# Improving IDE Recommendations by Considering Global Implications of Existing Recommendations

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

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

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

***Abstract*—Modern integrated development environments (IDEs) offer recommendations to aid development, such as auto-completions, refactorings, and fixes for compilation errors. Recommendations for each code location are typically computed independently of the other locations. We propose that an IDE should consider the whole codebase, not just the local context, before offering recommendations for a particular location. We demonstrate the potential benefits of our technique by presenting four concrete scenarios in which the Eclipse IDE fails to provide proper Quick Fixes at relevant locations, even though it offers those fixes at other locations. We describe a technique that can augment an existing IDE's recommendations to account for non-local information. For example, when some compilation errors depend on others, our technique helps the developer decide which errors to resolve first.**

## I. Introduction

Modern integrated development environments (IDEs) offer contextual information that can simplify common developer tasks. For example, the auto-complete mechanism recommends suffixes that a developer may use to complete partial variable or method names. As another example, Eclipse's Quick Fix recommends ways to resolve compilation errors.

IDEs make these recommendations independently at each location: the recommendations at one code location do not affect which recommendations are displayed at the others. However, applying a recommendation may affect other parts of the program. For example, in a project with multiple compilation errors that result from a single incorrect type declaration, fixing the declaration resolves the errors at all the other locations. The fix for this error is unique: no local recommendations at the other error locations will resolve all the errors.

We propose an approach that uses global information to improve Quick Fix recommendations. Our approach answers two kinds of developer questions:

1) "I want to resolve this error. Where should I look?"
2) "My project has multiple errors. Where should I start?"

Our approach creates no new recommendations. Rather, it augments the IDE by presenting its existing recommendations at different, appropriate code locations. Our approach is a general one that can be layered on top of any program analysis that makes concrete recommendations to the programmer.

The key idea is to precompute which errors each code recommendation resolves. To answer "Where should I look to resolve a particular error?", the IDE (augmented by our technique) tells the developer which code recommendations, regardless of their locations, resolve that error. To answer "Where should I start resolving errors?", the IDE tells the developer which code recommendations resolve the most errors and thus are likely to be good starting points.

Our vision is that IDE recommendations can be improved by considering a global view of the project and removing the common IDE assumption that errors are independent. While the independence assumption may improve the speed of displaying recommendations, we focus on programmer productivity instead. We hypothesize that global analyses can improve the relevance of Quick Fix recommendations, enabling developers to resolve compilation errors faster. Our approach preserves responsiveness by displaying existing recommendations immediately, then adding the results of global analysis as they become available.

The remainder of this paper is organized as follows. Section II presents four code examples for which our approach provides better results than today's IDEs. Section III investigates some ad hoc solutions for problems described in Section II and points out their limitations. Section IV describes a proof-of-concept tool that implements our approach. Section V compares our approach with related work. Finally, Section VI concludes.

## II. Compilation-Error Dependencies

Sometimes, in order to resolve a compilation error, one must first resolve another error at a remote location. In such cases, we say the compilation error depends on the remote location error. When the dependency is unidirectional, Eclipse only offers the fix at one location, instead of both.

This section presents four Java examples of such compilation error dependencies. In each case, the approach we will describe in Section IV can improve the recommendations at the locations of the dependent errors.

### A. Unresolvable Type Declaration

In Figure 1, the variable `name` is declared with type `string` instead of `String`. If the developer invokes Quick Fix where `name` is assigned a `String` value, none of the Eclipse recommendations resolve either of the errors. However, if the devel-

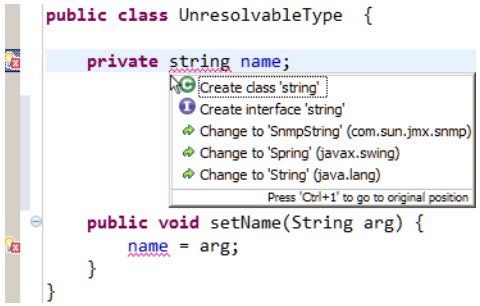


Figure 1. An unresolvable type declaration `string` causes two compilation errors. At the second error location, Quick Fix fails to recommend changing `string` to `String`.

oper invokes Quick Fix at the declaration error, one Eclipse recommendation is to change `string` to `String`, which eliminates both errors.

The developer bears two burdens: recognizing that the two errors are related and can be resolved at a single location, and recognizing that the second error cannot be resolved by the local recommendations. While in this example the two errors are close together, they could just as easily be in different classes and packages, making their relationship more difficult for the developer to understand.

### B. Undeclared Exception Throw

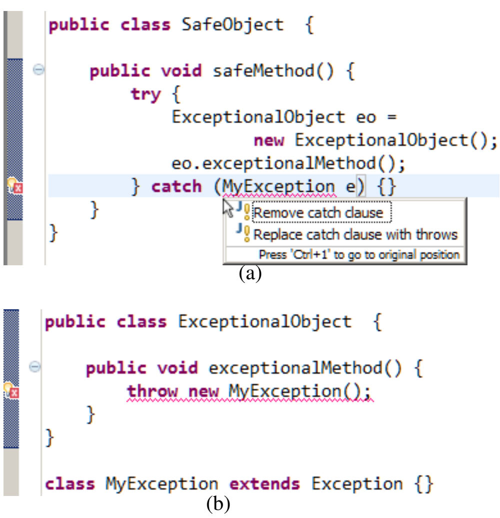


Figure 2. A missing `throws` declaration causes two compilation errors. At the first error location, Quick Fix fails to recommend declaring `exceptionalMethod()` to throw `MyException`.

Java requires that a *checked* exception be declared by all methods that throw the exception, and forbids catching an undeclared checked exception. The `catch`-clause error in Figure 2(a) needs to be resolved by adding a `throws` statement to the `exceptionalMethod` declaration. However, invoking Quick Fix at the `catch` clause will not recommend this change. Quick Fix recommends the desired change only where `exceptionalMethod` is declared. The locations of the `catch` clause and the method declaration are likely to be distant, obscuring their relationship from the developer.

### C. Abstract Method in a Concrete Class

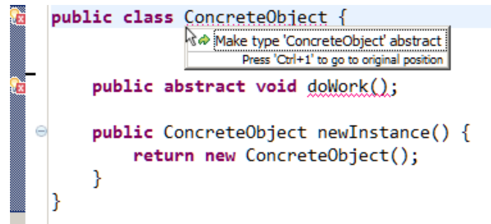


Figure 3. A missing method implementation causes two compilation errors. At the first error location, Quick Fix fails to recommend implementing the body of `doWork()`.

Declaring an abstract method within a concrete class causes two compilation errors: one for the class declaration and one for the method declaration, as in Figure 3. Eclipse's only recommendation for the class declaration is to make the class abstract. In the example, this recommendation resolves both errors but introduces a new compilation error where the class is instantiated (see the bottom of Figure 3). A better fix is to implement a body for `doWork`, which resolves all compilation errors without introducing any new ones. Eclipse makes this recommendation at the location where the method is declared, but not to a developer who is examining the class declaration.

### D. Multiple Inheritance

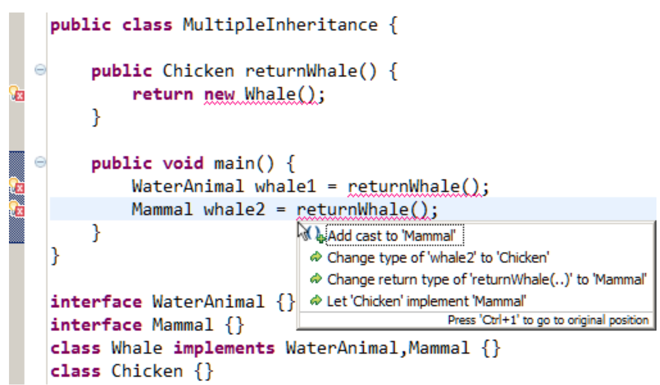


Figure 4. An incompatible return type causes three compilation errors. At the second and third error locations, Quick Fix fails to recommend changing the return type declaration of `returnWhale()` from `Chicken` to `Whale`.

A `return` statement that is incompatible with its method's type results in a compilation error. If the correct return type implements multiple interfaces not implemented by the incorrect return type, uses of the method expecting those interfaces will also result in compilation errors. For example, in Figure 4, the method `returnWhale` returns a `Whale`, but is incorrectly declared to return a `Chicken`. In addition to the error at the `return` statement, there are compilation errors at each of the two uses of the method, one that expects a `WaterAnimal` and one that expects a `Mammal`. At each of the latter two errors, Eclipse recommends changing the return type of `returnWhale` to the type of the left-hand side in the assignment: `WaterAnimal` and `Mammal`, respectively. Each of those fixes resolves only two of the three compilation errors. However, the correct fix, which resolves all three errors, changes `Chicken` to `Whale` and is only recommended at the `return` statement location.

## III. Identifying Dependencies with Dedicated Analyses

Section II showed that Eclipse does not consistently direct developers to the right locations to resolve dependent errors.

Eclipse developers could augment the Quick Fix engine to include analyses or heuristics specific to each of these situations. For the unresolvable type example from Section II-A, Eclipse could use type inference to determine that the right-hand side of the assignment statement is of type `String`, and then use this information to offer the correct fix — changing the type of `name` to `String` — at both the declaration and the assignment statements. The undeclared exception example from Section II-B could be handled by a dataflow analysis that investigates the `try-catch` block, identifies the exceptions that are caught but not thrown inside the `try` block, and analyzes where those exceptions may by thrown. Similar analyses could be used for the examples from Sections II-C and II-D.

In contrast to writing ad hoc analyses for each possible situation, our approach is more general and exploits the existing Quick Fix infrastructure.

## IV. Identifying Dependencies with Speculative Analysis

The error scenarios in Section II have a key element in common: in each scenario, Quick Fix recommends a fix that resolves multiple compilation errors. The problems we have outlined arise because Eclipse focuses on recommending fixes that resolve a specific error, ignoring the effects on other errors.

Our approach simultaneously considers the recommendations at *all* error locations and dynamically determines which errors they affect. When Quick Fix is invoked at a dependent error, our approach identifies the dependence and the recommendation that would resolve all related errors conjointly.

Our approach uses speculative analysis [2], which applies a set of possible actions in the background and analyzes each resulting artifact. The approach:

1) applies each Quick Fix recommendation to a copy of the project,
2) compiles the resulting code, and
3) identifies which compilation errors are resolved.

The output of this process is a list, for each error, of all the recommendations that resolve that error.

Computing the errors resolved by each recommendation has three benefits:

First, the IDE can present information about each recommendation at all the locations where the developer may need it. When compilation errors depend on others, this can ease the developer's burden of identifying which errors to fix first.

Second, the IDE can determine how many errors each fix resolves. This information can improve Quick Fix recommendation ordering, by prioritizing fixes that resolve the most errors. For example, changing `string` to `String` in the code from Figure 1 resolves two errors, whereas each of the recommendations generated for the second error resolves none.

Third, even without focusing on fixing a specific error, the IDE can determine which Quick Fix recommendation resolves the most errors. For a developer unsure where to start fixing, this recommendation may indicate a good starting point.

We have built a prototype implementation of our approach in a publicly-available Eclipse plug-in called Quick Fix Scout [7]. Figure 5 shows a screenshot of Quick Fix Scout inside Eclipse. The figure demonstrates the first and second benefits outlined above: the dialog includes (1) a recommendation from another location that resolves the dialog's error, and (2) the number of compilation errors that remain after each recommendation's application.

*Limitations:* We are aware of two potential weaknesses of our approach. First, our approach only reports recommendations that Quick Fix already suggests at some location. If the root cause location does not itself have a compilation error, our approach will not be able to identify the best fix. For example, if the mistake in Figure 1 was declaring `name` to be of type `int`, rather than the unresolvable `string`, Eclipse would not recommend changing `int` to `String` because no compilation error occurs at that location. However, our approach would do no worse than today's approaches. Second, our approach assumes that the number of compilation errors a fix resolves is a measure of that fix's correctness. While this assumption often holds (e.g., in all the situations we presented in Section II), it

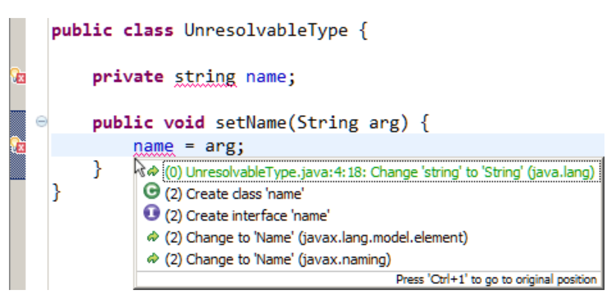


Figure 5. The first five Quick Fix recommendations for the code of Figure 1, when using the Quick Fix Scout plug-in. Quick Fix Scout has modified the Quick Fix recommendations in three ways. First, each recommendation is preceded (in parentheses) by the number of compilation errors that will remain after applying the recommendation. Second, a non-local suggestion has been added, at the top of the dialog box. Third, the recommendations are sorted by the number of remaining errors.

is possible that in some situations, Quick Fix Scout may tempt the developer with fixes that resolve more errors but are, in fact, the wrong fixes to apply.

## V. Related Work

Research and user requests continually push IDEs to improve their recommendation engines. To the best of our knowledge, these improvements have maintained the independence assumption. Bruch et al. [1] improve Eclipse's content assist (auto-complete) by taking historical data and previous users' habits into account. They show that it is possible to improve auto-complete by reordering and assigning a confidence value to the recommendations. Similarly, Hou and Pletcher [4] use historical data while extending recommendations by performing type-hierarchy filtering and grouping recommendations by functional roles. Perelman et al. [9] use partial expressions to generate well-typed, complete expressions to improve and complement Visual Studio's IntelliSense (auto-complete). These three approaches use heuristics — either historical data or types — to improve recommendation quality. In contrast, when resolving compilation errors with Quick Fix, our approach precisely determines the consequences of each recommendation at all sites in the program. In addition, our approach is robust and complementary to such IDE improvements and automatically incorporates them. For example, if Eclipse were to create new types of Quick Fix recommendations, our approach, and the Quick Fix Scout implementation, would automatically adapt to using the improvements.

Our approach relies on error-correcting compilers [5], [8] and would provide no benefit if compilers only identified a single error at a time. Although every modern compiler uses error correction, there is often a conceptual gap between what the error message reports and the solution that a developer must make to resolve the error. Seminal [6] reduces this gap by augmenting the ML type checker to identify — and to recommend, based on user-experience — corrections to obtuse error messages resulting from complex type-inference algorithms. We attempt to reduce this gap by easing the burden of identifying which errors should be resolved first. Today, developers may spend significant time attempting to understand the dependent errors' messages, and to resolve them, whereas our approach will guide the developers to resolve the underlying errors first.

Quick Fix Scout uses speculative analysis to identify recommendations relevant to each error. We have previously used speculative analysis to predict textual and higher-order conflicts in collaborative development [3], speculating over version control operations instead of Quick Fix recommendations.

## VI. Contributions

We have identified a problem with today's IDEs that offer contextual information to simplify common developer tasks: IDEs ignore dependencies between tasks and sometimes fail to make appropriate recommendations, even when the IDEs are aware of those recommendations. We have described four Java examples for which Eclipse Quick Fix fails to provide the relevant recommendations at the proper locations, and have presented an approach for improving IDE recommendations. Our approach considers all the recommendations in a codebase, computes the implications of those recommendations, and displays the recommendations at all of the relevant locations. We have built Quick Fix Scout [7], a prototype Eclipse plug-in implementation of our approach, applied to Quick Fix.

Recommendations that account for task dependencies, without requiring the developer to first explicitly and manually identify such dependencies, show promise in terms of making developers more effective. We focused here on fixing compilation errors, but our work should be applicable to many more types of recommendations. First, the independence assumption we identified is relevant to all recommendations that may affect another recommendation. Second, the speculative analysis approach we applied to Quick Fix can similarly be applied to most other recommendation engines. Future directions of research include (1) identifying other recommendation engines which can benefit from our approach, (2) developing approaches for recommendations that preclude automatic execution (such as requiring the developer to provide a new name for a class, for example), (3) and identifying tools other than recommendation engines that can benefit from the removal of an independence assumption.

## References


[1] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *ESEC/FSE*, Amsterdam, The Netherlands, 2009, pp. 213–222.

[2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis: Exploring future states of software," in *FOSER*, Santa Fe, NM, USA, November 2010.

[3] ——, "Proactive detection of collaboration conflicts," in *ESEC/FSE*, Szeged, Hungary, September 2011, pp. 168–178.

[4] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion," in *ICSM*, September 2011, pp. 233–242.

[5] E. T. Irons, "An error-correcting parse algorithm," *Communications of the ACM*, vol. 6, pp. 669–673, 1963.

[6] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *PLDI*, San Diego, CA, USA, June 2007, pp. 425–434.

[7] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Quick Fix Scout," http://quick-fix-scout.googlecode.com, 2010–2012.

[8] A. B. Pai and R. B. Kieburtz, "Global context recovery: A new strategy for syntactic error recovery by table-drive parsers," *ACM TOPLAS*, vol. 2, pp. 18–41, 1980.

[9] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *PLDI*, Beijing, China, June 2012.