# How Program History Can Improve Code Completion

Romain Robbes
REVEAL @ University of Lugano
Via G. Buffi 13, 6900 Lugano, Switzerland
romain.robbes@lu.unisi.ch

Michele Lanza
REVEAL @ University of Lugano
Via G. Buffi 13, 6900 Lugano, Switzerland
michele.lanza@unisi.ch

## Abstract

*Code completion is a widely used productivity tool. It takes away the burden of remembering and typing the exact names of methods or classes: As a developer starts typing a name, it provides a progressively refined list of candidates matching the name. However, the candidate list always comes in alphabetic order, i.e., the environment is only second-guessing the name based on pattern matching. Finding the correct candidate can be cumbersome or slower than typing the full name.*

*We present an approach to improve code completion with program history. We define a benchmark measuring the accuracy and usefulness of a code completion engine. Further, we use the change history data to also improve the results offered by code completion tools. Finally, we propose an alternative interface for completion tools.*

## 1 Introduction

In 2006, Murphy et al. published an empirical study on how 41 Java developers used the Eclipse IDE [6]. One of their findings was that each developer in the study used the code completion feature. Among the top commands executed across all 41 developers, code completion came sixth with 6.7% of the number of executed commands, sharing the top spots with basic editing commands such as copy, paste, save and delete. It is hardly surprising that this was not discussed much: Code completion is one of those features that once used becomes second nature. Nowadays, every major IDE features a language-specific code completion system, while any text editor has to offer at least some kind of word completion to be deemed usable for programming.

What is surprising is that not much is being done to advance code completion. Beyond taking into account the programming language used, there have been few documented efforts to improve completion engines. This does not mean that code completion cannot be improved, far from it: The set of possible candidates (referred from now on as suggestions or matches) returned by a code completion engine is often inconveniently large, and the match a developer is actually looking for can be buried under several irrelevant suggestions. If spotting it takes too long, the context switch risks breaking the flow the developer is in.

Language-specific completion engines can alleviate this problem as they significantly reduce the number of possible matches by exploiting the structure or the type system of the program under edition. However, if an API is inherently large, or if the programming language used is untyped, the set of candidates to choose from will still be too large. Given the limitations of current code completion, we argue that there are a number of reasons for the lack of work being done to improve it:

1. There is no obvious way to improve language-dependent code completion: Code completion algorithms already take into account the structure of the program, and if possible the structures of the APIs the program uses. To improve the state of the art, additional sources of information are needed.

2. Beyond obvious improvements such as using the program structure, there is no way to assert that a completion mechanism is "better" than another. A standard measure of how a completion algorithm performs compared to another on some empirical data is missing, since the data itself is not there. The only possible assessment of a completion engine is to manually test selected test cases.

3. *"If it ain't broken, don't fix it"*. Users are accustomed to the way code completion works and are resistant to change. This healthy skepticism implies that only a significant improvement over the default code completion system can change the status quo.

Ultimately, these reasons are tied to a single one: Code completion is "as good as it gets" with the information provided by current IDEs. To improve it, we need additional sources of information, and provide evidence that the improvement is worthwhile.

978-1-4244-2188-6/08/$25.00 ©2008 IEEE

In our previous work, we implemented Spyware, a framework which records the history of a program under development with great accuracy and stores it in a change-based repository [7, 9]. Our IDE monitoring plug-in is notified of the programmer's code edits, analyzes them, and extracts the actual program-level (i.e., not text-based) changes the developer performed on the program. These are then stored as first-class entities in a change-based software repository, and later used by various change-aware tools.

In this paper, we use change-based information to improve code completion. As a prerequisite, we define a benchmark to test the accuracy of completion engines. In essence, we replay the entire development history of the program and call the completion engine at every step, comparing the suggestions of the completion engine with the changes that were actually performed on the program. With this benchmark as a basis for comparison, we define alternative completion algorithms using change-based historical information to different extents, and compare them to the default algorithm which sorts matches in alphabetical order. We validate our algorithms by extensively testing each variant of the completion engine on the history of a medium-sized program developed for a number of years, as well as several smaller projects, testing the completion engine several hundred thousand times.

**Structure of the paper.** Section 2 details code completion algorithms and exposes the main shortcomings of these. We qualify those algorithms as "pessimist", and introduce requirements for "optimist" ones. Section 3 details the kind and the format of the data that we gather and store in change-based repositories, and how it can be accessed later on. Next, Section 4 presents our first contribution, the benchmarking framework we defined to measure the accuracy of completion engines. Section 5 introduces our second contribution, several "optimist" code completion strategies beyond the default "pessimist" one. Each strategy is evaluated according to the benchmark we defined. Section 6 presents our last contribution, a prototype implementation of a UI better suited for "optimist" completion algorithms. Finally, after a brief discussion in Section 7 and related work review (Section 8), we conclude in Section 9.

## 2 Code Completion

Word completion predates code completion and is present in most text editors. Since the algorithms used for it are very different, we do not cover these.

Code completion uses the large amount of information it can gather on the code base to significantly reduce the number of matches proposed to a user when he triggers it. For instance, a Java-specific code completion engine, when asked to complete a method call to a String instance, will only return the names of methods implemented in the class String. When completing a variable name, it will only consider variables which are visible in the scope of the current location. Such a behaviour is possible thanks to the amount of analysis performed in the IDE. At any time, an IDE such as Eclipse maintains a full queryable program model.

In the following, we focus on the completion *engine*, i.e., the part of the code completion tool which takes as input a token to be completed and a context used to access all the information necessary in the system, and outputs an ordered sequence of possible completions. We describe code completion in three IDEs: Eclipse (for Java), Squeak and VisualWorks (for Smalltalk).

### 2.1 Code completion in Eclipse

Code completion in Eclipse for Java is structure-sensitive, i.e., it can detect when it completes a variable/method name, and proposes different completions. It is also type-sensitive: If a variable is an instance of class String, the matches returned when auto completing a method name will be looked for in the classes "String" and "Object", i.e., the class itself and all of its superclasses.

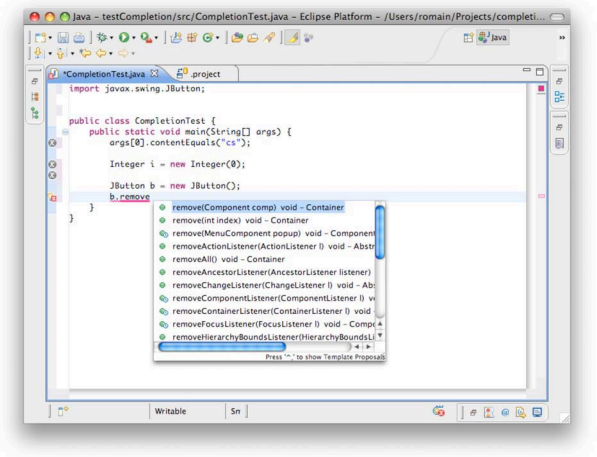


**Figure 1. Code completion in Eclipse**

Figure 1 shows Eclipse code completion in action: The programmer typed "remove" and attempts to complete it via Content Assist. The system determines that the object to which the message is sent is an instance of "javax.swing.JButton". This class features a large API of more than 400 methods, of which 22 start with "remove". These 22 potential matches are all returned and displayed in a popup window showing around 10 of them, the rest needing scrolling to be accessed. The matches are sorted in alphabetical order, with the shorter ones given priority (the first 3 matches would barely save typing as they would only insert parentheses).

This example shows that sometimes the completion system, even in a typed programming language, can break down and be more a hindrance than an actual help. As APIs grow larger, completion becomes less useful, especially since some prefixes tend to be shared by more methods than other: For instance, more than a hundred methods in JButton's interface start with the prefix "get".

### 2.2 Code completion in Visualworks

Visualworks is a Smalltalk IDE sold by Cincom [1]. Since Smalltalk is a dynamically typed language, Visualworks faces more challenges than Eclipse to propose accurate matches. The IDE can not make any assumption on the type of an object since it is determined only at runtime, and thus returns potential candidates from all the classes defined in the system. Since Smalltalk contains large libraries and is implemented in itself, the IDE contains more than 2600 classes already defined and accessible initially. These 2600 classes total more than 50,000 methods, defining around 27,000 unique method names, i.e., 27,000 potential matches for each completion. The potential matches are presented in a menu, which is routinely more than 50 entries long. As in Eclipse, the matches are sorted alphabetically.

### 2.3 Code completion in Squeak

Squeak's [2] completion system has two modes. The normal mode of operation is similar to Visualworks: Since the type of the receiver is not known, the set of candidates is searched for in the entire system. However, squeak features an integration of the completion engine with a type inference system, Roel Wuyts' RoelTyper [12]. When the type inference engine finds a possible type for the receiver, the candidate list is significantly shorter than it would be if matches were searched in the entire system (3000 classes, 57,000 methods totalling 33,000 unique method names). It is equivalent to the completion found in Eclipse. The type inference engine finds the correct type for a variable roughly half of the time. Both systems sort matches alphabetically.

### 2.4 Optimist and Pessimist Completion

All these algorithms have the same shortcoming: the match actually looked for may be buried under a large number of irrelevant suggestions because the matches are sorted alphabetically. The only way to narrow it down is to type a longer completion prefix which diminishes the value of code completion. To qualify completion algorithms, we reuse an analogy from Software Configuration Management. Versioning systems have two ways to handle conflicts during concurrent development [3]: *Pessimistic version control* –introduced first– prevents any conflict by forcing developers to lock a resource before using it. Conflicts never happen, but this situation is inconvenient when two developers need to edit the same file. In *optimistic version control* developers do not lock a resource to edit it. Several developers can freely work on the same file. Conflicts can happen, but the optimistic view states that they do not happen often enough to be counter-productive. Today, every major versioning system uses an optimistic strategy.

We characterize current completion algorithms as "pessimistic": They expect to return a large number of matches, and order them alphabetically. The alphabetical order is the fastest way to look up an individual entry among a large set. This makes the entry lookup a non-trivial operation: As anyone who has ever used a dictionary knows, search is still involved and the cognitive load associated to it might incur a context switch from the coding task at hand.

In contrast, we wish to introduce an "optimistic" completion algorithm. It would be free of the obligation to sort matches alphabetically, under the following assumptions:

1. The number of matches returned with each completion attempt are limited to a small quantity. The list of matches must be very quick to be checked. No scrolling should be involved, and reading it should be fast. In addition few keystrokes should be required to select the correct match. Our implementation (Section 6) limits the number of matches returned to 3.

2. The match the programmer is looking for has a high probability of being among the matches returned by the completion engine. Even if checking a short list of matches is fast, it is pointless if the match looked for is not in it. Hence the match looked for should be in the short list presented, preferably at the top spot.

3. To minimize typing, the completion prefix necessary to have the correct match with a high probability should be short. With a 10 character prefix, it is easy to return only 3 matches and have the right one among them.

To sum up, an optimistic code completion strategy seeks to maximize the probability that the desired entry is among the ones returned, while minimizing the number of entries returned, so that checking the list is fast enough. It attempts to do so even for short completion prefixes to minimize the typing involved by the programmer.

## 3 Change-based Software Repositories

The benchmark and some of the algoritms presented here rely on our previous work on *Change-Based Software Evolution* (CBSE). CBSE aims at accurately modelling how software changes by treating change as a first-class entity. This model has been previously used for software evolution analysis [8, 9].

### 3.1 Model and Implementation

CBSE models software evolution as a sequence of changes that take a system from one state to the next by means of syntactic (i.e., non text-based) transformations. These transformations are inferred from the activity recorded by the event notification system of IDEs such as Eclipse, whenever the developer incrementally modifies the system. Examples are the modification of the body of a method or a class, but also higher-level changes offered by refactoring engines. In short, we do not view the history of a software system as a sequence of versions, but as the sum of *change operations* which brought the system to its actual state.

CBSE is implemented in a prototype named SpyWare [10] for the Squeak Smalltalk IDE. SpyWare monitors the programmer's activity, converts it to changes and stores them in a change-based repository. We are also working on a prototype for the Eclipse IDE and the Java language called EclipsEye [11].

### 3.2 Program Representation

Our approach represents programs as domain-specific entities rather than text files. Since we focus on object-oriented programs, we consider constructs such as classes and methods. We represent a software system as an evolving abstract syntax tree (AST) containing nodes which represent packages, classes, methods, variables and statements. A node *a* is a child of a node *b* if *a* contains *b* (a superclass is not the parent of a subclass, only packages are parents of classes). Nodes have *properties*, which vary depending on the node type, such as: for classes, name and superclass; for methods, name, return type and access modifier (public, protected or private, if the language supports them); for variables name, type and access modifier, etc. The name is a property of entities since identity is provided by unique identifiers (ID).

### 3.3 Change Operations

Change operations represent the evolution of the system under study: They are actions a programmer performs when he changes a program, which in our model are captured and reified. They represent the transition from one state of the evolving system to the next. Change operations are *executable*: A change operation *c* applied to the state *n* of the program yields the state *n+1* of the program. Some examples of change operations are: adding/removing classes/methods to/from the system, changing the implementation of a method, or refactorings. We support *atomic* and *composite* change operations.

**Atomic Change Operations** Since we represent programs as ASTs, atomic change operations are, at the finest level, operations on the program's AST. Atomic change operations are executable, and can be undone: An atomic change contains all the necessary information to update the model by itself, and to compute its opposite atomic change. By iterating on the list of changes we can generate all the states the program went through during its evolution. The following operations suffice to model the evolution of a program AST:

**Creation:** creates a node *n* for an entity of a given type *t*.

**Addition:** adds a node *n* as a child of a given parent *p*.

**Removal:** removes node *n* from the children of parent *p*.

**Property change:** changes value *v* of property *p* of node *n*.

**Insertion:** inserts a node *n* as a child of a given parent (a method) *p*, at location m. This is necessary to model ordered parts of the AST such as the code in methods.

**Deletion:** deletes a node *n* from the location *m* in parent *p*. *m* is preserved to allow undo.

**Composite Change Operations** While atomic change operations are enough to model the evolution of programs, the finest level of granularity is not always the best suited. Representing the entire evolution of a system only by its atomic modifications leads to an overwhelming mass of information. Hence change operations can be abstracted into higher-level composite changes. Since we do not use composite changes in this article, we do not detail them further.

## 4 A Benchmark For Code Completion

The idea behind our benchmark is to use the information we recorded from the evolution of programs, and to replay it while calling the completion engine as often as possible. Since the information we record in our repository is accurate, we can simulate a programmer typing the text of the program while maintaining its structure as an AST. While replaying the evolution of the program, we can potentially call the completion engine at every keystroke, and gather the results it would have returned, as if it had been called at that point in time. Since we represent the program as an evolving AST, we are able to reconstruct the context necessary for the completion engine to work correctly, including the structure of the source code. For instance, the completion engine is able to locate in which class it is called, and therefore works as if under normal conditions.

The rationale behind the benchmarking framework is to reproduce as closely as possible the conditions encountered by the completion engine during its actual use. Indeed, one

might imagine a far simpler benchmark than ours: Rather than recording the complete history of a program, we could simply retrieve one version of the program, and attempt to complete every single message send in it, using the remainder of the program as the context. However, such an approach would disregard the order in which the code was developed and assume that the entire code base just "popped into existence". More importantly, it would not provide any additional source of information beyond the source code base, which would not permit any improvement over the state of the art. In contrast, by reproducing how the program was actually changed, we can feed realistic data to the completion engine, and give it the opportunity to use history as part of its strategy.

**Replaying a Program's Change History.** To recreate the context needed by the completion engine at each step, we execute each change in the change history of the program to recreate the AST of the program. In addition, the completion engine can use the actual change data to improve its future predictions. To measure the completion engine's accuracy, we use algorithm 1.

**Input**: Change history, completion engine to test
**Output**: Benchmark results

$results$ = newCollection();
**foreach** *Change $ch$ in Change history* **do**
  **if** *methodCallInsertion($ch$)* **then**
    $name$ = changeName($ch$);
    **foreach** *Substring $prefix$ of $name$ between 2 and 8* **do**
      $entries$ = queryEngine($engine$, $prefix$);
      $index$ = indexOf($entries$, $prefix$);
      add($results$, length($prefix$), $index$);
    **end**
  **end**
  processChange($engine$,$ch$);
**end**

**Algorithm 1**: The benchmark's main algorithm

While replaying the history of the system, we call the completion engine whenever we encounter the insertion of a statement including a method call. To test the accuracy with variable prefix length, we call the engine with every prefix of the method name between 2 and 8 letters –a prefix longer than this would be too long to be worthwhile. For each one of those prefixes, we collect the list of suggestions, and look up the index of the method that was actually inserted in the list, and store it in the benchmark results.

Using a concrete example, if the programmer inserted a method call to a method named "hasEnoughRooms", we would query the completion engine first with "ha", then "has", then "hasE", ..., up to "hasEnoughR". For each completion attempt we measure the index of "hasEnoughRooms" in the list of results. In our example, "hasEnoughRooms" could be 23rd for "ha", 15th for "has" and 8th for "hasE". One can picture our benchmark as emulating the behavior of a programmer compulsively pressing the completion key.

It is also possible that the correct match is not present in the list of entries returned by the engine. This can happen in the following cases:

1. The method called does not exist yet. There is no way to predict an entity which is not known to the system. This happens in a few rare cases.

2. The match is below the cut-off rate we set. If a match is at an index greater than 10, we consider that the completion has failed as it is unlikely a human will scroll down the list of matches. In the example above, we would store a result only when the size of the prefix is 4 (8th position).

In both cases we record that the algorithm failed to produce a useful result. When all the history is processed, all the results are analysed and summed up. For each completion strategy tested, we can extract the average position of the correct match in the entire history, or find how often it was in the first, second, or third position with a four letter prefix.

**Evaluating algorithms** To compare one algorithm with another, we need a numerical estimation of its accuracy. Precision and recall are often used to evaluate prediction algorithms. For completion algorithms however, the ranking of the matches plays a very important role. For this reason we devised a grading scheme giving more weight to both shorter prefixes and higher ranks in the returned list of matches. For each prefix length we compute a grade $G_i$, where $i$ is the prefix length, in the following way:

$$G_i = \frac{\sum_{j=1}^{10} \frac{results(i,j)}{j}}{attempts(i)} \quad (1)$$

Where $results(i, j)$ represents the number of correct matches at index $j$ for prefix length $i$, and $attempts(i)$ the number of time the benchmark was run for prefix length $i$. Hence the grade improves when the indices of the correct match improves. A hypothetical algorithm having an accuracy of 100% for a given prefix length would have a grade of 1 for that prefix length.

Based on this grade we compute the total score of the completion algorithm, using the following formula which gives greater weight to shorter prefixes:

$$S = \frac{\sum_{i=1}^{7} \frac{G_{i+1}}{i}}{\sum_{k=1}^{7} \frac{1}{k}} \times 100 \quad (2)$$

The numerator is the sum of the actual grades for prefixes 2 to 8, with weights, while the denominator in the formula corresponds to a perfect score (1) for each prefix. Thus a hypothetical algorithm always placing the correct match in the first position, for any prefix length, would get a score of 1. The score is then multiplied by 100 to ease reading.

**Limitations.** The benchmark we defined only takes into account the completion of method calls, and not other program entities. This is because the number of methods is usually the highest. Other entities, such as packages, classes, variables or keywords are less numerous. Hence the number of methods usually dwarfs the number of other entities in the system, and is where efforts should be first focused to get the most improvements.

**Typed and Untyped Completions.** As we have seen in Section 2, there are mainly two kinds of completion: Type-sensitive completion, and type-insensitive completion, the latter being the one which needs to be improved most. To address both types of completion, we chose the Squeak IDE to implement our benchmark. As Smaltalk is untyped, this allows us to improve type-insensitive completion. However since Squeak features an inference engine, we were able to test whether our completion algorithms also improves type-sensitive completion.

**Data used in the benchmark.** We used the history of SpyWare, our monitoring framework itself, to test our benchmark. SpyWare has currently around 250 classes and 20,000 lines of code. The data we used spanned from 2005 to 2007, totalling more than 16,000 thousands developer-level changes in several hundred sessions. In this history, more than 200,000 method calls were inserted, resulting in roughly 200,000 tests for our algorithm, and more than a million individual calls to the completion engine. We also used the data from 6 student projects, much smaller in nature and lasting a week. This allows us to evaluate how our algorithms perform on several code bases, and also how much they can learn in a shorter amount of time.

| Project | SW | SW(typed) | S1 | S2 | S3 | S4 | S5 | S6 |
|---|---|---|---|---|---|---|---|---|
| Attemps | 131 | 49 | 5.5 | 8.5 | 10.7 | 5.6 | 5.7 | 9.6 |

**Table 1. Completion attempts, in thousands**

# 5 Code Completion Algorithms

In this section we evaluate a series of completion algorithms, starting by recalling and evaluating the two default "pessimistic" strategies for typed and untyped completions. For each algorithm we present, we first give an intuition of why it could improve the performance of code completion, then detail its principles. We then detail its overall performance on our larger case study, SpyWare, with a table showing the algorithm's results for prefixes from 2 to 8 characters. Each column represents a prefix size. The results are expressed in percentages of accurate predictions for each index. The first rows gives the percentage of correct prediction in the first place, ditto for the second and third. The fourth rows aggregates the results for indices between 4 and 10. Anything more than 10 is considered a failure since it would require scrolling to be selected. After a brief analysis, we finally provide the global accuracy score for the algorithm, computed from the results. We discuss all the algorithms and their performances on the six other projects in the last section.

## 5.1 Default Untyped Strategy

**Intuition:** The match we are looking for can be anywhere in the system.

**Algorithm:** The algorithm searches through all methods defined in the system that match the prefix on which the completion is attempted. It sorts the list alphabetically.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 0.0 | 0.33 | 2.39 | 3.09 | 0.0 | 0.03 | 0.13 |
| % 2nd | 2.89 | 10.79 | 14.35 | 19.37 | 16.39 | 23.99 | 19.77 |
| % 3rd | 0.7 | 5.01 | 8.46 | 14.39 | 14.73 | 23.53 | 26.88 |
| % 4-10 | 6.74 | 17.63 | 24.52 | 23.9 | 39.18 | 36.51 | 41.66 |
| % fail | 89.63 | 66.2 | 50.24 | 39.22 | 29.67 | 15.9 | 11.53 |

**Table 2. Results for the default algorithm**

**Score: 12.1**. The algorithm barely, if ever, places the correct match in the top position. However it performs better for the second and third places, which rise steadily: They contain the right match nearly half of the time with a prefix length of 7 or 8, however a prefix length of eight is really long.

## 5.2 Default Typed Strategy

**Intuition:** The match is one of the methods defined in the hierarchy of the class of the receiver.

**Algorithm:** The algorithm searches through all the methods defined in the class hierarchy of the receiver, as indicated by the programmer or as inferred by the completion engine.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 31.07 | 36.96 | 39.14 | 41.67 | 50.26 | 51.46 | 52.84 |
| % 2nd | 10.11 | 11.41 | 13.84 | 16.78 | 13.13 | 13.51 | 12.15 |
| % 3rd | 5.19 | 5.94 | 4.91 | 5.15 | 3.2 | 1.94 | 2.0 |
| % 4-10 | 16.29 | 12.54 | 12.24 | 8.12 | 6.29 | 4.14 | 2.79 |
| % fail | 37.3 | 33.11 | 29.83 | 28.24 | 27.08 | 28.91 | 30.18 |

**Table 3. Results for typed completion**

**Score: 47.95**. Only the results where the type inference engine found a type were considered. This test was only run on the SpyWare case study as technical reasons prevented us to make the type inference engine work properly for the other case studies. The algorithm consistently achieves more than 25% of matches in the first position, which is much better than the untyped case. On short prefixes, it still has less than 50% of chances to get the right match in the top 3 positions.

### 5.3 Optimist Structure

**Intuition:** Local methods are called more often than distant ones (i.e., in other packages).

**Algorithm:** The algorithm searches first in the methods of the current class, then in its package, and finally in the entire system.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 12.7 | 22.45 | 24.93 | 27.32 | 33.46 | 39.5 | 40.18 |
| % 2nd | 5.94 | 13.21 | 18.09 | 21.24 | 20.52 | 18.15 | 22.4 |
| % 3rd | 3.26 | 5.27 | 6.24 | 7.22 | 10.69 | 14.72 | 10.77 |
| % 4-10 | 14.86 | 16.78 | 18.02 | 17.93 | 17.23 | 20.51 | 20.75 |
| % Fail | 63.2 | 42.26 | 32.69 | 26.26 | 18.07 | 7.08 | 5.87 |

**Table 4. Results for Optimist Structure**

**Score: 34.16**. This algorithm does not use the history of the system, only its structure, but is still an optimist algorithm since it orders the matches non-alphabetically. This algorithm represents how far we can go without using an additional source of information. As we can see, its results are a definite improvement over the default algorithm, since even with only two letters it gets more than 10% of correct matches. There is still room for improvement.

### 5.4 Recently Modified Method Names

**Intuition:** Programmers are likely to use methods they have just defined or modified.

**Algorithm:** Instead of ordering all the matches alphabetically, they are ordered by date, with the most recent date being given priority. Upon initialization, the algorithm creates a new dated entry for every method in the system, dated as January 1, 1970. Whenever a method is added or modified, its entry is changed to the current date, making it much more likely to be selected.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 16.73 | 23.81 | 25.87 | 28.34 | 33.38 | 41.07 | 41.15 |
| % 2nd | 6.53 | 12.99 | 17.41 | 19.3 | 18.23 | 16.37 | 21.31 |
| % 3rd | 4.56 | 6.27 | 6.83 | 7.7 | 11.53 | 15.58 | 10.76 |
| % 4-10 | 15.53 | 17.0 | 20.16 | 20.73 | 20.34 | 20.65 | 21.55 |
| % fail | 56.63 | 39.89 | 29.7 | 23.9 | 16.47 | 6.3 | 5.18 |

**Table 5. Results for recent method names**

**Results: 36.57**. Using a little amount of historical information is slightly better than using the structure. The results increase steadily with the length of the prefix, achieving a very good accuracy (nearly 75% in the top three) with longer prefixes. However the results for short prefixes are not as good. In all cases, results for the first position rise steadily from 16 to 40%. This puts this first "optimist" algorithm slightly less than on par with the default typed algorithm, albeit without using type information: This means that it will not resort to the default completion strategy when the type inferencer does not work.

### 5.5 Recently Modified Method Bodies

**Intuition:** Programmers work with a vocabulary which is larger than the names of the methods they are currently modifying. We need to also consider the methods which are called in the bodies of the methods they have recently visited. This vocabulary evolves, so only the most recent methods are to be considered.

**Algorithm:** A set of 1000 entries is kept which is considered to be the "working vocabulary" of the programmer. Whenever a method is modified, its name and all the methods which are called in it are added to the working set. All the entries are sorted by date, favoring the most recent entries. To better match the vocabulary the programmer is currently using, the names of the method called which are in the bodies of the methods which have been recently modified is also included in the list of priority matches.

**Score: 70.13**. Considering the vocabulary the programmer is currently using yields much better results. With a two-letter prefix, the correct match is in the top 3 in two thirds of the cases. With a six-letter prefix, in two-third of

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 47.04 | 60.36 | 65.91 | 67.03 | 69.51 | 72.56 | 72.82 |
| % 2nd | 16.88 | 15.63 | 14.24 | 14.91 | 14.51 | 14.04 | 14.12 |
| % 3rd | 8.02 | 5.42 | 4.39 | 4.29 | 3.83 | 4.09 | 4.58 |
| % 4-10 | 11.25 | 7.06 | 6.49 | 6.64 | 6.51 | 5.95 | 5.64 |
| % fail | 16.79 | 11.49 | 8.93 | 7.09 | 5.6 | 3.33 | 2.81 |

**Table 6. Results for recently modified bodies**

the cases it is the first one, and it is in the top three in 85% of the cases. This level of performance is worthy of an "optimist" algorithm.

### 5.6 Recently Inserted Code

**Intuition:** The vocabulary taken with the entire methods bodies is too large, as some of the statements included in these bodies are not relevant anymore. Only the most recent inserted statements should be considered.

**Algorithm:** The algorithm is similar to the previous one. However when a method is modified, we only refresh the vocabulary entries which have been newly inserted in the modified method as well as the name, instead of taking into account every method call. This algorithm makes a more extensive use of the change information we provide.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 33.99 | 52.02 | 59.66 | 60.71 | 63.44 | 67.13 | 68.1 |
| % 2nd | 15.05 | 16.4 | 15.44 | 16.46 | 16.38 | 17.09 | 16.52 |
| % 3rd | 9.29 | 7.46 | 5.98 | 5.64 | 5.36 | 4.74 | 5.45 |
| % 4-10 | 22.84 | 11.05 | 8.53 | 8.65 | 8.45 | 7.23 | 6.71 |
| % fail | 18.79 | 13.03 | 10.35 | 8.5 | 6.33 | 3.77 | 3.17 |

**Table 7. Results for recently inserted code**

**Score: 62.66**. In that case our hypothesis was wrong, since this algorithm is less precise than the previous one, especially for short prefixes. In all cases, this algorithm still performs better than the typed completion strategy.

### 5.7 Per-Session Vocabulary

**Intuition:** Programmers have an evolving vocabulary representing their working set. However it changes quickly when they change tasks. In that case they reuse and modify an older vocabulary. It is possible to find that vocabulary when considering the class which is currently changed.

**Algorithm:** This algorithm uses fully the change information we provide. In this algorithm, a vocabulary (i.e.,, still a set of dated entries) is maintained for each *programming session* in the history. A session is a sequence of dated changes separated by at most an hour. If a new change occurs with a delay superior to an hour, a new session is started. In addition to a vocabulary, each session contains a list of classes which were changed (or had methods changed) during it.

When looking for a completion, the class for the current method is looked up. To reconstruct the vocabulary the most relevant to that class, the vocabulary of all the sessions in which the class was modified is taken into account and given priority over the other vocabularies.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 46.9 | 61.98 | 67.82 | 69.15 | 72.59 | 75.61 | 76.43 |
| % 2nd | 16.88 | 15.96 | 14.41 | 15.01 | 14.24 | 14.44 | 13.8 |
| % 3rd | 7.97 | 5.73 | 4.64 | 4.3 | 3.45 | 3.0 | 3.4 |
| % 4-10 | 14.66 | 8.18 | 6.5 | 6.19 | 5.44 | 4.53 | 4.16 |
| % fail | 13.56 | 8.12 | 6.58 | 5.32 | 4.25 | 2.39 | 2.17 |

**Table 8. Results for per-session vocabulary**

**Score: 71.67**. This algorithm is the best we found as it reacts more quickly to the developer changing tasks, or moving around in the system. Since this does not happen that often, the results are only marginally better. However when switching tasks the additional accuracy helps. It seems that filtering the history based on the entity in focus (at the class level) is a good fit for an "optimistic" completion algorithm.

### 5.8 Typed Optimist Completion

**Intuition:** Merging optimist completion and type information should give us the best of both worlds.

**Algorithm:** This algorithm merges two previously seen algorithms. It uses the data from the session-based algorithm (our best optimist algorithm so far), and merges it with the one from the default typed algorithm. The merge works as follow:

The list of matches for the two algorithms are retrieved ($M_{session}$ and $M_{typed}$). The matches present in both lists are put at the top of $M_{session}$, which is returned.

| Prefix | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| % 1st | 59.65 | 64.82 | 70.09 | 73.49 | 76.39 | 79.73 | 82.09 |
| % 2nd | 14.43 | 14.96 | 14.1 | 13.87 | 13.17 | 13.09 | 12.08 |
| % 3rd | 4.86 | 4.64 | 3.89 | 3.27 | 2.92 | 2.23 | 1.85 |
| % 4-10 | 8.71 | 7.04 | 5.86 | 4.58 | 4.09 | 3.37 | 2.5 |
| % Fail | 12.31 | 8.51 | 6.03 | 4.75 | 3.4 | 1.54 | 1.44 |

**Table 9. Typed optimist completion**

**Score: 76.79**. The result is a significant, 5 points improvement, by 5 points (we ran it on SpyWare only for the same reasons as the default typed algorithm). This algorithm indeed performs better than anyone, since it merely reuses the already accurate session information, but makes sure that the matches corresponding to the right type are put before the other matches. In particular, with a two letter prefix, it gets the first match correctly 60 percents of the times.

### 5.9 Discussion of the results

Most of our hypotheses on what helps code completion were correct, except "Recently inserted code". We expected it to perform better than using the entire method bodies, but were proven wrong. We need to investigate if merging the two strategies yields any benefits over using only "Recent modified bodies". On the other hand, using sessions to order the history of the program is still the best algorithm we found, even if by a narrow margin. This algorithm considers only inserted calls during each session, perhaps using the method bodies there could be helpful as well.

When considering the other case studies (Table 10), we see that the trends are the same for all the studies, with some variations. Globally, if one algorithm performs better than another for a case study, it tends to do so for all of them. The only exception is the session-aware algorithm, which sometimes perform better, sometimes worse, than the one using the code of all the methods recently modified. One reason for this may be that the other case studies have a much shorter history, diminishing the roles of sessions. The algorithm has hence less time to adapt.

| Project | SW | S1 | S2 | S3 | S4 | S5 | S6 |
|---|---|---|---|---|---|---|---|
| Baseline | 12.15 | 11.17 | 10.72 | 15.26 | 14.35 | 14.69 | 14.86 |
| Structure | 34.15 | 23.31 | 26.92 | 37.37 | 31.79 | 36.46 | 37.72 |
| Names | 36.57 | 30.11 | 34.69 | 41.32 | 29.84 | 39.80 | 39.68 |
| Inserted | 62.66 | 75.46 | 75.87 | 71.25 | 69.03 | 68.79 | 59.95 |
| Bodies | 70.14 | 82.37 | 80.94 | 77.93 | 79.03 | 77.76 | 67.46 |
| Sessions | 71.67 | 79.23 | 78.95 | 70.92 | 77.19 | 79.56 | 66.79 |

**Table 10. Scores for the untyped algorithms of all projects**

Considering type information, we saw that it gives a significant improvement on the default strategy. However, the score obtained by our optimist algorithms –without using any type information– is still better. Further, our optimist algorithms work even in cases where the type inference engine does not infer a type, and hence is more useful globally. Merging the two strategies, e.g., filtering the list of returned matches by an optimist algorithm based on type information, gives even better results.

## 6 A User Interface for Optimist Completion

All user interfaces for completion tools suit "pessimist" completion algorithms. In all the cases we surveyed, the interface is a menu invoked by the programmer via a keyboard shortcut. Arrow keys are then used to select the right match.

We propose a completion interface Figure 2 suited for "optimist" completion algorithms. Since "optimist" completion algorithms often have the correct match in the top 3 spots – according to our benchmark, more than two thirds of the time after entering two letters the match looked for is in the top three spots considered by the algorithm–, we implemented a prototype user interface making suggestions without explicit programmer invocation. The interface shows the top three matches after the programmer has typed at least two letters of a method call. Depending on the algorithm chosen, the probability that the correct match is among those three vary between 3% for the default strategy, and nearly 80% for the typed optimist completion. With this difference in behavior, completion becomes even more of a second nature, since it does not have to be consciously invoked. The programmer uses a shortcut only when he sees the completion he needs in the short list given by the tool.

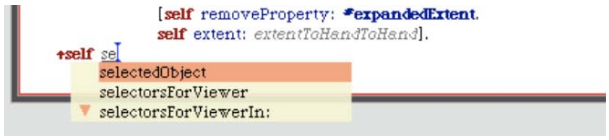


**Figure 2. Optimist completion in action**

Casual usage by the first author shows that the UI often gives a correct match before (e.g., with a shorter prefix) one would think of using the completion. However, a complete evaluation needs to be done as part of future work.

## 7 Discussion

Despite the provably more efficient completion algorithms we presented, our approach has a few shortcomings:

**Applicability to other programs.** We have tested several programs, but can not account for the general validity of our results. However, our results are relatively consistent among the different program we tested. If an algorithm performs better in one, it tends to perform better on the others.

**Applicability to other languages.** Our results are currently valid for Smalltalk only. However, the tests showed that our optimist algorithms perform better than the default algorithm using type inference, even without any type

information. Merging the two approaches shows another improvement. An intuitive reason for this is that even if only 5 matches are returned due to the help of typing, the position they occupy is still important. Thus we think our results have some potential for typed object-oriented languages such as Java. In addition, we are confident they could greatly benefit any dynamic language, such as Python, Ruby, Erlang, etc.

**Other uses of code completion.** Programmers use code completion in IDE at least for two reasons: (1) To complete the code they are typing, which is the part that we optimize, and (2) as a quick alternative to documentation. Code completion allows programmers to quickly discover the methods at their disposal on any object. Our completion algorithms do not provide this, and one could argue that they are detrimental to this usage, since they return only a few number of matches. However, we see the two systems as complementing each other. If our alternative GUI is used, programmers could use optimist completion while typing (without explicit invocation), and still invoke the regular code completion algorithm using the old keyboard shortcut.

**Resource usage.** Our benchmark in its current form is resource-intensive. Testing the completion engine several hundred thousands time in a row takes a few hours for each benchmark. We are looking at ways to make this faster.

## 8 Related Work

Beyond the classical completion algorithms, few works can compare with our approach. Mylyn's task contexts feature a form of code completion prioritizing elements belonging to the task at hand [4], which is very similar to our approach. We could however not reproduce their algorithm since our recorded information focuses on changes, while theirs focuses on interactions (they also record which entities were changed, but not the change extent). The data we recorded includes interactions only on a smaller period and could thus not be compared with the rest of the data. Another completion mechanism is Keyword Programming [5], in which free-form keywords are replaced by valid code found in the model of the program. It functions quite differently from standard completion algorithms, and hence could not be directly compared with other completion strategies.

## 9 Conclusions and Future Work

Code completion is a tool used by every developer, yet improvements have been few and far-between: Additional data is needed to both improve it and measure the improvement. We defined a benchmark to measure the accuracy of code completion by replaying the entire change history of seven projects, while calling the completion engine at every step. Using this historical information as an additional source of data for the completion engine, we significantly improved its accuracy by changing the alphabetical ordering of the results to an ordering based on entity usage.

Our "optimistic" completion algorithms have the correct match in the top 3 in 75% of the cases, whereas a "pessimistic" algorithm always have the correct match, but in a much larger list of candidates, and usually at a worse rank: The matches, when sorted alphabetically, have no semantic ordering. Hence using an "optimistic" algorithm involves less navigation and a lesser cognitive load to select a match.

In parallel, we implemented a completion tool prototype better adapted to optimist completion: its UI is always activated and proposes only three matches at a time.


### Acknowledgements

We thank D. Pollet and S. Krishnamurthi for insightful discussions about this work. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "REBASE" (SNF Project No. 115990). We thank the European Smalltalk User Group (http://www.esug.org) for sponsoring this work.


## References


[1] `http://smalltalk.cincom.com`, 2007.

[2] `http://www.squeak.org`, 2007.

[3] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[4] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of SIGSOFT FSE 2006*, pages 1–11, 2006.

[5] G. Little and R. C. Miller. Keyword programming in java. In *Proceedings of ASE 2007*, pages 84–93, 2007.

[6] G. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, jul 2006.

[7] R. Robbes. Mining a change-based software repository. In *Proceedings of MSR 2007*, page 15. ACM Press, 2007.

[8] R. Robbes and M. Lanza. An approach to software evolution based on semantic change. In *Proceedings of FASE 2007*, pages 27–41, 2007.

[9] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007*, pages 155–164, 2007.

[10] R. Robbes and M. Lanza. Spyware: a change-aware development toolset. In *ICSE*, pages 847–850, 2008.

[11] Y. Sharon. Eclipseye — spying on eclipse. Bachelor's thesis, University of Lugano, June 2007.

[12] R. Wuyts. Roeltyper: a fast type reconstructor for smalltalk. http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/, 2007.