economics convinced us that faults found early in the lifecycle are less expensive and time consuming to correct [2]. We have worked as practitioners and researchers in the area of verification and validation for over twenty years, and we have seen this confirmed many times. In fact, we are convinced that faults found early in the lifecycle can serve as predictors of faults that will be found later in the lifecycle. We further believe that textual analysis can assist. For example, an unsatisfied high level requirement (one that does not have design elements to satisfy it) may lead to a missing capability in the as-built software product.

Evidence of such fault links (the relationship of one fault to another) was presented by Hayes and Offutt [9, 12, 13]. Specifically, high-level interface requirements (textual) were examined using a technique called input validation analysis. Faults in the interface requirements were identified as well as potential faults (ambiguities, for example). Test cases were generated on the basis of these early life cycle faults. The test cases were then executed on the as-built software and 13% of these revealed later life cycle faults in the delivered product. Hayes postulated that these early life cycle faults were late life cycle predictors for two reasons: developers tend to make the same kinds of mistakes, regardless of the life cycle phase; and faults do not get repaired early in the life cycle and are detected later (latent defects) [9].

Knowing that faults caught early in the life cycle are easier and less costly to repair AND can assist us in predicting and discovering later life cycle faults is a call to action. We should fully explore techniques that allow us to analyze early lifecycle artifacts for such faults. We should not be dissuaded from our duty by the existence of textual narrative in these early artifacts.

4.2 Requirements Tracing

Requirements tracing, a bane of Independent Verification & Validation (IV& V) analysts, is a prolonged, tedious, but *incredibly important* task of making sure that all initial software requirements have been adequately reflected in the design specifications for the software, and, eventually, in the code. Traditional approaches to requirements tracing ivolve repeatedly going through hardcopies of requirements documents, building and manually maintaining spreadsheets, or, at best, using requirements management tools that allow manual assignment of keywords to requirements and use simple keyword matching algorithms to find candiate links. Such procedures reflect the nature of the task: requirements documents are written in natural, if somewhat

more bureaucratically formal, language. So far, human cognitive powers are unmatched in detecting correspondence between two (or more) text fragments: requirements and design elements, for example.

The only reason why, up to this day, such procedures are still employed is the relatively small size of the documents under consideration for the requirements tracing task. Even then, large projects have thousands of requirements and, potentially, tens of thousands of design elements: approaching the limits of what IV& V analysts are prepared to suffer through without extra help.

At the same time, in the core of the requirements tracing task, lies a problem well-known to computer scientists and, in fact, well-studied by them: *given a document collection, and a document (query) find all such documents in the collection that are similar (relevant) to it.* This problem, addressed by decades of intensive research in Information Retrieval (IR), is becoming ubiquitous, at the very least for those of us who use Internet on a daily basis. And our ability to search for, *and find*, information in the pits of the World Wide Web only attests to the success of Information Retrieval in dealing with this problem.

Thus, we have reached the conclusion that by taking the low level requirements (design elements) to be the document collection, and by treating high level requirements as queries, we can use the vast array of IR algorithms (see [1] for the starting point) to produce lists of candidate links for the requirements traceability matrix. Our preliminary experiments, reported in [10, 11] showed that:

- automated means of generating candidate links work much faster than humans (even when humans are assisted by existing requirements management software);
- automated means of generating candidate links retrieve more *true links* than the human analyst/requirements management software combination;
- automated means of generating cadidate links tend to report *more false positives* than human analyst/requirements management software combination;
- analyst participation in the process, as the validator of the candidate links, is still important.

Following this work, we have implemented additional IR algorithms, and incorporated *user feedback analysis* into the system, making the requirements tracing process interactive again and giving human analysts the last word in determination of the links. At the same time, feedback analysis has shown the ability to improve significantly both the *recall* (percentage of true links found) and *precision* (the measure of the signal-to-noise ratio in the list of candidate links), especially when combined with techniques for filtering outputs produced by our IR methods [11]. This lead to creation of RETRO (REquirements Tracing On-target), a standalone, IR-based requirements tracing tool for IV& V analysts [11].

In Table 1, we briefly summarize RETRO's achievements to date. The two main metrics of success of an IR task that are applicable to the requirements tracing problem itself are

Method	Precision	Recall
STP	38.80%	63.41%
Analyst+STP	46.15%	43.9%
TF-IDF	11.3%	57.3%
TF-IDF+ Feedback	18.6%	**76.2%**
TF-IDF+ Feedback+Filter	**60.9%**	59.5%
TF-IDF+ Thesaurus	12.2%	**64.2%**
TF-IDF+ Thesaurus + Feedback	18.1%%	**83.3%**
TF-IDF+ Thesaurus + Feedback + Filter	**39.5**%	**80.9%**
	73.8%	**73.8%**
LSI (10 dim, 0.32 coverage)	5%	**90.4%**
LSI (10 dim, 0.32 coverage)+ Thesaurus	5%	**92.85%**
LSI (40 dim, 0.92 coverage)	5%	**80.95%**
LSI (40 dim, 0.92 coverage)+ Thesaurus	5%	**85.71%**

Table 1. Using information retrieval to trace requirements.

recall, the percentage of all true links retrieved, and *precision*, the percentage of true links in the answer set. Recall measures coverage, while precision measures signal-to-noise ratio. The top two rows in Table 1 show the precision and recall obtained from a commercial requirements management tool, SuperTracePlus (STP) [8, 20], and from a senior analyst working with the output of the SuperTracePlus on a simple test set consisting of 19 high-level and 49 low-level requirements. The remaining rows show how different methods that we have implemented in RETRO fare on the same test. TF-IDF is a standard [1] IR method that computes similarity between documents as the cosine of the angle between their vector representations. This method has been enhanced with user feedback [1], various filtering techniques [11], and a simple thesaurus [11]. LSI stands for "Latent Semantic Indexing", a dimension reduction technique that proved to work very well on small document collections [4].

In each row we report the best, in our opinion, combination of recall and precision that was obtained during the experiments (for TF-IDF+Thesaurus+Feedback+Filter we report two different results achieved). As can seen from the table, most of the methods tested within RETRO consistently outperform humans and STP in recall. At the same time, these methods trail in precision, a feat that can be corrected by the use of filtering techniques at the price of some decrease in recall. In general, our findings today show that there is significant potential in the use of well-established IR methods for analyzing textual artifacts.

5 But What's the Price?

All things considered, *textual artifacts* are less well-understood than *code*. It stands to reason that the analysis of such artifacts must be conducted with more complicated algorithms, than the analysis, even mining, of code alone. At the same time, we should also be prepared for the taste of failure: not all methods of text analysis will work in our settings. Our experience with the use of IR algorithms for requirements tracing lead us to discover the following specific features of mining software-related text:

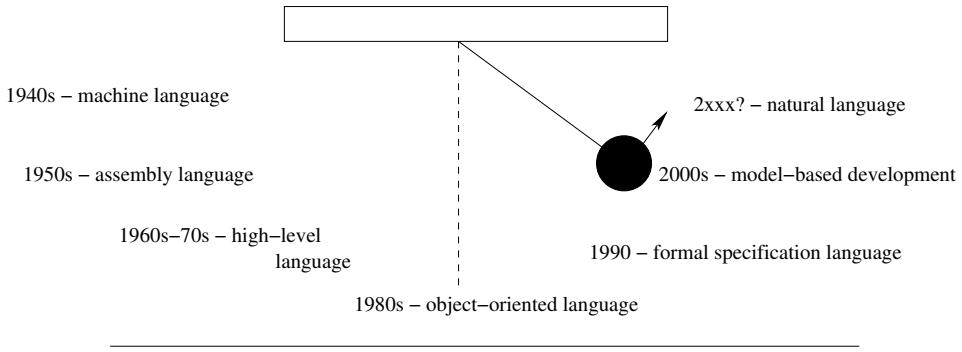

Figure 1. The Artifact Pendulum Swings from Structure to Less Structure.

- domain size: traditional IR algorithms represent documents as vectors of keyword weights. Such methodology works very well when the document collection is large enough to approximate the real use of different terms in English. Thus, the methods of determining the importance of a keyword for a document that work extremely well when there are billions of documents in the collection, at times, have strange effects when the number of documents is in tens or hundreds.
- document size: traditional IR algorithms assume that the individual documents contain significant text. Most of traditional test collections for IR algorithms [14] use documents that have on average more than one hundred words. At the same time, it is not unusual for a requirement to consist of one or two simple sentences. That is to say, the fewer the words in the document, the fewer keywords detected.
- incomplete vocabulary: requirements documents are, very often, written in a very specific lingo. Combined with the relatively small number of requirements, it makes the vocabularly of the entire collection, both high- and low- level requirements, incomplete, and sometimes, different from the traditional English usage vocabularly (in terms of frequency of use of words). Thus, some words, that are treated as almost stopwords[2] elsewhere, may suddenly give the appearance of very important keywords, only because they are used in only one or two requirements.
- recall vs. precision: typically, both recall and precision are equally important. However, their roles are drastically different in requirements tracing tasks. Recall appears to us as more important as, at the end of the day *all* matching requirement pairs must be found. Precision plays a role of a "filter": it determines how many false positives will be examined by the human analyst. Very low precision, makes automated candidate link generation useless, however, while our goal is always 100% recall, precision of even 40%–50% is excellent (analyst has to examine about one false positive per true link) as it drastically reduces human effort as compared to the manual process.

All of this leads to the observation that while we should attempt to take as much advantage of already designed information retrieval, text mining and/or natural language processing methods, we should also be ready to: (a) accept unapplicability of some of them to specific problems, and (b) not only *adopt* but ***adapt*** them to the needs and features of these problems.

6 Conclusion

One could argue that the software engineering artifact pendulum has been swinging from more formal, structured and parseable means of describing software to more text-based ever since the inception of the discipline. As can be seen in Figure 1, the beginning of it all was the pleasant-to-parse machine language (top left). Wise pioneers of our field realized that the price was too high for the poor human programmers, and came up with assembly language... and the pendulum came into motion. High-level procedural languages that came next attempted to record the algorithm rather than its direct execution by the computer. Assembly command abbreviations were replaced with keywords, control structures and identifiers of (practically) arbitrary length. Then, in the 1980s, we decided that an even higher level of abstraction was needed: the ability for developers to think of things in terms of objects.

All this high level thinking, however, had not been leading to drastically better software. In fact, requirements were just as poorly specified by software engineers using UML and use cases and other "texty" artifacts that came along with object oriented techniques.

The next step (the pendulum starts going up) lead to formal specification languages. The potential of these languages cannot be denied. The ability to parse such artifacts and even use specification provers to ensure that

[2]Stopwords are words that are not considered to be keywords: articles, prepositions, pronouns, modal verbs, and some common verbs and nouns (such as "be", "get", "thing", "stuff"). In addition, special collections of documents may have extra stopwords, e.g., "software" is a stopword in a collection of Software Engineering papers.

source code implements a formal specification is powerful indeed. Formal methods promise automatic verification and automatic generation of demonstrably correct code. However, the experience with such tools to date is not positive. Ph.D.-level mathematical skills may be required to specify the knowledge required for such tools [21]. Commercial practitioners may lack either the required training or the required time needed for such specification.

So the pendulum continues to rise. Other researchers have argued for lightweight modeling languages with formal semantics (e.g. [6]). Here, we propose something different. After decades of research, we have evidence that *mere* text can be more useful than previously believed. Our recommendation is that when repositories are built, we should always include all available text artifacts.

Text mining from software engineering text is a high risk, high return adventure. The translation steps from high level artifacts, such as concept documents and high level requirements statements, to low level implementation, such as source code, inject a tremendous amount of variance into the final artifacts. At the same time, it is precisely this variance that hurts software development process, especially the validation and verification part of it. Thus, we maintain that *achieving a better understanding of how text turns into code* will lead to improved software.

Acknowledgements

This research was conducted at West Virginia University, Portland State University, and the University of Kentucky under NASA contracts NCC2-0979, NCC5-685, NAG5-11732 and NNG04GA38G. The work was partially sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, Addison-Wesley, 1999.

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] P. Clark and T. Ng. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[4] S. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41(6):391–407, 1990.

[5] G. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *Proc., International Conference on Software Maintenance (ICSM)*, 2002.

[6] S. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, pages 4–14, 1998.

[7] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[8] J. Hayes. Risk reduction through requirements tracing. In *The Conference Proceedings of Software Quality Week*, 1990.

[9] J. H. Hayes. Input validation testing: A system level, early lifecycle technique. In *ICSE '97 Doctoral Consortium, published in the Proceedings of the Seventeenth International Conference on Software Engineering Doctoral Consortium*, May 1997.

[10] J. H. Hayes, A. Dekhtyar, and J. Osbourne. Improving requirements tracing via information retrieval. In *International Conference on Requirements Engineering, Monterey, California*, pages 151–161, 2003.

[11] J. H. Hayes, A. Dekhtyar, S. Sundaram, and S. Howard. Helping analysts trace requirements: An objective look. In *International Conference on Requirements Engineering (RE'2004)*, 2004.

[12] J. H. Hayes and J. Offutt. Input validation testing: A requirements-driven, system level, early lifecycle technique. In *Proceedings of the 11th International Conference on Software Engineering and its Applications*, October 1998.

[13] J. H. Hayes and J. Offutt. Increased software reliability through input validation analysis and testing. In *Proceedings of The Tenth IEEE International Symposium on Software Reliability Engineering*, pages 199–209, 1999.

[14] W. Hersh and P. Over. The trec-9 interactive track report. In *Proc. Text Retrieval Conference (TREC-9)*, pages 41–50, 2000.

[15] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] T. Joachims. Text categorization with suport vector machines: Learning with many relevant features. In *Proc. ECML*, pages 137–142, 1998.

[17] B. Lee and B. Bryant. Contextual knowledge representation for requirements documents in natural language. In *Proceedings of FLAIRS, the 15th International Florida Artificial Intelligence Research Symposium*, 2002.

[18] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[19] T. Menzies, J. S. D. Stefano, C. Cunanan, and R. M. Chapman. Mining repositories to assist in project planning and resource allocation. In *International Workshop on Mining Software Repositories (submitted)*, 2004. Available from `http://menzies.us/pdf/04msrdefects.pdf`.

[20] T. Mundie and F. Hallsworth. Requirements analysis using supertrace pc. In *Proc. American Society of Mechanical Engineers (ASME) for Computers in Engineering Symposium at the Energy and Environmental Expo*, 1995.

[21] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.