TASSAL: Autofolding for Source Code Summarization

Jaroslav Fowkes* Pankajan Chanthirasegaran* Razvan Ranca†
Miltiadis Allamanis* Mirella Lapata* Charles Sutton*

*School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, UK
{jfowkes, pchanthi, m.allamanis, csutton}@ed.ac.uk; mlap@inf.ed.ac.uk
†Tractable, Oval Office, 11-12 The Oval, London, E2 9DT, UK
razvan@tractable.io

ABSTRACT

We present a novel tool, TASSAL, that automatically creates a summary of each source file in a project by folding its least salient code regions. The intended use-case for our tool is the *first-look problem*: to help developers who are unfamiliar with a new codebase and are attempting to understand it. TASSAL is intended to aid developers in this task by folding away less informative regions of code and allowing them to focus their efforts on the most informative ones. While modern code editors do provide *code folding* to selectively hide blocks of code, it is impractical to use as folding decisions must be made manually or based on simple rules. We find through a case study that TASSAL is strongly preferred by experienced developers over simple folding baselines, demonstrating its usefulness. In short, we strongly believe TASSAL can aid program comprehension by turning code folding into a usable and valuable tool. A video highlighting the main features of TASSAL can be found at https://youtu.be/_yu7JZgiBA4.

1. INTRODUCTION

Developers spend more of their time reading and browsing source code than actually writing it [11, 13]. Despite much research [20], there is still a need for better tools that aid program comprehension and thereby reduce the cost of software development. This raises new opportunities for tools that can summarize source code and aid the developer in their comprehension task.

Often during development and maintenance, developers skim the code in order to quickly understand it [19]. A good *summary* of the source code aims to support this use case: by eliding less-important details, a summary can be easier to read quickly and help the developer to gain a high-level conceptual understanding of the code. This is particularly the case for the *first-look problem* when developers need to quickly familiarize themselves with the core parts of a large code base, for example when joining an existing open source project.

Many code editors provide a feature called *code folding*, which allows developers to selectively display or hide blocks of source code. This feature is commonly supported and familiar to developers [8, 12, 17]. However, in current IDEs, folding quickly becomes cumbersome because folding decisions must be done manually by the programmer, or based on simple rules, such as folding code blocks based on depth [2], that some IDEs take automatically. This creates an obvious chicken-and-egg problem, because the developer must already understand the source file in order to decide what should be folded.

In this paper, we propose that code folding can be a valuable tool for aiding program comprehension, provided that folding decisions are made automatically based on the *code's content*. We therefore introduce a novel autofolding method for source code summarization, called TASSAL, that automatically creates a code summary by folding non-essential code elements that are not useful on first viewing. The main idea behind TASSAL is to define a textual *salience* measure of how representative the code in each block is to the overall text of the source file. Less representative blocks are the first to be folded. Our demo of TASSAL can be found at http://groups.inf.ed.ac.uk/cup/tassal/demo.html and the source code for TASSAL itself is available at https://github.com/mast-group/tassal. A video highlighting the main features of TASSAL can be found at https://youtu.be/_yu7JZgiBA4. At present, the demo only supports Java source files, however the core tool itself is language agnostic and can be easily extended to any language for which an AST can be defined.

More broadly, we hope that TASSAL will aid program comprehension by turning code folding, perhaps an overlooked feature, into a useful, usable and valuable tool.

2. OVERVIEW OF TASSAL

Our tool, TASSAL or Tree-based Autofolding Software Summarization ALgorithm, is based on optimizing the similarity between a summary of a source file and the source file itself. This is, to our knowledge, the first content-based autofolding method for code summarization.

The outline of TASSAL is as follows: TASSAL takes as input a set of source files along with a desired compression ratio and outputs a summary of each file where uninformative regions of code have been folded. In order to achieve this TASSAL first parses the code's AST to obtain suitable

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-4205-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2889160.2889171

regions to fold. It then applies a source code language model to each foldable region. The aim of this model is to identify, for every source file, which tokens specifically characterize the file, as opposed to project-specific or Java-generic tokens that are not as informative for understanding the file. Using this ranking TASSAL then leverages an optimization algorithm to determine the most uninformative regions to fold while achieving the desired level of compression. This is a novel optimization procedure that takes the structure of the code into account.

When we say that TASSAL *folds* a source code region we mean that the region is replaced by a symbol indicating that the region was folded. To encourage intuitive summaries, we let TASSAL perform folding only on *code blocks*, *comment blocks*, *import statements* and *fields*. Our reasoning for this is that such code regions are natural units for summarization since they take advantage of the code structure specified by the programmer. However, since our approach works within the code's AST, it can be trivially extended to fold *any* contiguous region of interest, e.g. statements (or a carefully designed subset thereof).

To determine which non-essential regions should be folded, TASSAL has a choice of using two source code language models: the first, which enables us to summarize source files in real-time, is based on a *vector space model* (VSM) [14] for source code tokens. In the VSM, we choose to unfold (i.e., include in the summary) those source code regions that are closest in the vector space to the full unfolded file. The second, which gives better summaries but requires pre-training on the source files, makes use of a novel *topic model* [1] for code, which separates tokens according whether they best characterize their file, their project, or the corpus as a whole. Then, we choose to unfold (i.e., include in the summary) those code regions for which the largest number of tokens come from the file-specific topic rather than the project-specific or general-Java topics. For technical details of TASSAL and the above models, as well as more extensive evaluation, we refer the interested reader to our preprint [5].

3. TASSAL IN ACTION

We created a demo of TASSAL using the Play Framework (https://www.playframework.com) to showcase how it can be used to summarize open-source Java projects on GitHub (however note that TASSAL can summarize the source code of any Java project). A screenshot of the demo is shown in Figure 1 and as one can see from the figure, the basic layout of the demo is very simple. On the left hand side is a tree view showing all the Java source files for a user-selected project on GitHub. Upon clicking on a source file, the remainder of the screen uses the Javascript-based ACE code editor (https://ace.c9.io) to show a summary of the file where less informative code regions have been folded. The user can adjust the conciseness of the summary using the compression ratio slider at top-left, ranging from viewing the complete file (0%) to folding all the foldable regions (100%). As a sanity check, the fold icons () to the left of the line numbers denote the code regions that were marked as folded by TASSAL. Note that while TASSAL is able to fold fields, we did not find a satisfactory way to implement this in ACE and therefore omitted it from the demo (however the fold icons for fields are still displayed).

If the user wishes to unfold a folded region, they can do so by clicking on the symbol denoting the fold () and conversely, they can fold any foldable region by clicking on the down arrow (▼) to the right of the line numbers as is standard in modern editors. One can see from the example (`StatusLine.java` from the `bigbluebutton` project displayed at 50% compression) that the header has been folded, as have the `toString`, `getCode`, `getReason`, `clone` and `equals` methods, i.e., Java boilerplate code — precisely the less salient code regions. Note also, how by *folding* the less informative regions the code *remains readable and navigable* and *no information is lost*. This is not true of other summarization approaches to source code [4, 6, 7, 9, 15, 16, 18].

As for the choice of language model, using TASSAL with the topic model will in general produce better summaries, however training a topic model is too expensive for an interactive system. Therefore, we train the topic model in advance on a small set of projects and cache the topic assignments. If the user requests summaries of GitHub projects for which we have not run the topic model, we fall back to the the VSM model. When both language models are available, the user can toggle between them by means of a radio button in the top-left corner.

4. EVALUATION

We evaluated TASSAL against simple heuristic folding baselines and found that it was strongly preferred by experienced developers. The baselines we used were chosen to represent more naïve approaches for autofolding source code. All the baselines start from a fully folded source file and gradually unfold foldable regions until they reach the required compression ratio. For consistency, if a foldable region is to be unfolded, all of its parent regions in the AST are unfolded as well. The baselines are:

Shallowest unfold the shallowest available foldable region in the AST first, choosing randomly if there is more than one.

Largest unfold the largest available foldable region first, as measured by the number of tokens, breaking ties randomly.

Javadoc first unfold all Javadoc comments (in random order) and then fallback at random to an available foldable region, unfolding method blocks last.

Each of the baselines represents a possible assumption that we can make about summarizing source code. The Largest baseline assumes that the largest nodes are more valuable in a summary, the Shallowest baseline is representative of the folding approach used in the Code Bubbles IDE [2] and the Javadoc baseline is representative of the current rule-based defaults in IDEs such as Eclipse.

We asked developers to rate the summaries produced by our best content-based method, TASSAL using the topic model, and the three non-content-based baselines at a com-

	Conciseness		Usefulness	
Summary	Mean	St. dev.	Mean	St. dev.
TASSAL	**3.27**	**1.01**	**3.18**	**0.97**
Javadocs*	3.07	1.03	2.69	1.09
Shallowest*	2.97	1.05	2.50	1.15
Largest*	3.08	1.07	2.67	1.06

Table 1: Mean and standard deviation averaged across developer ratings for summaries produced by the four autofolding systems at a compression ratio of 50%. *significantly different from TASSAL.

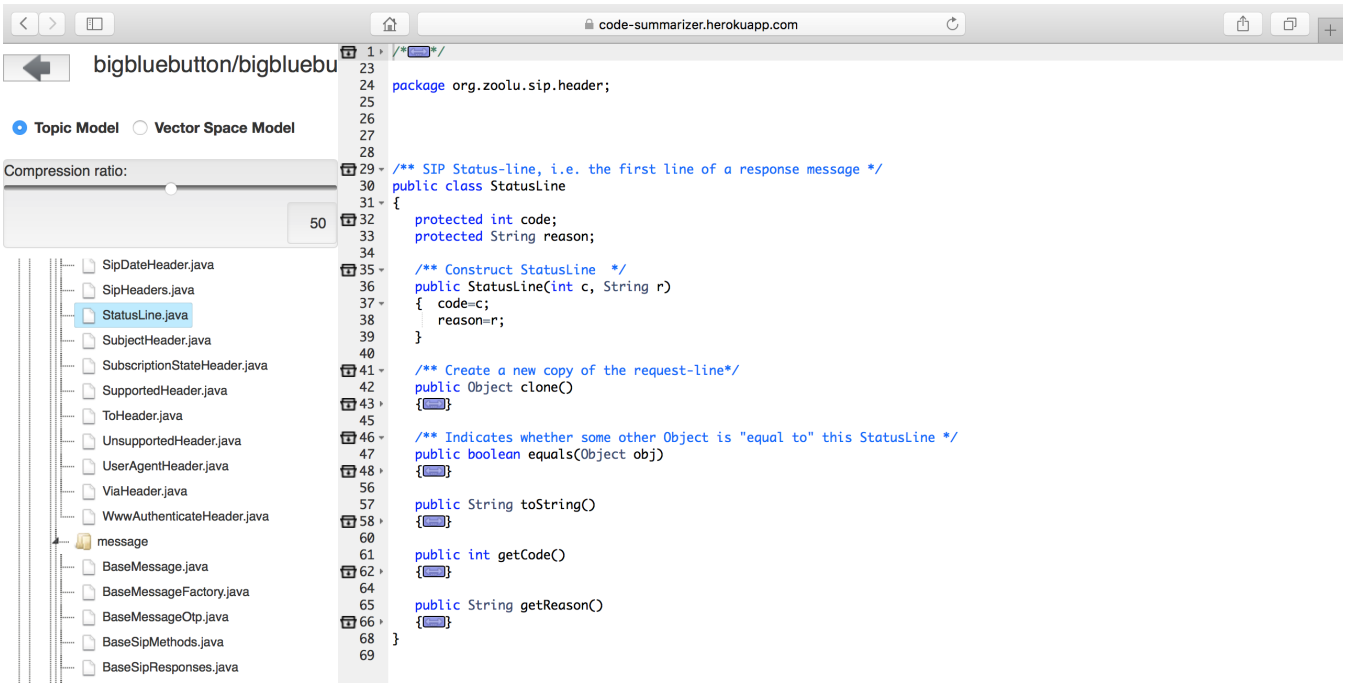

Figure 1: A screenshot of our source code autofolding tool being used to summarize `StatusLine.java` from `bigbluebutton`.

pression ratio of 50% (as can be seen in Figure 1, 50% is often a substantial compression ratio, because the uncompressed lines include blank lines, method headers, etc.). All four systems were allowed to fold code blocks, comment blocks and import statements. We recruited six experienced developers for our study. All were computer science masters students with an average 5.3 years Java programming experience and 4 years industry programming experience.

We randomly selected four projects from the top six high-quality popular projects on GitHub (`storm`, `elasticsearch`, `spring-framework`, `libgdx`, `bigbluebutton`, `netty`) and five files from each project for the study, resulting in 20 files in total. For every file, developers were presented with each of the possible summaries in random order and asked to rate the conciseness and usefulness of each summary on a five-point Likert scale (higher is better). Developers were allowed to browse the full source code of each project during the study.

We show the average ratings across all six developers in Table 1 along with the average standard deviations. One can see that summaries produced by TASSAL score around 0.2 points higher on conciseness and 0.5 points higher on usefulness than the three baselines. We performed ANOVA on the developer conciseness and usefulness ratings for the different summaries and found that the difference between TASSAL and the baselines was significant ($p < 0.05$) as denoted in Table 1. This lends support to our belief that TASSAL can improve upon existing rule-based autofolding systems present in modern IDEs and aid program comprehension.

5. RELATED WORK

There is some existing work on the use of **code folding** (also known as code elision) to aid comprehension. In particular, Cockburn et al. [3] find that illegible elision of all method bodies in a class improves programmer efficiency in editing and browsing tasks. Rugaber et al. [17] consider a conceptual model for manual folding, extending it to non-contiguous regions of code. Kullbach et al. [12] develop the `GUPRO` IDE to aid in the comprehension of C preprocessor code via rule-based folding of macro expansions and file includes. Also, Hendrix et al. develop the `GRASP` program comprehension tool, combining control structure diagramming with manual folding [8]. Bragdon et al. [2] perform code autofolding of long methods based on code block depth in their proposed Code Bubbles IDE. Additionally, most modern IDEs and code editors already have extensive support for folding *specific* code regions as well as the ability to fold regions based on user-specified *rules*. However, to the best of our knowledge the problem of automatically determining which regions to fold based on their *content* is novel.

We are aware of only a few previous methods that consider the problem of **code summarization**. One of the first approaches is program slicing [18, 9] which hides irrelevant LOC for a chosen program path – essentially a very specific form of query-based summarization. Most similar to our work are Haiduc et al. [6, 7] and the follow up work by Eddy et al. [4] and Rodeghero et al. [16], who also consider the problem of summarizing source code, particularly methods and classes, but in their work code fragments are summarized by a short list of keywords. For example, the `StatusLine` constructor in Figure 1 might be summarized by the list of terms (`status`, `line`, `code`, `reason`). McBurney et al. [15] take this idea further and present the keywords in a navigable tree structure, with more general topics near the top of the tree. In our work, we summarize code *with* code, which we would argue has the potential to provide a much richer and more informative summary.

Also, Ying et al. [21] consider the problem of summarizing a list of code fragments, such as those returned by a code search engine. They use a supervised learning approach at the level of lines of code. Because they consider the results of code search, their classifier uses query-level features, e.g., whether a line of code uses any identifiers that were present in the query. This is a source of information that is not avail-

able in our problem setting. In contrast, we target use cases in which the developer is skimming the source code to get an overview of its operation, rather than performing a directed keyword search. Kim et al. [10] develop a system that augments API documentation with code example summaries but these are mined from the web and are therefore limited to APIs which have examples already written for them — our approach is applicable to *any* source file.

On a more technical level, our folding-based summaries are distinguished from this previous work in that our summaries are coherent with respect to the programming language's syntax. Indeed, Eddy et al. [4] observe that developers prefer summaries with a natural structure. Folding on code blocks also enables us to retain method headers in the summary — identified by Haiduc et al. [7] as highly relevant to developers and accounting for the high scores of their best performing method.

6. CONCLUSIONS

We presented a novel tool that summarizes source code files by automatically folding their least informative code regions. Unlike existing work on code summarization, our tool demonstrates that the folding procedure common to IDEs can serve as the basis of an automatic summary. Indeed, our proposed method builds on this previous work using disjoint line- and term-level code summaries and introduces a new contiguous parse subtree as its summary. Our evaluation demonstrates that our summarizer is favoured by experienced developers over methods currently used as standard in modern IDEs. In future we would like to extend our tool to generate targeted summaries for specific software engineering tasks such as bug localization or code review.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (grant number EP/K024043/1) and by Microsoft Research through its PhD Scholarship Programme. We are also grateful to Rebecca Mason for letting us adapt her topic model implementation to source code and would like to thank Brian Doll for useful discussions.

References

[1] D. Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.

[2] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *ICSE*, pages 455–464, 2010.

[3] A. Cockburn and M. Smith. Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Computers*, 15(3):387–407, 2003.

[4] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver. Evaluating source code summarization techniques: Replication and expansion. In *ICPC*, pages 13–22, 2013.

[5] J. Fowkes, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton. Autofolding for source code summarization. *arXiv preprint arXiv:1403.4503*, 2014.

[6] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *ICSE*, pages 223–226, 2010.

[7] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, pages 35–44, 2010.

[8] T. D. Hendrix, J. H. Cross II, L. A. Barowski, and K. S. Mathias. Visual support for incremental abstraction and refinement in Ada 95. In *ACM SIGAda Ada Letters*, volume 18, pages 142–147, 1998.

[9] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In *PASTE*, pages 57–63, 2008.

[10] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *Transactions on Information Systems*, 31(1):1, 2013.

[11] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Transactions on Software Engineering*, 32(12):971–987, 2006.

[12] B. Kullbach and V. Riediger. Folding: An approach to enable program understanding of preprocessed languages. In *WCRE*, pages 3–12, 2001.

[13] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE*, pages 492–501, 2006.

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

[15] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger. Improving topic model source code summarization. In *ICPC*, pages 291–294, 2014.

[16] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE*, pages 390–401, 2014.

[17] S. Rugaber, N. Chainani, O. Nnadi, and K. Stirewalt. A conceptual model for folding. Technical Report GT-CS-08-09, Georgia Institute of Technology, 2008.

[18] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):12, 2012.

[19] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *ICSM*, pages 157–166, 2009.

[20] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *IWPC*, pages 181–191, 2005.

[21] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *FSE*, pages 655–658, 2013.