PeerJ Computer Science

Are suggestions from coupled file changes useful for perfective maintenance tasks?

Jasmin Ramadani and Stefan Wagner

Institute of Software Technology, University of Stuttgart, Stuttgart, Germany

ABSTRACT

Background. Software maintenance is an important activity in the development process where maintenance team members leave and new members join over time. The identification of files which are changed together frequently has been proposed several times. Yet, existing studies about coupled file changes ignore the feedback from developers as well as the impact of these changes on the performance of maintenance and rather these studies rely on the analysis findings and expert evaluation.

Methods. We investigate the usefulness of coupled file changes during perfective maintenance tasks when developers are inexperienced in programming or when they were new on the project. Using data mining on software repositories we identify files that are changed most frequently together in the past. We extract coupled file changes from the Git repository of a Java software system and join them with corresponding attributes from the versioning and issue tracking system and the project documentation. We present a controlled experiment involving 36 student participants in which we investigate if coupled file change suggestions influence the correctness of the task solutions and the required time to complete them.

Results. The results show that the use of coupled file change suggestions significantly increases the correctness of the solutions. However, there is only a minor effect on the time required to complete the perfective maintenance tasks. We also derived a set of the most useful attributes based on the developers' feedback.

Discussion. Coupled file changes and a limited number of the proposed attributes are useful for inexperienced developers working on perfective maintenance tasks where although the developers using these suggestions solved more tasks, they still need time to understand and organize this information.

Submitted 16 January 2017
Accepted 18 September 2017
Published 16 October 2017

Corresponding author
Jasmin Ramadani,
jasmin.ramadani@informatik.uni-stuttgart.de

Academic editor
Ahmed Hassan

Additional Information and Declarations can be found on page 28

DOI **10.7717/peerj-cs.135**

Subjects Data Science, Software Engineering
Keywords Data mining, Software repositories, Coupled changes, Git

INTRODUCTION

Software maintenance represents a very important part in software product development (*Abran & Nguyenkim, 1991*). Maintenance is often performed by maintenance programmers. Over time teams change when members leave and others join (*Hutton & Welland, 2007*). New members cannot be productively included to solve maintenance tasks immediately, so they need some support to successfully perform their tasks.

Perfective maintenance tasks represent changes dealing with new or modified user requirements (*Stafford, 2003*). They are related to activities which increase performance of

the system or enhance its user interface (*Van Vliet, Van Vliet & Van Vliet, 1993*). *Lientz & Swanson (1980)* reported that more than 60% of the software maintenance efforts are of perfective nature.

Software development produces large amounts of data which is stored in software repositories. These repositories contain the artifacts developed during software evolution. After some time, this data becomes a valuable information source for solving maintenance tasks.

One of the most used techniques for analyzing software repositories is data mining. The term *mining software repositories (MSR)* describes investigations of software repositories using data mining (*Hassan, 2008*).

Couplings have been defined as "the measure of the strength of an association established by a connection from one module to another" (*Stevens, Myers & Constantine, 1974*). Change couplings are also described as files having the same commit time, author and modification description (*Gall, Jazayeri & Krajewski, 2003*). Knowing which files were frequently changed together can support developers in dealing with the large amount of information about the software product, especially if the developer is new on the project, the project started a long time ago or if the developer does not have significant experience in software development.

Problem statement

Several researchers have proposed approaches of identifying coupled file changes to give recommendations to developers (*Bavota et al., 2013*; *Kagdi, Yusuf & Maletic, 2006*; *Ying et al., 2004*; *Zimmermann et al., 2004*; *Hassan & Holt, 2004*). Existing studies, however, focus on the presentation of the mining results and expert investigations and they neglect the feedback of developers on the findings as well as the impact of coupled changes on maintenance tasks.

Research objectives

The overall aim of our research is to investigate the usefulness of coupled file change suggestions in supporting developers who are inexperienced, new on the projects or supposed to work on unfamiliar parts of the project. We provide suggestions for likely changes so that we can explore how useful the suggestions are for the developers.

We identify couplings of files that are changed frequently together based on the information gathered from the software project repository. We use the version control system, the issue tracking system and the project documentation archives as data sources for additional attributes. We join this additional information to the coupled changes that we discover to build the suggestions.

The usefulness of coupled file changes is determined by analyzing their influence on the correctness of the solutions and the time required for solving maintenance tasks.

Contribution

We present a controlled experiment on the usefulness of coupled change suggestions where each of the 36 participants try to solve four different perfective maintenance tasks and report their feedback on the usefulness of the repository attributes.

RELATED WORK

Many studies have been dedicated to investigating software repositories to find logically coupled changes, e.g., *Bieman, Andrews & Yang (2003)*; *Fluri, Gall & Pinzger (2005)*; *Gall, Hajek & Jazayeri (1998)*. We identify two granularity levels, the first one investigates the couplings based on a file level (*Kagdi, Yusuf & Maletic, 2006*; *Ying et al., 2004*) and the second scenario examines coupled changes identified between parts of files like classes, methods or modules (*Fluri, Gall & Pinzger, 2005*; *Kagdi, 2007*; *Zimmermann et al., 2004*; *Zimmermann et al., 2006*; *Hassan & Holt, 2004*). In our study, we use coupled file change on a file level.

The majority of the studies dealing with identifying coupled changes use some kind of data mining for this purpose (*German, 2004*; *Hattori et al., 2008*; *Kagdi, Yusuf & Maletic, 2006*; *Shirabad, Lethbridge & Matwin, 2003*; *Van Rysselberghe & Demeyer, 2004*; *Ying et al., 2004*; *Zimmermann et al., 2004*). Especially the association rules technique is often used to identify frequent changes (*Kagdi, Yusuf & Maletic, 2006*; *Ying et al., 2004*; *Zimmermann et al., 2004*). This data mining technique uses various algorithms to determine the frequency of these changes. Most of the studies employ the Apriori algorithm (*Kagdi, Yusuf & Maletic, 2006*; *Zimmermann et al., 2004*). However, other faster algorithms like the FP-growth algorithm are also in use (*Ying et al., 2004*). We generate the coupled file changes using the frequent item sets analysis and the FP-growth algorithm.

Most of the studies use a single data source where some kind of version control system is investigated, typically CVS or Subversion. There are few studies which investigate a Git version control system (*Bird et al., 2009*; *Carlsson, 2013*). Other studies combine more than one data source to be investigated, like the version control system and an issue tracking system (*Canfora & Cerulo, 2005*; *D'Ambros, Lanza & Robbes, 2009*; *Fischer, Pinzger & Gall, 2003*; *Wu et al., 2011*) where the data extracted from these two sources is analyzed and the link between the changed files and issues is determined. We use three different sources for the additional attributes: the Git versioning history, the JIRA issue tracking system and the software documentation.

To the best of our knowledge, there are few studies investigating how couplings align with developers' opinion or feedback. Coupling metrics on structural and semantic levels are investigated in *Revelle, Gethers & Poshyvanyk (2011)*. The developers were asked if they find these metrics to be useful. They show that feature couplings on a higher level of abstraction than classes are useful. The developers' perceptions of software couplings are investigated in *Bavota et al. (2013)*. Here the authors examine how class couplings captured by different coupling measures like semantic, logical and others align with the developers' perception of couplings.

The interestingness of coupled changes is also studied in *Ying et al. (2004)*. This study defines a categorization of coupled changes interestingness according to the source code changes. In *Ramadani & Wagner (2016)*, the feedback on the interestingness of coupled file changes and attributes from the software repository was investigated. In our experiment we extend the findings of this case study and investigate the usefulness of coupled file changes and the corresponding attributes.

The categorization of changes of a software product related to the maintenance task categories defined in *Swanson (1976)* has been previously investigated (*Hindle, German & Holt, 2008*; *Purushothaman & Perry, 2005*). The authors in *Purushothaman & Perry (2005)* classified small changes based on their purpose and implementation. The changes on the software in large commits have been categorized in *Hindle, German & Holt (2008)*. They define categories to identify the types of the addressed issues and the types of the changes in large commits. Similarly, we classify the issues related to the changes on the system based on the defined maintenance categories. Furthermore, we classify the changes on the system based on the most involved functionalities.

Various experiments involving maintenance tasks have been described in the literature. *Nguyen, Boehm & Danphitsanuphan (2011)* deal with assessing and estimating software maintenance tasks. *De Lucia, Pompella & Stefanucci (2002)* investigate the effort estimation for corrective software maintenance. *Ricca et al. (2012)* perform an experiment on maintenance in the context of model-driven development. *Chan (2008)* investigates the impact of programming and application-specific knowledge on maintenance effort. In our experiment, we explore how the coupled file change suggestions influence the correctness of performing maintenance tasks and the time required to solve these tasks.

BACKGROUND

Software maintenance

Software maintenance includes program or documentation changes to make the software system perform correctly or more efficiently (*Shelly, Cashman & Rosenblatt, 1998*). Software maintenance has been defined in the IEEE 1219 Standard for Software Maintenance (*IEEE, 1998*) to be a software product modification after delivery to remove faults, improve performance or adapt the environment. In the ISO/IEC 12207 Life Cycle Processes Standard (*ISO/IEC, 2008*), maintenance is described as the process where code or documentation modifications are performed due to some problem or improvement.

Maintenance categories

Swanson (1976) defined three different categories of maintenance: corrective, adaptive and perfective maintenance. The ISO/IEC 14764 International Standard for Software Maintenance (*ISO/IEC, 2006*) updates this list with a fourth category, preventive maintenance, so we have the following maintenance categories (*Pigoski, 1996*):

- Corrective Maintenance: This type of maintenance tasks includes corrections of errors in systems. Software product modifications are performed to correct the discovered problems. It corrects design, source code and implementation errors.
- Adaptive Maintenance: This involves changes in the environment and includes adding new features or functions to the system. Software product modifications are performed to ensure software product usability in a changed environment.
- Perfective Maintenance: This involves changes in the system which influence its efficiency. It also includes software product modifications to improve maintainability or performance.

- Preventive Maintenance: Here, the changes to the system are performed to reduce the possibility of system failures in the future. It includes software product modification to detect and remove failures before they materialize.

Data mining

Coupled file changes

To discover coupled file changes using data mining, we introduce the data technique that we employ in our study. One of the most popular data mining techniques is the discovery of frequent item sets. To identify sets of items which occur together frequently in a given database is one of the most basic tasks in data mining (*Han, 2005*).

Coupled changes describe a situation where someone changes a particular file and also changes another file afterwards. Let us say that the developer changes file f_1 and then frequently changes file f_3. By investigating the transactions of changed files in the version control system commits we identify a set of files that are changed together. Let us have the following three transactions: $T_1 = \{f_1, f_2, f_3, f_7\}$, $T_2 = \{f_1, f_3, f_5, f_6\}$, $T_3 = \{f_1, f_2, f_3, f_8\}$. From these three transactions, we isolate the rule that files f_1 and f_3 are found together: f_1 and f_3 are coupled. This means that when the developers changed file f_1, they also changed file f_3. If these files are found together frequently, this can help other persons by suggesting that if they change f_1, they should also change f_3. Let $F = \{f_1, f_2, \ldots, f_d\}$ be the set of all items (files) f in a transaction and $T = \{t_1, t_2, \ldots, t_n\}$ be the set of all transactions t. As transactions, we define the commits consisting of different files. Each transaction contains a subset of chosen items from F called item set. An important property of an item set is the support count δ which is the number of transactions containing an item. We call the item sets frequent if they have a support threshold min_{sup} greater than a minimum specified by the user with

$$0 \leq \min_{\text{sup}} \leq |F|. \tag{1}$$

Data mining algorithm

Various algorithms for mining frequent item sets and association rules have been proposed in literature (*Agrawal & Srikant, 1994*; *Györödi & Györödi, 2004*; *Han, Pei & Yin, 2000*). We use the FP-Tree-Growth algorithm to find the frequent change patterns. As opposed to the Apriori algorithm (*Agrawal & Srikant, 1994*) which uses a bottom-up generation of frequent item set combinations, the FP-Tree-Growth algorithm uses partition and divide-and-conquer methods (*Györödi & Györödi, 2004*). This algorithm is faster and more memory-efficient than the Apriori algorithm used in other studies. This algorithm allows frequent item set discovery without candidate item set generation.

Change grouping heuristic

There are different heuristics proposed for grouping file changes (*Kagdi, Yusuf & Maletic, 2006*). We apply a heuristic considering the file changes done by a single committer to be related and do not include the changes done by other committers in the same group. We use the complete version history of the project, however based on the "developer heuristic" we group the commits performed by a single developer. From each group of these commits we extract files frequently changed together.

EXPERIMENTAL DESIGN

In this section we define the research questions, hypotheses and metrics used in our analysis.

Study goal

We use the GQM approach (*Basili, Caldiera & Rombach, 1994*) and its MEDEA extension (*Briand, Morasca & Basili, 2002*) to define the study goal. The *goal* of our study is to analyze the usefulness of coupled file change suggestions. The *objective* is to compare the correctness of the solution and the time needed for a set of maintenance tasks between the group using coupled change suggestions and the group that does not use this kind of help. The *purpose* is to evaluate how effective coupled file change suggestions are regarding the correctness of the modified source code and the time required to perform the maintenance tasks. The *viewpoint* is that of software developers and the targeted environment is open source systems.

Research questions

We investigate the usefulness of coupled file change suggestions and the corresponding repository attributes. In this study, we concentrate on perfective maintenance to have a similar set of tasks. For that purpose we define the following research questions:

RQ1: How useful are coupled file change suggestions in solving perfective maintenance tasks?

To determine the usefulness of the coupled file changes concept, we define the following sub-questions:

RQ1.1: Do coupled file change suggestions influence the correctness of perfective maintenance tasks?

We investigate if there is any difference in the correctness of the maintenance task solutions between the group of developers who used coupled file change suggestions and the group not using them.

RQ1.2: Do coupled file change suggestions influence the time needed to solve perfective maintenance tasks?

We explore if the time that the developers need to complete the maintenance tasks differs between the group using coupled change suggestions and the group not using these suggestions. We consider two scenarios: The first one includes only the time needed to solve the tasks, the second one also includes the time needed to select relevant coupled file changes.

RQ2: How useful are the attributes from the software repository in solving perfective maintenance tasks?

The second research question deals with the attributes from the versioning system, the issue tracking system and the documentation. We investigate the perceived usefulness of each attribute in the proposed set to understand which attributes are good candidates to be provided to the developers.

Hypotheses

We formulate the following hypotheses to answer the research questions in our study.

For **RQ1.1** we define the following hypotheses: $\mathbf{H_{0.1.1}}$: There is no significant difference in the correctness of perfective maintenance task solutions between the developers using coupled file change suggestions and those not using these suggestions.
$\mathbf{H_{A.1.1}}$: There is a significant difference in the correctness of perfective maintenance task between the developers who used coupled file change suggestions and those not using these suggestions.
For **RQ1.2** we address the following hypotheses:
$\mathbf{H_{0.1.2}}$: There is no significant difference in the time required to solve perfective maintenance tasks between the developers who used coupled file change suggestions and the developers not using these suggestions.
$\mathbf{H_{A.1.2}}$: There is a significant difference in the time required to solve perfective maintenance tasks between the developers who used coupled file change suggestions and those not using these suggestions.
To answer **RQ2** we formulate the following hypotheses:
$\mathbf{H_{0.2}}$: There is no significant difference in the perceived usefulness among the attributes from the software repository in the current set.
$\mathbf{H_{A.2}}$: There is a significant difference in the perceived usefulness among the attributes from the software repository in the current set.

Experiment variables

We define the following dependent variables: the correctness of the solution after the execution of the maintenance task, the time spent to perform the maintenance task and the usefulness of the repository attributes. For the first variable, the correctness of the task solution, we assign scores to each developer's solution of the maintenance tasks.

Our approach is similar to the one presented by *Ricca et al. (2012)* where the correctness of the solution of the maintenance task is manually assessed by defining scores from totally incorrect to completely correct task solution. We define three scores: 0 if the developers did not execute or did not solve the task at all, 1 if the task was partially solved and 2 if the developer performed a complete solution of the maintenance task. The solutions are tested using unit tests to ensure the correctness of the edited source code.

The second variable, the time required for executing the maintenance tasks is measured by examining the screen recordings. We mark the start time and the end time for every task. We calculate the difference to compute the total time needed to solve each task. We differentiate the time needed only to solve the tasks t_s and the time needed to determine the relatedness of the coupled files t_r. For the third variable, the usefulness of the repository attributes, we use an ordinal scale to identify the feedback of the developers. The participants can choose between the following options for each attribute: very useful, somewhat useful, neutral, not particularly useful and not useful. We code the usefulness feedback using the scoring presented in Table 1.

Experiment design

We distinguish two cases for the maintenance tasks: the first one includes tasks executed on Java Code in the Eclipse IDE without any suggestions and the second one includes tasks

Table 1 Usefulness score.

Very useful	Somewhat useful	Neutral	Not particularly useful	Not useful
5	4	3	2	1

Table 2 Experiment design.

Lab	Tasks	
Lab 1	Tasks 1–2 (−)	Tasks 3–4 (+)
Lab 2	Tasks 1–2 (+)	Tasks 3–4 (−)

executed with additional coupled files suggestions and corresponding attributes from the repositories. We use a similar approach to the one presented by *Ricca et al. (2012)* and define two values: − for Eclipse only and + for the coupled file suggestions.

We use a counterbalanced experiment design as described in Table 2. This ensures that all subjects work with both treatments: without and with coupled change suggestions. We split the subjects randomly into two groups working in two lab sessions of two hours each. In each session, the participants work on two tasks using only the task description and on two tasks using coupled file change suggestions and their related attributes. The participants in the second lab swapped the order of the tasks in the first lab.

Objects

The object of the study is an open source Java software called A-STPA. The source code and the repository were downloaded from SourceForge (https://sourceforge.net/projects/astpa/). The system was built mainly in Java by 12 developers at the University of Stuttgart during a software project between 2013 and 2014. It represents an Eclipse-based tool for hazard analysis. The source code contains 16,012 lines of code and 178 classes organized in 37 packages. The Git repository of the project contains 1,106 commits from which we extracted 205 coupled file changes.

Subjects

The experiment participants are 36 students from the Software Engineering course in their second semester at the University of Stuttgart (Germany). The students have basic Java programming and Eclipse knowledge and have not been related in any way with the software system investigated in the experiment.

Materials, procedure and environment

All subjects received the following materials which can be found in the supplemental material of this paper.

- Tools and code: The participants received the Eclipse IDE to work with, the screen capturing tool and the source code they need to edit.
- Questionnaires: The first questionnaire is filled in at the start of the experiment and it is related to their programming background. The second questionnaire performed at the end of the experiment is about their feedback on the usefulness of coupled changes and the additional set of repository attributes.

- Software documentation: We provided the technical documentation for the software system including the architecture description covering the sub-projects, the overview of the classes in the data model, the application controllers, the graphic editor and the package descriptions.
- Setup instructions: The participants received the instruction steps how to prepare the environment, where to find the IDE, the source code and how to perform the experiment.
- Maintenance tasks and description: Every participant received spreadsheets with four maintenance tasks and their free-text description. The maintenance tasks represent quick program fixes that should be performed by the participants according to the maintenance requests (*Basili, 1990*). The maintenance tasks used in the experiment require the participants to add various enhancements to the system. The changes do not influence the structure or the functionalities of the application. The tasks are related to simple changes of the user interface of the system. All four maintenance tasks are perfective and have been assigned to the participants in both groups.
- Set of coupled files: The files changed together frequently used to solve a similar tasks have been provided to the group that uses coupled file changes.
- Repository Attributes: The attribute set from the versioning system, the issue tracking system and the documentation about similar tasks performed in the system. They have been joined to the coupled files using a mapping between the commits containing the coupled files and the issues using their issue IDs.

The environment for the experiment tasks was Eclipse IDE on a Windows PC in both treatments. For each lab, we prepared an Eclipse project containing the Java source code of the A-STPA system. The project materials were made available to the subjects on a flash drive. The participants had a maximum of two hours to fill the questionnaires and perform the maintenance tasks.

Selection of change author

According to the used heuristic for grouping the change sets in the versioning history, we need to select the authors of the changes whose data will be included in the analysis.

The selection process of the developers as authors of the source code changes is presented in Fig. 1. Out of 12 developers who worked on the A-STPA software, after performing the frequent itemset analysis, we have eight developers left whose entries in the repository delivered coupled files.

We have four maintenance tasks to be solved in the experiment. For each of the tasks we use commits from a different developer to avoid the influence of the authorship of the commits on the tasks. Out of the eight developers we need to select four, one for each maintenance task.

Selection of coupled files

After selecting the developers, we continue with the selection of the coupled files. The process includes the selection of the most frequent coupled files followed by the selection of relevant coupled files as presented in Fig. 1.

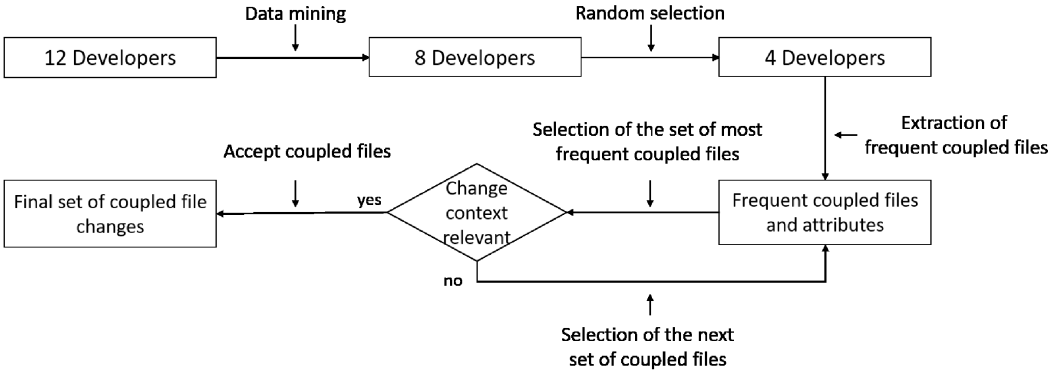

Figure 1 **Changes selection.**

Full-size DOI: 10.7717/peerjcs.135/fig-1

Selection of the most frequent coupled files

We need to select the coupled files which we will include in the suggestions for the developers in the experiment. For each of the four developers we list the most frequent coupled files we have extracted. We sort the sets of coupled file changes by their frequency in descending order, so on top of the list we have the most frequent set of coupled files. We start selecting the sets of coupled files from the top of the list.

We do this for two main reasons: (1) To avoid a potential subjectivity in the selection of the coupled files. (2) We want to use the strongest couplings, meaning the coupled files which are frequent and did not happen by chance.

Selection of relevant coupled files

After identifying the most frequent coupled files, we examine their broader change context. This means that we need to determine if they fulfill the requirements to be: (1) of perfective nature and (2) related to modifications in the user interface of the application.

We determine this change context using a manual analysis of the content of the commit messages where the coupled file changes were included as well as the description of the related issues. To perform this, we use the mappings between the commit messages and the issue IDs provided as part of the corresponding repository attributes we added to the coupled file changes.

Classification of issues

We classified the issues for the examined software systems using the approach proposed in *Hindle, German & Holt (2008)*. We determine the following classes of issues:

- Corrective: These issues cover failures related to the processing or performance of the software.
- Adaptive: These changes include modifications related to the data and the processing environment.
- Perfective: The changes include modifications related to performance, maintenance or usability improvements.
- Implementation: These tasks include new requirements for the software system.

- Other: These include changes that are not functionally related to the system like copyright or control version system related issues.

We go further and classify the perfective changes based on the most frequently involved system functionalities. For example, we want to know how many perfective issues have been defined for the user interface of the application and what are the main parts of this interface addressed in these issues. This way we expose the representativeness of the selected coupled file changes and the defined tasks for the software system we examine.

Definition of tasks

After we determined the sets of coupled file changes which fulfill the requirements of the experiment, we continue with the definition of the tasks the participants need to solve.

Firstly, we determine the change context of the selected coupled file sets more precisely by looking up repeatedly in the related commit messages and the issue description. This identifies the functionality the file changes are related to. We use the mapping of the issue IDs and the commit messages to follow up this information. After we identified the issues related to sets of relevant coupled file changes, we define perfective maintenance tasks related to similar functionalities covered in these issues. For example, in Table 3 we have an issue extracted from the issue tracking system of the A-STPA product which defines that a new item in the application view should be created using a keyboard shortcut. The commit message for the changes solving this task represents the comment of the developer who placed the shortcut. Considering the described functionality, we create a task where the developer needs to create a new shortcut combination for that purpose. In the same manner, we repeat the procedure for each of the relevant coupled files that we have selected and define four tasks.

The content of the text description of the tasks is related to the content of the issues we extracted from the issue tracking system. We keep the content of the task definitions very simple. They contain the functionality or the part of the system which has to be changed and the action to be performed. This makes it easier to replicate the process using other software products and their repositories.

Tasks and coupled file changes

Our goal is for each of the tasks to provide coupled file changes related to their context. This feature is of great importance for the study. Offering unrelated coupled file can be misleading and confusing for the developers.

We can extend the examination of the commit messages content and the issue descriptions to determine the change context as a part of a tool using natural-language processing techniques. We can compare the content of the user input or the issue content with the comments in the commit messages or issue description we mapped to the coupled file change sets. However, this exceeds the scope of this study and can be considered as future work.

Table 3 Task information and coupled file changes. .

User task	Task solution file set
Change the shortcut for adding new items in all the user interface views from "SWT.KeyDown and 'n'" into "SWT.KeyUp and 'y'"	ControlActionView.java SystemGoalView.java DesignRequirementView.java SafetyConstraintView.java HazardsView.java AccidentsView.java
Related commit	**Suggested coupled changes**
I have set a simple shortcut for new items to be "n", which can be quickly changed if needed.	ControlActionView.java DesignRequirementView.java SafetyConstraintView.java AccidentsView.java
Related issue	
Using a keyboard shortcut, a new item should be created in the application views.	

Solution of tasks

The complete list of files included in the task solutions are defined manually by analyzing the solutions of the related issues and evaluated by an independent party.

An example of the relation between the files included in the solution for a particular maintenance task and the set of coupled file changes is presented in Table 3. Here, we can see that to be able to solve the mentioned task, the developer needs to change six files which are related to the views of the application.

The coupled change suggestion based on an issue related to the defined task recommends four files to be changed. These files were extracted from the version history have been changed frequently together in the past.

We would like to point out that the file change suggestions do not represent the solutions for a particular task in the experiment. The solution usually contains more files than the provided suggestions. Although the provided suggestions contain a subset of the solution set, the developers still need to find the location in the source code meaning the method or the class that they need to modify in order to solve the tasks. This is finer grain information that we do not provide in our coupled files. The developers still have to read the repository attributes and decide if they want to follow the coupled file change suggestions.

Maintenance activities

After receiving the task description, the participants investigate the source code of the application, identify the files where the changes are needed and perform the changes according to the requirement. The scenario for solving the provided maintenance tasks includes the following activities (*Nguyen, Boehm & Danphitsanuphan, 2011*):

- Task understanding: First of all, the participants need to read the task description and the instructions and prepare for the changes. They can ask if they need some clarification about the settings and the instructions.

- Change specification: During this step, the participants locate the source code they need to change, try to understand and specify the code change.
- Change design: This step includes the execution of the already specified source code changes and debugging the affected source code.
- Change test: To specify the successfulness of the performed code changes, a unit test needs to be performed. This step is performed by the experiment organizers after the lab sessions.

Data collection procedure

We collect data from several sources: the software repository of the system, the questionnaires, the provided task solutions and the screen capture recordings.

Software repositories

- Version Control System: The first data source that we use is the log data from the version control system. The investigated project uses Git as a control management tool. It is a distributed versioning system allowing the developers to maintain their local versions of source code. This version control system preserves the possibility to group changes into a single change set or a so-called *atomic commit* regardless of the number of directories, files or lines of code that change. A commit snapshot represents the total set of modified files and directories (*Zimmermann et al., 2004*). We organize the data in a transaction form where every transaction represents the files which changed together in a single commit. From this data source we extract the coupled file changes and the commit related attributes.
- Issue Tracking System: It stores important information about the software changes or problems. In our case, the developers used JIRA as their issue tracking system. This data source is used to extract the issue-related attributes.
- Project Documentation: The software documentation gathered during the development process represents a rich source of data. The documentation contains the data model and code descriptions. From these documents, we discover the project structure. For example, in the investigated project, the package containing the files described by the following path: *astpa/controlstructure/figure/*, contains the Java classes responsible for the control diagram figures of this software. We use the documentation to identify the package description.

The complete set of attributes we extract from the software repository is presented in Table 4.

Questionnaire

The developers answer a number of multiple-choice questions. Using the first questionnaire, we investigate the developers' programming background. We use a second questionnaire after the tasks are solved in order to gather the feedback on the usefulness of coupled changes and the additional attributes.[1]

[1] The questionnaires are available in the supplemental material of this paper.

Table 4 Repository attributes description.

Attribute name	Attribute description
Commit ID	Unique ID of Git commit
Commit message	Free-text comment of the commit in Git
Commit time	Time-stamp of committed change in Git
Commit author	Person who executed the commit
Issue description	Free-text comment on issue to be solved
Issue type	Type of the issue: bug, feature
Issue author	Person who created the issue to be solved
Package description	Free-text description of the package: layer, feature

Tasks completion

Similar to other studies (*Chan, 2008*; *Nguyen, Boehm & Danphitsanuphan, 2011*; *Ricca et al., 2012*), we define two factors which represent the completion of the maintenance tasks:

- Correctness of solution: We determine the correctness of the solution by examining the changed source code if the solution satisfies the change requirements. We use the scoring presented previously where we summarize the points each developer gathers for each of the four tasks. The score is added next to each of the participants for both treatments, with and without using coupled file changes.
- Time of task completion (t_s): This represents the time measured in minutes the developers spent to solve the maintenance tasks. Having a scenario where the developers only need to solve the tasks, the selection of the coupled files is not included in the total time for the tasks. It does not include the time needed to determine the relatedness of the coupled files for a specific task. The completion time could be automatically determined using a tool implementation or as part of an analysis procedure and does not represent part of the developer task solution. We use a screen capturing device to record the time that each participant spends solving each of the four tasks. We record the time needed for each task in both treatments.
- Time required to determine the relevance of the coupled files (t_r): This represents the time needed to determine the change context of each of the coupled files related to the tasks. Considering a worst-case scenario, the selection of the coupled files has to be performed by the developers and the time needs to be calculated for the group using coupled file change suggestions. In this case, the total time needed for each of the tasks is the sum $(t_r + t_s)$ of the time needed to select the coupled files and the time to solve the task. Given the task list, the coupled files list and the issue list, we record the time the developers need to go through the process of determining the change context of the coupled files we examine for a given task. We use three additional developers to measure the time required to determine the context of each of the coupled file changes related to the tasks.

Data analysis procedure

To be able to test our hypotheses, we need to analyze the usefulness of the coupled file changes and the usefulness of the attributes from the software repository. We perform the analysis using the SPSS statistical software.

Usefulness of coupled file changes

The main part of the analysis is the investigation of the usefulness of the coupled changes. For this purpose we compare the scores of each task solution and the amount of time needed for solving the tasks in both groups: without using coupled file suggestions and with using of coupled file suggestions.

For the time needed for the solution, we only use the values for the accomplished tasks. This way we assure that the values for the unsolved tasks do not corrupt the overall values for the time needed to successfully solve the tasks.

Here we have two main scenarios. The first one includes only the time the developers need to solve the tasks. The second scenario also includes the time needed to select the coupled files set related to a specific maintenance task. We calculate the mean time for a particular task. Furthermore, we repeat the calculation for each participant on the task. At last, we determine the grand mean as the average of all the means of the time values for each of the tasks determined by the participants, weighted by the sample size. In our case this is the number of coupled files.

Having k populations or tasks, the ith observation is t_{ri} which is the $j(i)$th coupled files set. We write $j(i)$ to indicate the group associated with the observation i. Let i vary from 1 to n, which is the total number of samples, in our case, these are the coupled files, j varies from 1 to k, the total number of tasks. There are a total of n observations with n_j observations in sample j: $n = n_1 + \cdots + n_k$. The grand mean of all observations is calculated using the formula:

$$\overline{t_r} = \sum_{j=1}^{k} \left(\frac{n_j}{n}\right) \overline{t_{rj}} \tag{2}$$

here, $\overline{t_r}$ is the average of the sample means, weighted by the sample size (*Hanlon & Larget, 2011*).

To determine the usefulness of coupled file changes, we test the overall difference in the correctness of solving the tasks using the two-tailed Mann–Whitney U test. It is used to test hypotheses where two samples from the same population have the same medians or that one of them has larger values, so we test the statistical significance of difference between two value sets.

Determining an appropriate significance threshold defines whether the null hypothesis must be rejected (*Nachar, 2008*). If the p-value is small, the null hypothesis can be rejected meaning that the value sets are different. If the p-value is large, the values do not differ. Usually a 0.05-level of significance is used as threshold. The p-value is not enough to determine the strength of the relationship between variables. For that purpose we report the effect size estimate (*Tomczak & Tomczak, 2014*).

We use a conservative approach where we test the difference in the correctness of our tasks. Without differentiating the tasks, we compare all the solutions of the tasks using coupled file changes and the tasks performed without any suggestion. We repeat the same approach to test the overall difference between the time needed to solve the tasks using coupled change suggestions against the tasks solved without the help of coupled file changes.

We use the SPSS statistical software and its typical output for the Mann–Whitney U Test whereby the p-value of the statistical significance in the difference between the two groups is reported. The mean ranking determines how each group scored in the test. To support statistical difference between the samples, we calculate the r-value of the effect size proposed in (*Cohen, 1988*) using the z value from the SPSS output (*Fritz, Morris & Richler, 2012*). A value of 0.5 determines a large effect, 0.3 means medium effect and 0.1 identifies a small effect (*Coolican & Taylor, 2009*). Given that we have a study which is restricted to a small number of comparisons, we do not adjust the p-value using a procedure like the Bonferroni correction (*Armstrong, 2014*).

Usefulness of attributes

We analyze the feedback from the questionnaire investigating which attributes are useful. We investigate every attribute in the set extracted from the versioning system, the issue tracking system and the documentation as previously presented. For that purpose we use the Kruskal–Wallis H test, an extension of the Mann–Whitney U test. Using this test, we determine if there are statistically significant differences between the medians of more than two independent groups. We test the statistical significance between more than two value sets. The significance level determines if we can reject the null hypothesis. p-values bellow 0.05 mean that there is a significant difference between the groups (*Pohlert, 2014*). To determine the effect size for the Kruskal-Wallis H test, we calculate the effect sizes for the pairwise Mann–Whitney U tests for each of the attributes using the z statistic. We individually calculate the r-value for the effect size for each pair comparison. The r-value is calculated using the following formula:

$$r = \frac{z}{\sqrt{N}}. \tag{3}$$

Our approach tests the differences in the feedback about the usefulness between all the attributes for all 36 participants. This way we identify which attributes we should offer to the participants when solving their tasks together with the coupled file change suggestions. Using SPSS, we provide the statistical significance values of the difference between all eight attributes. The ranking of the means for the feedback on the usefulness values determine the most useful attributes.

Execution procedure

- Log Extraction: We extract the information from the Git log containing the committed file changes and the attributes. The log data is exported as a text file and the output is managed using proper log commands.
- Data preprocessing: After the text files with the log data have been generated, we continue with the preparation of the data for mining. Various data mining frameworks

use their own format, so the input for the data mining algorithm and framework needs to be adjusted.

- Support threshold: To begin the investigation, we need to extract coupled file changes from the software repository. We extract the coupled changes by defining the threshold value of the support for the frequent item set algorithm. We use the thresholds that give us a frequent yet still manageable number of couplings. This threshold is normally defined by the user. We use the technique presented in (*Fournier-Viger, 2013*) to identify the support level. These values vary from developer to developer, so we test the highest possible value that delivers frequent item sets. If the support value does not yield any useful results for a particular developer, we drop the value of the threshold. We did not consider item sets with a support rate below 0.2, meaning the coupled changes should have been found in 20 percent of the commits.
- Mining Framework: There is a variety of commercial and open-source products offering data mining techniques and algorithms. For the analysis, we use an open-source framework specializing in mining frequent item sets and association rules called the *SPMF-Framework* (http://www.philippe-fournier-viger.com/spmf). It consists of a large collection of algorithms supported by appropriate documentation.
- Experiment preparation: We prepare the environment by setting up the source code and the Eclipse where the participants will work on the tasks. We define the maintenance tasks and provide the free text description. We prepare the coupled file changes and the attributes from the software repository to be presented to the participants in the experiment.
- Solving tasks: The participants in both groups work for two hours in two labs and provide solutions for the maintenance tasks. The solution and the screen recording are saved for further analysis.
- Material gathering: We gather the questionnaires, the edited source codes and the video files of the participants screens for further analysis.
- Solution analysis: We analyze the scores for the correctness of the maintenance tasks, calculate the time needed for solving the tasks and determine the influence of the coupled file changes on the task solution.

RESULTS AND DISCUSSION

The complete list of the maintenance tasks, the coupled file changes, the software repository attributes, the questionnaires and the analysis results can be found in the supplemental material of this paper.

Participants

The participants' feedback about their background reports that most of them have around one year of programming experience and less than one year of experience with versioning and issue tracking systems. None of them answered to be in any way involved on the A-STPA project.

Table 5 Issue classification.

Issue category	Frequency	%
Corrective	217	31.77
Implementation	169	24.74
Perfective	146	21.38
Adaptive	85	12.45
Other	66	9.66

Table 6 Perfective issues.

Change category	Frequency	%
Views	74	49,01
Control structure	34	22,52
Menus	22	14,57
Non functional source changes	13	8,61

Issues classification

Based on the proposed classification from *Hindle, German & Holt (2008)*, we classified the issues from the issue tracking system related to commits in the Git version history as presented in Table 5. Here we can see that most changes of the system are corrective, implementation and perfective issues.

Further, we examined the perfective issues in more detail to determine to which parts of the system they are related. We have identified several classes of perfective issues related to the main functionalities of the system that we investigated in the experiment as presented in Table 6.

The most frequent perfective issues are related to changes to the view elements of the system user interface responsible for the visualization of the hazard analysis steps including their layout, tables, grids, text fields, buttons, icons and labels. These changes have been organized in the class called *Views*.

The next class called *Control Structure* is composed by the issues which handle the changes related to the control structure functionality of the user interface. It is responsible for drawing the diagrams, connects the layout components and includes changes on the diagram elements like objects, labels and connections.

The following class we call *Menus* is related to the issues associated to the user interface menus which are used to manipulate the creating and editing of project elements including changes in the wizards, the actions, labels and icons.

The last class includes the issues covering non-functional changes in the source code like cleanups, refactoring or formatting.

The task distribution in the experiment corresponds to this classification. We have defined two tasks for the application views, one task for the menus and one task for the control structure of the user interface.

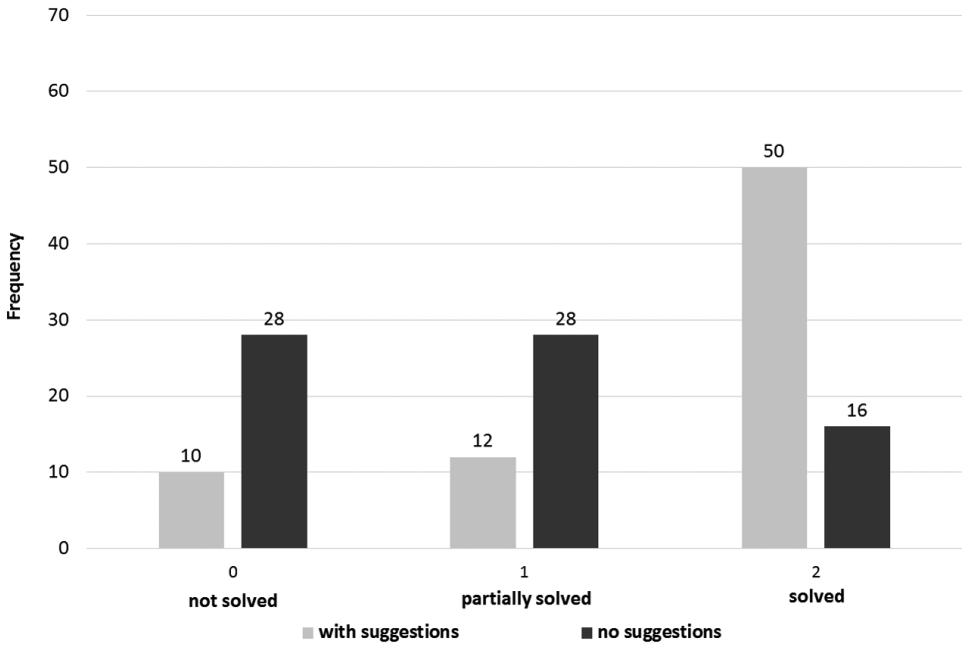

Figure 2 **Task correctness distribution.**
Full-size DOI: 10.7717/peerjcs.135/fig-2

Usefulness of coupled file changes

As we already explained, we operationalize the usefulness of coupled file changes by their influence on the correctness of the solutions and the time needed to solve the tasks.

Correctness

We summarize the correctness distribution as presented in Fig. 2. On the y-axis we have the frequency of occurrence and on the x-axis the score of solving of the tasks. Here, the observations are grouped based on the presence of coupled change suggestions during the maintenance task solution. From this figure we see that the participants achieved better scores using the coupled file change suggestions we provided.

We investigate the correctness difference of both groups by testing the first null hypothesis of the first research question claiming that there is no significant difference in the correctness of the task solutions.

Applying the Mann–Whitney U Test results in a p-value of 0.000 as presented in Table 7. This result has to be lower than the threshold of 0.05, so this null hypothesis can be rejected. This means that there is a statistically significant difference in the correctness of the solution for the provided tasks when using coupled file change suggestions against the correctness of the solutions only using the provided task description. The r-value of the effect size for the correctness is 0.448 which describes a strong statistical difference in the correctness of the maintenance task solutions between the groups that did or did not coupled change suggestions.

In Table 8, we represent the descriptive statistics for the correctness of the task solutions. The participants which used the suggestions solved 63.8% of the tasks completely, whereby

Table 7 Statistical significance (Coupled changes).

Depend. Variable	*p*-value	*r*-value
Correctness	0.000	0.448
Time effort	0.041	0.259

Table 8 Descriptive statistics for the correctness of the tasks.

Without suggestions (−)			With suggestions (+)		
Completely solved tasks	Median	MAD	Completely solved tasks	Median	MAD
22%	1	1	63.8%	2	0

the participants not using suggestions solved only 22% of the tasks. This shows a significantly higher score for the group using coupled change suggestions.

The median absolute deviation (MAD) value for the group using coupled changes is 0, whereby the value for the group not using coupled changes is 1. These values show that the correctness score is spread very close to the median for the score of the first group. The statistical results provide evidence that the coupled file changes significantly influenced the correctness of the maintenance tasks in the experiment. Inexperienced developers solved more tasks when using our suggestions which means they used the benefit of hints related to similar tasks. The coupled change suggestions allow the developer to follow a set of files and remind him/her that similar tasks include changes in various locations in the source code.

The improvement in the number of solved tasks for the group using the coupled change suggestions shows that developers have used the benefits of additional help in locating the features and the files to be modified to solve their tasks successfully. The group that did not use this kind of help did not succeed in solving the same or a higher number of tasks which points to the usefulness of our approach.

The use of coupled file changes has been especially noticed in cases where the developer needs to perform similar changes in several locations, like editing different views of the application GUI. Here, the developers not using coupled change suggestions missed implementing the change in all the files where the change should have been performed. Coupled file change suggestions help the developers not to miss other source code locations they need for their task.

Time

We analyzed the influence of using coupled file change suggestions on the time needed to successfully perform the tasks versus not using them. Many participants used split-screen and kept the documentation window open so we were not able to subtract the time spent reading the documentation from the total amount needed to solve the tasks.

The distribution of the values for the time needed to solve the tasks is presented in Fig. 3. We see that the distributions are similar with a slight tendency for more time needed to solve the tasks without suggestions.

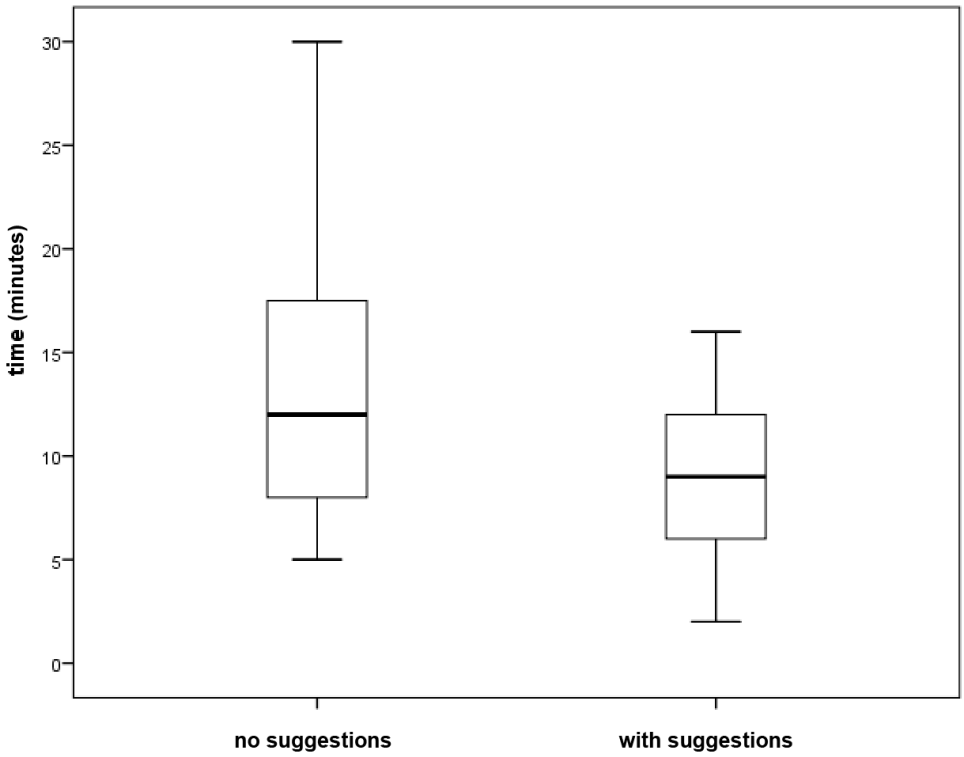

Figure 3 **Time Boxplots (t_s).**

Full-size DOI: 10.7717/peerjcs.135/fig-3

We test the second null hypothesis which claims that there is no influence of the coupled file changes on the time needed to solve the tasks.

The distribution including the time to determine the relatedness of the coupled files is presented in Fig. 4. Considering only the time needed to solve the tasks (t_s), the p-value for the two tailed test is 0.041. This value is slightly below the 0.05 threshold for the significance of the difference in the time needed to solve the tasks by the group using coupled file changes versus the group that didn't. Therefore, we reject the null hypothesis. The r-value for the time needed to solve the maintenance tasks is 0.259 which shows a relatively small statistical difference between the group that used coupled change suggestions and the group that did not.

Considering the case where we include the time to select the coupled files to the time needed to solve the tasks ($t_r + t_s$), we can see that there is almost no difference in the time measured for the group not using the coupled files and the group using coupled files. Here, the p-value for the total time is 0.987, which means that in this case the null hypothesis cannot not be rejected.

The r-value for the total time is 0.02, which emphasizes this small difference between using and not using coupled file change suggestions.

After calculating the grand mean for t_r, we added three more minutes to the amount of time for the task solution and included it in the analysis of the difference between both

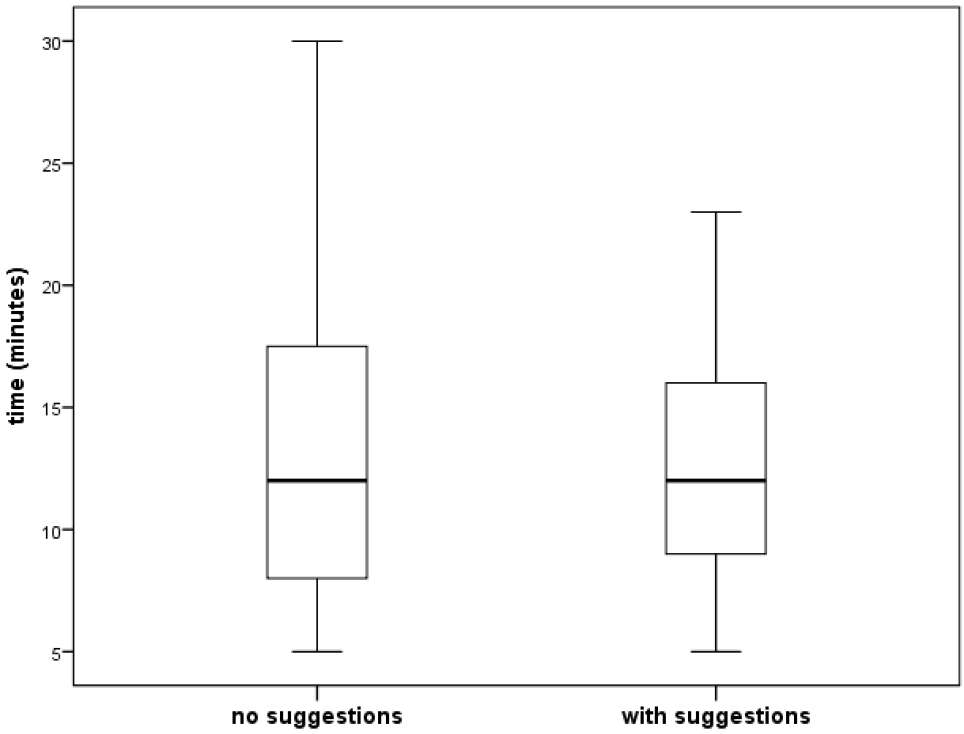

Figure 4 **Time Boxplots ($t_r + t_s$).**

Full-size DOI: 10.7717/peerjcs.135/fig-4

groups regarding the use of coupled file change suggestions. The time needed to determine the related coupled files for the additional participants is presented in Table 9.

For the total time including the time needed to select the coupled files, we add the number of considered coupled files per task and the mean time the developers needed to select the coupled files for the particular task.

The descriptive statistics in Table 10 for the time needed to solve the tasks report a decrease in the means for the time needed to solve the tasks by 26% for the group using coupled change suggestions. The means ranking reports slightly better results for the group using coupled file changes, which means that the participants of this group solved their tasks slightly faster. The standard deviation for the group using coupled changes is twice lower than for the group not using coupled changes which shows a higher spread-out for the first group. Including the time needed to select the coupled files, the values are almost the same for both groups.

From the results, we can see that in this case, because of the additional time we added for each of the participants, there is almost no difference between the mean values which tells us that the group using coupled files did not manage to solve the tasks faster.

The results related to the task selection time show a small improvement for the time needed to solve the tasks. The developers using coupled change suggestions needed less time to find the files to be changed. Without coupled file changes, they would need to search for the features and files in the source code they need to edit.

Table 9 Time to determine related coupled files.

	Time (minutes)			
	Task 1	**Task 2**	**Task 3**	**Task 4**
Participant 1				
Coupled files 1	3	4	3	3
Coupled files 2	5	3	2	2
Coupled files 3	3	2	–	–
Participant 2				
Coupled files 1	2	2	2	3
Coupled files 2	2	2	2	2
Coupled files 3	2	2	–	–
Participant 3				
Coupled files 1	2	4	5	2
Coupled files 2	2	4	2	2
Coupled files 3	2	2	–	–
All participants				
Mean (coupled files)	2.55	2.55	2.66	2.33
Grand mean (tasks)		3.41		

Table 10 Descriptive statistics for the time needed in minutes.

	Median	**Mean**	**Stand. Dev.**
Without suggestions	12	13.50	7.403
With suggestions (t_s)	9	9.11	3.837
With suggestions ($t_r + t_s$)	12	12.33	4.158

The improvement in the time needed to solve the tasks for the group using the coupled file changes is not as strong as the improvement in the correctness of the task solutions. It does not eliminate the time that the developers need to understand the features and the changes they need to perform in the source code. They still need time to organize this information and use it. Furthermore, they need to read and understand the suggestions. Coupled file change suggestions do not automatically provide a solution for solving their tasks.

If we include the time needed to select the coupled files, the results show that there is no improvement for the group using the coupled file change suggestions. If the coupled files need to be determined by the developers as a part of the task solution procedure, the small advantage for the groups using the suggestions disappears. An automated extraction of coupled file change suggestions including the determination of their relatedness could therefore be beneficial.

Usefulness of software repository attributes

The distribution of the usefulness of each repository attribute is presented in Fig. 5. The mean values for the usefulness of each of the repository attributes have been determined using the feedback of all participants in the experiment.

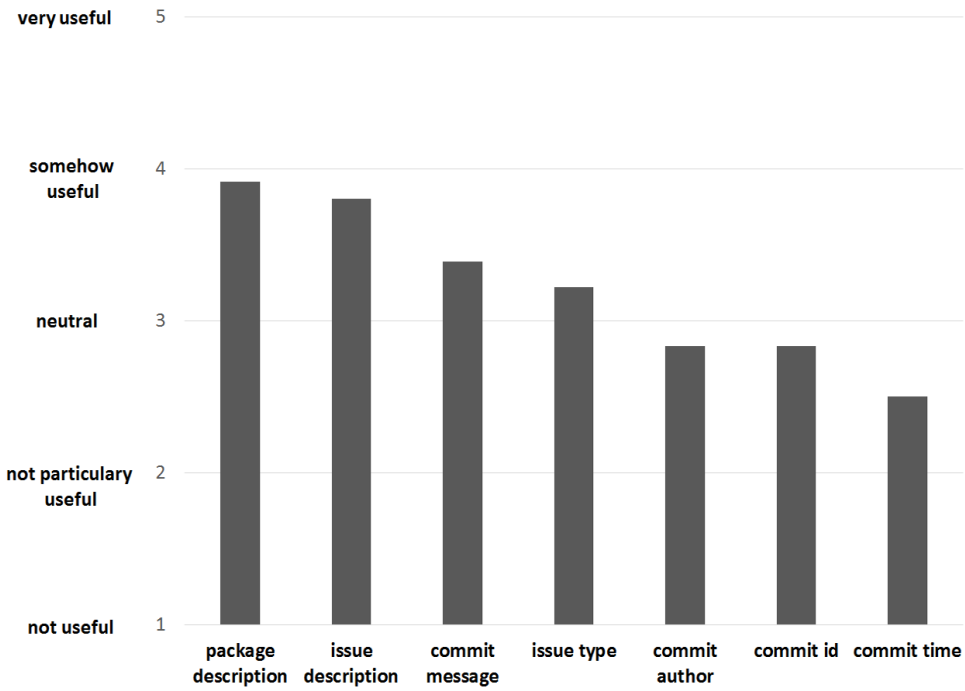

Figure 5 **Usefulness of attributes.**

Full-size DOI: 10.7717/peerjcs.135/fig-5

We test the third null hypothesis which claims that there is no difference in the usefulness between the attributes using the *p*-value of the Kruskal-Wallis H Test. In our case, the *p*-value for this test is 0.000 which is lower than the 0.05 threshold. This result leads us to reject the null hypothesis. This means that the alternative hypothesis claiming that there is a significant difference in the perceived usefulness among the attributes from the software repository is true.

We reported a set of various software attributes from the software repository. The participants reported their feedback on their usefulness at the end of the experiment lab after the tasks had been performed.

We gathered the descriptive statistics for the participants' feedback on the usefulness of each attribute presented in Table 11. The median values vary from 3 for the commit ID, the commit author, the commit time, the issue author and the issue time, to 4 for the commit message and the package description. This places the cutoff between "neutral" and "somewhat interesting" for most of the attributes. The MAD value for all attributes is 1, which shows a low spread out of the usefulness values around the median.

We calculated the *r*-value of the size effect for the repository attributes by creating pairs of each of the attributes where we determined the *z*-value of the Mann–Whitney test for each pair as presented in Table 12. We have 28 pairs of attributes.

The greatest difference in the usefulness is between the commit time and the issue description where the *r*-value is 0.566, followed by the difference between the commit time and the package description with an *r*-value of 0.557. This indicates a high statistical significance between these pairs of attributes. The lowest difference is between the commit

Table 11 Descriptive statistics (attributes usefulness).

Attribute	Median	MAD
Package description	4	1
Issue description	4	1
Commit message	4	1
Issue type	3	1
Commit ID	3	1
Commit author	3	1
Issue author	3	1
Commit time	3	1

Table 12 Statistical significance (coupled changes).

p-value	*r*-value	Repository attribute pairs	
0.180	0.279	commit ID	Commit message
0.972	0.004	commit ID	Commit author
0.249	0.136	Commit ID	Commit time
0.000	0.467	Commit ID	Issue description
0.108	0.190	Commit ID	Issue type
0.624	0.058	Commit ID	Issue author
0.000	0.465	Commit ID	Package description
0.022	0.270	Commit message	Commit author
0.001	0.400	Commit message	Commit time
0.048	0.233	Commit message	Issue description
0.582	0.065	Commit message	Issue type
0.004	0.336	Commit message	Issue author
0.220	0.269	Commit message	Package description
0.228	0.142	Commit author	Commit time
0.000	0.459	Commit author	Issue description
0.122	0.182	Commit author	Issue type
0.599	0.062	Commit author	Issue author
0.000	0.464	Commit author	Package description
0.000	0.566	Commit time	Issue description
0.008	0.311	Commit time	Issue type
0.476	0.084	Commit time	Issue author
0.000	0.557	Commit time	Package description
0.118	0.279	Issue description	Issue type
0.000	0.526	Issue description	Issue author
0.530	0.074	Issue description	Package description
0.039	0.244	Issue type	Issue author
0.009	0.308	Issue type	Package description
0.000	0.515	Issue author	Package description

ID and the commit author, here the r-value is 0.004, followed by the difference between the commit ID and the issue author with an r-value of 0.058. This shows that there are significant differences in the usefulness between individual attributes.

We determined that the attributes have different usefulness using the feedback of the participants. The median ranking defines which of the attributes are most useful. As the most useful attribute we identify the package description followed by the issue description and the commit message. This leads us to the conclusion that the inexperienced developers seek for help about the features of the source code that they need to edit and the task that they have to complete.

The issue type and the commit time are in the middle of the list. The most useless attribute is the commit author followed by the issue author and the commit id. Here, we suppose that the developers are not interested in the information regarding who performed the changes because they do not know this person. This could change if the developers were included in the project for a longer time.

Although we produced a list of typical repository attributes, the participants have identified a smaller set of attributes to be useful for them than we provided in this experiment. This means that we do not have to present all the attributes to the developers together with the coupled files for the reason that different developers can happen to find some attributes as obsolete to be included in the coupled file change suggestions. An individual choice of useful attributes can avoid confusion and increase the acceptance of the coupled file change suggestions concept.

Threats to validity

- Internal Validity: Potential internal validity threats can rise from the experiment design. To limit the learning effect, we use a counterbalanced design where every developer solves four different tasks where each of them solves two tasks without and two tasks using coupled change suggestions. This way the results are not directly influenced by the task supported by the coupled file suggestions.
  Other validity threats related to the experiment design are the selection of the coupled file changes, the creation of the maintenance tasks as well as their definition and solution.
  We extracted coupled files using a relatively high threshold which limits the possibility to provide suggestions for coupled changes that happened by chance.
  We selected the most frequent coupled files for each of the developers to avoid subjective interference. We also avoided delivering unrelated changes in order not to confuse the developer by providing suggestions out of the context.
  The maintenance tasks were constructed manually. However, they are related to issues from the issue tracking system and fulfill the conditions set in the experiment to be perfective and related to changes of the user interface.
  We classified the issues on the system based on the maintenance categories to show the representativeness of our maintenance tasks. The content includes a simple description of the functionalities and the required actions in order not to overwhelm the inexperienced developers by providing unnecessary information.

The set of files included in the solution of the tasks was provided by manually analyzing related issue solutions. We validated the task solutions using a third party.
The judgment of correctness of the developers' task solutions represents another internal threat whereby we test the solutions to determine the level of correctness.
The time needed to determine the relatedness of the coupled files can differ. To avoid an influence by particular tasks, we calculate the average time per coupled file set and calculate the grand mean for all tasks. We used independent student participants for the measurement of the time needed to select the related coupled files.
Also the metrics that we used to determine the usefulness can represent a threat. The subjective usefulness rating represents another construct validity whereby we evaluate the provided task solutions pairwise to minimize the errors in conducting the score distribution. For the time needed to solve the tasks, we play the captured screens of the participants to calculate the time the developers needed to solve the tasks.

- External Validity: The external validity threat concerns the generalization of the experiment. The main threats here are related to the choice of the coupled file changes, the type and description of the maintenance tasks as well as the participants and the system we investigate.
  We used a data mining technique that can be easily performed on other Git repositories to extract coupled file changes. Our approach uses mapping between the commits and the issues which excludes the projects not using them. However, this practice is used very often today. We can find many projects in various on-line software repository collections like GitHub using this kind of mapping and providing issue and project description.
  We chose simple perfective tasks that can be easily replicated and do not require large changes in the source code. The description of the tasks is simple and includes the source code functionalities to be changed and the activities without any specific format or structure. This way we maintain the possibility to repeat the process for other projects and limit the possibility of creating artificial conditions specially tailored for our experiment. Yet, it is not clear whether the results can be generalized for other types of maintenance tasks.
  The student participants in the experiment have basic programming experience which corresponds to the target group of our study to address inexperienced developers.
  The system we used for the experiment is an open source Java project with a clear project structure and repository. It does not contain specific information that can challenge the replication of the analysis.

CONCLUSION AND FUTURE WORK

From the provided results, we summarize that the coupled file change approach was successfully tested in the performed experiment. The participants working with coupled change suggestions provided significantly more correct solutions than the participants without these suggestions.

The participants using coupled file change suggestions did not manage to solve their tasks significantly faster in comparison to the participants working only with the issue descriptions.

We conclude that the coupled file change suggestions can be positively judged to be useful for inexperienced developers working on perfective maintenance tasks. The influence is particularly positive on the correctness of the solutions. The influence of the coupled change suggestions on the effort for solving the tasks is much lower than on the correctness of the solutions. Considering the time needed for the selection of the coupled files as part of the task solution procedure, the use of the coupled files does not give any advantage.

We extended the findings of *Ramadani & Wagner (2016)* where the participants judged the coupled file changes and the attributes as neutral to use in maintenance tasks. Our experiment outcomes are more positive compared to the results in *Ramadani & Wagner (2016)*. Working on real maintenance tasks and a real software product increases the acceptance of coupled change suggestions by the developers. Also, we rounded up the set of useful attributes based on the set of attributes presented in this study.

The next steps would be to transform the results and the findings into full-fledged tool implementation to support the developers working on maintenance tasks with the visual presentation of suggestions of the files they should also change. The final set of attributes presented in the tool should be adjustable so the developers will not be overwhelmed with information which could negate the positive effect we have found in this study.

ACKNOWLEDGEMENTS

We would like to thank Dr. Asim Abdulkhaleq for his help in the evaluation of the coupled files, the tasks and their solutions, the process of scoring and the analysis of the questionnaires. We also thank Nakharin Donsupae, Dominik Kesim and Adrian Weller for their help in the process of selecting the coupled files.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare there are no competing interests.

Author Contributions

- Jasmin Ramadani conceived and designed the experiments, performed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, prepared figures and/or tables, performed the computation work, reviewed drafts of the paper.
- Stefan Wagner conceived and designed the experiments, performed the experiments, wrote the paper, reviewed drafts of the paper.

Data Availability

The following information was supplied regarding data availability:

The raw data has been supplied as a Supplementary File.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.135#supplemental-information.

REFERENCES

Abran A, Nguyenkim H. 1991. Analysis of maintenance work categories through measurement. In: *Proceedings of the international conference on software maintenance*. Washington, D.C.: IEEE, 104–113 DOI 10.1109/ICSM.1991.160315.

Agrawal R, Srikant R. 1994. Fast algorithms for mining association rules in large databases. In: Bocca JB, Jarke M, Zaniolo C, eds. *Proceedings of the international conference on very large data bases*. San Francisco: Morgan Kaufmann Publishers Inc., 487–499.

Armstrong RA. 2014. When to use the Bonferroni correction. *Ophthalmic and Physiological Optics* **34(5)**:502–508 DOI 10.1111/opo.12131.

Basili VR. 1990. Viewing maintenance as reuse-oriented software development. *IEEE Software* **7**:19–25 DOI 10.1109/52.43045.

Basili VR, Caldiera G, Rombach HD. 1994. The goal question metric approach. In: *Encyclopedia of software engineering*. Los Alamitos: John Wiley and Sons, Inc.

Bavota G, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A. 2013. An Empirical study on the developers perception of software coupling. In: Notkin D, Cheng BHC, Pohl K, eds. *Proceedings of the international conference on software engineering*. Washington, D.C.: IEEE, 692–701.

Bieman J, Andrews A, Yang H. 2003. Understanding change-proneness in OO software through visualization. In: *Proceedings of the IEEE international workshop on program comprehension*. Washington, D.C.: IEEE, 44–53 DOI 10.1109/WPC.2003.1199188.

Bird C, Rigby PC, Barr ET, Hamilton DJ, Germán DM, Devanbu PT. 2009. The promises and perils of mining git. In: *Proceedings of the working conference on mining software repositories*. Washington, D.C.: IEEE, 1–10 DOI 10.1109/MSR.2009.5069475.

Briand L, Morasca S, Basili V. 2002. An operational process for goal-driven definition of measures. *IEEE Transactions on Software Engineering* **28**:1106–1125 DOI 10.1109/TSE.2002.1158285.

Canfora G, Cerulo L. 2005. Impact analysis by mining software and change request repositories. In: *Proceedings of the IEEE international software metrics symposium (METRICS'05)*. Washington, D.C.: IEEE, 9–29 DOI 10.1109/METRICS.2005.28.

Carlsson E. 2013. Mining git repositories: an introduction to repository mining. *Available at https://www.diva-portal.org/smash/get/diva2:638844/FULLTEXT01.pdf* (accessed on 13 March 2017).

Chan T. 2008. Impact of programming and application-specific knowledge on maintenance effort: a hazard rate model. In: *Proceedings of the IEEE international conference on software maintenance*. Beijing: IEEE, 47–56 DOI 10.1109/ICSM.2008.4658053.

Cohen J. 1988. *Statistical power analysis for the behavioral sciences*. L Hillsdale: Lawrence Erlbaum Associates.

Coolican H, Taylor F. 2009. *Research methods and statistics in psychology*. London: Hodder Education.

D'Ambros M, Lanza M, Robbes R. 2009. On the relationship between change coupling and software defects. In: *Proceedings of the working conference on reverse engineering*. Washington, D.C.: IEEE, 135–144 DOI 10.1109/WCRE.2009.19.

De Lucia A, Pompella E, Stefanucci S. 2002. Effort estimation for corrective software maintenance. In: *Proceedings of the international conference on software engineering and knowledge engineering*. New York: Association for Computing Machinery, 409–416 DOI 10.1145/568760.568831.

Fischer M, Pinzger M, Gall H. 2003. Populating a release history database from version control and bug tracking systems. In: *Proceedings of the international conference on software maintenance*. Washington, D.C.: IEEE, 23 DOI 10.1109/ICSM.2003.1235403.

Fluri B, Gall H, Pinzger M. 2005. Fine-grained analysis of change couplings. In: *Proceedings of the IEEE international workshop on source code analysis and manipulation*. Washington, D.C.: IEEE, 66–74 DOI 10.1109/SCAM.2005.14.

Fournier-Viger P. 2013. How to auto-adjust the minimum support threshold according to the data size. *Available at http://data-mining.philippe-fournier-viger.com/* (accessed on 13 March 2017).

Fritz CO, Morris PE, Richler JJ. 2012. Effect size estimates: Current use, calculations, and interpretation. *Journal of Experimental Psychology: General* **141**:2–18 DOI 10.1037/a0024338.

Gall H, Hajek K, Jazayeri M. 1998. Detection of logical coupling based on product release history. In: *Proceedings of the international conference on software maintenance*. Washington, D.C.: IEEE, 190 DOI 10.1109/ICSM.1998.738508.

Gall H, Jazayeri M, Krajewski J. 2003. CVS release history data for detecting logical couplings. In: *Proceedings of the international workshop on principles of software evolution*. Washington, D.C.: IEEE, 13–23 DOI 10.1109/IWPSE.2003.1231205.

German DM. 2004. Mining CVS repositories, the softchange experience. In: *Proceedings of the international workshop on mining software repositories*. 17–21 DOI 10.1049/ic:20040469.

Györödi C, Györödi R. 2004. A comparative study of association rules mining algorithms. *Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.2771rep=rep1type=pdf* (accessed on 13 March 2017).

Han J. 2005. *Data mining: concepts and techniques*. Burlington: Morgan Kaufmann Publishers Inc.

Han J, Pei J, Yin Y. 2000. Mining frequent patterns without candidate generation. In: *Proceedings of the ACM SIGMOD international conference on management of data*. New York: Association for Computing Machinery, 1–12 DOI 10.1145/335191.335372.

Hanlon B, Larget B. 2011. Analysis of variance. *Available at http://www.stat.wisc.edu/~st571-1/13-anova-4.pdf* (accessed on 2 June 2017).

Hassan AE. 2008. The road ahead for Mining Software Repositories. In: *Frontiers of software maintenance, 2008. FoSM 2008.* Washington, D.C.: IEEE, 48–57.

Hassan AE, Holt RC. 2004. Predicting change propagation in software systems. In: *Proceedings of the IEEE international conference on software maintenance.* Washington, D.C.: IEEE, 284–293 DOI 10.1109/ICSM.2004.1357812.

Hattori L, Dos Santos Jr G, Cardoso F, Sampaio M. 2008. Mining software repositories for software change impact analysis: a case study. In: *Proceedings of the Brazilian symposium on databases.* Porto Alegre: Sociedade Brasileira de Computação, 210–223.

Hindle A, German DM, Holt R. 2008. What do large commits tell us?: a taxonomical study of large commits. In: *Proceedings of the 2008 international working conference on mining software repositories.* New York: Association for Computing Machinery, 99–108 DOI 10.1145/1370750.1370773.

Hutton A, Welland R. 2007. An experiment measuring the effects of maintenance tasks on program knowledge. In: Kitchenham B, Brereton P, Turner M, eds. *Proceedings of the international conference on evaluation and assessment in software engineering.* London: British Computer Society, 43–52.

IEEE. 1998. *IEEE standard for software maintenance. IEEE Std 1219-1998.* Washington, D.C.: IEEE.

ISO/IEC ISO/IEC 14764: Software Engineering-Software Maintenance. 2006. *Available at https://www.iso.org/standard/39064.html* (accessed on 13 March 2017).

ISO/IEC ISO/IEC 12207: information Technology-Software life cycle processes. 2008. *Available at https://www.iso.org/standard/43447.html* (accessed on 13 March 2017).

Kagdi H. 2007. Improving change prediction with fine-grained source code mining. In: *Proceedings of the IEEE/ACM international conference on automated software engineering.* Washington, D.C.: IEEE, 559–562 DOI 10.1145/1321631.1321742.

Kagdi H, Yusuf S, Maletic JI. 2006. Mining sequences of changed-files from version histories. In: *Proceedings of the international workshop on mining software repositories.* Washington, D.C.: IEEE, 47–53 DOI 10.1145/1137983.1137996.

Lientz BP, Swanson EB. 1980. *Software maintenance management.* Boston: Addison-Wesley Longman Publishing Co., Inc.

Nachar N. 2008. The Mann–Whitney U: a test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology* **4**:13–20 DOI 10.20982/tqmp.04.1.p013.

Nguyen V, Boehm B, Danphitsanuphan P. 2011. A controlled experiment in assessing and estimating software maintenance tasks. *Information and Software Technology* **53**:682–691 DOI 10.1016/j.infsof.2010.11.003.

Pigoski TM. 1996. *Practical software maintenance: best practices for managing your software investment.* 1st Edition. Hoboken: Wiley Publishing.

Pohlert T. 2014. The pairwise multiple comparison of mean ranks package. *Available at https://cran.r-project.org/web/packages/PMCMR/vignettes/PMCMR.pdf* (accessed on 13 March 2017).

Purushothaman R, Perry DE. 2005. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering* **31(6)**:511–526 DOI 10.1109/TSE.2005.74.

Ramadani J, Wagner S. 2016. Are suggestions of coupled file changes interesting? In: Maciaszek L, Filipe J, eds. *Proceedings of the international conference on evaluation of novel software approaches to software engineering*. Setúbal: ENASE, 15–26 DOI 10.5220/0005854400150026.

Revelle M, Gethers M, Poshyvanyk D. 2011. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering* **16**:773–811 DOI 10.1007/s10664-011-9159-7.

Ricca F, Leotta M, Reggio G, Tiso A, Guerrini G, Torchiano M. 2012. Using UniMod for maintenance tasks: an experimental assessment in the context of model driven development. In: *Proceedings of the international workshop on modeling in software engineering*. Piscataway: IEEE Press, 77–83 DOI 10.1109/MISE.2012.6226018.

Shelly GB, Cashman TJ, Rosenblatt HJ. 1998. *Systems analysis and design*. 3rd Edition. Cambridge: International Thomson Publishing.

Shirabad J, Lethbridge T, Matwin S. 2003. Mining the maintenance history of a legacy software system. In: *Proceedings of the international conference on software maintenance*. Washington, D.C.: IEEE, 95–104 DOI 10.1109/ICSM.2003.1235410.

Stafford JA. 2003. Software maintenance as part of the software life cycle. *Available at http://hepguru.com/maintenance/Final_121603_v6.pdf* (accessed on 13 March 2017).

Stevens WP, Myers GJ, Constantine LL. 1974. Structured design. *IBM Systems Journal* **13**:115–139 DOI 10.1147/sj.132.0115.

Swanson EB. 1976. The dimensions of maintenance. In: *Proceedings of the international conference on software engineering*. Los Alamitos: IEEE, 492–497.

Tomczak M, Tomczak E. 2014. The need to report effect size estimates revisited. An overview of some recommended measures of effect size. *Trends in Sport Sciences* **21**:19–25.

Van Rysselberghe F, Demeyer S. 2004. Mining Version Control Systems for FACs (frequently Applied changes). In: Hassan AE, Holt RC, Mockus A, eds. *Proceedings of the international workshop on mining repositories*. 48–52.

Van Vliet H, Van Vliet H, Van Vliet J. 1993. *Software engineering: principles and practice*. New York: Wiley.

Wu R, Zhang H, Kim S, Cheung S-C. 2011. ReLink: recovering links between bugs and changes. In: *Proceedings of the ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering*. New York: Association for Computing Machinery, 15–25 DOI 10.1145/2025113.2025120.

Ying ATT, Murphy GC, Ng RT, Chu-Carroll M. 2004. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering* **30**:574–586 DOI 10.1109/TSE.2004.52.

Zimmermann T, Kim S, Zeller A, Whitehead Jr EJ. 2006. Mining version archives for co-changed lines. In: *Proceedings of the international workshop on mining software repositories*. New York: Association for Computing Machinery, 72–75 DOI 10.1145/1137983.1138001.

Zimmermann T, Weisgerber P, Diehl S, Zeller A. 2004. Mining version histories to guide software changes. In: *Proceedings of the international conference on software engineering*. New York: Association for Computing Machinery, 563–572 DOI 10.1145/1137983.1138001.