In In Proceedings of the AAAI Workshop on NLP for Software Engineering, February 2018.

Natural Language Processing and Program Analysis for Supporting Todo Comments as Software Evolves

Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric
The University of Texas at Austin
{pynie@, jessy@austin., khurshid@ece., mooney@cs., gligoric@}utexas.edu

Abstract

Natural language elements (e.g., API comments, todo comments) form a substantial part of software repositories. While developers routinely use many natural language elements (e.g., todo comments) for communication, the semantic content of these elements is often neglected by software engineering techniques and tools. Additionally, as software evolves and development teams re-organize, these natural language elements are frequently forgotten, or just become outdated, imprecise and irrelevant.

We envision several techniques, which combine natural language processing and program analysis, to help developers maintain their todo comments. Specifically, we propose techniques to synthesize code from comments, make comments executable, answer questions in comments, improve comment quality, and detect dangling comments.

Introduction

Natural language elements form a substantial part of software repositories. These elements are used to communicate between users and developers (e.g., API comments, bug reports, and feature requests), and among developers (e.g., todo comments). *Todo comments* contain invaluable data that describe changes to code that can increase software maintenance, reliability, and quality. Despite occurring frequently in practice and containing valuable information, these elements, because of their *informal nature*, are largely not exploited by existing software engineering tools.

Research on combining program analysis and natural language processing (NLP), which recently started to gain some traction, is in its infancy (Ernst 2017; Arnaoudova et al. 2015; Hindle et al. 2012; Oda et al. 2015; Allamanis, Peng, and Sutton 2016; Vasilescu, Casalnuovo, and Devanbu 2017; Raychev, Vechev, and Krause 2015; Nguyen et al. 2012), and the existing work, although novel, mostly neglected comments that are used to communicate *among the developers* (Storey et al. 2008; Sridhara 2016).

In this position paper, we argue about the importance of content in todo comments and envision several techniques to automatically maintain and resolve those comments.

This position paper is to a large extent inspired by our extensive analysis of a large corpus of open-source projects. Specifically, we analyzed over 30k open-source projects, which are available on GitHub, totaling 585 million lines of code (not counting comments). We found that these projects include over 297 million lines of comments ($\sim$30% of the total lines). Our analysis also uncovered more than 700k todo comments in the used corpus. We manually inspected (and discussed) hundreds of comments, code and comment changes, and commit messages. In the following subsections, we will frequently refer to this dataset and our findings related to this dataset. All examples of code and comments that we provide in this paper are taken from one of the analyzed open-source projects.

This paper mostly focuses on todo comments that contain valuable information on increasing software quality, performance, maintenance, and reliability. We consider the following three categories of todo comments. First, *task comments* explain what features are currently not supported or what optimizations need to be implemented (e.g., from the Google Guava project: "For optimal performance, use a binary search when `targets.size()` $<$ `size()`/`log(size())`"). Second, *trigger-action comments* talk about changes to the code repository that would be necessary if something else is modified by developers (e.g., from Guava: "check more preconditions (as `bufferSize` $>=$ `chunkSize`) if this is ever `public`"). Finally, *question comments* are concerned with alternative implementations, potential optimizations, and testing, which may be explored by developers only if time permits (e.g., from Guava: "Is this faster than `System.arraycopy()` for small arrays?").

Regardless of the category of todo comments, as software evolves and development teams re-organize, these comments may be *dangling*, i.e., resolved but forgotten (Storey et al. 2008; Sridhara 2016). For example, a trigger may hold (e.g., "if this is ever `public`") but the action may not be executed by developers (for very long time or ever), and developers may never have enough time to consider alternative algorithms and fine tune their existing implementations.

With the goal to help developers increase the reliability of their software, we propose several techniques to (1) synthesize code described in task comments, (2) make trigger-action comments executable, (3) answer question comments, (4) improve the quality of all todo comments, and (5) automatically detect dangling comments.

```
protected AbstractStreamingHasher(int chunkSize, int bufferSize) {
// TODO(kevinb): check more preconditions (as bufferSize >= chunkSize)
// if this is ever public
  if (TRIGIT.isPublic(TRIGIT.THIS_METHOD))
  checkArgument(bufferSize >= chunkSize);
checkArgument(bufferSize \% chunkSize == 0); ... }
```

(a) Example from Google Guava (AbstractStreamingHasher)

```
public void testDynamicAttributesSupport() throws Exception { ...
dispatcher.serviceAction(request, response, mapping);
// TODO : remove expectedJDK15 and if() after switching to Java 1.6
  if (TRIGIT.getJavaVersion() >5,
  TRIGIT.DELETE_NEXT, TRIGIT.VAR_DECL);
String expectedJDK15 = "<input type=\"text\" ...;
String expectedJDK16 = "<input type=\"text\" ...; ... }
```

(b) Example from Apache Struts (FreemarkerResultMockedTest)

Figure 1: Examples of trigger-action comments from open-source projects; we show how the existing comments (crossed out) can be encoded as executable statements in our TRIGIT framework (highlighted code)

Techniques

This section describes the basic idea behind each technique and the way we will approach the implementation.

Synthesizing Error-Reporting Code

We plan to develop lightweight synthesis techniques to generate error-reporting code for unsupported cases that are documented by developers in the *task comments* (e.g., from Guava: "support array types"). First, we will identify comments that document unsupported cases. To this end, we will explore possible supervision signals from resolved comments and their corresponding code changes, crowdsourcing annotation and semantic parsing of the comments. Second, we will synthesize error-reporting code that follows the style used in the codebase (e.g., throw an exception or return a special value from a function). Note that our goal is not to work on full-blown program synthesis, which would be interesting but challenging (e.g., Polikarpova, Kuraj, and Solar-Lezama (2016)), but rather to focus on a specific domain of error-reporting. Basically, our goal is to make the existing comments *observable* during program execution by reporting an appropriate message for unsupported cases.

Extracting Executable Comments

We will develop techniques to help software engineers to encode their *trigger-action* comments as *executable code statements*. This will help with repository maintenance, because developers will not need to manually check their todo comments; instead, the executable statements will be automatically triggered when appropriate.

We show several examples of trigger-action comments in Table 1 (the top half). We found that $\sim$10% of all todo comments (in our corpus) belong to this comment category. While it would be infeasible to support every comment written in the trigger-action style, we plan to focus on those tasks that update the codebase (e.g., transformations of abstract syntax trees) when triggers are satisfied.

Our initial step is to develop a domain specific language embedded in Java to be used to: (1) query the static features of the codebase, e.g., required Java version, and (2) specify code transformations, e.g., remove a method from a class. Figure 1 shows two examples of trigger-action comments encoded in our framework (named TRIGIT); the original todo comments are crossed out and the statements for our framework are highlighted.

In the first example, we use our framework to check a modifier of the current method; if the method becomes `public`, the code guarded by the trigger should become a part of the compiled class. In the second example, we specify that a variable should be removed if the required Java version is higher than 1.5; the required Java version can be obtained from a build script. (Note that the statements/expressions that use the variables need to be annotated too, but we do not show this due to space limitations.) The evaluation of the triggers will be done statically (once code is compiled), as the queries should not depend on the dynamic behavior of the program. Our tool, which can be implemented as a compiler plugin, will automatically remove the triggers and perform program transformations. Note that the user would still be able to inspect/approve the changes (e.g., by executing git diff). As the transformation engine we will use the existing open-source platforms, e.g., Eclipse, or program transformation systems, e.g., Cordy et al. (2004). The language design will be guided by examples, and we will evolve the language to support cases that we encounter in the future.

Our second step is to automatically discover trigger-action comments present in a codebase and recover the corresponding triggers and actions via mining explicit *condition* relations within the content of the todo comments; *explicit* discourse relations can be classified with adequate accuracy (Pitler et al. 2008).

In the third step, we will develop automated migration from comments to the TRIGIT specifications, which will follow our recent work on language to code for if-this-then-that (IFTTT) recipes (Quirk, Mooney, and Galley 2015). Specifically, we will train a semantic parser to map trigger-action comments into executable code using supervision automatically extracted from the *code changes* made when a todo comment is resolved. This supervision may be noisy, since not all code changes may be directly related to resolving the todo comment, but our previous work on IFTTT shows that noisy, automatically extracted supervision from pairing comments and code can be tolerated reasonably well.

Answering Questions From Comments

We will develop techniques to help software engineers to make informed decisions about questions that are asked in todo comments. In our preliminary studies, we discovered that developers ask questions in todo comments more than 10% of the time; we obtained this number by counting todo comments that contain "?". Some of these questions are shown in Table 1 (the bottom half). Many of the questions are related to code *optimization*, *program transformation*, or *testing*. Our plan is to focus on techniques that will address these three types of questions. First, to answer questions related to optimizations, we will extract suggested

Table 1: Example todo comments in open-source projects

	Project (on GitHub)	File (.java)	Todo Comments
trigger-action comments	Apache/Incubator-wave	Pretty	Remove this when `HtmlViewImpl` implements `getAttributes`
	Apache/Struts	FreemarkerResultMockedTest	Remove `expectedJDK15` and `if()` after switching to Java 1.6
	Apache/Poi	TextXSSBugs	Delete this test case when `MROUND` and `VAR` are implemented
	Google/Guava	Types	Once we are on Java 8, delete this abstraction
	Google/Guava	AbstractStreamingHasher	Check preconditions (as `bufferSize` $>=$ `chunkSize`) if this is ever `public`
	Google/Guava	MapTest	Replace with `Ascii.caseInsensitiveEquivalence()` when it exists
	KangProject/Frameworks_base	SslCertificate	If deprecated constructors are removed, this should always be available
	Morristech/Gwt	DefaultFilters	This class needs to be revisited, when Gwt's Ant is upgraded
	Morristech/Gwt	Simplifier	If the AST were normalized, we wouldn't need this
question comments	Andyglick/Hk2-fork	AbstractRepositoryImpl	Is it allowed to call the initialize method multiple times?
	Apache/Net	IMAPReply	Would `lookingAt()` be more efficient? If so, then drop trailing .* from patterns
	Google/Guava	ArrayTable	Add getters returning `rowKeyToIndex` and `columnKeyToIndex`?
	Google/Guava	EvictingQueue	Do we want to `checkNotNull` each element in `containsAll` and `retainAll`?
	Eclipse/CDT	LlvmEnvironmentVariableSupplier	Is this actually called anywhere?
	Eclipse/CDT	EvalBinary	What if the composite being accessed is not an array but a structure?
	Eclipse/Mwe	PluginExtensionManager	Test: what happens when a handler is not there? Exception?
	JetBrains/Jdk8u_jaxp	NodeSet	What happens if `index` is out of range?
	Square/OKhttp	Http2Reader	Test case for empty continuation header?

code modifications from comments, apply those modifications and profile the code (by executing existing test suites) and evaluate the performance with profiles (on various machines). Second, to answer questions related to tests, we will develop techniques that extract test inputs from a question and generate new tests with those inputs; these new tests will be obtained by adjusting an automated test generation tool (e.g., Randoop (Pacheco et al. 2007)) or by extending existing (manually written) tests. Third, to answer questions related to code structure, we will extract suggested changes (from Guava: "Add getters returning rowKeyToIndex and columnKeyToIndex?"), perform the changes, and measure quality of code in terms of naturalness (Hindle et al. 2012).

Our question classification system will also learn from how todo comments are answered as software evolves (e.g., files and functions that are modified and language artifacts that are added or edited); we can also learn from actions taken by developers. As some of the questions may be open-ended, we plan to develop an interactive dialog interface, which we recently used for language to code translation (Chaurasia and Mooney 2017). We plan to use dialog systems to clarify user intent and gather information—in our case, when a question is initially asked.

Improving Todo Comments

We will develop techniques to help software engineers to write meaningful todo comments. While manually analyzing hundreds of todo comments, we found a number of comments that were hard to understand even if we read the code near those comments. We were also in disagreement about their meaning in several cases, and although we could understand a comment (e.g., from the Square Retrofit project: "TODO non-suck message"), it was clear that any technique would have a hard time to extract any useful data.

Our initial task will be to detect todo comments that are not specific enough, as well as those comments that do not follow the conventions already used in the same project. The techniques that we will develop will build on our work on text specificity (Li and Nenkova 2015) and program analysis. When we detect an unspecific comment, we will either notify a developer to provide additional clarification, highlight a part of the comment that does not follow the style (in a similar way that spellcheckers highlight typos in comments inside IDEs), or automatically reformat the comment to be consistent with other comments in the same repository. We will also provide automated comment style checkers, where the rules can be expressed by developers; this is similar to code style checkers, which are used in practice. Having specific comments that follow the same style will enable techniques from prior sections.

Detecting Dangling Todo Comments

Prior work has shown that developers may resolve todo comments but forget to remove these comments from source code (Storey et al. 2008; Sridhara 2016); these *dangling comments* can waste developers' time during program comprehension and maintenance.

We are working on a technique, based on machine learning, to automatically detect dangling todo comments. Our detection technique learns from existing software repositories. As mentioned earlier, we have already collected more than 700k todo comments. This large dataset provides examples for todo comments that were removed by developers (over 20k). We are using these examples as distant supervision signals, where we are exploring automatic labeling of examples (e.g., todo comments that are in the same file with removed todo comments). Our models are exploiting commit messages and static code analysis of changes. In the future, we plan to also utilize software histories to extract necessary context when todo comments were introduced. We will also reason about co-evolution of code and comments from when a todo comment was introduced until

it was resolved by a developer. Specifically, for each code change, we will compute its distance from todo comments, word similarity with each comment, and code structure that may be described in a comment. These sources of information provide complementary views to feature development and complementary models, so we plan to build on our prior work in co-training and ensemble models.

Related Work

Li et al. (2006) used text classification to validate the representativeness of their study of bug characteristics. Fluri, Wursch, and Gall (2007) empirically showed that code and comments frequently co-evolve. Padioleau, Tan, and Zhou (2009) *manually* studied over one thousand comments, and found that 50% of comments can be leveraged by various techniques. Haouari, Sahraoui, and Langlais (2011) introduced a taxonomy of comments and found that todo comments are the second most common type of comments. Movshovitz-Attias and Cohen (2013) used topic modeling and language models to generate comments from Java source files. Several work tackled automated generation of commit messages and mining relation from commit messages (Linares-Vásquez et al. 2015; Jiang and McMillan 2017; Andersson, Ericsson, and Wingkvist 2014; Loyola, Marrese-Taylor, and Matsuo 2017).

Tan et al. (2007) detected inconsistencies between code and comments and proposed a technique to test Javadoc comments. Zhong et al. (2011) developed a technique to infer specification from natural language API documentation and used it to detect issues in client code.

Conclusion

We argued that comments used to communicate among developers (todo comments) contain invaluable content that is currently neglected. We described several techniques – synthesizing code from comments, making comments executable, answering questions in comments, improving comment quality, and detecting dangling comments. These techniques, based on natural language processing and program analysis, have potential to substantially simplify software maintenance and increase software reliability.

Acknowledgments

We thank Rishabh Rai for the initial discussion on this work. This work was partially supported by the US National Science Foundation under Grant No. CCF-1704790.

References

Allamanis, M.; Peng, H.; and Sutton, C. A. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*.

Andersson, R.; Ericsson, M.; and Wingkvist, A. 2014. Mining relations from Git commit messages: An experience report. In *SLTC*.

Arnaoudova, V.; Haiduc, S.; Marcus, A.; and Antoniol, G. 2015. The use of text retrieval and natural language processing in software engineering. In *ICSE*.

Chaurasia, S., and Mooney, R. 2017. Dialog for language to code. In *IJCNLP*.

Cordy, J. R. 2004. TXL - a language for programming language tools and applications. *ENTCS* 110.

Ernst, M. D. 2017. Natural language is a programming language: Applying natural language processing to software development. In *SNAPL*, volume 71.

Fluri, B.; Wursch, M.; and Gall, H. C. 2007. Do code and comments co-evolve? On the relation between source code and comment changes. In *WCRE*.

Haouari, D.; Sahraoui, H.; and Langlais, P. 2011. How good is your comment? A study of comments in Java programs. In *ESEM*.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *ICSE*.

Jiang, S., and McMillan, C. 2017. Towards automatic generation of short summaries of commits. In *ICPC*.

Li, J. J., and Nenkova, A. 2015. Fast and accurate prediction of sentence specificity. In *AAAI*.

Li, Z.; Tan, L.; Wang, X.; Lu, S.; Zhou, Y.; and Zhai, C. 2006. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *ASID*.

Linares-Vásquez, M.; Cortés-Coy, F.; Aponte, J.; and Poshyvanyk, D. 2015. ChangeScribe: A tool for automatically generating commit messages. In *ICSE*.

Loyola, P.; Marrese-Taylor, E.; and Matsuo, Y. 2017. A neural architecture for generating natural language descriptions from source code changes. In *ACL*.

Movshovitz-Attias, D., and Cohen, W. W. 2013. Natural language models for predicting programming comments. In *ACL*.

Nguyen, A. T.; Nguyen, T. T.; Nguyen, T. N.; Lo, D.; and Sun, C. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *ASE*.

Oda, Y.; Fudaba, H.; Neubig, G.; Hata, H.; Sakti, S.; Toda, T.; and Nakamura, S. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *ASE*.

Pacheco, C.; Lahiri, S. K.; Ernst, M. D.; and Ball, T. 2007. Feedback-directed random test generation. In *ICSE*.

Padioleau, Y.; Tan, L.; and Zhou, Y. 2009. Listening to programmers taxonomies and characteristics of comments in operating system code. In *ICSE*.

Pitler, E.; Raghupathy, M.; Mehta, H.; Nenkova, A.; Lee, A.; and Joshi, A. 2008. Easily identifiable discourse relations. In *COLING*.

Polikarpova, N.; Kuraj, I.; and Solar-Lezama, A. 2016. Program synthesis from polymorphic refinement types. In *PLDI*.

Quirk, C.; Mooney, R.; and Galley, M. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*.

Raychev, V.; Vechev, M.; and Krause, A. 2015. Predicting program properties from "Big Code". In *POPL*.

Sridhara, G. 2016. Automatically detecting the up-to-date status of ToDo comments in Java programs. In *ISEC*.

Storey, M.-A.; Ryall, J.; Bull, R. I.; Myers, D.; and Singer, J. 2008. TODO or to bug. In *ICSE*.

Tan, L.; Yuan, D.; Krishna, G.; and Zhou, Y. 2007. /*iComment: Bugs or bad comments?*/. In *SOSP*.

Vasilescu, B.; Casalnuovo, C.; and Devanbu, P. T. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *FSE*.

Zhong, H.; Zhang, L.; Xie, T.; and Mei, H. 2011. Inferring specifications for resources from natural language API documentation. *ASE Journal* 18(3).