JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Do Tools Support Code Integration? A Survey

Martín Dias[a] Stéphane Ducasse[a] Damien Cassou[a]
Verónica Uquillas-Gómez[b]

a. RMoD team, Inria Lille Nord Europe, University of Lille, CRIStAL

b. Norizzk.com

Abstract

Integrating changes made by other developers is a difficult and tedious process. To understand how to help integrators, we first need to know the main questions they ask themselves while integrating and then relate these questions to the tool support that is needed. With this information, researchers and tool developers will be able to focus on the important questions that have little tool support.

In this paper, we report on a 2-step study. In the first step, we did an open call to integrators. We ask them to list questions they ask themselves when they integrate a change. In the second step, based on the questions gathered during the first step and a literature survey, we built a list of 46 questions and run a survey to rank the importance of each question and if the level of tool support was adequate. We present the results we collected. Additionally, we present a taxonomy of the questions according to the kind of information that tools need to answer such questions.

We found out that some questions like "Who is the author of this changed code?" are important and have good tool support whereas others like "Do all the changes within the commit belong together? (Can we split the commit?)" are moderately to extremely important and have no tool support.

1 Introduction

Software is in constant evolution [DDN02]. In a software project, code changes represent bug fixes, enhancements, new features and adaptations due to changing domains. The evolution of a project codebase is usually managed in a *revision control system* that supports *branches*. Developers perform code changes in a branch and often such changes should be merged into another branch. This activity is called *integration*. Integration of changes is a difficult activity and poses substantial challenges [UG12, GZSvD15]. Focused on understanding development challenges, several research works [LM10, PTL+11, SMDV08, FM10] systematically characterize what questions developers need to answer when working. These works present catalogs (*i.e.,* lists) of questions that serve as a basement for research on new tools to improve the

Martín Dias, Stéphane Ducasse, Damien Cassou, Verónica Uquillas-Gómez. *Do Tools Support Code Integration? A Survey*.

In Journal of Object Technology, vol. 15, no. 2, 2016, pages 2:1–20. doi:10.5381/jot.2016.15.2.a2

performance of developers. Besides these results are useful for development in general, there are no catalogs focused specifically on integration. A recent survey [GZSvD15] proposes some questions to characterize pull requests but the authors do not focus on a systematic characterization of questions that integrators ask themselves. There is not enough research done for knowing qualitatively and quantitatively these challenges.

The main contributions of this paper are:

1. A catalog of 46 questions that integrators ask when performing integration of changes. The main motivation behind obtaining these questions is to identify and understand what are the information needs and tool support of developers that deal with integration activities (Section 4).

2. An evaluation of each question of this catalog. For each question, the participants had to rank the importance and the support that tools offer. In a period of 5 months we received the answers of 42 integrators who integrate changes on diverse software projects (Section 5).

3. A taxonomy of the questions according to the kind of information that tools need to answer such questions. Since the final purpose of collecting and evaluating the integrator's questions is to improve their tools, we present a tentative classification of the information required to answer such questions (Section 6).

2 Vocabulary

In this section we introduce the definitions and terminology that we use in the rest of this article. Not all of them are well-known or standard terminology in the context of integration of software changes, therefore we define them explicitly.

Code Changes. A code change is any alteration to the codebase of a program. Often, development tools manipulate code changes as mere text changes *i.e.,* as insertion and removal of lines of text in the a sources file. However, since the source files represent code entities of the program, code changes are removals, additions and modifications to code entities.

Commits. A commit in the context of *version control systems* (VCS) refers to the act of submitting code changes to a repository. A commit can also refer to the group of additions, modifications and removals made to the source code that developers submit to the repository and result in a new revision (also known as version). For example, if a set of changes (commit) cross-cuts three packages, such changes are submitted separately resulting in three package versions.

Deltas. A delta is a set of changes representing the differences between two successive commits in the history.

Sequences of changes. Sometimes, developers perform a single task (*e.g.,* implementing a new feature) spread over several commits. We call sequence of changes (or sequence of commits) to several commits in the history of a program which are related to a single task.

3 Related Work

Questions about Code. LaToza and Myers [LM10] conducted a survey to investigate the questions that developers consider *hard-to-answer*. From the answers of 179 developers at Microsoft, the authors collected 371 questions like "Are the benefits of this refactoring worth the time investment?", "Is this functionality already implemented?" or "How does this code interact with libraries?". The authors classified these questions in 10 categories: rationale, debugging, policies, history, implications, implementing, refactorings, teammates, building and branching, and testing. They concluded that having a better understanding of developers' information needs may lead to new tools, programming languages, and processes that make *hard-to-answer* questions less time consuming or error prone to answer.

Study on Pull-Request. Gougios *et al.* [GZSvD15] performed a study focused on the quality model that developers have in mind when they accept pull-requests on GitHub. Some questions characterize the projects (frequency of pull-requests, tools used to assess and perform the merge, kind of requests). Then, they asked developers to rank factors of acceptance or rejection: presence of tests, number of commits, comments, etc. They asked how the code is reviewed, how the requests are sorted. The work style of the developer is also considered. While the poll focuses on pull-requests, it is difficult to classify the underlying questions according to different perspectives.

Costs and Benefits of Branching. Bird and Zimmermann [BZ12] presented a survey on how developers use branches, and an empirical analysis of cost and benefit of branches in diverse scenarios. The analysis aims at determining the usefulness of a branch in terms of cost-benefit. The rationale is that useless (high-cost-low-benefit) branches can produce severe impact on the development process of a large project (*e.g.,* missing deadlines and increased failures). The authors consider integration as an error-prone activity and propose high-level operations to restructure branches to reduce the number of useless branches. While the survey is restricted to developers at Microsoft, the results are generalizable.

Study on Integration Decisions. Phillips *et al.* [PRS12] performed a study focused on how developers of a large-scale system make branching and integration decisions while managing releases. They evaluated a survey they previously elaborated [PSW11] by conducting semi-structured interviews with seven developers of a company. The authors found that developers making decisions need to consider 10 factors, such as potential conflicts, bug counts, and dependencies between branches. The authors also identified the information needed to support integration. Release decision makers need to predict storms of conflicts, detect pressure building up from non-integrated changes, monitor code flow between branches (what is the frequency of integrations), and track branch health (metrics such as test results, bugs, and task completion at branch level).

Empirical Study on Branching and Merging. Premraj *et al.* [PTL+11] presented an empirical study that observed developers branching without considering the consequences on merging. The goal of the study was to understand the implications of such branching for the cost of merging changes. The study had two parts: 1) A qualitative study where 16 developers were surveyed (5 questions oriented to branchers and 3 questions oriented to integrators) to learn their views on branching and merging files, and their experience with the development overhead from branching and merging; 2) a quantitative study that calculated the number of branches, the number of merges on a number of files, and the time spent on merging files. From the study they established (a) the roles of the branchers and mergers (*i.e.,* architects,

configuration managers, integrators and developers), and (b) the types of files that dictate the cost of merging (*e.g.,* configuration files). They concluded that VCS tools and VCS best practices (*e.g., branch only when necessary, branch late, propagate early and often*) are not sufficient to share files in an agile development environment. They also suggested that contents of shared files must be aligned with the responsibilities of the primary owners of those files, as a way to decrease conflicts of branching and merging files.

Questions Related to Evolution Tasks. Sillito *et al.* [SMDV08] proposed a catalog of 44 types of questions programmers ask during software evolution tasks. The authors' goals were to better understand what a programmer needs to know about a code base when performing a change task, how a programmer goes about finding that information, and how well today's programming tools support evolution. They performed two qualitative studies [SVFM05, SMDV06] observing 9 and 16 programmers respectively, making changes to medium and large sized code bases. From the analysis of the empirical information collected during both studies, they established the used tools, types of change tasks, paired versus individual programming, and the level of prior knowledge of the code base. 44 questions were classified in 4 categories: (a) finding focus points (*e.g.,* "Where in the code is the text in this error message or UI element?"), (b) expanding focus points (*e.g.,* "Where is this method called or type referenced?"), (c) understanding a subgraph (*e.g.,* "How are instances of these types created and assembled?"), and (d) questions over groups of subgraphs (*e.g.,* "What will the total impact of this change be?"). They also established that 34% of the questions was fully addressed by tools and 66% of the questions only partially addressed. From the results, they found that programmers need better tool support for asking more refined or precise questions, maintaining context, and piecing information together.

Information Fragment Model. Fritz and Murphy [FM10] presented a study in which they interviewed 11 professional developers to identify different kinds of questions they need answered during development, but for which support is weak. From the results, they established a catalog of 78 questions classified in several categories such as: (a) people specific (12 questions *e.g.,* "Which code reviews have been assigned to which person?"); (b) code change specific (35 questions *e.g.,* "What are the changes on newly resolved work items related to me?"); and (c) work item progress (11 questions *e.g.,* "Which features and functions have been changing?"). Alongside this study, they introduced the information fragment model (*i.e.,* a subset of development information for the system of interest) and associated prototype tool built on top of Eclipse for answering the identified questions by composing different kinds of information needed. This model provides a representation that correlates various software artifacts (source code, work items, team membership, comments, bug reports, and others). By browsing the model, developers can find answers to particular development questions.

As summarized in this section, several related works present catalogs of questions as a means to understand development activities (*e.g.,* maintenance or code comprehension) and to identify the developers' information needs. However, these works are not focused specifically in integration activities but on development activities in general.

4 Qualitative Study on Integrator Questions

In this section, we present a catalog of questions that integrators ask when performing integration of changes. The main motivation behind obtaining these questions is to identify and understand which are the information needs of developers that deal with integration activities.

Integration of changes is a difficult and tedious activity. By knowing which real questions are raised during integration and are troublesome to answer, the complexity of this process can be understood, common factors can be extracted to characterize changes, and future solutions can be assessed.

4.1 Methodology

This catalog is based on work done by V. Uquillas-Gomez for her PhD thesis [UG12]. We conducted an open call to the developers of three Smalltalk communities to compile the questions. Specifically, we sent a mail to three development mailing-lists (VisualWorks Users[1], European Smalltalk User Group[2], and Pharo project[3]) requesting input on the questions they ask themselves when integrating. We first provided an overview of the reasons of our study; next, we asked "What are the questions that you ask yourselves when you are merging (or want to merge) changes into your projects?"; finally, we added six typical questions raised by one of the main Pharo integrators (*e.g.,* "Is this change impacted by a change that happened in another branch of my software?") as a way to guide their answers.

In a period of 10 days we received the responses of 20 participants who integrate changes on small, medium and large Smalltalk projects. The answers were diverse among the group: (a) 8 participants provided concrete questions; (b) 9 participants provided concrete questions and extra feedback (merge situations they deal with, policies they follow when merging, explanations of why they ask these questions or think they are challenging, and broad ideas for tools supporting merging); (c) 3 participants did not list any question at all but rather included general feedback regarding their desiderata for merging tool support. This information was analyzed and yielded 56 questions. Moreover, we took into account related studies presented above [FM10, LM10, SMDV08], extending our findings with 8 questions taken from these studies. Finally, a Pharo integrator helped refining and verifying the questions.

The resulting catalog is composed of 46 questions that we clustered into 5 different categories: (a) authorship/ownership, (b) structural change characterization, (c) behavioral change characterization, (d) bug tracking infrastructure, and (e) temporal and change sequence.

4.2 Results

Following, we briefly describe each category prior to introducing its respective questions. Each question is accompanied by an identifier (*e.g.,* A_1) that is used to refer to the question in later sections.

Authorship/Ownership. The first category of questions is related to the owner of the original code, to the author of the changes, and to the committer. These questions assess the author's quality and the reliability level of his changes.

	Authorship/Ownership questions
A_1	Who is the author of this changed code?
A_2	Who was the previous owner of the changed code?
A_3	Has my own code been changed?
A_4	What is the general quality of the change committer?
A_5	How many people have contributed to this group of commits?

[1] vwnc@cs.uiuc.edu
[2] esug-list@lists.esug.org
[3] pharo-project@lists.gforge.inria.fr

Structural change characterization. The second category of questions is related to the structure of the original code as well as the changes. They cover various aspects in terms of volume, impact volume, dependencies (which packages, classes should be loaded before), and so on. From that perspective, they are not tailored to a sequence of changes but more to a single delta [UGDD10].

	Structural change characterization questions
S_1	How large is the change?
S_2	How many entities (packages/classes/methods) are impacted by the commit? (Impacted in the sense they can stop compiling, for example)
S_3	Is this commit confined to a single package or spread over the entire system?
S_4	What is the complexity of the changes?
S_5	Do all the changes within the commit belong together? (Can we split the commit?)
S_6	Are there other packages that will need to change as well to integrate this commit? (Can we identify the users of the changed code?)
S_7	Will the code compile after applying this commit?
S_8	Is the commit conflict free? (Does this change generate any syntactic merge conflicts when integrating?)
S_9	Which entities (packages/classes/methods) have been changed?
S_{10}	Does this change depend on other changes (in the source branch) to be functional (in the target branch)?

Behavioral change characterization. The third category of questions is related to the nature, behavior and intent of a change. Such questions can be mostly applied to changes within a single delta. Note that some of these questions are open-ended and therefore inherently difficult to answer automatically. Moreover they may require up front knowledge of the system as well.

	Behavioral change characterization questions
B_1	Does this commit follow rules and conventions?
B_2	Is the vocabulary used in the commit consistent with the one on the system?
B_3	Does this commit improve the quality of the system?
B_4	Does this commit correctly fulfil its goal? (Does it fix correctly a particular problem?)
B_5	What is the intention of this commit?
B_6	In a commit with 'strange code', was the strange code intentional (it has to be like that to turn around a special aspect of the system), or accidental (the author did not really know what he was doing)?
B_7	What kind of commit is it? (Bug fix/New feature/Refactoring/Documentation/...)
B_8	Did this commit fix/break tests? Which tests?
B_9	Is the commit covered by tests? What is the coverage? How can I test it?
B_{10}	If I apply the commit, what are the parts of my current system that it affects? What are the users (classes/methods/functions) potentially impacted by this change in the destination branch/fork?)

continued on next page...

. . . continued from previous page

	Behavioral change characterization questions
B_{11}	What are the implications of this commit on the (potentially undeclared) API? (Are there any unknown users of the API that will be impacted by the changes?)

Bug tracking infrastructure. The fourth category of questions is related to bug tracking trazability of changes.

	Bug tracking infrastructure questions
I_1	To which bug entry does this change relate?
I_2	What bug fixes also affected the part of the system that is being impacted by this change?

Temporal and change sequence. The final category of questions is related to situating changes within the context of a sequence of changes, as well as to the time at which the changes occur. Indeed, often a change does not happen in isolation, other changes may depend on it and fork analysis requires to understand change dependencies [UGDK14]. In particular, when working on a sequence of changes, these questions capture the place of a change within the sequence.

	Temporal and change sequence questions
T_1	How old is this commit (compared to the version to which it should be integrated)?
T_2	In which commit/version of the system was this method/function previously changed?
T_3	Did this class/method/function change (a lot) recently/in the past?
T_4	Is this change to a class/method/function the most recent one (in the branch)?
T_5	Is there any pending change in the sequence of commits (in the branch) that supersedes this one?
T_6	Is this commit part of a whole series of commits?
T_7	Does this commit depend on previous ones? (What are the other commits needed first to merge this commit?)
T_8	Is the change to a class/method/function ever used in subsequent changes?
T_9	Is this change to a class/method/function reverting the code to an old state?
T_{10}	What else changed when this code was introduced or modified (*e.g.,* documentation, website, database schema)?
T_{11}	What other classes/methods/functions changed when this code was introduced or modified?
T_{12}	What are the changes made by the same authors/during the same time period?
T_{13}	Did the changing classes/methods/functions of this commit change together in a previous commit?
T_{14}	If there were changes to class/methods/functions happening together in the past, can we suspect that there is still something missing in the current commit?
T_{15}	Were the classes/methods/functions affected by this change renamed in the past and, if so, in which version of the system?

continued on next page. . .

... continued from previous page

	Temporal and change sequence questions
T_{16}	What were the users (callers) of a changed method/function in a particular version of the system?
T_{17}	What are the current users (callers) of a changed method/function?
T_{18}	What commits of another branch have been integrated into this branch?

4.3 Threats to Validity

The fact that all the participants are Smalltalk experts, and that their answers are focused on the integration of changes in of Smalltalk projects may be considered a threat to validity of our study. However, our audience was diverse in several ways. The participants work on different Smalltalk projects which follow different development policies. This is a positive aspect because we received different points of view regarding the integration process. In addition, often programmers work with several programming languages.

Our study was performed by an heterogenous group of participants from both industry and academia. Since our academia participants integrate changes in open-source projects with many users, we consider their answers have the same weight as the answers of industry participants. Besides, academia participants were a minority.

Furthermore, this study covers a topic that is present in any collaborative development process independently of the programming language and infrastructure used. The related work discussed before also proposed the identification of developers' information needs by means of questions. Even though their studies covered other broad aspects, our participants proposed questions also present in these studies. This shows the generalizability of such studies, and the questions gathered in our study can be used to assess future solutions.

5 Quantitative Study of Integrator Questions

The questions presented in Section 4 express needs of integrators. Such needs may have different importance for the everyday's work of the integrators. Additionally, some of those questions may be currently supported by development tools, whereas others may not. Since the final purpose of our work is the improvement of integration toolset, we decided to evaluate the questions. Thus, we conducted a survey where integrators ranked each question of the catalog in two dimensions:

- *Importance*: Nothing, Little, Moderate, and Extreme.
- *Tool support*: No, Partially, and Yes.

5.1 Methodology

We called for participation in several software development communities, which include Smalltalk-related mailing-lists, the Twitter accounts of the Apache Software Foundation and the Eclipse Foundation. In a period of 5 months we received the responses of 42 participants who integrate changes on very diverse software projects. The survey included a "Participant Profile" part to categorize participants and their projects. We start by structuring the results of this part because of its impact on the results. The full and detailed results of this study are available in a technical report [DUGCD14].

5.2 Participant and System Profiles

As can be seen in Figures 1 and 2, the experience of the participants is quite diverse, both in development and in integration. The distribution still shows that the participants had a serious experience in development. Figure 2 shows that integration has a smaller spread period (from 2 to 20 years compared to the 5 to 45 years of experience) and that the duration of the integration activity is shorter (from 2 years compared to 5).

Most of the participants (88%), answered they are (or they were) developers of the system where they integrate changes; this is a result that we expected due to the complexity of integration activity, which requires a deep knowledge of the codebase of a project that generally only a developer that worked on it has. Turnover and open-source projects may change such fact. Future and broader studies may contradict this fact.

System characterization. Figure 3 reveals that most of participant's systems have between 3 and 16 developers. The participants reported to work in systems with around 5 forks (median). About the frequency of integrations, participants answered performing them mostly *on demand*, *i.e.,* not regularly but when necessary. Around 20% of the participants answered that they never perform integrations between forks. In the answers, approximately half of the systems are open-source. About the size (in lines of code), most of the projects have between $10k$ and $450k$ LOC. When we asked for the kind of software were they integrated changes, three quarters answered they work in *End user applications*, and the same number of responses for *Libraries, frames and platforms*. This means that many participants integrated changes both in client- and in provider-side of their software.

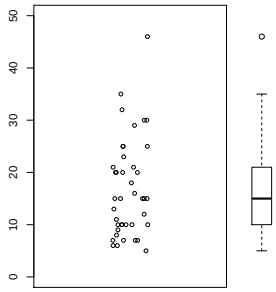


Figure 1 – How many years have you been developing software?

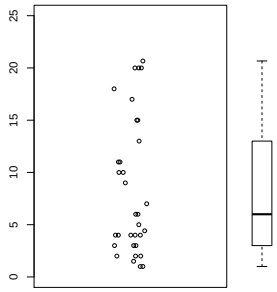


Figure 2 – How long have you been integrating changes?

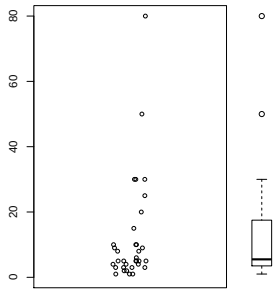


Figure 3 – How many developers are working on this system? (we removed two outliers: 250 and 600)

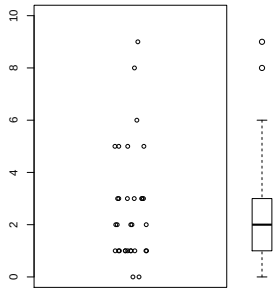


Figure 4 – How many integrators are working on this system? (we removed one outlier: 60)

Approximately 26% of participants answered they are not the main integrator of their system. This percentage shows how much the role of integration is shared by several project members. In fact, Figure 4 reveals that there is a median of 2 integrators per system. 93% of participants reported that they interact with developers when integrating changes. In the survey we asked what are the reasons of such interaction with developers. The main reason participants answered was solving merging conflicts. They answered as well that understanding the changes, and giving feedback about the quality of the changes are other reasons for interacting. Figure 5 shows that "Development" and "Release" are the most used types of branches, although "Feature", "Bug fix" and "Experimental/Prototype" are common as well. Figure 6 shows that the participants consider merge conflicts and regressions as the most significant problems.

In the results, we observe the same number of participants use general-purpose VCS (CVS, Git, SVN and TFS) and Smalltalk-specific VCS (Monticello, StORE and ENVY) (Figure 7). On the one hand, we can extract from this observation that half of the responses correspond to integrators working in projects that involve Smalltalk code. On the other hand, the fact that all (or most) of the Smalltalk environments only support a Smalltalk-specific VCS indicates that around half of the responses correspond to *non-Smalltalk projects*.

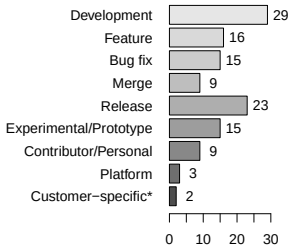


Figure 5 – What types of branches are defined for this system? (category "Customer specific" was extracted from "Other" field)

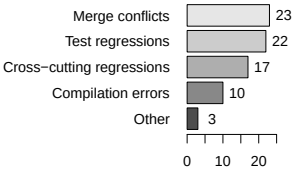


Figure 6 – What are the most significant problems that you have had with merges?

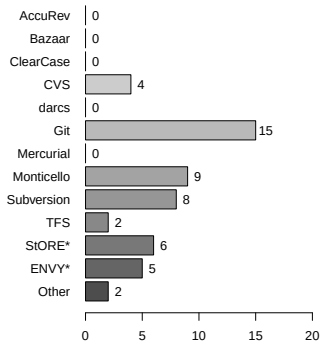


Figure 7 – Which versioning tool(s) are using for this system? (categories "StORE" and "ENVY" were extracted from "Other" field)

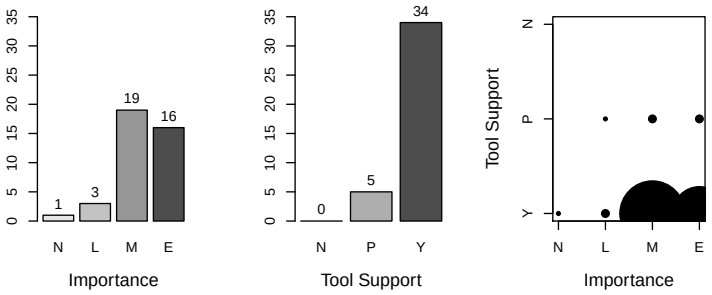
5.3 Results

We published all the collected data from this survey as a technical report [DUGCD14]. In this section, we classify the questions and report only the questions identified as important and with little tool support. We focus on them since they express how to improve the integration toolset.

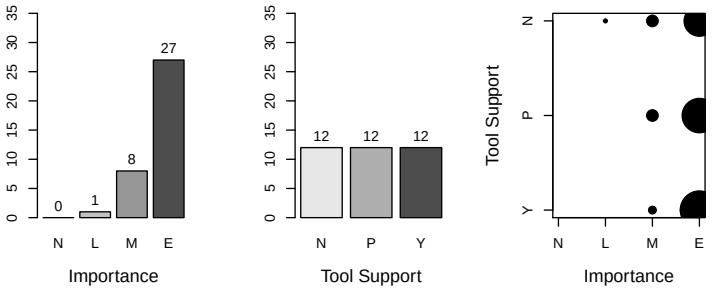
Importance: If a question's responses are concentrated among No and Little importance, we say the question has *Agreed No-Importance*; if responses are concentrated between Moderate and Extreme importance, we say the question has *Agreed Importance*. Instead, when there is not a clear agreement among the answers, we say the question has *Disagreed Importance*.

Tool Support: If a question's responses are concentrated around No, we say the question has *Agreed No-Support*; if responses are concentrated around Yes, we say the question has

Agreed Support. Instead, when there is not a clear agreement among the answers, we say the question has *Disagreed Support*.



(a) A_1 *Who is the author of this changed code?*



(b) B_8 *Did this commit fix/break tests? Which tests?*

Title	Legend
Importance	Nothing (N), Little (L), Moderate (M), Extreme (E)
Tool Support	No (N), Partially (P), Yes (Y)

Figure 8 – Participant responses to two questions. From the survey we conclude that question A_1 has *Agreed Importance* and *Agreed Support* question, while B_8 has *Agreed Importance* and *Disagreed Support*.

We illustrate our classification criteria in Figure 8, which presents the responses to two integration questions:

A_1 : Who is the author of this changed code?

B_8 : Did this commit fix/break tests? Which tests?

For A_1, participants mostly agreed that this question has from moderate to extreme importance, and that it does have tool support. Thus, we classify A_1 as a question with *Agreed Importance* and *Agreed Support*. For B_8, participants also agreed that it is an important question, but they disagreed in the tool support. Then, we classify B_8 as a question with *Agreed Importance* and *Disagreed Support*.

We applied this criteria to all the questions of the catalog. Results are presented in Table 1. Overall, in the dimension of Importance, 33 questions (72%) have *Agreed Importance*, 10 have *Disagreed Importance* (21%), and only 3 have *Agreed No-Importance* (7%). In the dimension of Tool Support, 12 questions (26%) have *Agreed Support*, 9 have *Disagreed Support* (20%), and 25 have *Agreed No-Support* (54%). In the following Subsection, we use this categorization to discuss the most relevant results.

	Agreed No-Importance	*Disagreed Importance*	*Agreed Importance*
Agreed No-Support	T_{12}, T_{13}	A_4, B_2, B_6, T_{14}, T_{15}	S_4, S_5, S_6, S_{10}, B_1, B_3,B_4, B_9, B_{10}, B_{11}, I_2, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{16}
Disagreed Support	A_5		S_2, S_7, B_8, I_1, T_3, T_{11}, T_{17}, T_{18}
Agreed Support		A_2, A_3, S_1, T_1, T_2	A_1, S_3, S_8, S_9, B_5, B_7, T_4

Table 1 – Classification of the questions according to surveyed integrators.

5.4 Top 9 Important Questions without Tool Support

In the following, we discuss 9 questions that are important and have no tool support. These questions are all in the upper-right corner of Table 1. The answers to these questions are summarized in Figure 9.

We found three conceptual axis in these questions, which we use to interpret the survey results: *understanding change impact*; *understanding change dependencies with cherrypicking*; and *understanding change scattering*.

5.4.1 Understanding Change Impact

Understanding the impact of applying a code change on the current system is a key concern of integrators. The effects of a change are of crucial importance since it can introduce unexpected behavior in the system. The following questions capture the problems faced by integrators when assessing the impact of code changes.

S_6 : Are there other packages that will need to change as well to integrate this commit? (Can we identify the users of the changed code?)

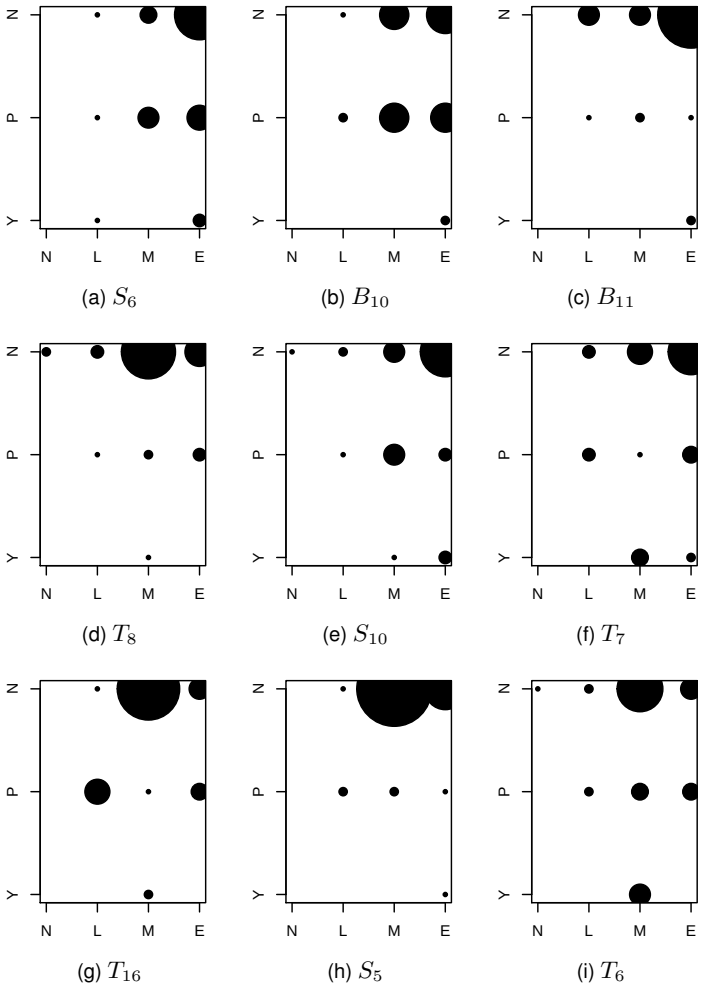
B_{10} : If I apply the commit, what are the parts of my current system that it affects? What are the users (classes/methods/functions) potentially impacted by this change in the destination branch/fork?)

B_{11} : What are the implications of this commit on the (potentially undeclared) API? (Are there any unknown users of the API that will be impacted by the changes?)

T_8 : Is the change to a class/method/function ever used in subsequent changes?

T_{16} : What were the users (callers) of a changed method/function in a particular version of the system?

These five questions share the need of understanding the impact of integrating a change in the destination branch. However, these questions do not refer to the same kind of impact: In one hand, B_{10} and T_8 refer to the local impact, *i.e.,* which are the entities in the codebase (in the target branch) that are affected by the change. In the other hand, S_6 and B_{11} refer to the external impact, *i.e.,* which entities in other codebases are affected with the change. The latter is usually called *ripple effect*. [YCM78]



(a) S_6 (b) B_{10} (c) B_{11}

(d) T_8 (e) S_{10} (f) T_7

(g) T_{16} (h) S_5 (i) T_6

Axis	Title	Legend
x	Importance	Nothing (N), Little (L), Moderate (M), Extreme (E)
y	Tool Support	No (N), Partially (P), Yes (Y)

Figure 9 – Top 9 questions with *Agreed Importance* and *Agreed No-Support*.

Questions T_8 and T_{16} complement each other: the former talks about impact in future commits of a branch, while the latter talks of impact in past commits. These questions address important issues when the commits around a change have to be understood.

5.4.2 Understanding Change Dependencies when Cherrypicking

Sometimes an integrator has to apply in a branch (*i.e.,* target or destination branch) some code changes selected from another branch (*i.e.,* source branch). This action is known as *cherrypicking* code changes. The main difference with a plain merge is that not every change from the source branch is reapplied in the destination branch, but only a selection of such changes.

Given a piece of code often depends on other pieces of code (*e.g.,* a method that invokes other methods during execution), cherrypicking is an error-prone and difficult activity. For example, the cherrypick of a method modification which adds a reference to a class that does not exist in the destination branch will lead to a compilation error. Since a commit can depend on other commits, and the identification of such dependencies is a complex task, the support from tools is important [UGDK14]. The following questions capture such problems.

S_{10} : Does this change depend on other changes (in the source branch) to be functional (in the target branch)?

T_7 : Does this commit depend on previous ones? (What are the other commits needed first to merge this commit?)

These two questions have much in common. Both focus in the understanding of the dependencies of a change. The main difference is that T_7 refers in general to dependencies in any commit of the source branch, while S_{10} is limited to the changes in the source commit (of the cherry-picking operation).

5.4.3 Understanding Change Scattering

Developers often bundle changes of unrelated tasks (*e.g.,* bug fix and refactoring) in a single commit, thus creating a so-called *tangled commit*. In a study, Herzig and Zeller [HZ13] analyzed several open-source projects and found that 20% of the bug-fixing commits are tangled, *i.e.,* these commits contain unrelated changes apart of the bug fix changes. In other cases, however, developers perform a single task (*e.g.,* implementing a new feature) spread over several commits, which also poses difficulties to understand changes.

The following questions can be interpreted as two sides of the same problem: the wrong spread of code changes along commits.

S_5 : Do all the changes within the commit belong together? (Can we split the commit?)

T_6 : Is this commit part of a whole series of commits?

In one hand, question S_5 is related to understanding changes in a *tangled commit*. In the other hand, question T_6 is about understanding changes related to one task, that are spread in several commits.

Question S_5 is the most important question without tool support according to the participants of our survey. Unlike the other questions discussed above, S_5 and T_6 have no direct reference to change dependencies or impact. However, any tool that answers such questions will certainly benefit from this information.

5.4.4 Conclusion

In this Section we presented a quantitative evaluation of the catalog of integration questions introduced in Section 5. We described and discussed the most relevant results obtained from the survey, highlighting some questions that integrators agree in that they are important and have little tool support. We think these highlights can be a guideline for new research on improving the work of integrators. We further analyze the information needs for answering all catalog questions in Section 6.

6 How to Answer Integrator Questions?

Since the final purpose of collecting and evaluating the questions of Section 4 is to improve the tools for integration, we have elaborated a taxonomy of the information required to answer such questions. We established four high-level categories: (1) descriptive information, (2) structural information, (3) semantic information, and (4) historical information. While the last category includes information related to changes in the context of a sequence of changes, the other categories provide information that can be used for characterizing changes within a single delta and within a sequence. In the following subsections, we explain the mentioned categories, and we conclude with a summary of the taxonomy.

6.1 Descriptive Information

Author/Owner. The developers who produced a code change (*i.e.,* the *author*) and the one who committed the code change to the VCS repository (*i.e.,* the *committer*) may be different persons. The *owners* of a piece of code are developers that had changed it in the past. Not every VCS support the distinction between author and committer. For example, CVS and SVN do not support this feature, while in Git a commit does have both author and committer signatures.

Information regarding code owner as well as code change author and committer is specially relevant in large development teams, where many people interact. With this information, integrators can take better decisions about code changes: For example, changes made by the owner of a piece of code could be considered more reliable than changes made by other developers [BNM+11]. Ownership can be derived by metrics that consider the number or size of changes that a developer has committed for a particular piece of code.

Time. The point in *time* at which a change happened is key to establish the order of a change, and helps identifying sequences of changes during the evolution of a system. IDEs may record the time when a source code change is made, and VCSs register the time when changes are submitted to the repository. Temporal information is heavily used to support several activities in the development process, such as *untangling changes* [HZ13, DBG+15] and *change impact analysis* [CRR05, GHR09, HZ11]. Additionally, temporal information is useful to establish the history of a system and dependencies of changes.

6.2 Structural Information

Structure. The packages, classes and methods are the core of programs. Identifying which entities changed and how they relate to each other can ease understanding these changes. For example, a pull up/push down method refactoring can be detected by identifying if their respective classes belong to the same class hierarchy [UGDD10]. Several recent approaches for untangling changes use structural information [BBBL15, DBG+15, HZ13].

Change Scope. Identifying if the changed source code is *local* to a method, class, hierarchy, or package, or that it *cross-cuts* multiple entities establishes the scope of a change (*e.g.,* multiple changes are contained in one single package) [DGK06]. Assessing a local change is often simpler than assessing one cross-cutting several packages. Cross-cutting changes that are not necessarily structurally related but evolve together may indicate a coupling between these changes [GDMR04, GJK03]. Knowing this information may help identify missing changes. Therefore, getting a fast overview of the location of changed program entities in the context of the hierarchy and package structure is important to assess changes.

Kind of Action. Understanding whether the changes are mainly *adding*, *removing* or *modifying* behavior is another level of characterization. Whether changes are at the level of entire methods (*i.e.,* a method was added or removed) or intra method (*i.e.,* a method's body was modified) is another element. Whether the changes were actually changing the semantics of the system (*e.g.,* not just changes to license or comments) is complementary to the other information. Identifying specific actions such as renamings (a renaming is usually stored as a removal and an addition) would definitely improve any characterization of changes. However, this can be a challenging process when the addition differs a lot from the removal.

Kind of Entity. Characterizing changes by kind of entity (*e.g.,* fields, methods, comments, etc.) they affect is straightforward. This, combined with previous characterizations such as *kind of action*, can ease in assessing the impact of changes. For example, if only comments were affected at class or method level, developers can rapidly identify that these changes have no semantical impact on the system. Therefore, they can integrate these changes without dedicating time to understand and assess their impact.

6.3 Semantic Information

Vocabulary. Identifiers (class names, field names, function names and parameter names) and comments are important elements of the source code that give hints about semantics and intent of the developers [TGM96, AL98, KDG05]. They represent the vocabulary present in a system, and such vocabulary is affected when the system evolves [AHM+09] (*e.g.,* if new features are introduced or if existing behavior is removed is reflected as additions and removals of identifiers). Assessing the difference in vocabulary between a change and the target branch can give information about whether or not the integrator should merge that change. Moreover, the amount of introduced or removed vocabulary can provide an overview of the changes and their impact. For example, having a large number of changes with a limited number of affected vocabulary (*e.g.,* one added and one removed identifier) could mean that a function was renamed, or that a call was replaced among its clients.

Reason. A system is constantly evolving due to fixes of bugs, enhancements, new features or adaptations to changing environment. The reason behind *why* a piece of code changed is fundamental to aid in understanding and assessing the impact of a change. Committers can add a description (*i.e.,* commit message) about the changes they submit into the repository. Unfortunately, committers may omit important details of *what* and *why* changed, even more when they submit multiple unrelated changes in one commit.

6.4 Historical Information

History. The history of a system contains a wealth of information that can be used to understand the system and its evolution, to detect problems in the system, to predict future

problems, and so on. A representation of the history, where the subjects of change are code entities instead of just files, can help integrators to better understand the changes and their potential impact.

Change Dependency. A specific change can require several other prior changes. This is more relevant in the case when changes come from different branches or forks. For example, if class C subclasses class B, then class C *depends on* class B. If the branch in which class C is intended to be integrated does not contain class B, then the change adding class C must be integrated with its dependency (*i.e.,* change adding class B). To support the integration of changes from one branch into another – *cherry picking* – it is fundamental to establish dependencies between changes. Such dependencies can be used to characterize changes and deltas within the sequence, and partition changes that should be integrated. The characterization of deltas can guide integrators to prioritize changes, for example deltas that do not depend on any prior change can be tagged as the easy ones. Therefore the integrator could first concentrate on the complex cases, or vice-versa. Moreover, by means of dependencies it can be possible to establish which entities have been changing together. This can be key in spotting problems with a change, and help integrators assessing these changes.

6.5 Classification of Top 9 Questions

Making use of the taxonomy of information described above in this section, we present in Table 2 a classification of the top 9 questions presented in Section 5.4.

Descriptive Information	
Author/Owner	S_{10}, T_7
Time	S_5, T_6, S_{10}
Structural Information	
Structure	S_5, S_{10}, T_{16}, B_{10}
Change Scope	B_{10}, S_{10}
Kind of Actions	S_5, S_{10}, T_6, T_7, T_8, B_{10}
Kind of Entities	B_{10}, B_{11}, S_5, S_6, S_{10}, T_6, T_7, T_8, T_{16}
Semantic Information	
Vocabulary	B_{11}, S_6, S_{10}
Reason	S_{10}
Historical Information	
History	S_5, S_{10}, T_6, T_7, T_8, T_{16}
Change Dependencies	S_5, S_{10}, T_6, T_7

Table 2 – Classification of the Top 9 questions according to the information that tools need to answer such questions.

7 Conclusion

Git's strong capabilities to support and encourage branching as well as a the expansion of Git as a distributed VCS stress the task of integrators. Based on a large literature analysis and an open call, in this article we presented a catalog of 46 questions that integrators may need to answer when they integrate changes. We then performed a validation of these questions by asking developers to assess the importance and tool support for each of the questions in the catalog. We also discussed a subset of 9 questions that are both important and have no

tool support. Finally, we presented a taxonomy of the questions according to the kind of information that tools need to answer them.

We hope that other researchers will conduct alternate surveys based on our catalog and develop new generation tools to help integrating changes.

Acknowledgments We thank Nicolas Anquetil for his help in the survey.

References

[AHM+09] Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the evolution of the source code vocabulary. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR'09, pages 189–198. IEEE Computer Society, 2009. doi:10.1109/CSMR.2009.61.

[AL98] Nicolas Anquetil and Timothy C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON'98, pages 213–222. IBM Press, 1998. URL: http://portal.acm.org/citation.cfm?id=783160.783164.

[BBBL15] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 134–144, Piscataway, NJ, USA, 2015. IEEE Press. doi:10.1109/ICSE.2015.35.

[BNM+11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE'11, pages 4–14. ACM, 2011. doi:10.1145/2025113.2025119.

[BZ12] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11, New York, NY, USA, 2012. ACM. doi:10.1145/2393596.2393648.

[CRR05] Ophelia C. Chesley, Xiaoxia Ren, and Barbara G. Ryder. Crisp: A debugging tool for Java programs. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM'05, pages 401–410, 2005. doi:10.1109/ICSM.2005.37.

[DBG+15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *SANER'15: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, Montreal, Canada, 2015. (candidate for IEEE Research Best Paper Award). doi:10.1109/SANER.2015.7081844.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. URL: http://www.iam.unibe.ch/~scg/OORP.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, ICSM'06, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society. doi:10.1109/ICSM.2006.22.

[DUGCD14] Martin Dias, Verónica Uquillas-Gomez, Damien Cassou, and Stéphane Ducasse. Software integration questions: A quantitative survey. Technical report, INRIA Lille, 2014. URL: https://hal.inria.fr/hal-01093496.

[FM10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE'10, pages 175–184. ACM, 2010. doi:10.1145/1806799.1806828.

[GDMR04] Tudor Gîrba, Stéphane Ducasse, Radu Marinescu, and Daniel Rațiu. Identifying entities that change together. In *Ninth IEEE Workshop on Empirical Studies of Software Maintenance*, 2004. URL: http://scg.unibe.ch/archive/papers/Girb04dEntitiesChangeTogether.pdf.

[GHR09] Daniel M. German, Ahmed E. Hassan, and Gregorio Robles. Change impact graphs: Determining the impact of prior code changes. *Journal of Information Software Technology*, 51(10):1394–1408, October 2009. doi:10.1016/j.infsof.2009.04.018.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–23. IEEE Computer Society, 2003. doi:10.1109/IWPSE.2003.1231205.

[GZSvD15] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *ICSE'15: Proceedings of the 37th International Conference on Software Engineering*, pages 358–368, 2015. doi:10.1109/ICSE.2015.55.

[HZ11] Kim Herzig and Andreas Zeller. Mining cause-effect-chains from version histories. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering*, ISSRE'11, pages 60–69. IEEE, 2011. doi:10.1109/ISSRE.2011.16.

[HZ13] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press. doi:10.1109/MSR.2013.6624018.

[KDG05] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press. doi:10.1109/WCRE.2005.16.

[LM10] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU 10, pages 8:1–8:6. ACM, 2010. doi:10.1145/1937117.1937125.

[PRS12] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. Information needs for integration decisions in the release process of large-scale parallel development. In *Proceedings of the ACM 2012 conference on Computer*

Supported Cooperative Work, CSCW'12, pages 1371–1380. ACM, 2012. doi:10.1145/2145204.2145408.

[PSW11] Shaun Phillips, Jonathan Sillito, and Rob Walker. Branching and merging: an investigation into current version control practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE'11, pages 9–15. ACM, 2011. doi:10.1145/1984642.1984645.

[PTL+11] Rahul Premraj, Antony Tang, Nico Linssen, Hub Geraats, and Hans van Vliet. To branch or not to branch? In *Proceedings of the 2011 International Conference on Software and Systems Process*, ICSSP'11, pages 81–90. ACM, 2011. doi:10.1145/1987875.1987890.

[SMDV06] J. Sillito, G.C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th International Symposium on Foundations on Software Engineering*, SIGSOFT '06/FSE-14, pages 23–34. ACM, 2006. doi:10.1145/1181775.1181779.

[SMDV08] J. Sillito, G.C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, jul 2008. doi:10.1109/TSE.2008.26.

[SVFM05] Jonathan Sillito, Kris De Volder, Brian Fisher, and Gail Murphy. Managing software change tasks: An exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 23–32. IEEE Computer Society, 2005. doi:10.1109/ISESE.2005.1541811.

[TGM96] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Languages*, 4(3):143–167, 1996. URL: http://dblp.uni-trier.de/rec/bibtex/journals/jpl/TakangGM96.

[UG12] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, October 2012. URL: http://rmod.lille.inria.fr/archives/phd/PhD-2012-Uquillas-Gomez.pdf.

[UGDD10] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Visually supporting source code changes integration: the Torch dashboard. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pages 55–64, October 2010. doi:10.1109/WCRE.2010.15.

[UGDK14] Verónica Uquillas Gómez, Stéphane Ducasse, and Andy Kellens. Supporting streams of changes during branch integration. *Journal of Science of Computer Programming*, 2014. doi:10.1016/j.scico.2014.07.012.

[YCM78] Stephen S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 60–65. IEEE Press, nov 1978.