Supporting Developers' Coordination in The IDE

Anja Guzzi[*], **Alberto Bacchelli**
Delft University of Technology
Delft, The Netherlands
{a.guzzi, a.bacchelli}@tudelft.nl

Yann Riche
Microsoft
Redmond, WA, USA
yannr@microsoft.com

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
{arie.vanDeursen}@tudelft.nl

ABSTRACT

Teamwork in software engineering is time-consuming and problematic. In this paper, we explore how to better support developers' collaboration in teamwork, focusing on the software implementation phase happening in the integrated development environment (IDE). Conducting a qualitative investigation, we learn that developers' teamwork needs mostly regard coordination, rather than concurrent work on the same (sub)task, and that developers successfully deal with scenarios considered problematic in literature, but they have problems dealing with breaking changes made by peers on the same project. We derive implications and recommendations. Based on one of the latter, we analyze the current IDE support for receiving code changes, finding that historical information is neither visible nor easily accessible. Consequently, we devise and qualitatively evaluate BELLEVUE, the design of an IDE extension to make received changes always visible and code history accessible in the editor.

Author Keywords

Developers' coordination; IDE extension; qualitative study.

ACM Classification Keywords

D.2.6 Software Engineering: Programming Environments

INTRODUCTION

Software engineering is often a team effort. It is not unusual for hundreds of professionals collaborate to design, build, evaluate, and maintain software systems [71]. However, teamwork remains one of the most difficult and pervasive problems of software engineering, and developers face a plethora of teamwork problems at different levels [18].

Key to this teamwork are tools and processes that revolve around it, source code management, and development. Most of developers' time is spent within the Integrated Development Environments (IDE) [48], thus researchers are trying to leverage them by augmenting their collaborative capabilities (*e.g.,* [17, 27, 32, 33]). Nevertheless, the IDE remains a tool that primarily helps individual programmers to be more effective in the classical edit-compile-run cycle [73].

In this paper, we explore how to better support collaboration in teamwork within the IDE. Our research is set up in two phases: An exploratory investigation, followed by the design and evaluation of a medium fidelity click-through prototype.

In our investigation, we explored how developers in large development teams experience collaboration and identify problems they face when working in team. To this end, we (1) conducted a brainstorming session with 8 industry experts working on the development of IDE solutions at Microsoft; (2) identified three core opportunities for the design of improved collaborative solutions, which we refined through semi-structured interviews with developers; and (3) interviewed 11 professional developers with various degrees of experience and seniority, from 9 different companies to contextualize those opportunities.

In our investigation, we report how, while our participants reported collaborating with others on code, they spend a limited amount of time actively engaged in collaboration. As a consequence, one of the core issue emerging revolves around managing dependencies between activities, rather than working together contemporarily on the same (sub)task. Our interviews with developers also confirm that issues arise due to the lack of information (*i.e.,* imperfect information) when working on shared code, and uncover existing strategies to prevent or resolve them. Dependencies are often mediated through code, in the form of code changes. Yet, our investigation illustrates how dealing with code changes remains a common source of issues despite existing tools and strategies when working on the same project. This emerges as one of the main causes of frustration in interviewees' experience of teamwork. From our findings we derive implications and recommendations for improving coordination in modern IDEs.

In the second phase of this study, we focus on an investigation of how to improve handling teams code changes from within the IDE. Using common usability heuristics [56], we describe opportunities to better support teamwork in the IDE by supporting information needs about change history. Consequently, we leverage this analysis to: (1) derive requirements for an IDE extension, (2) describe the design of BELLEVUE, a prototype fulfilling these requirements, and (3) iteratively evaluate a design called BELLEVUE with 10 senior developers from different companies and backgrounds.

BELLEVUE makes incoming code changes always visible during development, eases the use of that history in the context of the developer's tasks and flows. It shows historical

[*]Anja Guzzi was an intern with the User Experience Team, Microsoft Developer Division, Microsoft, Redmond, USA in the summer of 2012 when this work was carried out.

This paper is a pre-print of: Anja Guzzi, Alberto Bacchelli, Yann Riche, Arie van Deursen. Supporting Developers Coordination in The IDE. In Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work, *CSCW 2015*, March 14–18, 2015, Vancouver, BC, Canada.
Copyright 2014, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved.
No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

information within the active code editor to let users modify the code without context switch. To achieve this, BELLEVUE takes already available historical change data and offers an interactive view that shows detailed historical information for files and code chunks with respect to a previous version.

EXPLORATORY INVESTIGATION: METHODOLOGY

In this section, we define the scope of our research, and our research methodologies (illustrated in Figure 1), divided into the following main steps: brainstorming, semi-structured interviews, and data analysis with card sorting.

Scoping

We scoped our initial focus by tapping in the collective knowledge of eight industry experts engaged in the design and implementation of development tools (including one of the co-authors, and the first author as researcher intern). To do so, we organized a brainstorming on the challenges and opportunities revolving around team development. The brainstorming led to the identification of the following areas for further investigation: *awareness* (*i.e.,* knowing other people's work to anticipate issues and to proactively seek out information when needed), *work dependencies* (*i.e.,* anticipating and understanding who I depend on and who depends on me), and *breaking changes* (*i.e.,* anticipating what and whom will be impacted by the breaking change I am about to release). As we will explore in more depth later, those areas describe times where lack of information can lead to resource- and time-consuming situations. Such situations are common not only in development situations, but in teamwork in general where situation of *imperfect information* is the norm [38].

Consistently, literature reported that developers often have difficulties facing questions such as: "*What have my coworkers been doing?*" "*How have resources I depend on changed?*" "*What information was relevant to my task?*" "*What code caused this program state?*" [7, 26, 45, 68] In our work, we iterate on this by providing a more in-depth view of some of those specific problems, and by illustrating and validating ways of addressing them within the flow of activities developers engage in.

Semi-structured Interviews

To gather more details about the context in which those issues emerge, and current strategies for addressing them, we conducted *semi-structured* interviews [50] with professional developers. In total, we interviewed 11 developers from a varying in experience, team sizes, and companies. Table 1 summarizes interviewees' characteristics.

We conducted each 90-min interview on the phone, and transcribed the interviews for latter analysis. After each interview, we analyzed the transcript and split it into smaller *coherent units* (*i.e.,* blocks expressing a single concept), which we summarized by using an interview quotation or an abstractive sentence. In addition, the interviewer kept notes (*i.e., memos*) of relevant and recurring observations in a document iteratively refined and updated. Out of the interviews, 56 memos emerged.

ID	Overall experience	In current team Time	In current team Role	Team Size
D1	7.5 years	4.5 months	dev	4
D2	10 years	6 years	dev lead	7
D3	2 months	2 months	junior dev	4
D4	1.5 years	1.5 years	sql dev	5
D5	20 years	10 years	senior dev	4
D6	25+ years	7 years	senior dev	15
D7	21 years	2 weeks	senior dev	10
D8	25+ years	10 years	senior dev	16
D9	1 years	1 years	dev	5
D10	15 years	5 years	dev lead	5
D11	20 years	6 years	senior dev	11

Table 1. Interviewed developers

To analyze our data, we created the 562 cards from the transcribed coherent units and the memos. Each card included: the source (transcript/memo), the interviewee's name (if from the transcript), the unit/memo content, and an ID for later reference. We used *card sorting* [69] to extract salient themes, leveraging a combination of open and closed card sorts. After macro categories were discovered, we re-analyzed their cards to obtain a finer-grained categorization. Finally, we organized the categories using *affinity diagramming* [52].

EXPLORATORY INVESTIGATION: DATA

In this section, we present the results of our exploratory investigation. When quoting raw data, we refer to the individual cards using a [D*X.Y*] notation, where *X* represents the developer, and *Y* (optional) the card (*e.g.,* [D2.03] refers to card 03 from the interview with D2).

Teamwork from the developers' perspectives

According to the interviewees, collaboration in teamwork is defined by a wide spectrum of activities:

Communication - Collaboration is communication: As D8 said: "*It is all about communication. If you have good communication, you have good collaboration.*" Communication can be both one-to-one and one-to-many, can be formal or informal, and goes through different channels. Channels are "conventional," such as email and instant messaging (IM), but also more domain specific ones, such as interactions via project management tools (*e.g.,* source code management systems and requirement tracking systems) and source code; as D4 explains: "[to communicate] *typically we use* [IM]*, but we also have an internal wiki that we use*". [D4.02, D7.09, D8.(01,02)]

Helping each other by sharing information - As D9 said: "*Collaboration is just sharing information and ideas.*" In particular, according to interviewees, collaboration means being proactive and sharing useful information to make it easier for others to complete their work (*e.g.,* "*make* [the acquisition of information] *as easy as possible on the other co-workers, so that they don't have to struggle*" [D7]). This aspect of collaboration involves voluntarily sending notifications (*e.g.,* "*FYI–for your information*" messages) and starting discussions (*e.g.,* "*let's coordinate on this change*

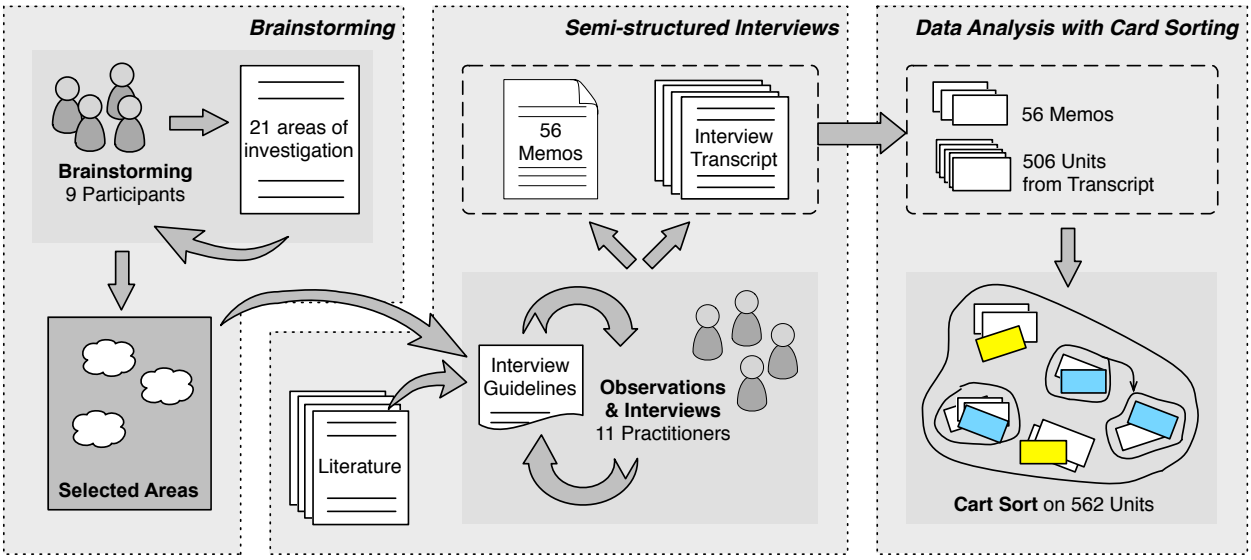

Figure 1. The research method applied in the first phase

I need to make") [D7]. Resource sharing involves not only knowledge on the actual source code of the project, but also information from external resources, for example about new technologies or coding style; as D9 stated: "*We also like send each other things, like, style tips and like interesting articles about how other companies do things*". [D2.02, D7.(02,03,06), D9.(01,04,05)]

Knowing what others know - Collaboration, from the interviewees' perspective, also means to stay aware of the experts on the different parts of the system (*i.e.,* "*the domain experts*") and to understand artifacts and code changes done by colleagues. D11 explains: "[collaboration] *is keeping track of what everybody is working on and being able to know how the different pieces are in place*". According to interviewees, knowing what the others know has the aim both of preventing problems and of reacting faster when they occur. [D2.01, D4.03, D7.04–07, D11.47]

Working on the same goal, doing different things - Overall, developers see collaboration as working toward the same goal (*e.g.,* product releases), by acting in parallel on different parts of the same project (*e.g.,* working separately on different code artifacts): "*Collaboration is we are all working towards the common goal, but maybe working on different parts of it, but these components do interact*" [D7]; "[Collaborating meant] *we divided up the work* [...]*, we went off these directions, and as things started to merge together, we go on* [merging] *on a case by case base.*" [D3]. [D1.2, D3.01, D4.01, D7.01, D9.(02,06)]

Dealing with imperfect information in teamwork

To investigate how developers deal with imperfect information, we outlined three concrete teamwork scenarios in which the existence of imperfect information can generate problems. The scenarios were derived from teamwork situations commonly described as problematic in literature.

Inefficient Task Assignment

Scenario. One developer is assigned to a task, while another is already working on a similar or related task. This introduces inefficiencies in the team (and potential collisions).

Related literature. Software engineering researchers have recognized task partition and allocation as important endeavors in the context of teamwork [44, 46]. If dependencies and common traits among tasks are not correctly handled, developers can reduce their efficiency and generate unexpected conflicts [37]. Literature suggests different techniques, with varying results, for efficient task assignment (*e.g.,* [16, 24]). In particular, the assignment of bug fixes (or new features to implement), from a repository of issues or requests to the most appropriate developers, is one of the main instances of task assignment investigated by researchers [1]. Researchers reported that bug triaging is a time consuming task, which requires non-trivial information about the system, and that often leads to erroneous choices [2]. A number of techniques has been devised to tackle the triaging problem, *e.g.,* [53, 41].

Interviews' outcome. Although considered realistic, our participants did not see this scenario as a core issue. In fact, the task assignment processes of teams are in general considered effective to prevent the problematic scenario to take place. In some teams supervising figures (*e.g.,* managers) do the task assignment (*e.g.,* "[a new task] *goes to a manager, who decides whom to assign*" [D8], and "*the boss will tell you about a task* [to do]" [D9]); in the other teams, tasks are divided during team meetings, with various degrees of developers' interaction (*e.g.,* "*we are using daily SCRUM meetings*" [D1], and "*we break up the code, and if the* [task] *is in your code, it's yours*" [D5]).

Simultaneous Conflicting Changes

Scenario. Developers find themselves in a situation where there is a merge conflict (*i.e.,* when different people are touching the code at the same time).

Related literature. A recent significant effort in the software engineering research community is devoted to detect concurrent modifications to software artifacts (*e.g.,* [35, 36, 59, 65]). In fact, developers making inconsistent changes to the same part of the code can cause merge conflicts when changes are committed to the code repository, which leads to wasted developers' efforts and project delays [13, 42, 66]. Grinter conducted one of the first field studies that investigated developers' coordination strategies [29]. She observed that it is sometimes difficult for developers (even for expert ones) to merge conflicting code without communicating with the other person who worked on a module. Later, de Souza *et al.*, in an observation of the coordination practices of a NASA software team, observed that developers in some cases think individually trying to avoid merging, while in others think collectively by holding check-ins and explaining their changes to their team members [21].

Interviews' outcome. Our participants reported only rarely encountering a situation where more than one person was working on the same file at the same time ("*we don't run into those situations a lot*" [D2]). Most of our participants' teams were organized to make developers work on different parts of the system, with a key person in charge of coordinating development to prevent those issues, typically a lead developer or an architect. Some participants also used technical solutions to avoid concurrent edits (*e.g.,* "*We put a lock on* [the file]*, so it does not get edited* [by others]" [D1]). When a merge conflict happens, our participants reported resolving it quickly and easily (*e.g.,* "*The best and quickest solution you have is to undo, we roll back and* [fix it]." [D1]; "*typically, it is solved really quick*" [D2]), and often using merging tools (*e.g.,* "*we don't have to do much manually*" [D8]). Although automatic merging was used, our participants also explained that they manually checked each conflict, revealing that it is not entirely trusted.

Breaking changes

Scenario. A developer/team changed some parts of the code, rendering the work of another developer or team unusable or obsolete.

Related literature. Researchers consider breaking changes problematic, not only for developers who receive the change and have to understand and adapt their code, but also for developers who are making a change that might break the functionalities of many clients [3]. Literature shows investigations (*e.g.,* [22]) on the effect of dependencies that break other people's work, and proposed methods to address subsequent problems, at the scale of both single system (*e.g.,* [10, 72]) and ecosystems (*e.g.,* [32, 61]).

Interviews' outcome. The reaction of our participants was different according to the *origin* of the breaking changes. When the origin was considered to be *external* to the team or company or when participants felt they had opportunity for neither intervening nor displaying their disappointment to 'the breaker', they accept the breaking changes with no strong negative emotions but rather as inevitable. This happened even when resolving the issue might take long time (*e.g.,* "*more than year*" [D2]) or when it resulted in large operational or maintenance costs (*e.g.,* "*this* [break] *was costing the company many thousands of dollars per minute.*" [D7]).

However, when the origin was *internal* to the company/team, participants reported strong negative emotions (*e.g.,* frustration). This seemed in part due to the mismatch between the expected communication that is made possible by being in the same company/team, and the "waste" of time spent finding the cause of the issue, which might in turn be resolved relatively quickly (*e.g.,* "*I spend a couple of hours to find out the error* [...] *fixed in 5 minutes.*" [D3] and "*I spent a day fixing the problem I spent three days finding.*" [D8]). Generally, breaking changes leading to syntactical errors were not considered an issue, because they could easily be spotted with static analysis tools (e.g. pre-compilers) and fixed. In effect, those particular breaks were considered as a direct consequence of the lack of coordination effort from the person introducing the breaking code [D1]. Some of our participants insisted that breaking changes when the origin of the break is internal to the team or company should be handled more smoothly and proactively. For example, some would prefer stricter rules to avoid internal breaking changes: "*people breaking other's people code* [...]*, I'd like to see management being more rigorous about it*" [D8].

Receiving a code change

Managing internal breaking changes is the most problematic scenario that emerged from our analysis, in this section, we analyze how developers deal with changes made by peer developers working on the same project.

Our participants reported that they investigated changes in the code-base when they were notified of them (*e.g.,* via automatic emails from versioning systems). However, they mostly did so to discover whether they had an impact on their own code. While they are not interested in building a holistic view of the whole code base, they use this approach to discover whether the changes will impact their own work. In doing so, they first need to assess how relevant the change is to their current or upcoming work to decide whether or not to investigate further. In some rare occasions, developers use this opportunity to explore other's work not as it impacts theirs, but from a style/aesthetics perspective, looking at coding styles, norms, approaches, and solutions, especially is changes are emitted by a respected colleague (for learning/inspiration) or a novice (for peer reviewing).

When our participants reported discovering an error caused by a change made by someone else, their most prominent complain regarded the lack of coordination that they felt should have accompanied the change (*e.g.,* they would have expected an "heads up" email). However, in the case of clear syntactic errors (*e.g.,* compilation errors, or runtime errors generating a stack trace), participants did not feel the kind of frustration they expressed in the case of semantic errors (*e.g.,* caused by a library that changes some sorting order, therefore impacting the dependent code) or unclear alteration in the behavior. In fact, semantic errors required participants to

perform a deeper and more time-consuming analysis to understand their cause [D3.(47,49), D8.(28,29,36)].

Once they found the cause, they explained they proceeded to measure the impact on their code by, for example, measuring how many files or lines of code needs to be changed (as D8 explained: "*I measure the impact of a change* [looking at] *how many files/lines it affects. A few? Hundreds?*"). Usually, the developers receiving the breaking change were those automatically responsible of adapting and fixing their own code. However, when a change had a deep impact on the code-base, and required more information about the change (*e.g.,* the rationale of the change) and the code-base, developers usually wanted to contact the author.

Participants also reported that, when the change introduced a defect, those receiving were responsible for deciding to file an issue report against the change to the change author (*e.g.,* D5 explained: "*if the bug is in your code, it's your bug to fix.* [...] *I send a bug request.*") [D3.(06,09,31), D5.06], or, occasionally, if the fix took little time, they would directly change the code to fix it (D3 said: "*If it's small I just fix it and notify the author*") [D3.59, D5.107, D8.38, D9.60]. The time spent fixing the problem in the code does not seem to bring frustration *per se* (*e.g.,* "*Diagnosing is almost always harder than to fix it. With the majority of bugs, once you know where the problem is, it's easy to fix*" [D8.29]). The rationale to fix it directly is that the developer already has the necessary information, which would require time to share with the author of the faulty code.

Some participants mentioned that the lack of testing contributes to faulty changes being committed to the repository (*e.g.,* "*we are really bad at testing* [...]*, you pull and you get a file you try to run and it fails*" [D9], "*If I'd tested it better, I wouldn't have put [this code] in the build*" [D5]). Nevertheless they also warned that running all the tests for each change would be too expensive ("*all tests, to run them all, it would take 3 weeks. Unfeasible to take 3 weeks for each check in*" [D6]). They also warned that testing performed on a setup might unreliable on a different one ("*we test and it's all good, but then they test on their end and it might break* [...]*. It's something to do with customizing.*" [D2]), and that many semantic changes could not be detected by tests ("*even if there are tests that check* [most] *things, you'd still end up with edge cases.* [...] *You still need to see that you break, and then react, and then fix it*" [D6]).

EXPLORATORY INVESTIGATION: INTERPRETATION

Teamwork Collaboration Is Coordination

The terminology used in many disciplines [51] defines coordination as "*managing dependencies between activities,*" and collaboration as "*peers working together.*" In this light, what participants consider as collaboration in teamwork is mostly *coordination*, needed to organize the individual work toward achieving a shared goal.

By analyzing the data from the interviews, coordination emerged—after individual work—as the dominant interaction level when working in team, rather than collaboration. In particular, our participants described that:

1. They spend most the time doing individual work;
2. Most of their interaction is to coordinate (*e.g.,* through daily stand-ups);
3. In their work, collaboration happens infrequently and on a need basis (*e.g.,* with (bi-)weekly sprint meetings).
4. Most of the time, their intention for collaboration is coordination leading to individual work.

By abstracting the explanations of interviewees, we model developers' interaction in three levels (from lowest to highest degree of interaction): individual work, coordination, and collaboration. Individual work corresponds to no interaction (*e.g.,* a developer working on her own), while collaboration means developers working together at the same time on the same (sub)task (*e.g.,* pair programming). Coordination is an intermediate level of interaction, where developers interact but are not working together on the same (sub)task (*e.g.,* during task assignment).

An activity is a working state that can be placed as a point on the corresponding level of interaction. In a set of activities done to accomplish a certain subtask (*i.e.,* 'working situation'), particular events often serve as a transition between levels of interaction, for example, steps from individual work to coordination (*e.g.,* "[when a file is locked] *we just* [ask]*: 'hey what are you working on? And then when you think I can do it?' to the author.*" [D2.10]), and from coordination to collaboration (*e.g.,* "*sometimes we* [...] *get together and talk about* [similar things]*, then realize how we can do these things together and do them together*" [D11.45]). Figure 2 depicts our model of developers' interactions.

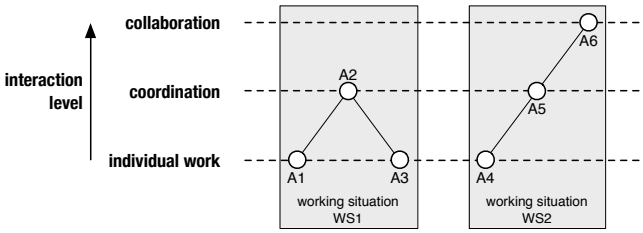

Figure 2. Model of developers' interactions in working situations

Figure 2 shows two working situations: In the first (WS1), a developer doing individual work asks another developer to make a change in their code (*e.g.,* "*I asked one of the guys: '*[...] *I need a method that would return* [a special object]*, can you write* [it] *for me?' He was able to write* [it] *and knew exactly where to go*" [D3.(09,15)]). This requires going from individual work (A1) to coordination (A2) when asking to make a change to the other, and back to individual work (A3) when they reach an agreement, without reaching a state of collaboration. In the second situation (WS2), two developers decide to work together on a subtask. This requires moving from individual work (A4) to coordination (A5) when they decide, then to collaboration (A6) for the actual work.

The steps between the different levels of interaction in the model are not necessarily discrete: Intermediate interaction

levels can be reached. For example, while the activity of task assignment can generally be placed on the coordination level, when the task assignment is *discussed together in a meeting* it can be put on a level between coordination and collaboration.

Implications

Our participants report that most of their time is spent in doing individual work, while, unexpectedly, they report to spend very little time *collaborating* on the same subtask. A direct consequence is that interactions revolving around coordination are a more actionable area, with better research opportunities and with greater potential impact, than areas considering purely the collaboration aspects. For example, better support to communication would have more relevance than tools for concurrent real-time coding.

In addition, techniques for supporting information sharing among developers should take into account that developers spend most of their time doing individual work. Considering that most of this individual work is spent in the development environment (the IDE) [48], tools that support coordination within the IDE have potential to lead to greater impact.

The role of information.

In our study, we uncovered how available information was pivotal in transitioning between levels of interaction (Figure 3). This happened when our participants acted on information they had already acquired earlier, reacted to incoming information, or sought out to gather new information, for example, through communication or by understanding code changes done by colleagues.

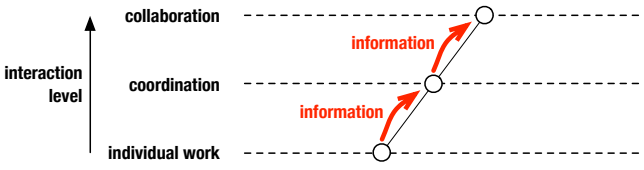

Figure 3. The role of information in developers' interaction

Researchers have been studying the importance of information sharing for teamwork over the years from different angles. For example, Cataldo *et al.* introduced the notion of *socio-technical congruence*, and provided evidence that developers who share information when making changes in dependent software components complete tasks in less time [15]. Other researchers similarly showed that missing information correlates with coordination problems and project failures [14, 63, 47]. Ko *et al.* analyzed the information needs of developers in collocated teams and found they have difficulties in answering questions related to information about colleagues' work and software component dependencies [45].

From our interviews, developers reported to know how to deal with the investigated scenarios involving imperfect information, except when they received an internal breaking change. We suggest that this is connected to how easy it is to access the information they need to address the problem. Analyzing the ways developers/teams successfully deal with a condition of imperfect information, we see that the solutions to the problematic scenarios require information to be shared in two ways: (1) via direct communication (*e.g.,* during a meeting), and (2) by making it visible (*e.g.,* in a tool).

Scenario	Needed Information Communicated	Visible
Task assignment	✓	✓
Simultaneous changes	✗	✓
Breaking changes	✗	✗

Table 2. Information in investigated scenarios

Table 2 shows that for non-problematic scenarios, the needed information is communicated or visible. In the case of task assignment, the inefficiencies are avoided by centralizing the task assignment to the team leader, who has all the information "visible" in mind, or by conducting group meetings in which the information is communicated. Other researchers report evidence of this behavior: Begel *et al.* described that industrial program managers and developers have regular team meetings to effectively prioritize bugs and to coordinate component completion schedules [8]; and Guzzi *et al.* reported that when open source software developers meet in person, they manage to advance the coordination of their projects better [31]. In the case of simultaneous changes that were not avoided with the team policies (*i.e.,* through modularization and technical locks), the information necessary to solve the merge conflict is immediately visible through the merge tool. In their analysis of parallel development practices, de Souza and Redmiles similarly reported that issues were averted through the mediation of configuration management tools [22]. In the case of breaking changes, we suggest that the needed information is neither communicated in time nor easily accessible/visible. As a result, developers can spend a long time finding the information they need to coordinate. This is in agreement with other studies that report how breaking changes are due to missing information and lead to significant loss of time (*e.g.,* [61]).

Implications

Our analysis showed that the efforts spent in gaining the information developers are missing can be a source of negative emotions. This underlines the importance of information sharing practices, both active (*e.g.,* communicated) and passive (*e.g.,* visible via a tool).

Researchers proposed a number of tools (*e.g.,* Palantìr [65] and FASTDash [9]) to detect merge conflicts and tested them in laboratory settings with seeded conflicts. These tools helped developers to spend less time to resolve conflicts and encouraged communication. An interesting venue for future research is to verify the overall impact of these tools on teams whose structure maps the software architecture, as our participants reported not encountering this issue.

In addition, in most of our investigated scenarios, we observed that—unexpectedly—developers already had means to deal with missing information, or did not considered these scenarios as issues. In contrast, the results of the study by deSouza and Redmiles put in evidence the significant differences that two unrelated companies have when they deal with

the management of dependencies and the underlying information sharing [22]. This suggests that what is considered a critical issue for a company/project could not important for another. As a consequence, it is important, when investigating potential problems generated by lack of information, to first study whether and how the target developers already employ any method to supply this missing information.

Changes and dependencies

As de Souza and Redmiles explained: "it is not possible to study changes in software systems without studying dependencies" [22]. In this light, our analysis of coordination and receiving changes is related to the work by Begel *et al.* [8] and by de Souza and Redmiles [22].

Begel *et al.* conducted a survey of Microsoft developers, testers, and program managers to see how these coordinate on dependencies (*e.g.,* tasks) within the same team and among different teams. The study reported that most Microsoft developers (63%) minimize code dependencies to mitigate problems. This is similar to our interviewees who use software modularity to avoid inefficient task assignment or merge conflicts. Similarly to our findings, Begel *et al.* also reported that lack of communication often led to coordination problems, and that email is the common means by which developers kept track dependencies. In contrast, our study outlines the difference between internal and external dependencies and changes. Begel *et al.* found that internal dependencies are managed by "send[ing] an email and pay[ing] a personal visit to the person blocking their work," [8], and surveyed developers do not report any negative emotion. Our findings underlined that, in the case of internal breaking changes, the process preceding the communication with the person blocking the work (*i.e.,* finding the source of the problem) is cause of dissatisfaction and frustration in cases where the expected communication did not take place. Moreover, the two studies present different definitions of *external* dependencies and breaking changes: (1) According to Begel *et al.*, dependencies are 'external' if in different teams within the same company, with which it is possible to communicate personally; (2) according to our findings, dependencies are 'external' if in different companies, with which it is extremely difficult to communicate. In the former case, Begel *et al.* reported that developers have to maintain personal communication with external teams to remain updated of changes, and the existence of unexpected changes from external teams generates anxiety. In the latter case, our interviewed developers did not report anxiety (even though unexpected changes happen and lead to loss of time), rather acceptance of the situation as part of the natural business model of the industry.

In their work, de Souza and Redmiles investigated the strategies software developers use to handle the effect of software dependencies and changes in two industrial software development teams [22]. The two teams deal with *internal* dependencies according to our definition. One team (MVP) allows parallel development and the modularity of the system is low, the other team (MCW) focuses on modularity by using a reference architecture. Our interviewed developers have complains similar to those in the MCW team, which also has strikingly similar practices: In both studies these teams avoid inefficient task assignment with modularity, their developers have problems identifying their impact network (they do not know who is consuming their code or whether changes can modify the component they depend on) and are only interested in changes in particular parts of the architecture that impact them. Moreover, developers in both MCW and our study have expectation that major changes are accompanied by notifications about their implications, yet are worried about information overload resulting from too frequent notifications. Conversely, the MVP practices seems to align with our participants' description of an ideal scenario where emails are sent to update about changes, everybody reads notification emails, management urges developers to notify about breaking changes, and such email even suggest courses of action to be taken to minimize the impact. As a result, despite the parallel development, coordination in MVP seems smoother than in our participants' experiences. One important characteristic of MVP, mentioned by de Souza and Redmiles, is that most developers have worked on the project for at least two years, and their experience could also be the cause of the difference with MWC, which is a newer project. Our results, though, do not seem to corroborate this hypothesis, since interviewed developers reported similar issues regardless of project maturity and personal experience. Our additional analysis of code changes looks at coordination from a low level perspective; we found that most information developers need to coordinate is typically available, but not necessarily accessible.

Implications

Our study confirms that lack of coordination leads to late discovery of unexpected errors or behaviors in the code, followed by a time-consuming analysis to find the code changes that are the source of the issue. This calls for better support for coordination when developers make and receive changes, and for when they need to investigate a change to determine its impact. As the existing body of research suggests, impact analysis and support for change understanding in software development scenario remains problematic. Research prototypes have not yet reached widespread usage in the IDE, and our findings underlines the substantial practical relevance of further research in these areas.

The differences between coordination practices between our interviewees' teams and the MVP team described by de Souza and Redmill [22] are an interesting venue for future research. In fact, compelling is the hypothesis that the modularity adopted by our interviewees' teams and MWC could create asymmetries in engineers' perceptions of dependencies [30], thus being at the basis of the differences and generating the reported coordination issues.

By investigating how developers currently handle received code changes in the IDE, we realized that they do many tasks manually, and spend a lot of effort to collect and remember change information. The data that would help developers in their tasks is available (*e.g.,* data recorded by the versioning systems), but not easily accessible. This implies that better support for integrating change information in the IDE is needed and it would impact development and coordination.

DESIGNING AND EVALUATING BELLEVUE

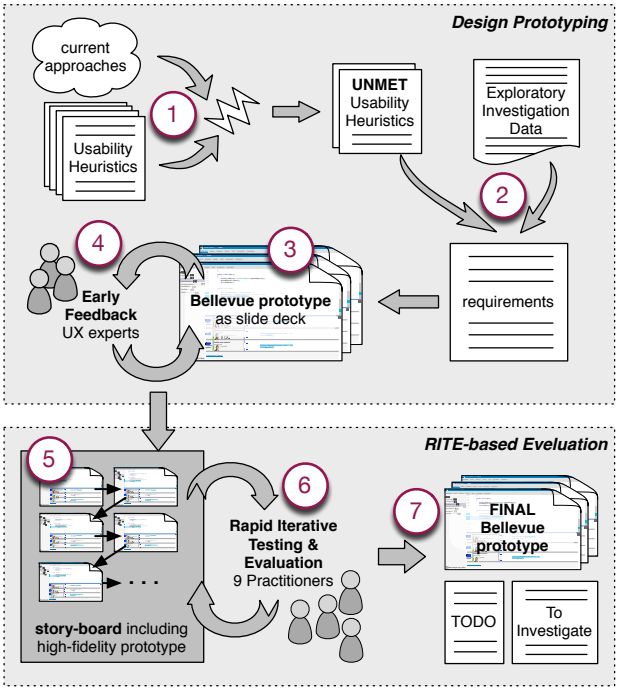

Figure 4. The research method applied in the second phase

Building upon our findings from our interviews, we aimed to design a tool to help developers anticipate, investigate, and react to breaking changes in a collaborative development environment. Figure 4 outlines the process.

Design requirements

We first analyzed the current approaches for receiving changes in the IDE under the light of widespread usability heuristics [56] (Point 1 in Figure 4). We found several unmet heuristics that, together with the data collected in the exploratory investigation, we used as a basis to derive requirements for our IDE extension to improve receiving changes and support teamwork (Point 2).

Recognition over recall

"Memory for recognizing things is better than memory for recalling things" [49].

Once a developer decides to merge a received change with the local version, the information about the integrated change disappears. For this reason, when developers encounter a bug, they must recall which change occured and whether any of them could have generated the problem. One participant explained that the frustration when he encounters a bug comes from "*figuring out where the problem is: Trying to figure out what really has changed*" [D5]. We suggest that when looking for the cause of a bug, developers' memory can be aided by tools to navigate change history, but existing tools require to switch from the current development context, and typically give the information outside of the development context.

Visibility of system status

"The system should always keep users informed about what is going on" [56].

Once changes are integrated, development tools provide no distinction between code already present before merge, and the newly integrated one. Therefore, there is no clear visibility of the system status with respect to its history. "*It's kind of impossible to know every single line of code that everybody else on your team changed*" [D3]. While historical information is available, it typically resides in dedicated tools or views, out of the current development context, thus the status is neither self-evident nor easily accessible: "*there isn't really an easy method [...] that let you see [that] these ten files are different from what you had in your current time*" [D5].

Clearly Marked Exits

"A system should never capture users in situations that have no visible escape" [55].

In software engineering, code repositories typically provide change history which gives developers an escape: If they find something not working after they merged some changes into their local working copy, they can roll back to the status prior to the merging. Problems with this approach are: (1) The exits are not evident, and (2) the exit strategy is binary.

The first issues means developers sometimes do not realize that their problems could be addressed by undoing the merge, instead of trying to find an error in their own code. The second issue means that developers can only undo *all* the merged changes at once, although the error can be caused by a mistake in a small fraction of the changed code. Once the code is rolled back, developers have to reconsider all the undone changes and realize which ones could have caused the error, without having the full IDE context at disposal, but only the change information, and integrate all the unrelated changes back again. D1 explained: "*It's a loss of time, we have to roll back, figure out* [what the problem was]*, and roll again. It's a loss of time, definitely.*"

Help and Documentation

"*[Documentation should]* be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large" [56].

In development processes, documentation also consists in the explanations software developers write as they commit their changes to the shared repository. It also includes other sources of information, such as descriptions of work items or bugs, stored in bug management or work item management tools. These pieces of information are accessible to the developer, and the commit messages are available to inspect upon receiving code changes, but once the changes are integrated they disappear, unless the user performs a number of steps navigating the history of the code in specialized windows or applications. Additionally, comments in the code commit and the work management tools are often disjointed. For example D5 complained: "*When you get the latest* [changed files] *you get tons of files*"; he found it very difficult to search the necessary help or information due to information overload. Finally, when developers integrate more than one commit into their local copy, often they see only the last commit message, even though a line of code could have been changed several times between their local copy and the current status.

Help users recognize, diagnose, and recover from errors
Current code change management in IDEs make it difficult to recognize and diagnose errors generated by integrated code changes, because they are not visible and the history has to be analyzed outside of the current development context. One interviewed developer explained that, despite the availability of external history tools, "*one of the problems is trying to figure out what really has changed [and] what's the impact on your code*" [D5]. In fact, as D3 explained, external tools are not helpful because "*version control gives you a list of files that changed and not the specific lines*": Seeing exactly which part changed and how takes many steps. Moreover, the only possibility to recover from errors is to do a complete undo of the merged changes, while it might be be enough to modify a small part of code to fix the error.

System design requirements
To address our current concerns with imperfect or missing information in development tasks, we suggest the following requirements for development tools:

(1) Received code changes should always be visible, (2) Information should be provided in context, both semantic (code) and procedural (history, project) without undue actions by the user, both at the project and file level (3) History of code chunks should be easily accessible, possibly using progressive disclosure to prevent information fatigue (4) Error identification ad diagnostics should be supported through a fluid integration of code history, (5) Code changes should be reversible at the sub-file level, and (6) Requiring context switches to acquire the necessary knowledge to solve a task should be avoided.

Prototype and evaluation

Consequently, we devised an IDE extension, named BELLEVUE, to fulfill the requirement outlined above and to serve as a tool to explore our preliminary design ideas (Point 3 in Figure 4). The prototype allowed us to communicate our ideas to various experienced designers and practitioners at Microsoft, and to get their feedback, reveal early problems, and improve the initial concept (Point 4 in Figure 4).

We devised a detailed storyboard including a high-fidelity prototype of BELLEVUE (Point 5). This was implemented as a PowerPoint presentation with a sequence of believable action steps of interaction with the prototype. Each step was devised to let the participants of the evaluation phase observe what was happening, explain what they would do, and describe the effects they would expect as a consequence of their actions. We used this prototype to evaluate BELLEVUE with professional software developers, using the RITE (Rapid Iterative Testing & Evaluation) method [54], to evaluate and identify problems in the prototype, quickly fix them, and then empirically verify the efficacy of the fixes (Point 6).

Participants in the RITE study were selected among a population with the following characteristics: More than three years as a professional developer, more than one year in the current company, and more than three months in the current team. Moreover, interviewees had to spend at least 20 hours per week on writing and editing code, their team had to use a source control management system, and they had to have at least browsed the change history, encountered a change merge, or used the file diff comparison view in the month before the RITE. Evaluation invitees were thanked for their participation by a gratuity in the form of Microsoft software.

Each session occured in a full usability lab on the Microsoft campus, and was video recorded for later analysis. To mitigate the *moderator acceptance bias* [28], we explained that the researcher guiding the session did *not* created the product. Moreover, to mitigate any *social desirability bias* [28], and to encourage discussion, the storyboard plot was describing the actions taken by a proxy developer named James. Following the storyboard plot described by the slides and the researcher, participants were solicited to follow a think-aloud protocol, and indicate what they saw, would do, and would expect as a result of their actions on each screen page.

After 9 iterations we reached a stable and validated design. At the end of the process (Point 7 in Figure 4), we had: (1) the finalized BELLEVUE prototype, (2) a set of changes to implement but that were not eventually integrated, and (3) a set of candidate aspects to be investigated as future work.

We designed BELLEVUE as a prototype code editing and navigation solution aimed at being a lightweight, ready to be used, without requiring developers to change their working style. It takes the historical change information that is already available, but currently neither visible nor easily accessible, and displays it in a non-obtrusive way. BELLEVUE offers an interactive view that shows detailed historical information for files and specific chucks with respect to a previous version. We detail the features of BELLEVUE, as they were at the end of the RITE phase, and the feedback from participants (mentioned as R1–9). The final version of the slide-deck used in the RITE is available as a file accompanying this paper.[1]

Recognizable changed files and blocks

BELLEVUE decorates changed files with an arrow (Figure 5, Point 1), and denotes changed lines with a blue[2] colored sign, both at a fine-grained granularity (Point 2), to see them in the context of the current text window, and a more coarse-grained one (Point 3), to see them in the context of the entire file. One can decide (Point 4) to see only the files that were just merged into the current local version. This design supports recognition over recall: Once new changes are merged into the local version, their traces remain visible. It also enhances the visibility of the system status, with respect to changes.

RITE participants' feedback—All the participants appreciated this feature. In particular, they liked that it helps filtering out irrelevant information when looking for the reason of an error that could have been introduced by a received change: "*Knowing what I can ignore is huge, the larger the project, the more beneficial it comes*" [R1]. Concerning the way in which changes are made recognizable, some users did not find it intuitive, or appropriate: "*I'd prefer a bar or something*

[1] Also available at: `http://www.st.ewi.tudelft.nl/~guzzi/`
[2] This color has been chosen because it is currently considered a neutral color in the IDE. As opposed to green or red, which are often associated to versioning systems or debuggers.

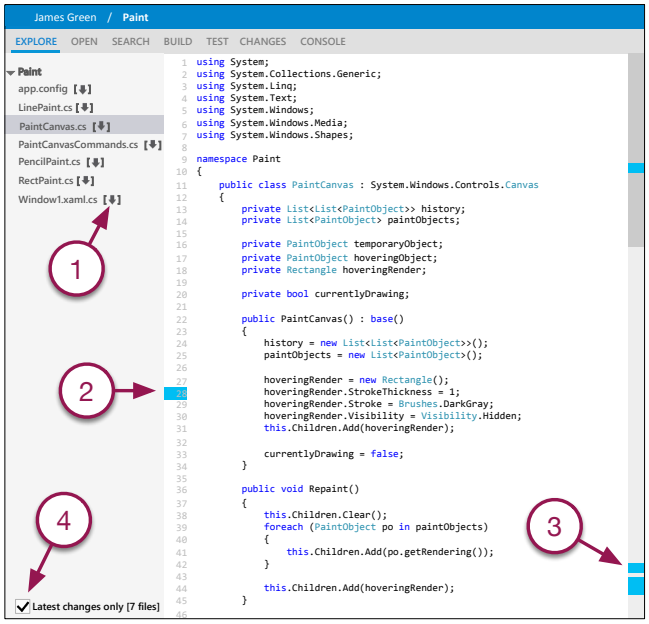

Figure 5. Recognizable changed files and blocks, and filtering

much more visibile [than a blue-colored sign] *to see that it's different*" [R2]. Nevertheless, after they continued in the scenario and experienced the following features of BELLEVUE, they withdrew their concerns. Some participants suggested to let the users personalize the color to denote changes; other participants suggested to use different colors to clearly distinguish added, removed, or modified lines, as it currently happens in tools that display code differences.

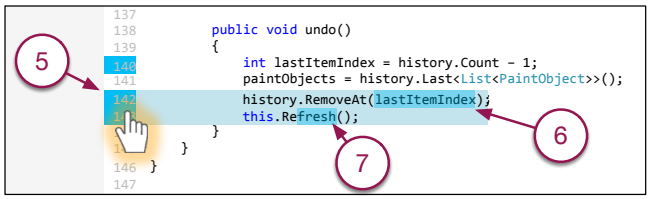

Figure 6. Visibility of changes' effect by mouse hovering

Visible changes' effect

To show the effect of the change in the code, the user can hover on any colored block to see the latest changes. For example, in Figure 6, the user decided to look at the changed block that was not visible in Figure 5. Then, by hovering on the colored sign on the left (Point 5), (s)he can see the effect of that change: The argument of the `RemoveAt` method call has changed (Point 6), and the `Refresh` method call has replaced a call present before on the same object (Point 7).

RITE participants' feedback—This feature was introduced in the third iteration of the tool, after considering the feedback received by the first participants. As an example, one participant had some expectations when hovering the lines indicating a change: "*toggle to highlight what's different from the last version, to quickly diagnose, I don't need a true side by side*" [R3]. Once introduced, this feature was well received by all the remaining participants (*e.g.,* "*ok, good! I can see here how* [this part] *changed!*" [R6]), because it also helps with the progressive disclosure of the information about the changes: Users can quickly verify whether the changes seem relevant and, only if necessary, investigate more.

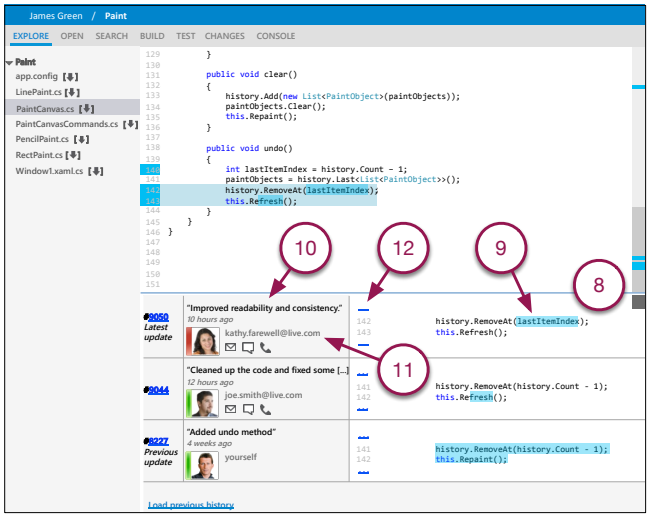

Figure 7. Accessible historical details

Accessible historical details

In BELLEVUE the user can see the code history of any block that was changed with respect to the previous local version. This is achieved with one click on the colored sign on the left of the interesting block. For example, in Figure 7, the user decided to further inspect the history of lines 142–143 because they led to an unexpected behavior. Once the block is clicked, a pane appears from the bottom (Point 8): It contains the historical details of the changes happened to that block since the last update of the user. Each item represents a change and shows the changed code with respect to the previous commit (Point 9), the commit message to document it (Point 10), and the contact information of the change author (Point 11). The changed code only regards the chosen block, but it is possible to extend it by clicking on the '...' before and after. It is also possible to inspect also previous history (Point 12).

RITE participants' feedback—As for the other steps, before showing what would happen next, the interviewer asked the participants how they would interact with the design and what their expectations would be. In particular, for this feature, the interviewer asked what participants expected it would happen by clicking on the colored sign on the left (Point 5). In this way, we learned that the participants wanted to have something similar to a diff with the previous version (*e.g., "I'd do a compare versus the previous version, and just look at those particular changes*" [R3]). The BELLEVUE solution was, thus, very much appreciated and it often exceeded their expectations: "*All the details! This is exactly what I was looking for: It tells me who* [...] *and it tells me what did each one, and how long ago!*" [R1]; "*oh I see, so this is exactly what I was looking for. It's even better!*" [R8]. Seeing the version that could have introduced the error (*i.e.,* #9044) was a clearly

marked exit: Some participants considered to recover the error by reverting that particular change, because that would not imply reverting entirely to a more complex change set.

Through the iterations, we added the clickable revision number (to open a panel to see all the changes in a revision), and the hovering function to show the full commit comment.

Participants' suggestions that we did not eventually include in the iterative evaluation, due to time reasons, mostly regarded the possibility of selecting specific parts to see the history, instead of the contiguous block (*e.g.,* "*I want to see the whole function* [history, by] *right clicking on a function*" [R2]).

Editable code

BELLEVUE allows editing code while reviewing the history (Figure 8), because it integrates history within the active editing context. It also highlights the new code differently (Point 13 in Figure 9) and automatically adds a new item to the list (Point 14) to put it in its historical context. This differs from current approaches for visualizing history, which involve opening a distinct context or application, and do not make it possible to edit code (*e.g.,* to fix a bug) and see history in the same context, at the same time.

RITE participants' feedback—This feature was very well received by all the participants. In particular, many were positively surprised and realized the possibilities of having code changes better integrated in the IDE: "*I have a diff view, but I am not trapped in that* [...] *I got my editor and my diff view, so the split view is very very helpful* [...]. *Let me do what I want to do, while looking at the information I needed to make my change*" [R1]; "*Now that I see, I know what is happening* [...]. *That is intuitive to me: Just clicking, edit, and go*" [R7]. They also appreciated the immediate feedback of the change in the local history (Figure 9): "*Oh, I like it shows it's local*" [R4].

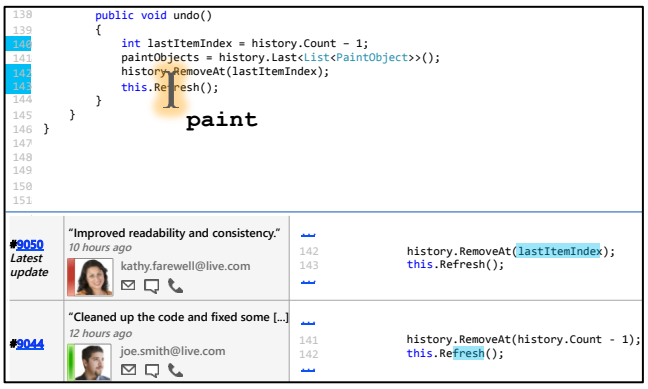

Figure 8. Editable code while accessing historical details

Contacting change's author

The author's photo and contact pane is inspired by CARES [32], a tool to help developers discover and choose relevant colleagues to speak with when seeking information or coordinating action. In BELLEVUE, the communication

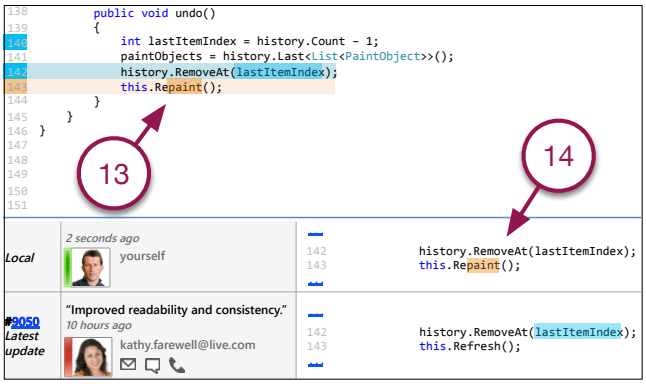

Figure 9. New local change added to history

icons are for contacting the author of a given commit. Figure 10 shows the email template automatically generated by clicking on the email icon for commit #9044; it includes the diff with the previous commit, for easier reference.

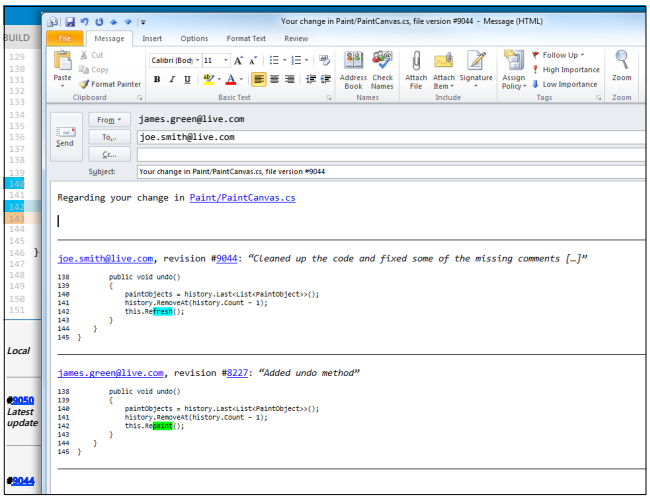

Figure 10. Contacting the author of a change from the IDE

RITE participants' feedback—The social interaction within the view was extremely well received by all the participants. They especially appreciated the possibility of quickly using email and IM: "*I really like that. I'd click on chat*" [R6]. When discussing the email they would write to the author of the buggy change, they all specify the things they would like to ideally see in the email, and when they see it, they like how it includes everything they wanted: "*That is perfect.* [It is] *exactly what I would have sent*" [R1]. However, some would have liked to obtain the diff view as BELLEVUE shows it in the IDE, while "*now it's like standard diff*" [R6].

Participants' suggestions not integrated for time reasons are: Adding an *email all* feature, change the email title to give information about method and class in which the new change is taking place, support for copy and paste from history to email, and add communication clients (*e.g.,* IRC or Skype).

Evaluation Debriefing

After each RITE session, participants filled two short questionnaires about their experience with the tool: A System Usability Scale (SUS) questionnaire [11] and a proprietary 7-point Likert scale questionnaire standardly used at Microsoft. The SUS answers were overall positive: The mean SUS score is 85.1 (answers had $\sigma = 0.66$, on average), which is considered a high value across different domains [5, 6, 67]; as an example, the statement "*I think that I would like to use this product frequently*" scored 4.7/5.0 ($\sigma = 0.50$). The proprietary survey was equally positive: Mean score was 5.4/7.0 (the higher the better: items only included positive wordings [40]), with $\sigma = 1$ on average. For example participants gave 5.4/7.0 ($\sigma = 1.13$) to the statement "*This product has powerful functionality and excels at what it was designed for*" and "*This product is something I am likely to share information about*" scored 5.9/7.0 ($\sigma = 0.78$).

COLLABORATIVE SOFTWARE DEVELOPMENT TOOLS

Coordination in software development has been studied in the fields of Software Engineering and Computer Supported Cooperative Work since the 1980s, and researchers have produced a wide range of analyses and tools [62].

BELLEVUE uses historical change information to support developers' coordination. Sarma *et al.* present a comprehensive review of coordination tools and defines a framework that classifies those technologies according to multiple coordination paradigms [66]. In this framework, tools such as versioning systems and issue tracking systems support the development process and are at the basis of the more sophisticated tools that provide meaningful and automatically aggregated information: These are research prototypes and industrial applications conceived to better support developers coordination in the IDE. Such tools includes full-fledged platforms, specific workspace awareness solutions, information discovery approaches, and code information visualization tools.

Full-fledged platforms, such as Jazz [39] and Mylyn [25], are at the far end of the spectrum in terms of complexity [66], and aim at transforming the IDE experience. Jazz, or Rational Team Concert, is an IDE platform, built on top of Eclipse and Visual Studio, that integrates several aspects of the software development process, including integrated planning, tracking of developer effort, project dashboards, reports, and process support. Relations between artifacts can be defined and leveraged to gather project information. Jazz also offer support for communication within the IDE (*e.g.,* instant messaging), more advanced than BELLEVUE. Mylyn and its successor, Tasktop Dev [70], are based on Eclipse and Visual Studio and use task context to improve the productivity of developers and teams [43]; for example, they reduce information overload by providing developers with just the artifacts and information necessary for their current code modification task, and offer a comprehensive task repository to support teamwork by sharing information on tasks and their context. Both platforms support the creation of novel information (*e.g.,* tasks and work items, and relations among artifacts) to support developers productivity, and encourage a task or work item based approach to evolution. BELLEVUE aims at using already available data and visualizing it in a non-obtrusive way. Another example of improved communication in the IDE is REmail [4], which integrates developers' email communication in the IDE to support program comprehension; REmail can be used in conjunction with BELLEVUE to extend the communication feature of the latter.

Workspace awareness solutions, such as Palantìr [64], Lighthouse [19], CollabVS [23], Syde [34], and Crystal [12] are concerned with changes before they are committed to the source code repository, to address the conflict detection or real-time development information. For example, Syde tracks fine-grained real-time changes and alerts developers on the code editor and on a view when potential conflicts are emerging. Given the goal of these tools, differently from BELLEVUE, they do not show change history related information.

Interestingly, BELLEVUE design could be included in environments such as Mylyn and Jazz, and could be used concurrently with workspace awareness tools, in order to offer coordination support from a complementary perspective.

Information discovery approaches, such as Ariadne [20] and Tesseract [63], seek and assemble information to perform tasks such as expert finding and socio-technical network analysis. Recommender systems, such as Seahawk [57, 58], exploit change information and externally generated data to support software development and comprehension. Similarly to BELLEVUE some of these approaches also use historical code information to inform their users. Given their goal, they offer different, complementary views on data and integration with the development environment.

Code information visualization tools include the "blame" functionality offered, for example, by git or svn.This feature allows to see who did the last change on each line of code of a file, and when. Another tool is the concept presented by Rastkar and Murphy, in which the developer is able to see for a summary of commit messages connected to a line of code in the IDE [60]. In contrast, BELLEVUE offers an interactive view that shows detailed historical information for specific chucks with respect to a previous version. BELLEVUE always displays which files and lines changed, so it does not require the developer to actively ask for the commit message of the line, because the developer may not be already aware of the relevance of the file and the line. In our exploratory investigation narrowing down a breaking change to the file and line causing the issue emerged as one of the most problematic and time-consuming efforts for developers.

FINAL REMARKS

In our study we explored how to support developers' collaboration in teamwork. We focused on teamwork in the software implementation phase, which takes place in the IDE, and we conducted a qualitative investigation to uncover actionable areas for improvement. We identified internal breaking changes as one of the most important areas for improvement, because current IDE support for receiving changes is not optimal. Consequently, we designed BELLEVUE to enable developers better coordinate, by making historical information visible and more accessible in the IDE.

Overall, this paper makes the following main contributions:

1. A qualitative analysis indicating that teamwork needs mostly regard coordination, that developers are able to face scenarios considered problematic in literature, and that dealing with breaking changes is hard, but it only generates frustration if the breaker is internal to the project.

2. Recommendations on how to improve collaboration in teamwork in the software implementation phase, such as to focus on interactions revolving around coordination rather than on collaboration on the same (sub)task.

3. Requirements for a tool to support teamwork based on currently unmet usability heuristics and the results of our qualitative analysis. For example, to favor recognition of code changes over recall, and to increase the visibility of the codebase status with respect to received changes.

4. The design and evaluation of BELLEVUE, an IDE extension to support teamwork by improving the integration of code changes in the IDE. BELLEVUE makes received changes visible inside the editor, and makes the history of code chunks easily accessible using progressive disclosure.

ACKNOWLEDGMENTS

We want to express our gratitude to the anonymous reviewers, whose valuable comments significantly helped to improve the paper. We warmly thank Andrew Begel for his first-class feedback on the revision of this paper, and Monty Hammontree for his support during Anja's internship.

REFERENCES

1. Anvik, J., Hiew, L., and Murphy, G. C. Who should fix this bug? In *Proceedings of ICSE 2006 (28th International Conference on Software Engineering)*, ACM Press (2006), 361–370.
2. Anvik, J., and Murphy, G. C. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology 20*, 3 (Aug. 2011), 10:1–10:35.
3. Arnold, R., and Bohner, S. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996.
4. Bacchelli, A., Lanza, M., and Humpa, V. RTFM (Read The Factual Mails) –augmenting program comprehension with REmail. In *Proceedings of CSMR 2011 (15th IEEE European Conference on Software Maintenance and Reengineering)* (2011), 15–24.
5. Bangor, A., Kortum, P., and Miller, J. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction 24*, 6 (July 2008), 574–594.
6. Bangor, A., Kortum, P., and Miller, J. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies 4*, 3 (May 2009), 114–123.
7. Begel, A., Khoo, Y. P., and Zimmermann, T. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, ACM (2010), 125–134.
8. Begel, A., Nagappan, N., Poile, C., and Layman, L. Coordination in large-scale software teams. In *Proceedings of the CHASE 2009 (2nd International Workshop on Cooperative and Human Aspects of Software Engineering)*, IEEE Computer Society (2009), 1–7.
9. Biehl, J. T., Czerwinski, M., Smith, G., and Robertson, G. G. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems*, ACM (2007), 1313–1322.
10. Black, S. Computing ripple effect for software maintenance. *Journal of Software Maintenance 13*, 4 (Sept. 2001), 263–.
11. Brooke, J. SUS: A 'quick and dirty' usability scale. In *Usability Evaluation in Industry*, P. W. Jordan, B. Thomas, I. L. McClelland, and B. Weerdmeester, Eds. CRC Press, 1996, ch. 21, 189–194.
12. Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. Proactive detection of collaboration conflicts. In *Proceedings of ESEC/FSE 2011 (8th Joint Meeting on Foundations of Software Engineering)*, ACM (2011), 168–178.
13. Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering 39*, 10 (2013), 1358–1375.
14. Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering 35*, 6 (Nov. 2009), 864–878.
15. Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of CSCW 2006 (20th Anniversary Conference on Computer Supported Cooperative Work)*, ACM (2006), 353–362.
16. Chen, W.-N., and Zhang, J. Ant colony optimization for software project scheduling and staffing with an event-based scheduler. *IEEE Transactions on Software Engineering 39*, 1 (Jan. 2013), 1–17.
17. Cheng, L.-T., de Souza, C. R., Hupfer, S., Patterson, J., and Ross, S. Building collaboration into IDEs. *ACM Queue 1*, 9 (2003), 40–50.
18. Curtis, B., Krasner, H., and Iscoe, N. A field study of the software design process for large systems. *Communications of the ACM 31*, 11 (Nov. 1988), 1268–1287.
19. da Silva, I., Chen, P., der Westhuizen, C. V., Ripley, R., and van der Hoek, A. Lighthouse: Coordination through emerging design. In *Proceedings of ETX 2006 (OOPSLA Workshop on Eclipse Technology eXchange*, ACM Press (2006), 11–15.
20. de Souza, C. R. B., Quirk, S., Trainer, E., and Redmiles, D. F. Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proceedings of GROUP 2007 (International ACM SIGGROUP Conference on Supporting Group Work)*, ACM (2007), 147–156.
21. de Souza, C. R. B., Redmiles, D., and Dourish, P. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of GROUP 2003 (International ACM SIGGROUP Conference on Supporting Group Work)*, ACM Press (2003), 105–114.
22. de Souza, C. R. B., and Redmiles, D. F. An empirical study of software developers' management of dependencies and changes. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference on Software Engineering)*, ACM (2008), 241–250.
23. Dewan, P., and Hegde, R. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of ECSCW 2007 (10th European Conference on Computer Supported Cooperative Work)*, Springer (2007), 24–28.
24. Duggan, J., Byrne, J., and Lyons, G. J. A task allocation optimizer for software construction. *IEEE Software 21*, 3 (May 2004), 76–82.
25. Eclipse Foundation. Mylyn. [Software]. Available: https://www.eclipse.org/mylyn/ [Accessed: Jun 4, 2014], 2014.
26. Fritz, T., and Murphy, G. C. Using information fragments to answer the questions developers ask. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, ACM (2010), 175–184.
27. Frost, R. Jazz and the eclipse way of collaboration. *IEEE Software 24*, 6 (2007), 114–117.
28. Furnham, A. Response bias, social desirability and dissimulation. *Personality and Individual Differences 7*, 3 (1986), 385 – 400.
29. Grinter, R. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work 5*, 4 (1996), 447–465.

30. Grubb, A. M., and Begel, A. On the perceived interdependence and information sharing inhibitions of enterprise software engineers. In *Proceedings of CSCW 2012 (ACM Conference on Computer Supported Cooperative Work)*, ACM (2012), 1337–1346.
31. Guzzi, A., Bacchelli, A., Lanza, M., Pinzger, M., and van Deursen, A. Communication in open source software development mailing lists. In *Proceedings of MSR 2013 (10th IEEE Working Conference on Mining Software Repositories)* (2013), 277–286.
32. Guzzi, A., Begel, A., Miller, J. K., and Nareddy, K. Facilitating enterprise software developer communication with CARES. In *Proceedings of ICSM 2012 (28th IEEE International Conference on Software Maintenance)* (2012), 527–536.
33. Hattori, L. *Change-centric Improvement of Team Collaboration*. PhD thesis, Università della Svizzera Italiana, February 2012.
34. Hattori, L., and Lanza, M. Syde: A tool for collaborative software development. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)* (2010), 235–238.
35. Hattori, L., Lanza, M., and D'Ambros, M. A qualitative analysis of preemptive conflict detection. Tech. Rep. 2011/05, University of Lugano, Sept. 2011.
36. Hegde, R., and Dewan, P. Connecting programming environments to support ad-hoc collaboration. In *Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering*, IEEE CS Press (2008), 178–187.
37. Henderson, R. M., and Clark, K. B. Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms. *Administrative Science Quarterly 35*, 1 (Mar. 1990), 9–30.
38. Herbsleb, J. D., Mockus, A., and Roberts, J. A. Collaboration in software engineering projects: A theory of coordination. In *Proceedings ICIS 2006 (International Conference on Information Systems)* (2006).
39. IBM. Rational Team Concert. [Software]. Available: http://jazz.net/projects/rational-team-concert/ [Accessed: Jun 4, 2014], 2014.
40. Jeff, and Lewis, J. R. When designing usability questionnaires, does it hurt to be positive? In *Proceedings of CHI 2011 (29th Conference on Human Factors in Computing Systems)*, CHI '11, ACM (2011), 2215–2224.
41. Jeong, G., Kim, S., and Zimmermann, T. Improving bug triage with bug tossing graphs. In *Proceedings of ESEC/FSE 2009 (7th Joint Meeting on Foundations of Software Engineering)*, ACM (2009), 111–120.
42. Kasi, B. K., and Sarma, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, IEEE Press (2013), 732–741.
43. Kersten, M., and Murphy, G. C. Using task context to improve programmer productivity. In *Proceedings of FSE 2006 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, ACM (2006), 1–11.
44. Kirsch, L. J. The Management of Complex Tasks in Organizations: Controlling the Systems Development Process. *Organization Science 7*, 1 (Jan. 1996), 1–21.
45. Ko, A. J., DeLine, R., and Venolia, G. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, IEEE Computer Society (2007), 344–353.
46. Kraut, R. E., and Streeter, L. A. Coordination in software development. *Communications of the ACM 38*, 3 (Mar. 1995), 69–81.
47. Kwan, I., Schroter, A., and Damian, D. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *IEEE Transactions on Software Engineering 37*, 3 (May 2011), 307–324.
48. LaToza, T. D., Venolia, G., and DeLine, R. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, ACM (2006), 492–501.
49. Lidwell, W., Holden, K., and Butler, J. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*, 2nd ed. Rockport Publishers, January 2010.
50. Lindlof, T. R., and Taylor, B. C. *Qualitative Communication Research Methods*. SAGE Publications, Inc., 2010.
51. Malone, T. W., and Crowston, K. The interdisciplinary study of coordination. *ACM Computing Surveys 26*, 1 (Mar. 1994), 87–119.
52. Martin, B., and Hanington, B. *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Rockport Publishers, 2012.
53. Matter, D., Kuhn, A., and Nierstrasz, O. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of MSR 2009 (6th International Working Conference on Mining Software Repositories)*, IEEE Computer Society (2009), 131–140.
54. Medlock, M. C., Wixon, D., Terrano, M., Romero, R. L., and Fulton, B. Using the RITE method to improve products: A definition and a case study. In *Proceedings of UPA 2002 (Usability Professionals Association)* (2002).
55. Molich, R., and Nielsen, J. Improving a human-computer dialogue. *Communications of the ACM 33*, 3 (Mar. 1990), 338–348.
56. Nielsen, J. 10 usability heuristics for user interface design. http://www.nngroup.com/articles/ten-usability-heuristics/, January 1995.
57. Ponzanelli, L., Bacchelli, A., and Lanza, M. Leveraging crowd knowledge for software comprehension and development. In *Proceedings of CSMR 2013 (17th European Conference on Software Maintenance and Reengineering)*, IEEE CS Press (2013), 57–66.
58. Ponzanelli, L., Bacchelli, A., and Lanza, M. Seahawk: Stack overflow in the ide. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, IEEE CS Press (2013), 1295–1298.
59. Proenca, T., Moura, N., and van der Hoek, A. On the use of emerging design as a basis for knowledge collaboration. *New Frontiers in Artificial Intelligence 6284* (2010), 124–134.
60. Rastkar, S., and Murphy, G. C. Why did this code change? In *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, IEEE Press (2013), 1193–1196.
61. Robbes, R., Lungu, M., and Röthlisberger, D. How do developers react to api deprecation?: The case of a smalltalk ecosystem. In *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering)*, ACM (2012), 56:1–56:11.
62. Sarma, A. A survey of collaborative tools in software development, isr. Tech. rep., Institute for Software Research, University of California, Irvine, 2005.
63. Sarma, A., Maccherone, L., Wagstrom, P., and Herbsleb, J. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of ICSE 2009 (31st International Conference on Software Engineering)*, IEEE Computer Society (Washington, DC, USA, 2009), 23–33.
64. Sarma, A., Noroozi, Z., and van der Hoekvan der Hoek. Palantìr: Raising awareness among configuration management workspaces. In *Proceedings of ICSE 2002 (23rd International Conference on Software Engineering)*, IEEE CS Press (2003), 444–454.
65. Sarma, A., Redmiles, D., and van der Hoek, A. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proceedings of FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press (2008), 113–123.
66. Sarma, A., Redmiles, D., and van der Hoek, A. Categorizing the spectrum of coordination technology. *IEEE Computer 43*, 6 (June 2010), 61–67.
67. Sauro, J. *A Practical Guide to the System Usability Scale: Background, Benchmarks and Best Practices*. CreateSpace, 2011.

68. Sillito, J., Murphy, G. C., and Volder, K. D. Questions programmers ask during software evolution tasks. In *Proceedings of FSE 2006 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, ACM (2006), 23–34.

69. Spencer, D. Card sorting: a definitive guide. http://boxesandarrows.com/card-sorting-a-definitive-guide/, April 2004.

70. Tasktop. Tasktop Dev. [Software]. Available: http://www.tasktop.com/dev [Accessed: Aug 1, 2014], 2014.

71. Whitehead, J. Collaboration in software engineering: A roadmap. In *Proceedings of FOSE 2007 (Future of Software Engineering)*, IEEE Computer Society (2007), 214–225.

72. Yau, S. S., Colofello, J. S., and MacGregor, T. Ripple effect analysis of software maintenance. In *Proceedings of COMPSAC*, IEEE Computer Society Press (1978), 60–65.

73. Zeller, A. The future of programming environments: Integration, synergy, and assistance. In *Proceedings of FOSE 2007 (Future of Software Engineering)*, IEEE Computer Society (2007), 316–325.