Speculative Analysis of Integrated Development Environment Recommendations

Kıvanç Muşlu [†], Yuriy Brun [♮], Reid Holmes [♣], Michael D. Ernst [†], David Notkin [†]

[†] Computer Science & Engineering
University of Washington
Seattle, WA, USA
{kivanc, mernst, notkin}@cs.washington.edu

[♮] Department of Computer Science
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

[♣] School of Computer Science
University of Waterloo
Waterloo, ON, Canada
rtholmes@cs.uwaterloo.ca

Abstract

Modern integrated development environments make recommendations and automate common tasks, such as refactorings, auto-completions, and error corrections. However, these tools present little or no information about the consequences of the recommended changes. For example, a rename refactoring may: modify the source code without changing program semantics; modify the source code and (incorrectly) change program semantics; modify the source code and (incorrectly) create compilation errors; show a name collision warning and require developer input; or show an error and not change the source code. Having to compute the consequences of a recommendation — either mentally or by making source code changes — puts an extra burden on the developers.

This paper aims to reduce this burden with a technique that informs developers of the consequences of code transformations. Using Eclipse Quick Fix as a domain, we describe a plug-in, Quick Fix Scout, that computes the consequences of Quick Fix recommendations. In our experiments, developers completed compilation-error removal tasks 10% faster when using Quick Fix Scout than Quick Fix, although the sample size was not large enough to show statistical significance.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: General; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments

General Terms Algorithms, Experimentation, Human Factors

Keywords Quick Fix Scout, Eclipse, Quick Fix dialog, Quick Fix, speculative analysis, IDE, recommendations

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

1. Introduction

Integrated development environments (IDEs), such as Eclipse and Visual Studio, provide tools that automate common tasks, such as refactoring, auto-complete, and correction of compilation errors. These tools have two goals: increasing developer speed and reducing developer mistakes. These tools are widely used: they are the most frequent developer actions after common text editing commands such as delete, save, and paste [12].

Despite their popularity, these recommendations are provided with little or no information about their consequences. For example, a rename refactoring changes the name of a variable everywhere in a program. However, this refactoring cannot be correctly and automatically applied when there are compilation errors or name collisions in the project. In those cases, if the developer is lucky, the IDE will detect the failure and either roll back the refactoring or assist the developer in performing the refactoring manually. For an unlucky developer, the IDE will perform an incomplete refactoring and break the code without notification, causing the developer to spend time determining if and why the refactoring failed, and fixing the code.

As another example, whenever there is a compilation error in an Eclipse project, Eclipse offers *Quick Fix* proposals: transformations that may resolve the error. However, some of these proposals may not resolve the compilation error and may even introduce new errors. When this happens, a developer may waste time undoing the proposal or trying other proposals, and may even give up on Quick Fix.

Figure 1 shows a Quick Fix dialog with proposals for a compilation error. Clicking on a proposal shows an additional yellow window previewing the changed code. However, the developer still needs to answer the following questions about each proposal:

- Does it resolve this compilation error?
- Does it resolve other compilation errors?
- Does it introduce new compilation errors?

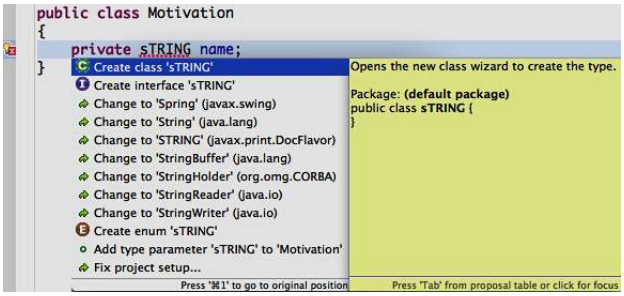

Figure 1. A Quick Fix dialog with 12 proposals. The window on the right previews the highlighted proposal.

Considering Quick Fix proposals in isolation can be limiting because developers may further wish to consider the following question:

- Which proposal, among those that would be offered for all compilation errors in the project, resolves the largest number of errors?

The Quick Fix dialog does not answer these questions. The developer can try to compute the answers mentally, or the developer can apply a proposal and manually investigate its effects on the programs. Both approaches are error-prone and time-consuming.

We aim to improve Quick Fix by informing developers of the consequences of each proposal, specifically of the proposal's effect on the number of compilation errors. As a proof-of-concept, we have built an Eclipse plug-in, Quick Fix Scout, that computes which compilation errors are resolved by each proposal. When a user invokes Quick Fix, Quick Fix Scout augments the standard dialog with additional, relevant proposals, and sorts the proposals with respect to the number of compilation errors they resolve.

This paper makes the following contributions:

- A novel technique for automatically computing the consequences of Quick Fix recommendations.
- An open-source, publicly-available tool — Quick Fix Scout: `http://quick-fix-scout.googlecode.com` — that communicates the consequences of a Quick Fix proposal to the developer.
- A case study that shows that most of the time (93%) developers apply one of the top three proposals in the dialog (Section 5.1).
- A controlled experiment with 20 users that demonstrates that Quick Fix Scout allows developers to remove compilation errors 10% faster, compared to using traditional Quick Fix (Section 5.2).

The rest of the paper is organized as follows. Section 2 explains the problem with Eclipse's Quick Fix. Section 3 presents speculative analysis and the Quick Fix Scout implementation. Section 4 introduces global best proposals — the additional proposals Quick Fix Scout adds to the dialog. Section 5 details the case study and controlled experiment design and results, and Section 6 discusses threats to the validity of these results. Section 7 places Quick Fix Scout in the context of related work. Finally, Section 8 concludes the paper.

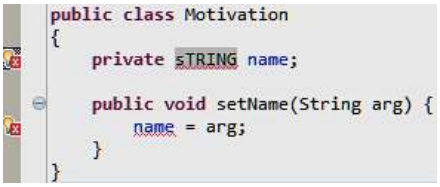

Figure 2. A Java program with two compilation errors. There is only one logical error: the type `sTRING` should be `String`.

2. Not Knowing the Consequences

Eclipse uses a fast, incremental compiler to identify and underline compilation errors with a "red squiggly". A developer who invokes Quick Fix at an error sees a pop-up dialog with a list of actions each of which may fix the error. The Eclipse documentation notes that Quick Fix can be used not only to provide suggestions but also as a shortcut for more expert users. [1]

Figures 2–4 demonstrate a situation in which Quick Fix falls short. Figure 2 shows a program with two compilation errors due to a single type mismatch between a variable declaration and a use site. The variable `name` should be declared to be of type `String` but is instead declared as `sTRING`. Invoking Quick Fix at the declaration error shows 12 proposals (Figure 3). The correct proposal — Change to '`String`' — is the fourth choice in the list. Ideally, Eclipse would provide the correct proposal as the first recommendation. Lower positions in the list likely cause the user to spend more time studying the choices or to cancel Quick Fix and address the error manually.

Invoking Quick Fix at the use error is worse for the developer. Figure 4 shows the 15 Quick Fix proposals, *none* of which resolves either error. Sophisticated users may realize this, cancel the invocation, and finish the change manually. Others may apply a proposal and either quickly realize that this was a poor choice and undo it, or perform more actions attempting to resolve the error, creating successively more difficult situations to recover from.

2.1 Visualizing Quick Fix consequences

Quick Fix Scout pre-computes the consequences of each proposal and visualizes this information by augmenting the Quick Fix dialog in three ways:

[1] `http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F`

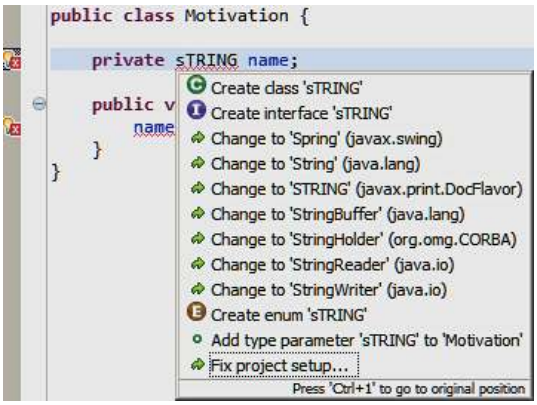

Figure 3. 12 Quick Fix proposals to resolve the type error from Figure 2.

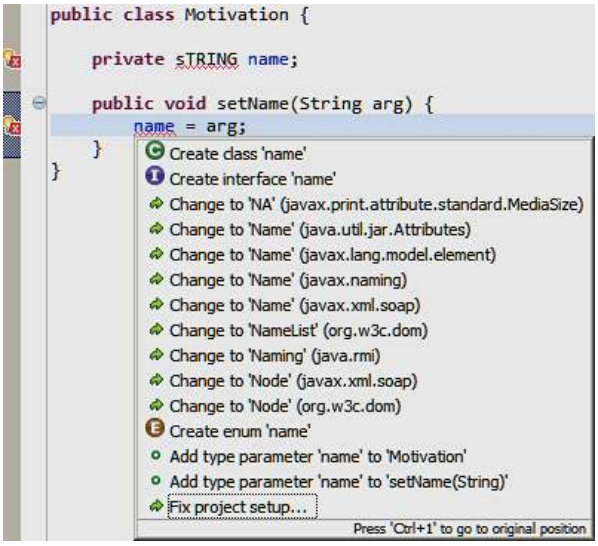

Figure 4. 15 Quick Fix proposals to resolve the assignment error from Figure 2. None of these proposals resolves either compilation error.

1. To the left of each proposal, add the number of compilation errors that remain after the proposal's hypothetical application.
2. Sort the proposals with respect to the number of remaining compilation errors.
3. Color the proposals: green for proposals that reduce the number of compilation errors, black for proposals that do not change the number of compilation errors, and red for proposals that increase the number of compilation errors.

Figure 5 — the Quick Fix Scout equivalent of Figure 3 — shows all these augmentations, except the red coloring.

Section 4 discusses one additional feature of Quick Fix Scout: *global best proposal*, which addresses the problem in Figure 4. Changing `sTRING` to `String` (offered only at the first error's site) resolves both compilation errors. However, Quick Fix does not present this proposal at the second error's site, even though it is relevant. Quick Fix Scout addresses this problem by providing the relevant proposal at both compilation error locations.

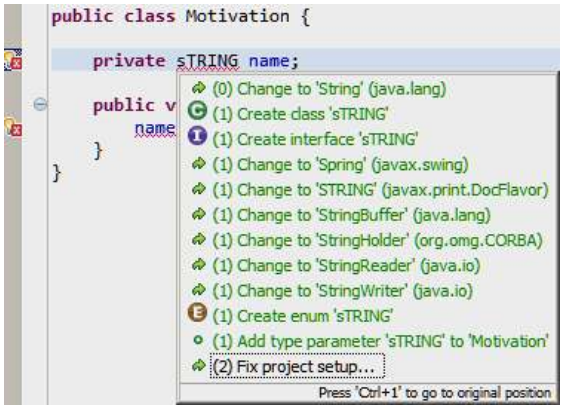

Figure 5. Quick Fix Scout sorts the 12 proposals offered by Eclipse (shown in Figure 3) by the number of errors that the proposal fixes.

Quick Fix Scout uses the number of remaining compilation errors as the main criterion to reorder and color the proposals. Since the primary intent of Quick Fix is to resolve compilation errors, we assume that a proposal that resolves more compilation errors is likely to be preferred by the developer. The proposals that resolve the same number of compilation errors are sorted using Quick Fix's standard ordering. This allowed us to measure the effects of the main criterion more accurately when evaluating Quick Fix Scout. The empirical data support the premise that the developers prefer proposals that resolve the highest number of compilation errors. Case study participants (Section 5.1) who used Quick Fix Scout selected a proposal that resolved the most compilation errors 90% of the time. Similarly, controlled experiment participants (Section 5.2) who used Quick Fix Scout selected such proposals 87% of the time, and those who used Quick Fix, 73% of the time.

3. Quick Fix Scout

Speculative analysis [2] explores possible future development states to help the developer make a decision that may lead to one or more of those states. Quick Fix Scout [10] is an Eclipse plug-in that speculatively applies each available Quick Fix proposal and compiles the resulting program. Quick Fix Scout augments the Quick Fix dialog to show how many compilation errors would remain after each proposal's hypothetical application, and sorts the proposals accordingly.

Next, Section 3.1 details the mechanism for computing Quick Fix proposals' consequences. Then, Section 3.2 describes the requirements for seamless background computation. Section 3.3 explains additional optimizations specific

```
1  while (true) {
2    waitUntilChangeInErrors();
3    for (Error err: copy.getErrors()) {
4      for (Proposal p: err.quickFixes()) {
5        copy.applyProposal(p);
6        copy.saveAndBuild();
7        results.add(p, copy.getErrors());
8        copy.applyProposal(p.getUndo());
9      }
10     publishResults();
11   }
12 }
```

Figure 6. A high-level description of the speculative analysis algorithm for computing the compilation errors that remain after applying each Quick Fix proposal. The publishResults() method augments the Quick Fix dialog with the proposal consequences.

to Quick Fix Scout. Section 3.4 discusses implementation limitations. Finally, Section 3.5 provides insight into generalizing the technique and the implementation to other IDEs and recommendations.

3.1 Computing Quick Fix consequences

Quick Fix Scout uses the speculative analysis algorithm, described at a high level in Figure 6, to compute the consequences of Quick Fix proposals. Quick Fix Scout maintains a separate, hidden *copy* of the developer's code and performs all its analysis on that copy, to avoid disturbing the developer's workspace. (Section 3.2 further describes the use of the copy.) Whenever the developer introduces a new compilation error or fixes an old one (line 2), Quick Fix Scout applies each proposal to the copy (line 5), one at a time, saves and builds the copy (line 6), and associates that proposal with the set of compilation errors that remain (line 7). Quick Fix Scout then undoes the proposal to restore the copy's state (line 8). Quick Fix Scout updates the Quick Fix dialog after computing the consequences of all the proposals (line 10).

3.2 Managing a copy of the developer's code

Quick Fix Scout maintains a copy of the developer's workspace. The copy is changed in two ways:

- Whenever the developer edits the main workspace, the copy is edited to keep it in sync with the mail workspace. Quick Fix Scout uses Eclipse's resource change listeners to listen for edits.
- Quick Fix Scout applies Quick Fix proposals, analyzes the consequences, and reverts the modifications.

Suppose the developer makes an edit while Quick Fix Scout is applying and reverting proposals. If the edit does not change the current compilation errors, then Quick Fix Scout buffers the changes until its speculative analysis completes, and only then applies them to the copy. If the edit does change the current compilation errors, then Quick Fix Scout abandons and restarts its speculative computation. This prevents stale results from being displayed and improves responsiveness.

3.3 Optimizations for a responsive UI

Ideally, Quick Fix Scout computes the consequences of a new error's proposals in the time between when the developer introduces the error and invokes Quick Fix. Quick Fix Scout includes the following optimizations and heuristics:

- It only recomputes consequences if a code change affects the compilation errors, as described in Section 3.2.
- It uses a user-adjustable typing session length to identify atomic sets of changes. A series of edits without a typing-session-length pause constitute an atomic set of edits. Quick Fix Scout waits for an entire atomic session to complete before recomputing consequences. Thus, for example, Quick Fix Scout ignores the temporary compilation errors that arise in the middle of typing a complete token.
- It considers first the errors that are closest to the cursor in the currently open file.
- It caches the consequences (i.e., the remaining compilation errors) for each proposal and uses the cache whenever Eclipse offers the same proposal at multiple locations.
- It updates the Quick Fix dialog incrementally, as results for errors (but not individual proposals for each error) become available. This is shown in Figure 6.

In general, each proposal application is a small change and, even for large projects, Eclipse can incrementally compile the updated project extremely quickly. Therefore, Quick Fix Scout's computation scales linearly with the number of proposals (which is proportional to the number of compilation errors), and independently of the size of the project. During typical coding, at any particular time, a project has several compilation errors with several proposals for each. The total number of proposals is typically less than a couple hundreds. As a worst-case example, we experimented with an 8K-line project with 50 compilation errors and 2,400 proposals. A 2.4GHz Intel Core i5 (quad core) MacBook Pro with 8GB of RAM computed all the consequences in 10 seconds, on average (computed over 10 consecutive computations, after allowing Eclipse's incremental compiler to optimize). This suggests Quick Fix Scout can scale well to large projects. Finally, since each proposal is analyzed separately, the analysis can be parallelized, though we have not yet implemented that functionality.

3.4 Current implementation limitations

There are at least four ways to invoke Quick Fix in Eclipse: (1) by pressing the keyboard shortcut, (2) by selecting Quick Fix through the context menu, (3) by clicking on the icon on

the left of the screen, and (4) by hovering the mouse over the compilation error. Internally, the first three methods create a Quick Fix dialog and the last method creates a Hover Dialog. The Hover Dialog is handled by `org.eclipse.jdt.ui` plug-in and the Eclipse installation does not permit us to modify this plug-in as we modified `org.eclipse.jface.text`. Though we have an implementation that works in debug mode for the Hover Dialog, our installation fails when it includes a modified `jdt.ui`. A future version of Eclipse will include a public API for reordering content assist type recommendations (e.g., auto-complete and Quick Fix),[2] which would simplify our implementation and might remove this limitation.

For each proposal, the Eclipse API provides an *undo change* that rolls back the associated proposal application. After analyzing each proposal, Quick Fix Scout uses this mechanism to return the copy project to its initial state. The proposals "Change compilation unit to '`typeName`'" and "Move '`typeName`' to '`packageName`'" have a bug in their implementation: the corresponding undos do not restore the project to its original state.[3] We have reported both bugs to Eclipse and they have been reproduced by the developers, but they have not yet been resolved. Quick Fix Scout must either skip analyzing these two proposals or re-copy the copy project after their analysis. Since re-copying can take considerable time for large projects, for performance reasons, the current implementation skips the analysis of these proposals and produces no consequence information for them, leaving the appropriate lines in the Quick Fix dialog unaugmented.

Quick Fix Scout uses an internal Eclipse API to apply proposals to the copy project. By default, this API acts as a no-op for the proposals that require user interaction. Therefore, currently, Quick Fix Scout does not compute the consequences of these proposals and leaves the appropriate lines in the Quick Fix dialog unaugmented. However, to our best knowledge, there are only four such proposals: Create class, interface, annotation, and enum '`typeName`'. These proposals do not modify existing code, but instead create new code. Therefore, it is relatively simple for developers to mentally predict their consequences.

3.5 Generalizing beyond Quick Fix and Eclipse

The ideas we demonstrated on Quick Fix Scout within Eclipse also apply to engines that produce other types of recommendation, such as refactorings and automatic code completions, and to other IDEs, such as NetBeans, IntelliJ, and Visual Studio.

Analysis of the possible future states of a non-pure recommendation — that modifies the source code when applied — cannot be applied to the developer's working copy as it might interfere with development. Most popular types of recommendations, such as refactorings, automatic code completions, and automatic code corrections, are non-pure code transformations. Section 3.2 proposes one way to separate the analysis from the developer's working copy using a code copy. Although this method uses Eclipse-specific constructs, such as resource change listeners, these constructs are a common design pattern available in all major IDEs. Therefore, the use of code copies for background analysis integration, without disturbing the developer's possibly active code, generalizes to other IDEs and recommendations.

Any recommendation may become obsolete when the code changes. Thus, most of the optimizations and heuristics in Section 3.3 apply to other recommendations. For example, automatic code completions that are closest to the current cursor position can be prioritized and computed first.

Finally, Quick Fix Scout is an instantiation of speculative analysis: the future states are generated via Quick Fix proposals, and the consequences are represented by the number of remaining compilation errors. By generating future states and representing consequences in other ways, speculative analysis can generalize to other consequences and recommendation engines. For example, refactoring suggestions can generate future states, and failing tests could represent the consequences.

4. Global Best Quick Fixes

Quick Fix Scout helps developers to quickly locate the best local proposals — the proposals that resolve the most compilation errors — by sorting them to the top in the Quick Fix dialog. However, sometimes, Eclipse offers the best proposal to fix an error at a *different* location than the error itself (recall Section 2). Quick Fix Scout's speculative analysis handles such situations because the analysis is global and applies to all compilation errors and proposals in the project, thus computing the information necessary to offer the global best proposal at all the relevant locations [11]. Figure 7 (the Quick Fix Scout equivalent of Figure 4) shows a global best proposal at the top of the dialog. That proposal is suggested by Eclipse at a different compilation error location, and is not displayed by the original Quick Fix.

For global best proposals, Quick Fix Scout adds the following context information:

1. The location of the error where Eclipse offers the proposal (`Motivation.java:5:17` in Figure 7).
2. The remote context that will be modified (`'sTRING'` is added to the original message `Change to 'String'` in Figure 7).

While this context information is not necessary for local proposals, it is useful when the proposal is displayed at a different location than the error to which it directly applies. For example, a developer may interpret `Change to 'String'`

[2] `http://blog.deepakazad.com/2012/03/jdt-3842-m6-new-and-noteworthy.html`

[3] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=338983` and `https://bugs.eclipse.org/bugs/show_bug.cgi?id=339181`

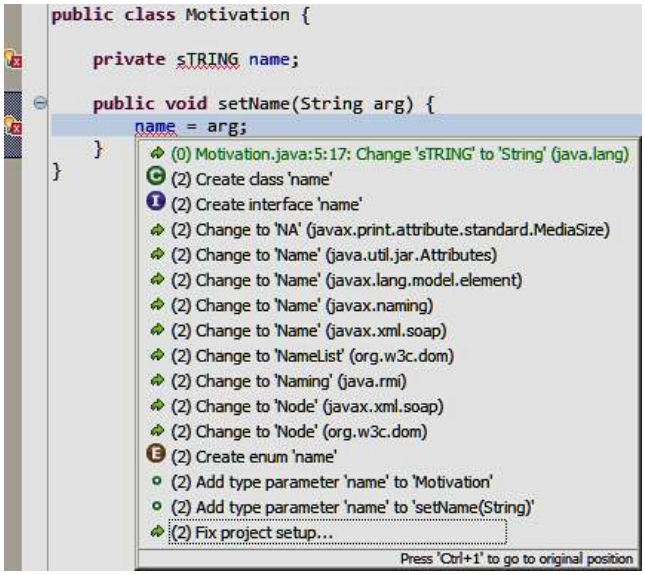

Figure 7. Quick Fix Scout computes the global best proposal for each compilation error and adds it to the Quick Fix dialog for that error. The addition of the associated error location (`Motivation.java:5:17`) and the associated error context (`'sTRING'`) distinguish global best proposals from normal proposals. If the global best proposal is already one of the local proposals, Quick Fix Scout makes no additions.

incorrectly, without knowing what token, and on what line, will be changed to `'String'`.

As a consequence of the above process, global best proposals are only shown if they resolve the local error, among other errors. While it is possible to augment the dialogs of all errors with the proposal that resolves the most errors in the project overall, we believe that showing a fix for an unrelated error might confuse developers. However, if invoked on a location without a warning or a compilation error, Quick Fix Scout does show the proposal that resolves the most errors (Figure 8).

One of the controlled experiment (Section 5.2) participants articulated the usefulness of global best proposals:

> "[Global best proposals] were great, because honestly the source of error is often not at the [location where I invoke Quick Fix]."

5. Evaluation

Our evaluation was based on two activities. First, over a roughly one-year period, we distributed a version of Quick Fix Scout to a collection of 13 friendly users (including three of the authors) and gathered information about their Quick Fix and Quick Fix Scout behavior in their normal workflow (Section 5.1). Second, we ran a controlled experiment with a within-participants mixed design across 20 participants, asking them to resolve various compilation errors on code they had not previously seen (Section 5.2).

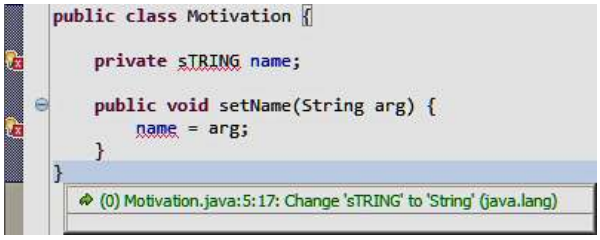

Figure 8. If invoked on a location without a warning or a compilation error, Quick Fix Scout shows the proposals that resolve the most errors whereas the default implementation would only inform the user that there are no available proposals for that location.

The friendly users selected, at their discretion, to use either Quick Fix or Quick Fix Scout during each logged session. The design of the controlled experiment determined the situations in which participants used Quick Fix and Quick Fix Scout.

For both activities, we acquired data with an instrumented version of the tool. The tool logs:

- whether Quick Fix or Quick Fix Scout is is running,
- the proposals offered by Quick Fix or Quick Fix Scout,
- whether the user selected a Quick Fix proposal or canceled the invocation,
- which proposal the user selected, if any, and
- how long it took the user to either make a selection or cancel the invocation.

The tool also tracks information that lets us detect some situations in which a user applies a proposal but soon after undoes that proposal.

5.1 Case study: friendly users

The goal of our informal case study was to understand how Quick Fix is used "in the wild" by developers. We wished to investigate the following questions:

- Does the ordering of the displayed proposals affect which proposal is selected?
- Does the number of proposals displayed affect which proposal is selected?
- Does the kind of proposal displayed affect which proposal is selected?

5.1.1 Case study design

Over approximately one year, 13 developers — including three of the authors — ran our tool and allowed us to view its logs. For each Eclipse session, each participant was free to use either the standard Quick Fix or our Quick Fix Scout; all sessions were logged.

	Standard Quick Fix					Quick Fix Scout				
User ID	# completed sessions	QF selection rate 1st	2nd	3rd	top 3	# completed sessions	QF selection rate 1st	2nd	3rd	top 3
1	4	100%	0%	0%	100%	1	100%	0%	0%	100%
2	0	—				1	100%	0%	0%	100%
3⋆	45	64%	16%	13%	93%	362	81%	15%	2%	98%
4	167	78%	20%	1%	99%	0	—			
5	17	47%	24%	0%	71%	0	—			
6⋆	25	40%	24%	8%	72%	22	55%	27%	0%	82%
7⋆	82	70%	22%	2%	94%	28	68%	18%	0%	86%
8	9	67%	22%	11%	71%	0	—			
9	7	71%	0%	0%	71%	10	60%	10%	10%	80%
10	6	33%	17%	33%	83%	0	—			
11	0	—				0	—			
12	6	17%	0%	17%	34%	0	—			
13	0	—				2	50%	0%	0%	50%
All	368	69%	20%	4%	93%	426	78%	15%	2%	95%

Figure 9. Case study information. A ⋆ in the User ID indicates the participant is an author of this paper. Completed sessions are the number of times the user invoked Quick Fix and selected a proposal. For each of the first three proposals in the Quick Fix menu, we report how often that proposal was selected. For example, user 9 never selected the second or third offered proposal from a standard Quick Fix menu, but did so when using Quick Fix Scout.

5.1.2 Case study results

Figure 9 shows that users selected the first (top) proposal 70% of the time, one of the top two proposals 90% of the time, and one of the top three proposals 93% of the time. For Quick Fix Scout sessions, the percentages are slightly higher, at 78%, 93%, and 95%. Given the small difference, and that three of the participants are authors, this data does not confirm a hypothesis that Quick Fix Scout is different from Quick Fix in this dimension.

For the completed sessions, Quick Fix offered as many as 72 (mean=5.7, median=2) proposals. For the canceled sessions, Quick Fix offered as many as 80 (mean=6.4, median=4) proposals. In contrast, for the completed sessions, Quick Fix Scout offered as many as 38 (mean=4.2, median=2) proposals. For the canceled sessions, Quick Fix Scout offered as many as 27 (mean=5.1, median=3) proposals. These data may suggest that when a user does not find an expected or a useful proposal easily, the invocation is more likely to be canceled. To investigate this further, we looked for a correlation between the number of proposals offered in the dialog and the session completion rate. We found no such correlation, further suggesting that as long as the correct proposal is located near the top of the list, the number of proposals shown might not have an effect on developers' decisions.

Eclipse documentation categorizes the Quick Fixes (see the column headings in Figure 10).[4] Five out of the nine proposal types represent 92% of all selected proposal types.

Figure 11 presents the most-frequently selected proposals and their selection ratios. Except for one user, these six proposals constitute about 80% of the selected proposals. Note the similarity in selection ratio between the proposals "Import ...", "Add Throws Declaration", and "Add Unimplemented Methods" and their types "Types", "Exception Handling", and "Constructor" respectively. The "Change to ..." proposal falls into "Methods" and "Fields & Variable", depending on its recipient.

Though there is some variation between participants, the results suggest that all proposals *do not* have the same importance: there are a few proposals that are favored by the developers. This observation can be explained by the nature of these proposals. For example, the "Import ..." proposal is offered whenever the developer declares an unresolvable type. If the developer makes this mistake intentionally, most of the time, she either wants to import that type or create a new type with that name. Therefore, there is a high probability that one of these proposal will be selected. "Add throws declaration" and "Surround with Try/Catch" are two proposals that are always offered for exception-handling-related compilation errors. When there is an exception-handling error, it is very likely that the developer will either propagate that exception or handle it immediately, which suggests that one of these proposals will be selected.

The imbalance in proposal selection rate can be used to improve Quick Fix by prioritizing proposals with respect to the user's history. Bruch et al. [1] have already done this for auto-complete.

[4] http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-java-editor-quickfix.htm

User ID	Types	Exception Handling	Methods	Constructors	Fields & Variables	Other	Unknown	Package Declaration	Imports	Build Path Problems
1	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
2	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
3	26%	22%	29%	11%	9%	1%	3%	0%	0%	0%
4	2%	65%	11%	10%	1%	5%	2%	0%	4%	0%
5	94%	0%	0%	0%	6%	0%	0%	0%	0%	0%
6	51%	19%	17%	0%	2%	0%	6%	0%	4%	0%
7	49%	0%	13%	9%	14%	7%	3%	5%	0%	1%
8	44%	0%	0%	11%	33%	11%	0%	0%	0%	0%
9	59%	18%	6%	6%	12%	0%	0%	0%	0%	0%
10	67%	0%	0%	0%	33%	0%	0%	0%	0%	0%
11	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
12	0%	0%	83%	0%	17%	0%	0%	0%	0%	0%
13	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
All	30%	27%	21%	10%	8%	3%	3%	3%	1%	1%

Figure 10. Proposal types and their selection ratios during the case study. The proposals whose type was unclear are listed as "Unknown".

5.2 Controlled experiment: graduate students

The goal of our controlled experiment was to determine whether users behave differently when using Quick Fix and when using Quick Fix Scout.

Each participant performed two sets of tasks — α and β task sets — of 12 tasks each. Each task presented the participant with a program that contained at least two compilation errors and required the participant to resolve all the compilation errors. The non-compilable program states were chosen randomly from the real development snapshots captured during the case studies from Section 5.1. For 6 of the tasks in each task set, we manually seeded each task with either 1 or 2 additional mutation errors, such as changing a field type or a method signature. The mutations introduced an average of 2.8 extra compilation errors per task.

Our study answers two research questions:

RQ 1: Does the additional information provided by Quick Fix Scout — specifically, the count of remaining compilation errors, and the coloring and reordering of proposals — allow users to remove compilation errors more quickly?

RQ 2: Does Quick Fix Scout affect the way users choose and use Quick Fix proposals?

5.2.1 Controlled experiment design

We recruited 20 participants, all graduate students who were familiar with Quick Fix but had never used Quick Fix Scout.[5]

We used a within-participants mixed design. We considered two factors: the tool or treatment factor (Quick Fix vs. Quick Fix Scout), and the task factor (α vs. β task sets). To reduce the confounding effects from developer differences and learning effects, we defined four blocks — the cross-product of the two factors. We used a balanced randomized block protocol, randomly selecting which participants perform which block with a guarantee that each block is performed an equal number of times. (We rejected a full within-participants factorial design because of the learning effects we would anticipate if a participant performed the same set of tasks twice using Quick Fix and then Quick Fix Scout or vice versa.)

Each participant received a brief tutorial about Quick Fix Scout, performed the two blocks (task sets), and took a concluding survey comparing Quick Fix Scout and Quick Fix around the two blocks. The two blocks differed in both the tool/treatment factor (from Quick Fix to Quick Fix Scout, or vice versa) and also the task factor (from the α task set to the β task set, or vice versa).

To answer **RQ 1**, we measured the time it took participants to complete tasks. In addition to the time per task group (α and β), we calculated per-task time by using the screen casts. The beginning of a task is defined to be the time when the participant opens the related project for the first time and the end of a task is defined to be the time when the participant resolved all compilation errors in the task and was satisfied with the implementation.

To answer **RQ 2**, we measured whether the user selected a proposal after invoking the Quick Fix menu or canceled the menu, how long it took the user to make that decision, which proposal the user selected, and whether the user undid a selected proposal.

5.2.2 Controlled experiment results

We used R to perform a 4-way blocked MANOVA test utilizing all independent and dependent variables. This minimizes the risk of a type 1 statistical error. All independent variables (user, Quick Fix vs. Quick Fix Scout, task, and order of

[5] Approved human subject materials were used; participants were offered a $20 gift card.

User ID	Import ...	Add Throws Declaration	Create Method ...	Change to ...	Add Unimplemented Methods	Surround with Try/Catch	Total
1	100%	0%	0%	0%	0%	0%	100%
2	100%	0%	0%	0%	0%	0%	100%
3	24%	21%	21%	10%	7%	0%	83%
4	2%	47%	11%	1%	8%	14%	83%
5	76%	0%	0%	6%	0%	0%	82%
6	34%	11%	2%	26%	0%	6%	79%
7	37%	0%	11%	7%	9%	0%	64%
8	44%	0%	0%	33%	11%	0%	88%
9	53%	18%	0%	24%	6%	0%	100%
10	50%	0%	0%	50%	0%	0%	100%
11	0%	0%	0%	0%	0%	0%	0%
12	0%	0%	0%	83%	0%	0%	83%
13	0%	0%	0%	0%	0%	0%	0%
All	25%	23%	15%	10%	7%	4%	84%

Figure 11. Most-frequently selected proposals and their selection ratios for the case study. Proposals that are selected less than 3% overall (represented by the "All" row) are excluded.

treatment type		1st treatment	2nd treatment	all treatments
α	QF	27m	19m	23m
	QFS	17m	15m	16m
β	QF	31m	22m	27m
	QFS	36m	21m	29m
QF		29m	20m	25m
QFS		26m	18m	22m

Figure 12. Mean time to remove compilation errors, in minutes.

task) had statistically significant effects, so we examined the analysis-of-variance results of the MANOVA test.

RQ 1 Participants completed tasks 10% faster, on average, when using Quick Fix Scout than Quick Fix (Figure 12). However, this result was not statistically significant (p=.11).

All the other independent variables did have statistically significant effects on task completion time: user ($p=5\times10^{-7}$), task ($p=2\times10^{-16}$), and order ($p=3\times10^{-6}$).

Even the task group had an effect ($p=3\times10^{-5}$): tasks in the β group were harder, and in fact five participants could not complete all tasks in β. We had not anticipated any difference between the task groups. Future work should investigate how the β tasks differ from the α tasks and why Quick Fix Scout caused a slight (but not statistically significant) slowdown on the β tasks. Per-task descriptive statistics appear in Figure 13.

There is a learning bias ($p=4\times10^{-8}$): the participants completed a task set 22% faster if it was their second task set. Possible explanations for this bias include participants getting used to resolving compilation errors and participants becoming familiar with the code (since multiple tasks were drawn from the same development projects).

RQ 2 Figure 14 summarizes the data we collected regarding user behavior with respect to Quick Fix.

Use of Quick Fix Scout improved the best proposal selection rate by 14% ($p=10^{-8}$). This increase and the frequent (75%) usage of global best proposals suggest that the participants were resolving more compilation errors per Quick Fix invocation with Quick Fix Scout. Though the difference between the total number of Quick Fix invocations is low (36) between treatments, we believe that the Quick Fix Scout increased usefulness of completed Quick Fix invocations, which helped the participants to save time overall. One participant noted:

> "With [Quick Fix Scout] I had a much better idea of what the error was... I found [Quick Fix] to be more vague..."

Use of Quick Fix Scout increased by .8 seconds the time spent selecting a proposal (p=.004). Possible explanations for this phenomenon include that (1) the Quick Fix Scout dialog contains extra information that the participants took extra time to process, and (2) Quick Fix Scout may take time to compute causing the participants to wait for the information to appear. Explanation (1) also explains the overall productivity improvement. If the participant gets enough information from the dialog, she could resolve the error without having to investigate the related code. Supporting this hypothesis, half of the participants agreed that they needed to type more manually — instead of using Quick Fix proposals — to resolve compilation errors when not using Quick Fix Scout (Figure 17).

Use of Quick Fix Scout did not have any other statistically significant effects. This stability between treatments strengthens our hypothesis that Quick Fix Scout did not change the way participants used Quick Fix, rather the extra information

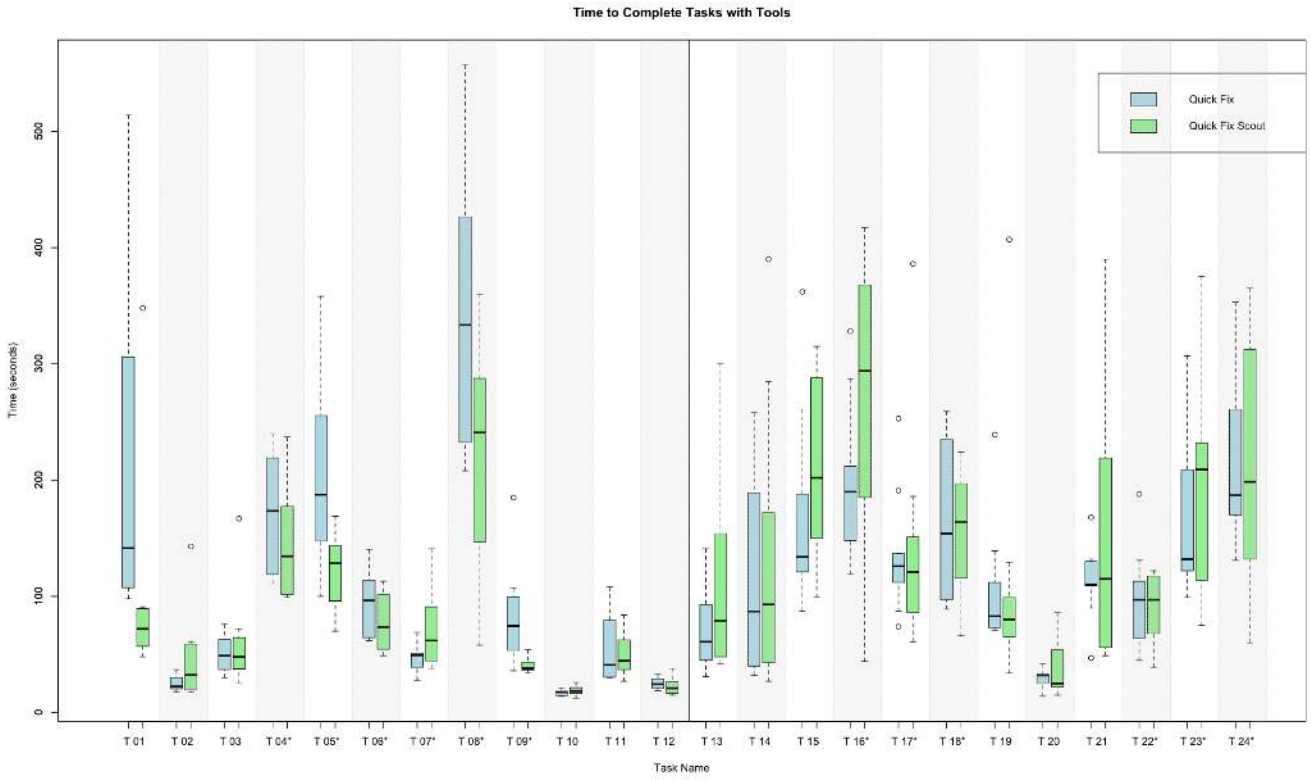

Figure 13. Median, minimum, and maximum time spent to complete each task by participants in seconds with and without Quick Fix Scout. The first 12 tasks make up the α set and the last 12 tasks make up the β set. The tasks with seeded errors are followed by an asterisk. Outliers are represented as small circles.

provided by Quick Fix Scout increased participants' understanding of the code and helped them make better decisions. One participant noted:

> "It was pretty apparent after using regular Quick Fix second, that [Quick Fix] Scout sped things up. I got frustrated as I'd have to scan from error to error to fix a problem rather than just go to the first error I saw. What's more, I had to spend more time staring at the [Quick Fix dialog] often to find that there was nothing relevant."

The data, the analysis, and the qualitative insights from the case study and controlled experiment participants suggest that **RQ 2** holds: Quick Fix Scout indeed changes the way in which users choose and use Quick Fix proposals. We have not teased out which aspects of Quick Fix Scout have the most influence.

6. Threats to Validity

We assess our evaluation activities in terms of simple characterizations of internal and external validity. Internal validity refers to the completeness and the correctness of the data collected through the experiments. External validity refers to the generalizability of our results to other settings.

One threat to internal validity is that, due to implementation difficulties, we log all Quick Fix invocations *except* those invoked through the Hover Dialog. We tried to limit this threat by rejecting participants who indicated that they consistently use Hover Dialog for invoking Quick Fix and by mentioning this restriction to accepted participants, recommending that they invoke Quick Fix in a different way. So the data we logged about invocations is accurate, although it may be incomplete.

Another threat to internal validity is in our computation of which proposal resolves the most errors. Since the developer might complete a Quick Fix invocation before the speculation computation completes, and because some proposals are omitted *a priori* (for example, a "Create class" proposal), we may not always log the number of compilation errors that would remain for every Quick Fix proposal. In some cases, these omitted proposals could resolve more compilation errors than the ones we identify as resolving the most errors. In our case study, only 6% of all completed Quick Fix invocations are completed before the speculative analysis finishes. Further, in our case study, none of the users selected instances of the *a priori* omitted proposals.

In addition to common external validity threats (such as having students rather than professional developers as

treatment		# invocations (invs.)	undone invs. rate	+invs. rate	avg. time +invs.	avg. time -invs.	1st prop. rate	2nd prop. rate	3rd prop. rate	BP rate	GBP rate
α	QF	554	17%	58%	3.0s	6.8s	79%	18%	0%	76%	
	QFS	449	10%	68%	4.6s	8.8s	79%	16%	0%	90%	79%
β	QF	572	13%	56%	4.3s	7.9s	73%	20%	2%	71%	
	QFS	631	17%	55%	4.4s	6.8s	71%	22%	2%	85%	67%
QF		1116	15%	57%	3.7s	7.4s	76%	19%	1%	73%	
QFS		1080	14%	60%	4.5s	7.4s	75%	19%	1%	87%	75%

Figure 14. Quick Fix and Quick Fix Scout invocation duration and proposal selection rate results. Invocations that were immediately undone by the participant are excluded. + and - invocations are ones for which the participant selected and did not select a proposal, respectively. For each treatment, we report the rates with which participants chose the 1st, 2nd, and 3rd proposal, as well as the best (BP) and global best proposals (GBP). Best proposals are defined as the proposals that resolve the highest number of compilation errors for a given Quick Fix invocation.

participants), a key threat is the decisions we made about which programs to use in the controlled experiment:

- Using small programs with multiple compilation errors.
- Using snapshots from the case study participants to populate our tasks in the controlled experiment.
- Adding seeded errors to half of the snapshots using mutation operators, as well as using a specific set of mutation operators.

Although there are strong motivations for each of these decisions in our experimental design, they could still, in principle, lead to inaccurate conclusions about how Quick Fix Scout would work if it were broadly distributed and used.

7. Related Work

The interest in software recommendation systems — "software ... that provides information items estimated to be valuable for a software engineering task in a given context" [15] — has grown over the past few years, with an increasing number of research results and tools, as well as an ongoing workshop [6].

Work in recommendation systems includes: defining recommendations for new domains, such as requirements elicitation [5] and team communication [17]; frameworks for defining recommendation systems [9]; and techniques for choosing recommendations to include in a system, such as data mining [16].

Some of efforts that are more directly relevant to Quick Fix Scout also aim to improve IDE recommendations. Robbes and Lanza propose eight different ways to reorder code-completion recommendations; they evaluated these techniques on a realistic benchmark, and show that reordering the matches based on historical usage provides the greatest improvement [14].

Bruch et al. reorder and filter the Eclipse auto-complete dialog using mined historical data and developer usage habits [1]. Recently, Perelman et al. showed that Visual Studio auto-complete (IntelliSense) can be improved by using developer-supplied partial types, to search all APIs for auto-completions that would transform the input type to the expected output type [13].

In contrast to the first two of these approaches, which rely on past usage patterns, Quick Fix Scout reorders recommendations based on information about properties of the program that will be created if a recommendation is selected. In contrast to the third approach, the developer need not add any information to the program for Quick Fix Scout (or, of course, Quick Fix) to work. In addition, we have recently shown how to improve IDE recommendations by considering the interactions between existing recommendations [11]; this is a key motivation for global best proposals (Section 4).

In addition to industrial efforts related to Quick Fix,[6] some research efforts address various aspects of Quick Fix. For example, a paper on automatic refactoring in the face of problems such as references to unavailable declarations mentions an experimental participant's idea to augment the system with invocations to Quick Fix [8]. As another example, a recommendation system approach to increasing reuse also suggests integrating their system through Quick Fix [7].

Quick Fix Scout is built on speculative analysis: a technique that computes precise information about likely future states of a program and presents this information to the developer so that she can make better and more informed decisions [2]. Applying speculative analysis on collaborative software development [4], we built Crystal [3]: a tool that notifies developers as soon as a conflict emerges. The biggest difference between Crystal and Quick Fix Scout is the granularity of the speculation. For collaboration conflicts, it is acceptable if the developer is notified after the conflict emerges since she would still be able to find the reason of the conflict and coordinate it. As a result, Crystal does not have to work on the most recent copy of the project and might report results with some delay. However, when a developer invokes Quick Fix, she needs the results for the recent version of the project as the results from any previous version is not acceptable and

[6] http://eclipse.org/recommenders

User ID	Status in the Department	Java Knowledge (Years)	Eclipse Knowledge (Years)	QF Usage Frequency	Keyboard Shortcut	Quick Fix Icon	Context Menu	Hovering
CS01	4th year BS	6	3	2	✔		✔	✔
CS02	1st year PhD	4	1.5	1	✔			
CS03*	2nd year PhD	7	7	4	✔	✔		
CS04	1st year Post-doc	13	9	3	✔	✔		
CS05	6th year PhD	2	2	3	✔	✔	✔	✔
CS06*	Assistant Prof.	11	4	4		✔	✔	✔
CS07*	Assistant Prof.	12	10	4	✔	✔		
CS10	5th year PhD	7	3	2	✔		✔	✔
CS12	3rd year PhD	7	6	4		✔		✔
CE01	1st year PhD	3	3	3		✔	✔	✔
CE02	2nd year PhD	11	5	3	✔	✔	✔	✔
CE03	2nd year PhD	10	10	3				✔
CE04	5th year PhD	10	9	1		✔		✔
CE05	1st year PhD	3	2	2	✔			✔
CE06	2nd year PhD	1	1	1			✔	
CE07	2nd year PhD	2	2	1		✔		✔
CE08	3rd year PhD	10	8	3	✔		✔	✔
CE09	2nd year PhD	8	2	3		✔		✔
CE10	2nd year PhD	5	5	2		✔		✔
CE11	2nd year PhD	5	5	2		✔	✔	
CE12	3rd year PhD	3	2	1			✔	✔
CE13	6th year PhD	10	3	3	✔			✔
CE14	2nd year PhD	6.5	6.5	4	✔	✔		✔
CE15	1st year PhD	7	3	2		✔		✔
CE16	2nd year PhD	6	7	1	✔	✔		
CE17	3rd year PhD	3	1	1				✔
CE18	2nd year PhD	8	3	1		✔	✔	✔
CE19	3rd year PhD	4	2	3				✔
CE20	2nd year PhD	4	4	3		✔	✔	✔

Figure 15. Case study and controlled experiment participants familiarity with Java, Eclipse and Quick Fix. "CS" and "CE" prefixes in user id represent case study and controlled experiment participants respectively. QF Usage Frequency is how frequently the participants use Quick Fix on a scale from 0 to 4, 0 meaning never and 4 meaning very frequently. A check mark in the last four columns represent that the participant prefers Quick Fix using the method in the column header. Case study participants # 8, 9, 11, and 13 are not shown since they have not completed our survey.

actionable. In addition, the developers want to see the results as soon as the Quick Fix dialog is created since it takes a couple of seconds for them to decide what to choose.

8. Contributions

Quick Fix Scout is an enhancement of Eclipse's standard Quick Fix that computes and reports to the user the number of compilation errors that would remain in the program for each Quick Fix proposal. Our prototype Eclipse plug-in addresses issues ranging from challenges in the user interface (additional information must be presented in roughly the same space used by the Quick Fix dialog) to challenges in keeping a background copy of the developer's code in sync with the dynamically changing code (Quick Fix Scout uses a copy to speculatively apply the proposals). We evaluated Quick Fix Scout in two ways: an informal case study of how a set of friendly users use both Quick Fix and Quick Fix Scout in their own work, and a 20-user, within-subjects, mixed-design controlled experiment that compares Quick Fix and Quick Fix Scout. Users fixed compilation errors 10% faster, on average, when using Quick Fix Scout, although this improvement was not statistically significant.

We are considering several improvements to Quick Fix Scout. First, if errors remain after applying a proposal, Quick Fix Scout can apply the approach repeatedly until all compilation errors are resolved or no progress can be made. Such multiple-ply speculation can identify proposals that resolve few errors but lead Quick Fix to generate new proposals that may resolve more errors. One goal of the approach would be to produce the smallest set of consecutive proposal appli-

Question	S. Agree	Agree	Neutral	Disagree	S. Disagree	N/A
Number of remaining compilation errors was helpful.	4	13	2	1	0	0
Reordering of proposals was helpful.	7	13	0	0	0	0
Coloring of proposals was helpful.	5	7	5	0	0	3
I liked the user experience provided by Quick Fix Scout.	7	12	1	0	0	0

Figure 16. The four-question survey, and a summary of the participants' responses, administered after each participant used Quick Fix Scout. "S. Agree" (resp. Disagree) represents "Strongly Agree" (resp. Disagree).

Question	Both	Quick Fix Scout	Quick Fix	Neither
Quick Fix (Scout) is helpful when resolving compilation errors.	19	1	0	0
There was no performance issues before Quick Fix dialog is updated.	10	1	2	7
For some tasks, I undid a proposal when using Quick Fix (Scout).	12	1	7	0
I manually resolved errors more often with Quick Fix (Scout).	2	0	10	8

Figure 17. The four-question survey, and a summary of the participants' responses, administered at the end of each participant's experiment session.

cations that resolves all, or the most possible, errors. This improvement raises several performance concerns, as the search space of proposals may be large. Second, Quick Fix Scout can use more-complex analyses to identify the consequences of possible future states. For example, if each of multiple proposals removes all compilation errors, Quick Fix Scout can speculatively run tests to determine which proposal makes the most tests pass. This information would likely allow users to make better decisions. As another example, successfully-compiling code might still cause version control conflicts; Quick Fix Scout could integrate our approach to proactively identify version control conflicts [4]. Again, these analyses may be computationally intensive and raise performance concerns for Quick Fix Scout.

The use of speculative analysis in software development is promising but full of technical challenges. Quick Fix Scout is an exemplar for speculative analysis. The underlying environment (Eclipse's Quick Fix) defines the actions that generate likely future states, and the computation of consequences is made efficient by Eclipse's incremental compiler. Finding other domains, and other implementation techniques, that provide an effective balance between performance and information to usefully guide developers, is a difficult but worthwhile effort.

Acknowledgments

We thank Deepak Azad and Dani Megert for explaining, on the Eclipse JDT forum, the internal Eclipse API and several implementation details. We thank Daniel Perelman, Colin Gordon, and Ivan Beschastnikh for piloting our experiment. We especially acknowledge the detailed and pertinent reviews that helped us significantly improve the paper.

This material is based upon work supported by the Bradley Chair in Computer Science & Engineering, the National Science Foundation under Grants CNS-0937060 to the Computing Research Association for the CIFellows Project and CCF-0963757, and by Microsoft Research through a Software Engineering Innovation Foundation grant.

A. Appendix

This section presents additional details of the data gathered during the experiments. Figure 15 summarizes the participants' familiarity with Java, Eclipse, and Quick Fix. Figures 16 and 17 summarize the surveys given to the controlled experiment participants.

References

[1] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE09)*, pages 213–222, Amsterdam, The Netherlands, 2009. doi: 10.1145/1595696.1595728.

[2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: Exploring future states of software. In *Proceedings of the 2010 Foundations of Software Engineering Working Conference on the Future of Software Engineering Research*, FoSER '10, Santa Fe, NM, USA, November 2010. doi: 10.1145/1882362.1882375.

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Crystal: Proactive conflict detector for distributed version control. http://crystalvc.googlecode.com, 2010.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Confer-*

ence and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '11, pages 168–178, Szeged, Hungary, September 2011. doi: 10.1145/2025113.2025139.

[5] C. Castro-Herrera, C. Duan, J. Cleland-Huang, and B. Mobasher. A recommender system for requirements elicitation in large-scale software projects. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1419–1426, 2009. doi: 10.1145/1529282.1529601.

[6] R. Holmes, M. Robillard, R. Walker, T. Zimmermann, and W. Maalej. International Workshops on Recommendation Systems for Software Engineering (RSSE). `https://sites.google.com/site/rsseresearch`, 2012.

[7] W. Janjic, D. Stoll, P. Bostan, and C. Atkinson. Lowering the barrier to reuse through test-driven search. In *Proceedings of the 2009 31st International Conference on Software Engineering Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 21–24, 2009. doi: 10.1109/SUITE.2009.5070015.

[8] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 726–738, 2010. doi: 10.1145/1869459.1869518.

[9] F. M. Melo and Á. Pereira Jr. A component-based open-source framework for general-purpose recommender systems. In *Proceedings of the 14th International ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '11, pages 67–72, 2011. doi: 10.1145/2000229.2000239.

[10] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Quick Fix Scout. http://quick-fix-scout.googlecode.com, 2010.

[11] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Improving IDE recommendations by considering global implications of existing recommendations. In *Proceedings of the 34th International Conference on Software Engineering, New Ideas and Emerging Results Track*, ICSE '12, Zurich, Switzerland, June 2012. doi: 10.1109/ICSE.2012.6227082.

[12] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006. doi: 10.1109/MS.2006.105.

[13] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *Proceedings of Programming Language Design and Implementation*, PLDI '12, Beijing, China, June 2012. doi: 10.1145/2254064.2254098.

[14] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, L'Aquila, Italy, 2008. doi: 10.1109/ASE.2008.42.

[15] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27:80–86, 2010. doi: 10.1109/MS.2009.161.

[16] K. Schneider, S. Gärtner, T. Wehrmaker, and B. Brügge. Recommendations as learning: From discrepancies to software improvement. In *Proceedings of the International Workshop on Software Recommendation Systems*, RSSE '12, pages 31–32, 2012. doi: 10.1109/RSSE.2012.6233405.

[17] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: a recommendation tool for promoting communication in software teams. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, RSSE '08, pages 2:1–2:1, 2008. doi: 10.1145/1454247.1454259.