How Software Developers Use Work Breakdown Relationships in Issue Repositories

C. Albert Thompson, Gail C. Murphy and Marc Palyart
University of British Columbia
{leetcat,murphy,mpalyart}@cs.ubc.ca

Marko Gašparič
Free University of Bozen-Bolzano
marko.gasparic@stud-inf.unibz.it

ABSTRACT

Software developers use issues as a means to describe a range of activities to be undertaken on a software system, including features to be added and defects that require fixing. When creating issues, software developers expend manual effort to specify relationships between issues, such as one issue blocking another or one issue being a sub-task of another. In particular, developers use a variety of relationships to express how work is to be broken down on a project. To better understand how software developers use work breakdown relationships between issues, we manually coded a sample of work breakdown relationships from three open source systems. We report on our findings and describe how the recognition of work breakdown relationships opens up new ways to improve software development techniques.

1. INTRODUCTION

For many software development projects, issue repositories hold key information defining what the system under development will do, who will work on different parts of the system, what defects occur as the system is being built, and more. When defining issues, software developers often expend manual effort to record relationships between issues, capturing such information as how work is to be broken down, how functionality in the system relates and which defects are similar to each other. Figure 1 shows the relationships defined for the `GATEWAY-3941` issue from the open source `Connect` project [2].These relationships describe on which other issues `GATEWAY-3941` depends, which issues depend on it, which issues it supports and which issues describe sub-tasks to complete as part of the work associated with `GATEWAY-3941`.

In the issue repository systems of which we are aware, there is little consistency as to the kinds and names of relationships supported. As one example, in a Bugzilla repository[1], relationships are typically entered as *depends-on* or *blocks*. The semantics of these relationships can vary between projects using Bugzilla. As another example, in a JIRA repository [4], users can define a variety of fields to hold relationships. Many JIRA systems are configured with four default relationships: *relates-to*, *duplicates*, *blocks* and *clones*. When JIRA is used for a project following an agile project management style, it is also common to see such relationships as *is-supported-by* and *subtasks*.

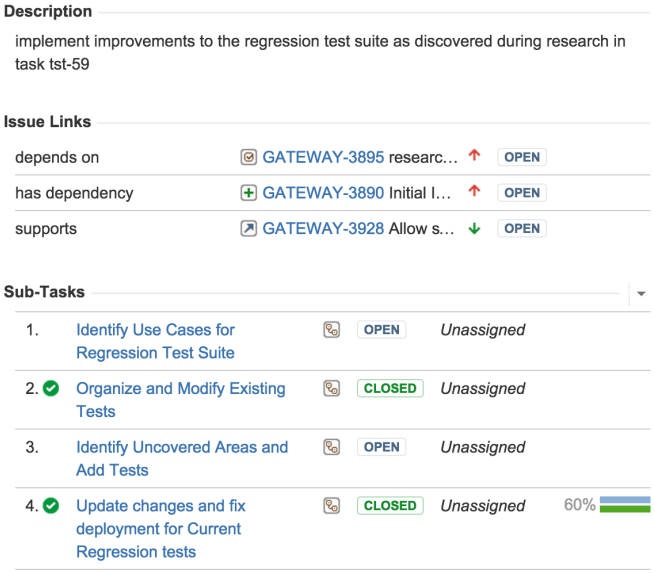

Figure 1: Example issue relationships

Table 1 shows the many kinds and instances of relationships developers specified largely manually in the issue repositories for three open source systems[1]: `Mylyn` [5], `Connect` [2] and `HBase` [3]. The data in Table 1 shows that thousands of relationship instances were specified for each system. When we examined the kinds of relationships in these repositories—by asking project developers on the forums they use and by analyzing documentation—we learned that the most frequently occurring relationships describe work breakdowns,[2] causing us to ask "how are software developers using work break down relationships in issue repositories"?

To investigate this question, we performed a qualitative study of a sample of work breakdown relationships from the

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901779

[1]The data reported is from April 30, 2015.

[2] In `Mylyn`, 59% of the relationships are `depends-on`, which represent work breakdowns. In `HBase` `sub-tasks` are work breakdowns representing 30% of all relationships specified. In `Connect`, 79% of relationships are work breakdowns via the `supported-by` and `subtask` relationships.

Table 1: Relationships instances from repositories using Bugzilla(*) or Jira(†)

	Mylyn *	Connect †	HBase †
Blocked			299
Breaks			47
Clone			32
Contains			12
Depends-on	2174	205	286
Duplicates	1520	43	159
Incorporates			208
Part-of-epic		559	
Requires			189
Relates-to			1901
Subtask		1643	1368
Supercedes			31
Supported-by		1361	
Total	3694	3811	4532

three aforementioned repositories. We had three researchers (authors of this paper) code the how these relationships were being used based on an analysis of the titles of selected issues. Through this coding, we determined six codes that describe the kinds of work breakdown relationships, ranging from describing particular cases in which a more general problem must be solved to describing how functionality should be verified.

A better understanding of how relationships are used in issue repositories and an ability to recognize different uses of relationships provides opportunities to create new, and improve existing, software development tools and methods. For instance, a new tool might predict missing tasks based on patterns in the issue repository. Or existing approaches might be improved, such as approaches to associate commits with issues; this kind of data is used to analyze and predict software development, such as change-inducing fixes [16]. Bird and colleagues showed that when such software engineering data is incomplete, bias can occur [9]. We discuss these possibilities in Section 4.

We begin by describing earlier efforts in characterizing issue repositories (Section 2). We then describe the qualitative study (Section 3) and discuss how better understanding of relationships can improve software development approaches (Section 4). We summarize the paper (Section 5).

2. RELATED WORK

A number of researchers have investigated how issues are used in development.

Mockus and colleagues characterize aspects of problem reports (i.e., issues), such as time to resolve a problem, as part of a characterization of open source development [13]. Anvik and colleagues describe how bug triage and duplicate bugs occur in source repositories [6]. Ko and colleagues analyzed titles of individual issues to determine such aspects as the degree to which issues refer to particular parts of a system and how much regularity there is between issue titles [12]. Bettenburg and colleagues consider what additional information should be included in an issue to assist a developer [8]. Banerjee and colleagues examined Mozilla and Eclipse repositories, finding that the maturity of a reporter reduces how often insignificant, poor quality, and duplicate bugs are detected [7]. Jankovic and colleagues find issues and commits can be used to reconstruct software processes; they define issues as parallel or sequential with the existence or nonexistence of a "block" relationship link between issues [11]. The work we present adds to these earlier efforts by identifying the prevalence and use of relationships between issues that are manually specified by developers.

3. QUALITATIVE STUDY

Our qualitative study involved sampling pairs of related issues from the Mylyn, Connect and HBase issue repositories and performing an open coding of the sampled pairs.

3.1 Coding Process

We began by selecting pairs of issues related in work breakdown relationships from the Mylyn and Connect repositories. We chose these two systems to start our open coding process [17] because we had knowledge that they each follow an agile development process and thus might share commonalities in how they use relationships in the issue repository.

Three coders (the first, third and fourth authors of the paper) read the titles of each issue in a selected pair and discussed the meaning of the relationship between the pair. If the meaning had not yet been seen, a code was developed to recognize and describe how the issues are related and was recorded in a codebook.[3] In the first iteration, 40 issue pairs from Mylyn and 60 issue pairs from Connect were randomly selected and coded.

After coding the first 100 issue pairs, we randomly selected a different set of 60 issue pairs from Mylyn and 90 issue pairs from Connect. Each of the three authors involved in the original iteration then coded 2 sets of 20 issue pairs (for a total of 40) from Mylyn and 2 sets of 30 issue pairs (for a total of 60) from Connect. In this way, each set of 20 or 30 issue pairs respectively was coded by two authors. The pairs of coders compared results and tried to reach a consensus on which code applies, updating the codes as necessary.

To ensure the updated codes were appropriately applied, all three coders then re-coded all previously coded pairs. At this point no new codes were found.

To determine if the codebook was sufficiently general to cover another system for which it had not been developed, two coders (the first and third authors of this paper) coded 50 issue pairs from the HBase repository, which was chosen as having different development characteristics from the other two projects. Based on this coding, some guideline refinements to clarify code selection were made to the codebook. The two coders then coded an additional 30 issue pairs from HBase to check if saturation had been reached. To assess the inter-coder reliability, we computed Cohen's kappa on the final 30 issue pairs coded; the coders achieved a 0.56 kappa value. As will be explained in the next section, some codes are related through a hierarchy; coders sometimes had differences in the level of code in the hierarchy assigned. If all sub-codes are collapsed to the super-code in the hierarchy, the kappa scores rises to 0.88.

[3]See www.cs.ubc.ca/labs/spl/projects/issueRelationships/

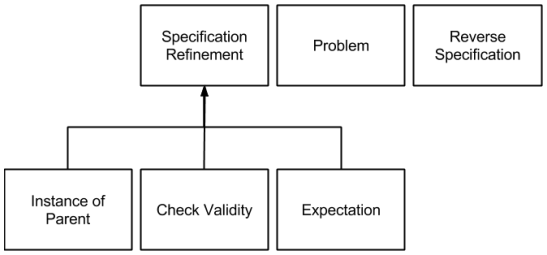

Figure 2: Codes developed through open coding

3.2 Codes

Six codes were identified from the coding process to more precisely describe the meaning of issues related through a work breakdown relationship: *specification refinement*, *instance of parent*, *expectation*, *problem*, *check validity*, and *reverse specification*. Pairs for which the meaning of the relationship could not be determined were coded as *unknown*. As Figure 2 shows, three of the codes are a specialization of the *specification refinement* code. Any pair of issues coded was assigned only one code from this set.

We describe the guidelines for each code in turn. For clarity in these guidelines, we refer to the issue that is the source of the relationship as the *parent* and the issue that is the target of the relationship as the *child*.

Specification refinement.

This code applies when a child issue describes one step of the work breakdown for the parent issue. The following example from `Mylyn` illustrates this code as the child issue specifies actions to take towards improving tooltip presentation.

Parent 205861: Improve tooltip presentation and content

Child 238292: Show reporter and beginning of description text on new issue tooltips.

Instance of parent.

This code applies when each child issue is a particular case in which the work described by the parent issue should be performed. The following example from `Connect` illustrates a child issue that specifies work from the parent is to occur on Windows machines, the same work can be done also on other types of machines.

Parent GATEWAY-1664: Create new VMs for final release installation testing. Needed by Friday 3/9

Child GATEWAY-1667: 4 *Windows* machines

Check validity.

This code applies when a child issue describes a verification activity for a parent issue. For instance, in `HBase`, the child issue describes adding tests to show the feature, described by the parent issue, works correctly.

Parent HBASE-10070: HBase read high-availability using timeline-consistent region replicas git

Child HBASE-10791: Add integration test to demonstrate performance improvement

Expectation.

This code applies when the child issue describes constraints or suggestions on how a parent issue can be fulfilled. The child issue in these cases often uses words like should, must, need, ensure, and improve. The following example from `Mylyn` shows how the child issue constraints the parent issue requirements.

Parent 158921: Improve the issue editor usability and information density

Child 212953: Depends on field in issue editor should fill available horizontal space

Problem.

This code applies when the child issue describes a problem that occurs in a parent issue. For instance, from `Connect`, the parent issue describes performing transaction logging and the child issue describes a particular part of the system requiring attention.

Parent GATEWAY-2151: Transaction Logging

Child GATEWAY-2782: non-unique messageid causes transaction not to be logged in transaction repo

Reverse specification.

This code applies when a parent issue describes one step of the work breakdown for the child issue. In other words, it is the reverse of *specification refinement*. For instance, from `Connect` the parent issue is a specific case of the child issue to investigate tests for concurrent messages.

Parent CONN-910: Execute concurrent tests from 3.3 gateway to 4.3 to ensure turning off of replay attacks fixes the issue

Child CONN-859: Investigate and research issue when concurrent messages are sent from connect 3.x gateway to connect 4.2 gateway

Unknown.

When none of the six codes just described apply to an issue pair, or when both reverse specification and specification refinement seem applicable, we consider the relationship meaning for an pair to be *unknown*. For instance, in the following `Connect` example, the parent issue describes an action to perform but the child issue is a noun phrase.

Parent CONN-1094: Create static screens for Direct configuration in Administrative GUI

Child CONN-1105: Trust Bundles

3.3 Summary

Table 2 shows the results of coding 330 issue pairs across the three repositories. The most prevalent code across all repositories was `specification refinement`. Interestingly, the more specialized version, `check validity`, occurs much more often in `Connect` than in the other repositories; perhaps the other repositories do not explicitly record their quality assurance related tasks. `Mylyn` contains more issue pairs describing `problem`s than the other repositories; this may be due to the high number of issue reporters who are not contributing developers. The `expectation` code occurs more frequently in `HBase`, perhaps because developers perform

Table 2: Codes occurring in each repository

	Mylyn		Connect		HBase	
Code	%	#	%	#	%	#
Spec. Refinement	48.0%	48	42.0%	66	47.5%	38
Check Validity	4.0%	4	14.0%	21	5.0%	4
Instance of Parent	5.0%	5	21.3%	32	7.5%	6
Expectation	14.0%	14	3.3%	5	21.3%	17
Problem	22.0%	22	6.7%	10	11.3%	9
Reverse Spec.	3.0%	3	2.7%	4	0.0%	0
Unknown	3.0%	3	7.3 %	11	6.3%	5
No consensus	1.0%	1	2.7%	4	1.3%	1
Total		100		150		80

more analysis of work breakdowns before specifying child tasks. The `instance of parent` occurs more frequently in `Connect`, suggesting that the developers more frequently refer to structural parts of the system when specifying work breakdown issues.

3.4 Threats to Validity

The codes may be biased by the knowledge and experience of the coders. The use of three coders helps minimize this bias. By separating into pairs to code after the development of the initial code book, we helped mitigate the persuasive affect of any one coder.

With each repository we coded, we clarified the code book. More refinements may be necessary if applied to other repositories, limiting the external validity of our results. Coding only the title of an issue also limits our results. More clarity on how a relationship is used might have been gained by using more information from the issue, such as comments about the work actually undertaken as part of the issue.

4. DISCUSSION

An understanding of how relationships are used may help information the development of new tools and may help improve the interlinking of commit and issue data.

4.1 Improving Software Development

The reliance on issue repositories for recording work on the system provides opportunities for tools to help ensure process is being followed. For example, tools may be created to analyze issues and learn the development process being used; this knowledge can then be used to automatically suggest tasks that may be missing. For example, `GATEWAY-2903`[4] in the `Connect` system, with six related work breakdown issues, was assumed complete, but developers had forgotten to specify and implement the work on a particular part of the system. Much later, 17 days after `GATEWAY-2903` was thought to be completed, a new issue `GATEWAY-3183`[5] was added to the system that explicitly referred to work as being missed in the issue `GATEWAY-2903`. This missing issue might have been predicted using process learned from issue relationships, allowing work to progress in a more timely and complete manner.

Analysis of the work described by issues and differences found between related issues might also help identify if there are any patterns that lead to certain kinds of bugs reported into the system. For example, do bugs correlate with particular semantic codes associated with the issues? Does this provide new opportunities to enhance bug prediction techniques (e.g., [14, 18, 10])?

4.2 Improving Software Engineering Data

Relationships between issues might be used to improve interlinking of commit and issue data to enable traceability between features and defects and the code that implements the feature or fixes the defect. Although tool support exists to help automate the collection of this data, the data is still often incomplete. Schermann and colleagues describe two scenarios derived in which missing links occur: 1) loners, which are single commits that lack a link to an issue and 2) phantoms, which are unlinked commits in a series of commits for which at least one commit in the series is linked [15]. The heuristics that Schermann and colleagues introduce to reduce the loners and phantoms in a system do not consider how issues in the issue repository are related. As a result, phantoms, in particular, may be incorrectly linked to an issue describing the overall work to be performed rather than an issue that describes the specific work undertaken by a phantom commit. Consider an example that Schermann and colleagues describe as a phantom: `CAMEL-7354` from the Apache Camel system[6] and the unlinked commit `14cd8d6fb`[7]. `CAMEL-7354` has seven sub-tasks; an inspection shows that commit `14cd8d6fb` should be linked to one of the sub-tasks, `CAMEL-7675`[8]. An ability to recognize and automatically tag the kind of relationship between `CAMEL-7354` and its sub-task `CAMEL-7675` could improve interlinking heuristics for associating commits with the appropriate issue.

5. SUMMARY

Developers often expend manual effort to specify how issues in an issue repository relate, especially to express how work is to be broken down and performed on the system. To investigate what kinds of work breakdowns are being expressed, we performed an open coding of a sample of 330 related issue pairs from the issue repositories of three open source systems: `Mylyn`, `Connect` and `HBase`. Our open coding progress resulted in six codes that describe a variety of kinds of work breakdowns, including cases where the work breakdowns express steps of verification and express constraints on work to be performed.

This study is the first to provide insight into the richness of information embedded in relationships in issue repositories. This information offers new opportunities to create new software engineering tools, such as to detect when necessary work may be missing and to improve interlinking of commit and issue data.

[4]https://connectopensource.atlassian.net/browse/GATEWAY-2903, verified 29/01/16

[5]https://connectopensource.atlassian.net/browse/GATEWAY-3183, verified 29/01/16

[6]https://issues.apache.org/jira/browse/CAMEL-7354, verified 29/01/16

[7]https://git1-us-west.apache.org/repos/asf?p=camel.git;a=commit;h=14cd8d6b

[8]https://issues.apache.org/jira/browse/CAMEL-7675, verified 29/01/16

6. REFERENCES

[1] Bugzilla. www.bugzilla.org. Accessed: 27/1/2016.

[2] Connect, project supporting health information exchange. www.connectopensource.org. Accessed: 27/1/2016.

[3] HBase. hbase.apache.org. Accessed: 27/1/2016.

[4] JIRA. www.atlassian.com/software/jira. Accessed: 27/1/2016.

[5] Mylyn. www.eclipse.org/mylyn. Accessed: 27/1/2016.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, 2005.

[7] S. Banerjee, J. Helmick, Z. Syed, and B. Cukic. Eclipse vs. mozilla: A comparison of two large-scale open source problem report repositories. In *International Symposium on High Assurance Systems Engineering*, pages 263–270, 2015.

[8] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *The Joint Meeting of the European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, pages 308–318, 2008.

[9] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *The Joint Meeting of the European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, 2009.

[10] A. E. Hassan. Predicting faults using the complexity of code changes. In *The International Conference on Software Engineering*, pages 78–88, 2009.

[11] M. Jankovic and M. Bajec. Comparison of software repositories for their usability in software process reconstruction. In *The International Conference on Research Challenges in Information Science*, pages 298–308, 2015.

[12] A. J. Ko, B. A. Myers, and D. H. Chau. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing*, pages 127–134, 2006.

[13] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[14] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *The International Conference on Software Engineering*, pages 181–190, 2008.

[15] G. Schermann, M. Brandtner, S. Panichella, P. Leitner, and H. Gall. Discovering loners and phantoms in commit and issue data. In *The International Conference on Program Comprehension*, 2015.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *The International Conference on Mining Software Repositories*, pages 1–5, 2005.

[17] A. Strauss and J. M. Corbin. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc, 1990.

[18] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *The Joint Meeting of the European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, pages 91–100, 2009.