Topic$_{XP}$: Exploring Topics in Source Code using Latent Dirichlet Allocation

Trevor Savage, Bogdan Dit, Malcom Gethers, Denys Poshyvanyk
Department of Computer Science
The College of William and Mary
Williamsburg, Virginia, USA
{tcsava, bdit, mbgethers}@email.wm.edu, denys@cs.wm.edu

***Abstract*—Acquiring general understanding of large software systems and components from which they are built can be a time consuming task, but having such an understanding is an important prerequisite to adding features or fixing bugs. In this paper we propose the tool, namely Topic$_{XP}$, to support developers during such software maintenance tasks by extracting and analyzing unstructured information in source code identifier names and comments using Latent Dirichlet Allocation. Topic$_{XP}$ enables developers to gain an overview of a software system under analysis by extracting and visualizing natural language topics, which generally correspond to concepts or features implemented in software classes. Topic$_{XP}$ is implemented as an open-source Eclipse plug-in, which proposes interactive visualization of topics along with structural dependencies between underlying classes implementing these topics. The paper also presents the results of a preliminary user study aimed at evaluating Topic$_{XP}$.**

I. Introduction

The task of maintaining or adding features to a large software system demands that developers have some general understanding of the system's architecture. An important prerequisite to writing new code or changing existing code is to know where to look for or to find the appropriate places to add code or change the system. But for a developer approaching an unfamiliar system, gaining even a high-level understanding of what components make up a part of software and how these components work together can be a time-consuming task. While an experienced developer with a system may know exactly where to start a particular task in the source code, a newcomer might not even be able to formulate textual queries to find an appropriate location.

One technique which promises to automate a portion of the task of understanding source code is the technique of extracting linguistic topics from source code. Topic extraction via Latent Dirichlet Allocation (LDA) [4] models source code documents as mixtures of probabilistic topics. These topics tend to correspond to concepts or features implemented by the software [2] and are highly related to classes throughout a software system, which may work towards a similar purpose. By using these topics to guide their investigation, a developer should be able to understand the structure of the system that may not be reflected by the package hierarchy or by the system's documentation.

Linguistic topics in source code have been examined before from a variety of angles [1, 2, 8, 12], using both Latent Semantic Indexing (LSI) [5] and LDA [4], but the task of visualizing these topics to the developer in an interactive manner has not been given as much attention. In this paper we introduce Topic$_{XP}$[1], a plug-in for the Eclipse[2] IDE, which provides a practical topic extraction system, an interactive visualization for exploring these topics and other useful structural information.

While our tool currently works only with systems written in the Java programming language, its concepts should be applicable to any system with discrete source documents organized in a hierarchical fashion.

This tool demo paper makes the following contributions:

- Topic$_{XP}$ implemented as a plug-in for Eclipse, which provides linguistic topic extraction using LDA.
- A set of related visualizations, implemented in the Topic$_{XP}$ plug-in, which aims at presenting linguistic topics to developers in a useful manner.
- A preliminary study which asserts the utility of the visualizations proposed and of the topics themselves.

II. Exploring Linguistic Topics

The tool described in this paper first extracts linguistic topics and other relevant information from source code, then visualizes this information for the user, and finally provides some mechanisms for the user to interact with this information. In section A we present the process used to capture linguistic topics from textual corpora. In section B we describe what information our tool extracts and how it extracts it. Section C illustrates our visualizations and describes the rational behind them. Finally, section D explains some of the additional ways that users can interact with the tool and the proposed visualizations.

A. Capturing Lingustic Topics with Latent Dirichlet Allocation

LDA [4] is a technique for deriving probabilistic topic models from textual corpora by means of a generative process. LDA models each document in a corpus as a mixture of linguistic topics. That is, each document is represented by a probability distribution over the set of linguistic topics inferred by LDA. In doing so, documents are not limited to being associated with a single topic, but instead are modeled in a way that considers the possibility that documents may address multiple topics. The linguistic

[1] http://www.cs.wm.edu/semeru/TopicXP
[2] http://www.eclipse.org/

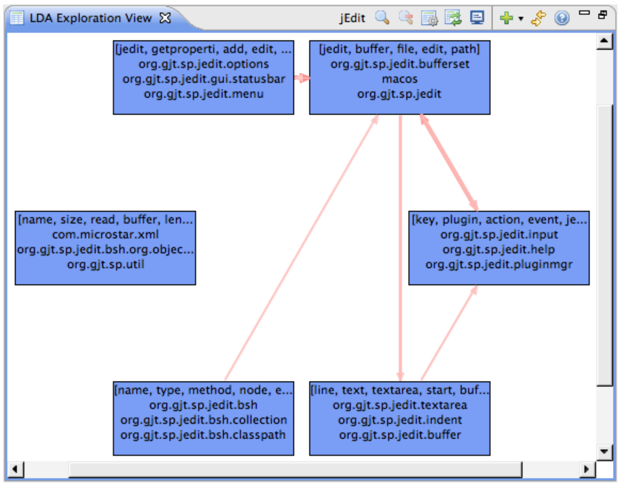

LDA Exploration View
jEdit
[jedit, getproperti, add, edit, ...
org.gjt.sp.jedit.options
org.gjt.sp.jedit.gui.statusbar
org.gjt.sp.jedit.menu
[jedit, buffer, file, edit, path]
org.gjt.sp.jedit.bufferset
macos
org.gjt.sp.jedit
[name, size, read, buffer, len...
com.microstar.xml
org.gjt.sp.jedit.bsh.org.objec...
org.gjt.sp.util
[key, plugin, action, event, je...
org.gjt.sp.jedit.input
org.gjt.sp.jedit.help
org.gjt.sp.jedit.pluginmgr
[name, type, method, node, e...
org.gjt.sp.jedit.bsh
org.gjt.sp.jedit.bsh.collection
org.gjt.sp.jedit.bsh.classpath
[line, text, textarea, start, buf...
org.gjt.sp.jedit.textarea
org.gjt.sp.jedit.indent
org.gjt.sp.jedit.buffer

Figure 1 The Topic Dependency View. Each box represents a topic and indicates the topic's most associated words and packages or classes. The arrows indicate that code in one topic calls code in another topic.

topics inferred by LDA are modeled as probability distributions over the set of words in a textual corpus.

Given *m* documents containing *k* topics expressed over *v* unique words (*w*) the distribution of i^{th} topic t_i over *v* words can be represented by φ_i and the distribution of j^{th} document d_i over *k* topics can be represented by θ_j. The LDA-based model assumes a prior Dirichlet distribution on θ, thus allowing the estimation of φ without requiring the estimation of θ. That is, LDA assumes the following generative process for each document d_i in a corpus *D*:

1. Select $N \sim$ Poisson distribution (ξ)
2. Select $\theta \sim$ Dirichlet distribution (α)
3. For each of the *N* words w_i:

   (a) Select a topic $t_k \sim$ Multinomial (θ).

   (b) Select a word w_i from $p(w_i|z_n,\beta)$, a multinomial probability conditioned on topic t_k.

We refer the interested reader to the original work of Blei *et al.* [4] for more details pertaining to LDA.

B. *Extracting Topics, Dependencies, and Cohesion*

Upon being invoked by the user right clicking on a project in Eclipse and selecting Explore with LDA, Topic$_{XP}$ extracts and computes a number of pieces of information to be visualized: topics, dependencies between those topics (based on structural dependencies among classes implementing these topics), and a cohesion metric derived from the LDA model. Steps 1-6 are completed three times: one time using all of the classes within the project as the documents, one time using packages, and one time using methods. The class topic map is used for most purposes. The package topic map, which uses the same topics as the class map, is used only to help describe the topics. The method topic map is used only to compute the cohesion metric detailed in step 8.

1. **Extracting Words from Source Code**. Before we can begin computing topics, we prepare the collections of words which make up the documents that LDA can process. This is done by extracting all the natural language information associated with a given document; all the words used in comments or identifiers inside the method, class or package are extracted by exploring the Abstract Syntax Tree as computed by Eclipse.
2. **Splitting Words.** We preprocess the words that can be found in source code, starting by running them through a tokenizer. This allows us to split identifier names written with camel case or using underscores (*i.e.*, CamelCase or under_score) into their component words, giving us a better idea of what natural language topics and words are used in implementation.
3. **Removing Stop Words.** By default, we remove a set of Java reserved keywords and some other words that are very commonly used, such as "get" and "set", since these words are typically used throughout majority of Java source code classes [14]. We also allow the user to specify additional stop words.
4. **Stemming Words**. The next step involved in pre-processing the input for LDA is to stem the words that make up our documents. Stemming involves removing the endings from words so that we can recognize a given word no matter what grammatical context it appears in. For example, in a stemmed corpus the term 'tabl' would appear as a representative of both 'table' and 'tables'. In order to do this, we use Snowball[3], which is a language for implementing stemming algorithms and a package containing a number of stemmers. We use Snowball's default English stemmer.
5. **Extracting LDA Topics.** With these pre-processing steps completed, we compute the topics with LDA. We use the LDA implementation provided by the JGibbLDA Library[4]. The user is asked for the parameters to use in the computation, with some sensible defaults offered as suggestions by Topic$_{XP}$. Once we have the options the user would like to use, we extract topics from the documents, or, in the case of packages, we assign the documents to the topics already extracted from the system's classes.
6. **Assigning Documents to Topics.** Using probabilistic topic distributions of documents acquired by LDA from source documents, these documents are assigned to a top

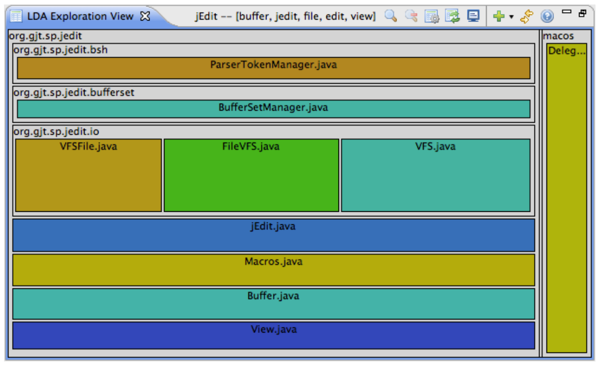


Figure 2 The Topic Contents View. Each box represents a class. Box size is determined by the degree the class belongs to the topic. Box color is tied to the MWE Cohesion metric.

[3] http://snowball.tartarus.org/
[4] http://jgibblda.sourceforge.net/

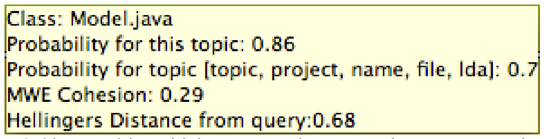
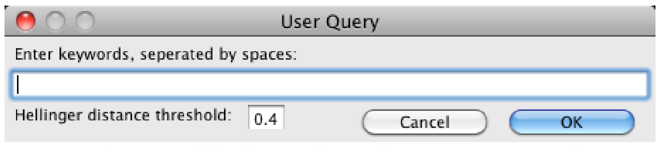

Class: Model.java
Probability for this topic: 0.86
Probability for topic [topic, project, name, file, lda]: 0.7
MWE Cohesion: 0.29
Hellingers Distance from query:0.68

Figure 3 Class tooltip, which appears when a user hovers over a class box in the topic contents view. This tooltip shows the actual numeric values for the metrics and models that our tool generates.



Enter keywords, seperated by spaces:
Hellinger distance threshold: 0.4

Figure 4 The query dialog box; Hellinger distances are computed between the user query and each class, and classes with distance above a threshold are hidden from view.

topic or topics for the purpose of the visualization either via a threshold level or a cut-off point, at the choice of the user.

7. **Extracting Dependencies**. In order to extract dependencies between the linguistic topics, we utilize X-Ray[5], an open source tool which computes and visualizes dependencies between classes or packages. Each call from one class to a method in another class is considered a dependency from the topic or topics to which the first class belongs to the topic or topics to which the second class belongs. Thus, the total weight of the dependency from one topic to another is simply the number of such method calls that exist from code in the first topic to code in the second.
8. **Computing Maximal Weighted Entropy (MWE)**. Finally, the method topic map created is used to calculate the MWE cohesion for each of the classes in the system being analyzed. This is an implementation of the metric described by Liu *et al.* [10] from our previous work. This metric is based on the idea that a cohesive class as modeled with LDA will represent a mixture of relatively few topics, since such a class would implement only closely related functions. A less cohesive class, on the other hand, would likely to be made up of many different topics as it implements many distinct functions, each of which likely involve the use of different words in the comments and identifiers. Thus, MWE models cohesion as a function of the entropy of the topic mixtures which comprise software class.

C. Visualizing Topics and their Relationships

The visualization that we designed for presenting the information to the user consists of two linked views (described in this section), which the user switches between as they first examine the topics and then explore the classes which are associated with each topic.

For this section we will be looking at the source code of *jEdit*[6] as our subject software to illustrate the functionality of Topic$_{XP}$. *jEdit* is an open source text editor written in Java, and since it is a relatively large, mature system it should provide a good idea on how Topic$_{XP}$ works on real systems. The tool extracted 11,729 unique tokens from *jEdit* version 4.3, found in 547 classes, 5,610 methods, and 40 packages.

Topic Dependency View. The first view, seen in Figure 1, presents to the user the topics extracted with Topic$_{XP}$. Each topic is represented as a box labeled with information that gives the user an idea of what the topic contains. A list of important keywords associated with the topic is displayed on the first line, followed by a list of the class or package documents most associated with the topic. From this information, the user should be able to start to ascertain the concepts that each topic encapsulates. For instance, in Figure 1 we can see the topic with keywords *line, text, textarea, start and buffer*, which clearly is associated with text areas, and the topic with keywords *key, plugin, action, event, and jedit*, which appears to be associated with keyboard input and other GUI functions.

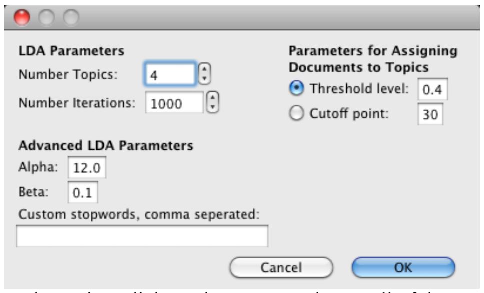
This view also displays the dependency information that the tool extracts. Arrows are drawn to indicate a dependency in the originating topic on the topic that the arrow points to, with the size and color of the arrows tied to the weight of the dependency. In Figure 1, we can see that the topic associated with text areas is dependent on the topic associated with keyboard input and other GUI functions.

Topic Contents View. When the user is ready to further investigate a particular topic, they click on the topic and are redirected to the second view provided by our tool, as illustrated in Figure 2. This view visualizes the contents of a topic, whose most relevant keywords are displayed in the toolbar, as a tree-map. Conceptually, tree-maps visualize data points such that each data point is assigned space proportional to a numeric property that it possesses. Tree-maps also preserve hierarchy in their visualizations, by drawing siblings next to each other and by assigning groups an amount of space proportional to the values of all their child data points. Algorithmically, we use a simple slice-and-dice approach which recursively splits the available space between all of the sibling elements at a given hierarchical level, assigning each a slice of the available space proportional to the element's total weight over the weight of it and all its siblings [3].

Thus, in our visualization classes are represented as boxes within the package hierarchy. The size allocated to a particular class corresponds to the probability with which that class is associated with the topic being examined. In other words, the size is proportional to the probability squared, since this makes it easier to visually discern the probability. In this way, the user is able to see, at a glance, the most important documents in a topic and move from a conceptual understanding to program comprehension that takes into account the class structure of the system. Double clicking on a class box allows the user to easily open the corresponding class in the Eclipse editor to further examine it at the level of the source code itself.

The main views of Topic$_{XP}$, which were discussed in detail in this section, are part of the typical user scenario, as summarized in Table 1. (*i.e.*, developer explores the Topic Dependency View, finds a relevant class in a selected topic, and then it analyzes its source code).

[5] http://xray.inf.usi.ch/xray.php
[6] http://www.jedit.org/



LDA Parameters

Number Topics: 4

Number Iterations: 1000

Parameters for Assigning Documents to Topics

[✓ Threshold level:] 0.4

Cutoff point: 30

Advanced LDA Parameters

Alpha: 12.0

Beta: 0.1

Custom stopwords, comma seperated:

Figure 5 The options dialog. The user can change all of the parameters used to generate the LDA-based model.

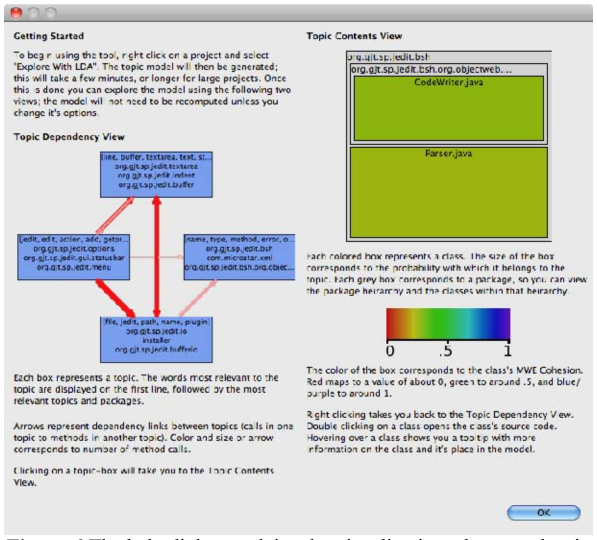
The topic contents tree-map view also displays some other information. The MWE cohesion of each class is visualized through the color of the box representing that class. The hue of the color is tied to WME cohesion metric, ranging from red for low values up through yellow and green and finally to blue for highly cohesive classes (see Figure 6). Additionally, hovering over a class brings up a tooltip, as shown in Figure 3, which allows the user to view the numeric values of all the measures computed by the tool.

Specifically, in Figure 2 we see a visualization for one of the topics from *jEdit* that deals with file accesses. Since this particular visualization used a cut-off point of ten documents per topic, all the visible classes have relatively high probabilities of being associated with this topic. However, these classes have a wide variety of MWE cohesion values, ranging from ParserTokenManager's value of 0.15 up to View's value of 0.91.

D. *Interactivity: Exploring the Visualization*

In addition to allowing the user to navigate between the views interactively, we also provide a few other ways for the user to interact with the implemented visualization.

Changing Model Options. The user can invoke the options dialog (see Figure 5), in order to change the options used to create the model. Most notably, the user can change the number of topics to be generated, as well as other LDA parameters, such as a number of iterations used and values of alpha and beta.



Getting Started

To begin using the tool, right click on a project and select "Explore With LDA". The topic model will then be generated; this will take a few minutes, or longer for large projects. Once this is done you can explore the model using the following two views; the model will not need to be recomputed unless you change it's options.

Topic Dependency View

line, buffer, textarea, text, st...
org.gjt.sp.jedit.textarea
org.gjt.sp.jedit.indent
org.gjt.sp.jedit.buffer
jedit, edit, action, add, getpr...
org.gjt.sp.jedit.options
org.gjt.sp.jedit.gui.statusbar
org.gjt.sp.jedit.menu
name, type, method, error, o...
org.gjt.sp.jedit.bsh
com.microstar.xml
org.gjt.sp.jedit.bsh.org.objec...
file, jedit, path, name, plugin
org.gjt.sp.jedit.io
installer
org.gjt.sp.jedit.bufferio

Each box represents a topic. The words most relevant to the topic are displayed on the first line, followed by the most relevant topics and packages.

Arrows represent dependency links between topics (calls in one topic to methods in another topic). Color and size of arrow corresponds to number of method calls.

Clicking on a topic-box will take you to the Topic Contents View.

Topic Contents View

Each colored box represents a class. The size of the box corresponds to the probability with which it belongs to the topic. Each grey box corresponds to a package, so you can view the package heirarchy and the classes within that heirarchy.

The color of the box corresponds to the class's MWE Cohesion. Red maps to a value of about 0, green to around .5, and blue/purple to around 1.

Right clicking takes you back to the Topic Dependency View. Double clicking on a class opens the class's source code. Hovering over a class shows you a tooltip with more information on the class and it's place in the model.

Figure 6 The help dialog explains the visualizations that our plug-in provides and how to navigate them. Notably, the spectrum of colors seen at right illustrates how class color varies with respect to MWE cohesion.

Table 1 Main usage scenarios of Topic$_{XP}$

Step	Output	Description
Invoke tool	Topic model	User supplies parameters for analysis and selects a project to analyze.
Explore Topic Dependency View	Topics & dependencies	Browsing topics and their keywords, user gains an idea of the concepts found in the source. (Figure 1)
Explore Topic Contents View	Classes & packages relevant to each topic	Exploring the packages and classes assigned to each topic allows user to learn more about distribution of concepts in source docs. (Figure 2)
Explore source code	Source code classes	User finally drills down to the actual source code which implements the concept, being guided by the Topic$_{XP}$ instead of searching for it manually.

A text field is also provided to add custom stop words specific to the project being analyzed. Finally, the user can also change the way that documents are associated with topics, switching between a threshold and cutoff and altering the values at which the threshold or cutoff occurs.

User Query. We also provide a mechanism by which the user can filter the classes displayed by the use of natural-language queries. The user enters a query and a threshold for hiding classes based on their distance from the query, as seen in Figure 4. The query is then processed by JGibbLDA so that it can be given probabilities with respect to already generated topics in the class topic model. The Hellinger distance between the query and each class is then calculated, and classes whose distance is greater than the user's threshold are filtered from the view.

This provides a form of concept location through LDA [11], allowing users to limit the visualization to a set of classes with relevancy to a concept represented by their query. When the user is done with their query, they can remove it to return to the full visualization.

Edge Filtering. Another parameter that the user can change is the threshold for displaying the arrows which represent dependencies between packages. In this way, the user can hide edges with less weight and only look at the edges which indicate potentially stronger dependencies.

III. EXPERIMENTAL EVALUATION

We conducted a user study to investigate the usefulness of the Topic$_{XP}$ tool during software maintenance tasks. More specifically, we were interested in finding out if this tool helps developers perform their tasks faster or if it helps developers find more relevant results.

Systems. We conducted this study on two open-source systems. The first one is *jEdit*[7] version 4.3, a textual editor written in Java, which has 98K lines of code in 483 files. The second system is *muCommander*[8] version 0.84, a cross-platform file manager implemented in Java, with 63K lines of code in 879 files.

Participants. The participants of this user study were four Ph.D. students from W&M. Two of them had experience working in Eclipse and with open-source software (identified as E_1 and E_2) and the other two had less

[7] http://www.jedit.org/

[8] http://www.mucommander.com/

Table 2 Distribution of tasks for developers. E_1 and E_2 are the experienced Eclipse developers while N_1 and N_2 are the developers with less experience with Eclipse. T_1 and T_2 are $Task_1$ and $Task_2$. The bold tasks were performed using $Topic_{XP}$, and the other tasks were performed using Eclipse

Subjects	Tasks (completed form left to right)			
E_1, N_1	**T_1 jEdit**	**T_2 jEdit**	T_1 muCmd	T_2 muCmd
E_2, N_2	T_1 jEdit	T_2 jEdit	**T_1 muCmd**	**T_2 muCmd**

experience (identified as N_1 and N_2). The participants were remunerated $15 for 1.5 hours of their time for this study.

Tasks. The participants were required to perform concept location [15] for four maintenance tasks, two for each system. For two of the tasks, the participants were required to use only the features available in the Eclipse IDE (*i.e.*, searching by queries, browsing files, following static dependencies, etc.), and for the other two tasks, the participants were required to use the features provided by the $Topic_{XP}$ tool. Table 2 shows the distribution of tasks for each participant in the study.

Collecting Data. In order to collect the data, we used the DevMon [6] plug-in to monitor the interactions of developers with the Eclipse IDE, such as what files were opened/closed and from what Eclipse view, what queries the developers formulated, etc. We extended DevMon plug-in to capture specific interactions between the developers and the $Topic_{XP}$ plug-in, such as what parameters they used, what topics they selected, what classes they opened from a topic, etc. The detailed interactions, which included the exact time and event, were stored for later analysis.

Procedure. The developers were given a package, which contained Eclipse 3.5 IDE, $Topic_{XP}$ and DevMon plug-ins installed, and the code for *jEdit* and *muCommander* in the workspace. They were also provided instructions on how to run the experiment, how to use $Topic_{XP}$, the order in which they had to perform the tasks, and the actual maintenance tasks[9]. They were required to spend no more than 5-15 minutes on each task, and after completing a task, they selected an option from DevMon to record the end of their tasks, as well as the list of files they considered to be related to the task, which was used for analysis of the results.

Time analysis. Figure 7(a) shows box plots for the amount of time (expressed in minutes) the subjects needed to complete their tasks. The leftmost box plot represents the time needed by subjects E_1 and N_1 to complete the *jEdit* tasks using the $Topic_{XP}$ plug-in, whereas the second box plot represents the time needed by E_2 and N_2 to finish their tasks for *jEdit* using only the Eclipse features. Similarly, the last two box plots are for the *muCommander* system. Note that each box plot presented in this paper is composed from four data points. We can observe that for both systems the participants were able to finish their tasks in about the same amount of time, regardless of using Eclipse or $Topic_{XP}$. These results are extremely encouraging considering the lack of experience and little training of participants with $Topic_{XP}$. We are confident that with more training with $Topic_{XP}$ developers will be able to finish their tasks much faster. We also observed that developer E_1 used the "User Query" feature of $Topic_{XP}$ (see section II.D) 15 times, which due to

[9] http://www.cs.wm.edu/semeru/TopicXP

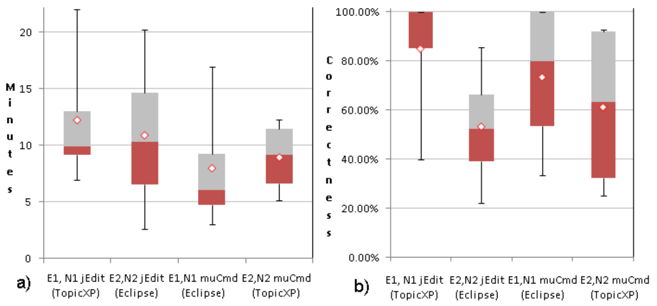


Figure 7 a) Box plots for time spent (minutes) on tasks; b) Box plots for correctness (percentage) of tasks

the implementation of the tool may take up to two minutes to complete (*i.e.*, there was a lot of idle time waiting for the tool to finish). This means the data presented in Figure 7(a) is an upper bound time, and with some optimization to the $Topic_{XP}$ tool, the tasks will be completed much faster.

Correctness Analysis. Each of the 16 tasks performed by the subjects resulted in a list of files the subjects considered relevant to the task. One of the authors of this paper, who used these systems in the past, and who is familiar with their internals, examined these files individually and determined if they are relevant to the task or not. After that, a correctness measure was associated with each task, and the correctness measure is defined as the number of files considered relevant by one of the authors, divided by the total number of files considered relevant by the subjects. Note that this measure was used as we do not have a gold set for these tasks to compute the traditional precision or recall measures. Figure 7(b) shows box plots for the correctness of the tasks, and we can observe that the correctness for the *jEdit* tasks is much higher when using the $Topic_{XP}$ plug-in, than in the case of using Eclipse. However, for the *muCommander* tasks, the correctness using Eclipse is slightly higher than the correctness of using $Topic_{XP}$. One explanation for the superiority of $Topic_{XP}$ for *jEdit* is that the topics produced by $Topic_{XP}$ were very relevant to the task. However, further investigation is needed.

Other observations. Figure 8(a) shows that in the case of *muCommander*'s tasks, the participants in the study opened on average 11.5 files using Eclipse features (Figure 8(a), third box plot), whereas using $Topic_{XP}$ the participants opened on average 18.5 files (Figure 8(a), last box plot).

We also looked at the number of times developers used the Eclipse search feature and the $Topic_{XP}$ user query feature. The results from Figure 8(b) show that the developers performed about the same amount of searches and query features for *jEdit*, but not as many $Topic_{XP}$ user queries for *muCommander*. Looking at the data, we saw that developer E_2 did not perform any $Topic_{XP}$ user queries, which may be due to the fact that she was unfamiliar with this feature. This can be corrected in the future with more thorough training.

We also observed that developers used the Topic Contents View of $Topic_{XP}$ on average ten times for *jEdit* and about eight times for *muCommander*.

One of the developers (E_2) used the $Topic_{XP}$ tool to complement Eclipse. She identified a few classes using $Topic_{XP}$, and after that he explored other classes (using

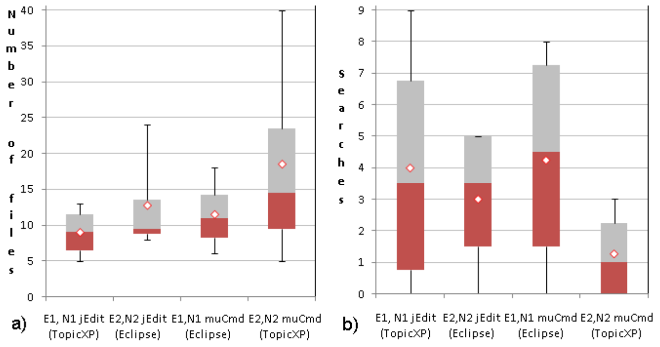


Figure 8 a) Number of files opened using Topic$_{XP}$ or Eclipse during the maintenance tasks; b) The number of Eclipse searches (*i.e.*, Eclipse File Search and Eclipse Java Search) and Topic$_{XP}$ searches (*i.e.*, user queries).

Eclipse's Package Explorer view) that were in the same package as the classes identified by Topic$_{XP}$.

Threats to validity. In this study we decided to use two software systems to minimize the chances that the tool would be useful only for a particular type of system. In addition, we used programmers with different levels of experience to test whether experience is a determining factor for using this tool, but so far it seems that both experienced and less experienced developers benefited from this tool. Among other factors that might have influenced the results of this study, we mention possible unfamiliarity of the subjects with Topic$_{XP}$ or the lack of understanding properly prearranged maintenance tasks.

Final remarks. Given the number of subjects we cannot generalize these results, however, we can state that Topic$_{XP}$ has a great potential. This is because we observed that even with little training developers could use it to get meaningful results and in some cases even better results than using Eclipse alone, at the cost of more time, which in turn can be reduced by optimizing the tool. As Topic$_{XP}$ will improve, we plan in the future to conduct user studies that would aim at statistically evaluating key features of this tool.

IV. Related Work

A number of techniques have been tried previously to help developers acquire an overview of the source code that makes up software. Many of these have involved graphical visualizations of the relationships of classes, such as the Software Maps by Kuhn *et al.* [9]. Kuhn *et al.* also proposed a topic-based visualization of software classes, called a distribution map [8], however the distribution map was designed for a system using LSI and clustering to create topics as opposed to using LDA as per our tool. It should also be noted that Topic$_{XP}$ utilizes structural information and computes conceptual cohesion of classes (*i.e.*, WME) based on measuring probabilistic distribution of topics among methods in classes using information entropy.

The idea of using topic maps generated via LDA to examine source code has also been previously investigated. Maskeri *et al.* [12] studied using topics extracted with LDA from a software system, treating each source file as a document. Ahindle *et al.* [7] treated CVS commits during a certain time period as documents, and then examined how extracted topics changed between time periods. Baldi *et al.* [2] proposed equating latent topics scattered through different source files, as discovered using LDA, with aspects, and conducted a study using LDA at both the repository level and the project level to validate this claim. Tian *et al.* [16] applied latent topics to the task of automatically categorizing software systems. Finally, LDA has been used for traceability link recovery [1] and compared to other similar Information Retrieval based approaches [13].

V. Conclusion

This paper presents Topic$_{XP}$, an open-source Eclipse plug-in, for exploring topic-mapped classes and packages. This tool is designed to allow developers to learn about the concepts, or latent topics which comprise a software system while also observing the distribution and dependencies of these topics, as well as the cohesiveness of the underlying source code classes. The results of a user study indicate that Topic$_{XP}$ is a functional tool, which can effectively assist developers in completing routine software maintenance tasks.

VI. Acknowledgments

This work is supported by NSF CCF-1016868 and NSF CCF-0916260 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Asuncion, H., Asuncion, A., and Taylor, R., "Software Traceability with Topic Modeling", in ICSE'10.

[2] Baldi, P., Linstead, E., Lopes, C., and Bajracharya, S., "A Theory of Aspects as Latent Topics", in OOPSLA'08, pp. 543-562.

[3] Ben, S., "Tree visualization with tree-maps: 2-d space-filling approach", *ACM Trans. Graph.*, vol. 11, no. 1, 1992, pp. 92-99.

[4] Blei, D. M., Ng, A. Y., and Jordan, M. I., "Latent Dirichlet Allocation", *Journal of Machine Learning Research*, vol. 3, 2003, pp. 993-1022.

[5] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[6] Dit, B., *Monitoring the Searching and Browsing Behavior of Developers in Eclipse during Concept Location*, Wayne State University, Detroit, MI, 2009.

[7] Hindle, A. J., Godfrey, M. W., and Holt, R. C., "What's Hot and What's Not: Windowing Developer Topic Analysis", in ICSM'09.

[8] Kuhn, A., Ducasse, S., and Gîrba, T., "Semantic Clustering: Identifying Topics in Source Code", *IST*, vol. 49, no. 3, March 2007, pp. 230-243.

[9] Kuhn, A., Loretan, P., and Nierstrasz, O., "Consistent Layout for Thematic Software Maps", in WCRE'08, pp. 209-218.

[10] Liu, Y., Poshyvanyk, D., Ferenc, R., Gyimóthy, T., and Chrisochoides, N., "Modeling Class Cohesion as Mixtures of Latent Topics", ICSM'09, 233-242.

[11] Lukins, S., Kraft, N., and Etzkorn, L., "Source Code Retrieval for Bug Location Using Latent Dirichlet Allocation", in WCRE'08, pp. 155-164.

[12] Maskeri, G., Sarkar, S., and Heafield, K., "Mining Business Topics in Source Code using Latent Dirichlet Allocation", in ISEC'08.

[13] Oliveto, R., Gethers, M., Poshyvanyk, D., De Lucia, A., "On the Equivalence of IR Methods for Automated Traceability Link Recovery", in ICPC'10.

[14] Pierret, D. and Poshyvanyk, D., "An Empirical Exploration of Regularities in Open-Source Software Lexicons", ICPC'09, pp. 228-232.

[15] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *TSE*, 33/6, pp. 420-432.

[16] Tian, K., Revelle, M., and Poshyvanyk, D., "Using LDA for Automatic Categorization of Software", MSR'09, pp. 163-166.