ReUse: A Recommendation System for Implementing User Stories

Heidar Pirzadeh, Andre de Santi Oliveira, Sara Shanian
SAP SE, SAP Hybris
Montreal, Quebec, Canada
email: {heidar.pirzadeh, andre.de.santi.oliveira, sara.shanian}@sap.com

Abstract— **In agile software development, user stories contain feature descriptions that are used as the entry points of discussions about the design, specification, requirements, and estimation of the software features. The first step in implementing a user story is to find proper files in the code base to make changes. To help the developers, in this paper, we describe a new approach that automatically recommends the files where a feature will most likely be implemented based on a given user story that describes the feature.**

Keywords—Recommendation System; Text Mining; Program Comprehension; Information Retrieval; Agile Software Development

I. INTRODUCTION

Since 2010 more than 200 big software companies have adopted Agile software development methodology [1]. Important motivations behind the widespeard adoption of Agile software development are fast delivery to the market, efficient handling of new requirements, and increased overall productivity of development teams.

The development team in an Agile setting typically receives the new requirements in the form of a user story (hereinafter interchangeably referred to as story). A user story is a very high-level definition of a requirement, which contains enough information so that the developers can produce a reasonable estimate of the effort to implement it. Given a story, developers go through three basic steps of 1) identification of code locations as starting points, 2) finding and applying a solution, and 3) testing and validating the implemented change.

Any delay in one of the above-mentioned steps will result in a delayed implementation of the story and undermines the important goal of fast delivery. In fact, it has been shown that developers could get stuck in the first step of finding the right location to start making their changes to implement the request [2], [3]. While experienced developers are usually faster in identifying and understanding the subset of the code relevant to the intended change, studies have shown that developers spend up to 50 percent of their time searching for information [4], [5] to answer their questions about the system under development.

The reason that experienced developers are faster in their identification step is because of their higher familiarity with the system and with the previously implemented similar requirements. Their knowledge and experience makes them a valuable source for answering others' questions during their program comprehension [6]. If an experienced developer leaves the team, usually, part of that knowledge will also go with him. We think that externalizing this knowledge and how it is gained could enable everybody in the team to speed up and improve their deliveries and, in turn, make the team less prone to personnel changes. A recommendation system that could provide team members with suggestions to help with identification of changes locations seems like a perfect fit for this scenario.

Once implemented, a user story could be usually mapped to the creation or modification of one or more classes in the code base. Many companies use ticket management (e.g., JIRA [12]) and code management (e.g., Bitbucket [13]) ecosystems to respectively maintain and store the stories, the code and the mapping between them. For example, each story in JIRA has a ticket number. When committing the code that implements a story to Bitbucket, the developer could include the story's ticket number in the commit message so that JIRA creates a link between the story and the committed code. The information accumulated in ticket/code management systems could be leveraged by the recommendation system in creating insightful recommendations that could help developers in faster and more reliable deliveries.

In this paper, we propose ReUse a recommendation system that employs techniques from information retrieval, text mining, and the field of recommender systems to automatically suggest a list of files where a story will most likely be implemented. We evaluated the effectiveness of our recommendation system in an industrial setting on the Order Management System (OMS) product at SAP Hybris. The results show that our recommendations are of 71% precision.

We start by reviewing a few related works in Section II. In Section III, we describe the proposed approach. Section IV discusses the results of our case study on OMS. In Section V, we discuss the threats to validity. We conclude the paper and describe future avenues in Section VI.

II. RELATED WORK

To the best of our knowledge, no other work has been reported on a recommendation system for user stories in Agile software development. However, a few works have been done on recommendation systems for bug localization during software maintenance. Kim et al. [11] propose a prediction model to recommend files to be fixed. In their model, the features are created from textual information of already existing bug reports, then Naive Bayes algorithm is applied to train the model using previously fixed files as classification labels, and then use the trained model to assign multiple source files to a given new bug report. Our evaluation of the effectiveness of a recommendation is also quite different. They "consider the prediction results to be *correct* if at least one of the recommended files matches one of the actual patch files for a given bug report". We think for a developer to go through the recommended files he needs to be assured about the precision of the list. Zhou et al. [3] proposed a revised Vector Space Model approach for improving the performance for bug

 ISBN: 978-1-61208-498-5

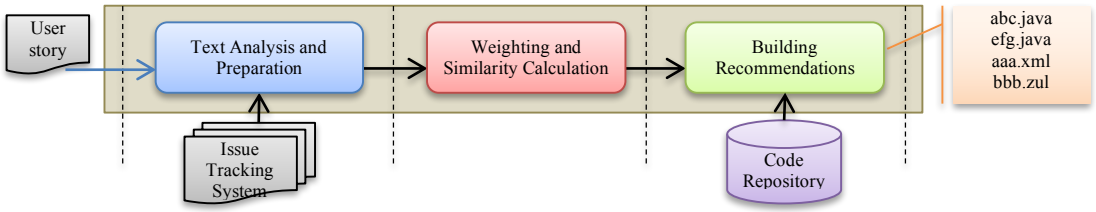

Figure 1. Overview of the ReUse recommendation system

localization. They measure the lexical similarity between a new bug report and every source file and also give more weight to larger size files and files that have been fixed before for similar bug reports. Their approach relies on good programming practices in naming variables, methods and classes. In comparison, our approach is independent of file names or the content of the files.

III. The Proposed Approach

The idea behind our approach is based on the assumption that there are naturally occurring groups of user stories in any project that can be identified by looking at their description. By using such information, our system could provide recommendations for a new user story by first finding the group of user stories it belongs to by computing its similarity to existing user stories. Thus, a user story could have a set of nearest neighbors that can be used to make recommendations about the files needed to implement that user story based on the files that were modified during the implementation of similar user stories.

As shown in Fig. 1, our recommendation system performs its task through three main steps:

- Analyzing the textual content of the new story to prepare it for next steps by tagging the content with meta information,
- Creating a weighted vector of the new story and use it in finding other stories that are closely similar to it, and
- Preparing a recommendation for the new story by providing a precise set of recommended files to be used by developers.

A. Text Analysis and Preparation

A common pre-processing step in many information retrieval approaches is one that removes stop-words - The words that add little value to the process of finding relevant information. Stop-word identification, which is the process of identifying these words, makes use of domain and global information. For example, in the domain of English literature, stop-words include auxiliary verbs (e.g., have, be), pronouns (he, it), or prepositions (to, for).

In our text analysis and preparation, our stop-word remover component uses Lucene's StandardAnalyzer to remove the terms in a user story that are listed in a set of common English stop-words dictionary. Another pre-processing step in our approach is *Stemming*. A stemmer maps different forms of a term to a single form. A stemmer, for example, could strip the "s" from plural nouns, the "ing" from verbs, and so on to extract the stem of the term. That way, a stem could act as a natural group of terms with a similar meaning. Our stemmer component uses Lucene's EnglishStemming to find the stems; the English stemmer is an updated version of the famous Porter Stemmer [9].

Whether it is in a query or in a document some terms can represent different meaning depending on the role that they take. The role is even more important when there are processes like *Stemming* that changes a term to an alternative (usually simpler) that could result in an unjustifiable similarity while the original form of the term had a different meaning because of the role it had. For example, the term "dogs" has a different meaning in the sentence "The sailor walked the dogs" in comparison to the meaning that it has in "The sailor dogs the hatch" because of its roles that are correspondingly *noun* in the first sentence and *verb* in the second one. This role is also referred to as the part-of-speech (POS) for that term. The similarity between two similar looking terms should increase only if their roles are the same in the places that the word has appeared in. That is why we perform a POS tagging on a user story before we pass it to our stemming component. Our current POS tagger component is implemented using Maxent part-of-speech tagger from the Stanford NLP group.

B. Weighting and Similarity Calculation

We use a weighting process for finding representative terms in each user story and add them to a corresponding *terms vector* that is weighted based on the representativeness of each term for that user story. Our weighting function is implemented as a Term Frequency, Inverse Document Frequency (TF-IDF) [10] schema. The goal of TF-IDF term weighting is to obtain high weights for terms that are representative of a document's content and lower weights for terms that are less representative.

In our case, the weight of a term depends both on how often it appears in the given story (term frequency, or *tf*) and on how often it appears in all the stories (document frequency, or *df*) of the ticket management system. In general, a high frequency of a term (high *tf*) in one story shows the importance of that term while if a term is scattered between different stories (high *df*), then it is considered less important. Therefore, if a term has high *tf* and low *df* (or high *idf* -inverse document frequency) it will have a higher weight. Since the importance of a term does not increase proportionally with the term's frequency, the weight of term i in story k is calculated as shown in (1):

$$w_{i,k} = \frac{\left(log\left(tf_{i,k}\right) + 1\right) * \log\left({}^{N}/_{n_i}\right)}{\sqrt{\sum_{j=1}^{e}\left[\left(\log\left(tf_{j,k}\right) + 1\right) * \log\left(\frac{N}{n_j}\right)\right]^2}} \quad (1)$$

where term frequency $tf_{i,k}$ of term i in story k is the number of times that i occurs in k, N is the total number of stories, n_i is the number of stories where the term i has appeared in and e is the total number of terms. The factor $\log\left({}^{N}/_{n_i}\right)$ is the "*idf*" factor that decreases as the terms are used widely in all user stories. The denominator in the equation is used for weight

normalization. This factor is used to adjust the terms vector of the story to its norm, so all the stories have the same modulus and can be compared no matter the size of the story.

Once we have the terms vector of each story ready, we need to measure the similarity between stories. The similarity between two objects is in general regarded as how much they share in common. In the domain of text mining, the most commonly used measure for evaluating the similarity between two documents is the cosine of the angle between term vectors representing the documents. In the same way, as shown in (2), we calculate the similarity between two stories x and y by measuring the cosine similarity between their terms vectors V_x and V_y:

$$S(V_x, V_y) = \frac{\sum_{i=1}^{n} w_{i,x} * w_{i,y}}{\sqrt{\sum_{i=1}^{n} (w_{i,x})^2} * \sqrt{\sum_{i=1}^{n} (w_{i,y})^2}} \quad (2)$$

where $w_{i,x}$ and $w_{i,y}$ are respectively the weight of term i in vectors V_x and V_y, and the denominator of the fraction is for normalization. The weights cannot be negative and, thus, the similarity between two vectors ranges from 0 to 1, where 0 indicates independence, 1 means exactly the same, and in-between values indicate intermediate similarity.

C. Building Recommendations

Recommendation systems are now part of many applications in our daily life. These systems provide the user with a list of recommended items and help them to find the preferred items in the bigger list of available items [7], [8]. Our recommendation system is based on collaborative filtering. In collaborative filtering, the items are recommended to the users based on the previously rated items by the other users. Mapping the idea back to our case, our recommendation system should recommend files for a new user story based on the previously modified files by other user stories. More formally, as shown in (3), the usefulness of file f for implementing story s noted as the utility $u(s, f)$ of file f for story s has the following form

$$u(s, f): \text{Func}(u(s_i, f)) \qquad \forall s_i \in C_s \quad (3)$$

where $u(s_i, f)$ is the utility assigned to the file f for story s_i in C_s the set of stories that are similar to story s. Different utility functions could be plugged into our recommendation system to be used in creating recommendations. When building the recommendation, our goal is to provide the user with a highly precise list of recommended files that our recommendation system deems necessary to implement the user story.

The recommendation can help developers in building their mental model in a quicker and more accurate way. If the developer is already familiar with the code base, she could use the recommendation as a potential checklist to increase her confidence in the changes that are planned. The basic idea in creating a precise recommendation is to find the files that are associated with similar stories and are frequently changed to implement those stories. More formally, as presented in (4), to build the recommendation, we calculate the utility u_F of file f for a given story s as follows

$$u_F(s, f) = \begin{cases} \sum_{i=1}^{n} u(s_i, f) * S(V_s, V_{s_i}) & \text{if } S(V_s, V_{s_i}) > t \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

where $S(V_s, V_{s_i})$ is the calculated similarity between the given story s and a similar story s_i, t is a certain cut-off threshold, n is the maximum number of similar stories to be considered, and $u(s_i, f)$ is the utility of file f for story s_i which is defined as $u(s_i, f) = c_{i,f}$ where $c_{i,f}$ is the number of commits in which the file f has appeared for implementing story s_i.

IV. Evaluation

To evaluate the effectiveness of the ReUse recommendation system, we use it in an industrial setting on the OMS project at SAP Hybris. The OMS enables customers to flexibly pick and choose from a set of omni-channel order management and fulfillment functionalities. We use release 5.7 of OMS in 2015. This version of OMS contains 3018 files in 928 packages. The total number of tickets (excluding bugs) for this release was 176. All 176 tickets were already implanted at the time of evaluation and each ticket was linked to the modified files in the Git repository management system Bitbucket. The tickets were extracted from JIRA as a CSV file. Although the exported file contained many attributes for each ticket we only kept *Summary* (the name of the ticket), *Description*, *Issue Type*, *Ticket ID*, *Sub-task ID*, *Parent ID* for the experiment.

Technically, the goal in our evaluation is to find out how effectively our recommendation system can predict the set of files that need to be changed for each story and compare the recommended set with the actual set of files that were modified for that story. This way, our recommendation problem could be seen as a classification problem where our recommendation algorithm tries to classify the source code files into two class of relevant and irrelevant for each story. The effectiveness of classification in this case would be the rate of true and false predictions that the algorithm makes. These rates can be arranged in a contingency table that is called the confusion matrix (see Table I).

TABLE I. CONFUSION MATRIX

	Relevant	Irrelevant	
Recommended	TP	FP	TP + FP
Not recommended	FN	TN	FN +TN
	TP + FN	FP + TN	

As seen in Table II, True Positive (TP) is the number of correctly predicted the relevant files. False Positive (FP) is the number of incorrectly predicted relevant files. False Negative (FN) is the number of incorrectly predicted irrelevant files. True Negative (TN) is the number of correctly predicted irrelevant files.

As shown in (5), *Precision* or true positive accuracy is calculated as the ratio of recommended files that are relevant to the total number of recommended files:

$$precision = \frac{TP}{TP + FP} \quad (5)$$

Recall or true positive rate, as presented in (6), is calculated as the ratio of recommended files that are relevant to the total number of relevant files:

$$recall = \frac{TP}{TP + FN} \quad (6)$$

 ISBN: 978-1-61208-498-5

Specificity or true negative accuracy is calculated as the ratio of not recommended files that are irrelevant to the total number of irrelevant files as shown in (7):

$$specificity = \frac{TN}{TN + FP} \quad (7)$$

Then as presented in (8) *Accuracy* is calculated as the ratio of correct predictions:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (8)$$

In our evaluation, we needed to have a portion of our tickets as a training set for the recommendation system to use them to build recommendation and a second portion as our test set (set of stories that we want to feed to the recommendation system and evaluate the suggestions that the system provides for each of those stories). To avoid any bias in the selection of the training and the test set we use k-Folds Cross Validation. In k-Folds cross validation, sometimes called rotation estimation, the data set D is randomly spilt into k mutually exclusive subsets (the folds) $D_1, D_2, \ldots, D_k$ of approximately equal size. The algorithm is trained and tested k times; each time $t \in \{1, 2, \ldots, k\}$, it is trained on $D \backslash D_t$ (i.e., D minus D_t) and tested on D_t.

The result presented in this section uses the following configuration: we fold the data by splitting our set of 176 stories randomly into 18 sets (roughly 10 stories per set). On each iteration, we use 17 sets as our training set and 1 set as the test set. That is, a cross validation (k = 18). In our experiment we only consider the most similar story (n = 1) and the cutoff threshold is (t = 0.5). The number of files that will be recommended in this case is equal to the number of files modified files for the most similar story. The following table shows the result of our evaluation as the average of 18 iterations.

TABLE II. EVALUATION RESULTS

Metric	Value
Precision	0.7125751
Recall	0.4658216
Accuracy	0.9178191
Specificity	0.9363258

Execution of the experiment on a typical developer machine (Intel Core i7 2.5 GHz processor of 4 cores and 16 GB of Ram) took less than 30 seconds. This time includes the time for training and the time to run the test of each iteration. As shown in the Table I, the files that our recommendation system suggests to the developers in the recommendation are 71% of the time the files that they certainly needed to make a change to implement the user story. At the same time, our recommendation system scores a very high specificity and accuracy. Our system, is successful in avoiding the recommendation of irrelevant files 93% of the time while in general makes a correct prediction 91% of the time in its recommendation.

We also performed a case by case analysis for the stories for which our recommendation system scored lower than 50% precision. One of such stories was OMSE-31, for which the precision of the recommended files was only 4%.

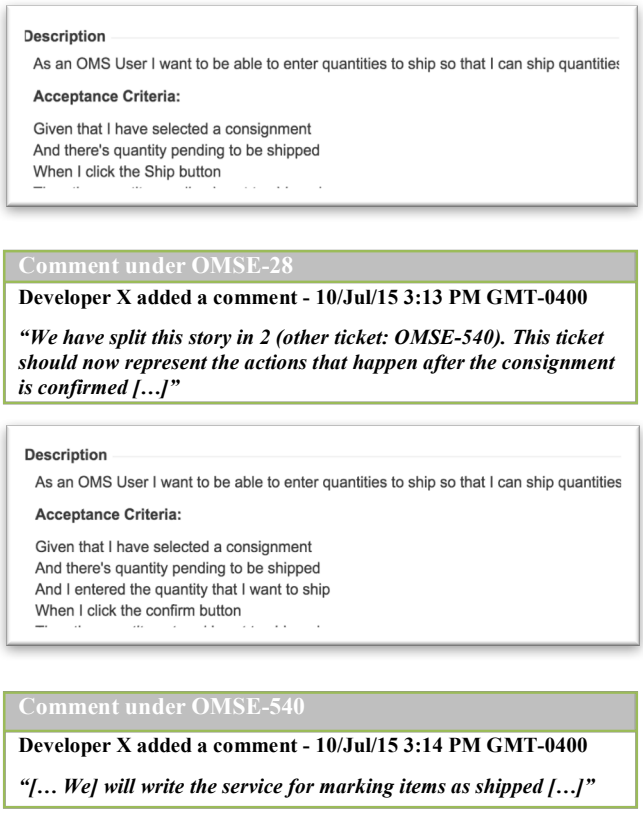

Figure 2. OMSE-28 and its comment (top), OMSE-540 and its comment (bottom)

Our investigation showed that, the story description was updated during a sprint but the previous content was not deleted (the content was rather formatted with ~~strikethrough~~). The csv parser in our system ignores all text formatting and could not detect such situations and as a result recommended files associated to a story that was similar to OMSE-31 considering the content that should have not been considered.

Another case was for story OMSE-540 with only 8% of precision. The recommendation system detected OMSE-28 as highly similar story and recommended the modified files accordingly. However, the list of files that was actually modified was significantly different than the predicted one. Further investigation showed that OMSE-28 was describing a feature from the end user perspective. While, as shown in Fig. 2, later on, the story was split into two smaller stories one to implement the user interface and the second one to implement the services in the backend that will be used by the user interface to implement the feature. For this, the developer cloned the original user story (OMSE-28) and created OMSE-540 and made a minimal change to the description. However, he left two comments, one for each story. The current version of our recommendation system does not take comments into account.

V. THREATS TO VALIDITY

There are potential threats to the validity of our work. The effectiveness of our recommendation system is highly dependent on the quality of the stories that the members of the Agile team maintain for their project. Although our recommendation system showed an impressive effectiveness

 ISBN: 978-1-61208-498-5

on the commercial project of OMS at SAP Hybris, not all teams or companies have similar level of standards when it comes to creating and maintaining their backlog of the stories. TF-IDF alone is prone to misspellings and multi-word verbs and expressions. To have a resilient approach we need to check the frequency of those cases and remove them.

VI. Conclusion And Future Work

In this paper we proposed an approach to help developers in during their implementation tasks by taking benefit from the suggestions that our recommendation system provides them on where to make code changes. Our recommendation system called ReUse, builds a precise recommendation list of files that are need to be changed with high probability. We evaluated our recommendation system on the OMS at SAP Hybris and the results show 71% precision in recommending the files that need to be changed.

For our future work, we would like to look into automatic fine tuning of parameters in our recommendation builder along with plugging in new utility functions to increase the recall and take advantage of our recommendation system on other projects at SAP. Taking other sources of information such as comments or the links to other tickets (the hierarchy of tickets) into account could help us take advantage of relations other than the textual and conceptual relation between stories to improve the results.

Handling the rich texts by the parser in our work, as shown in our case study, could potentially reduce the chances of inaccurate similarities and result in better recommendations. Also, adding components to our system to deal with misspelled words and expressions could also potentially be beneficial in detecting the similarities between stories.

References

[1] J. Little, "The List of Firms Using Scrum", [Online]. Available from: http://scrumcommunity.pbworks.com/w/page/10148930/Firms Using Scrum 2016.07.12

[2] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen, "A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report," Proc. IEEE/ACM 26th Int'l Conf. Automated Software Eng., pp. 263-272, 2011.

[3] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed?—More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports," Proc. 34th Int'l Conf. Software Eng., pp. 14-24, 2012.

[4] A. J. Ko, R. DeLine, and G. Venolia "Information Needs in Collocated Software Development Teams", In Proceedings of the 29th International Conference on Software Engineering, ICSE'07. IEEE, 2007.

[5] G. C. Murphy, M. Kersten, and L. Findlater "How Are Java Software Developers Using the Eclipse IDE?" IEEE Software pp. 76–83, 2006.

[6] Th. D. LaToza, G. Venolia, and R. DeLine "Maintaining mental models: a study of developer work habits". In ICSE '06: Proceeding of the 28th international conference on Software engineering. ACM, New York, NY, USA, 492–501, 2006.

[7] Y. Koren, R. Bell, "Advances in Collaborative Filtering", ch. 5. Springer,. Recommender Systems Handbook, 2011.

[8] L. Baltrunas, and F. Ricci. "Experimental evaluation of context-dependent collaborative filtering using item splitting" User Modeling and User-Adapted Interaction 24, no. 1-2: 7-34, 2014.

[9] M. F. Porter, "An algorithm for suffix stripping". Program, 14(3):130–137, 1980.

[10] T. Joachims. "Text Categorization with Support Vector Machines: Learning with Many Relevant Features". In Proc. of ECML98, 137-142, 1998.

[11] K. Dongsun, Y. Tao, S. Kim, and A. Zeller. "Where should we fix this bug? a two-phase recommendation model" Software Engineering, IEEE Trans. on 39, no. 11: 1597-1610, 2013.

[12] JIRA, [Online]. Available from: https://www.atlassian.com/software/jira 2016.07.12

[13] Bitbucket, [Online]. Available from: https://bitbucket.org/ 2016.07.12

 ISBN: 978-1-61208-498-5