

Improving Code Completion with Program History

Romain Robbes · Michele Lanza

Received: date / Accepted: date

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 usually comes in alphabetic order, i.e., the environment is only second-guessing the name based on pattern matching, relying on human intervention to pick the correct one. Finding the correct candidate can thus be cumbersome or slower than typing the full name.

We present an approach to improve code completion based on recorded program histories. We define a benchmarking procedure measuring the accuracy of a code completion engine and apply it to several completion algorithms on a dataset consisting of the history of several systems. Further, we use the change history data to improve the results offered by code completion tools. Finally, we propose an alternative interface for completion tools that we released to developers and evaluated.

Keywords Software Evolution · First-class Changes · Integrated Development Environments · Code Completion · Benchmark

Romain Robbes
PLEIAD Lab, Computer Science Department (DCC), University of Chile
Blanco Encalada 2120, of. 308, Santiago, Chile
Tel.: +56 2 978 4974
Fax: +56 2 689 5531
E-mail: rrobbes@dcc.uchile.cl

Michele Lanza
REVEAL @ Faculty of Informatics - University of Lugano
Via G. Buffi 13, 6904 Lugano, Switzerland
Tel.: +41 58 666 4659
Fax: +41 58 666 4536
E-mail: michele.lanza@usi.ch

1 Introduction

In 2006, Murphy et al. published an empirical study on how 41 Java developers used the Eclipse IDE [7]. One of their findings was that *every* developer in the study used the code completion feature. Among the top commands executed across all 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 to 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 dynamically typed, the set of candidates to choose from will still be too large. Given the limitations of current code completion engines, 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 a program, and if possible the structure 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 the fact that 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.

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 [13, 15]. 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 stored as first-class entities in a change-based software repository, and later used by various change-aware tools.

In this article we illustrate how one can improve code completion with the use of change-based information. 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. In this article we make the following contributions:

- The definition of a benchmarking procedure for code completion engines based on the recorded, real-world usage of the IDE by programmers.
- The definition of several variants of completion engines, completing method calls, and their evaluation on a benchmark with several programmers, against standard completion algorithms.
- Following the same procedure, we evaluate several variants of completion engines for class names on the same dataset.
- The implementation of our algorithms in a tool named OCompletion, which takes advantage of the increased accuracy of the algorithms to automatically propose completions without explicit user interaction.
- A qualitative evaluation of OCompletion. We released it to the open source communities around the Squeak and the Pharo projects, and collected their feedback in the form of a survey.

Structure of the article. Section 2 details code completion algorithms and exposes the main shortcomings of these. We classify those algorithms as “pessimistic”, and introduce requirements for “optimistic” ones. Section 3 contrasts benchmark-style evaluations with user studies and motivates our choice for a benchmark-style evaluation. Section 4 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 5 presents the benchmarking framework we defined to measure the accuracy of completion engines. Section 6 introduces several optimistic code completion strategies beyond the default pessimistic one. Each strategy is evaluated according to the benchmark we defined. Section 7 introduces several completion algorithms at the class level evaluated with the same benchmark-style evaluation. Section 8 presents a prototype implementation of a UI better suited for optimistic completion algorithms. Section 9 presents a qualitative evaluation of our code completion tool based on the feedback we received after its public release. Finally, after a brief discussion in Section 10 and related work review (Section 11), we conclude in Section 12.

2 Code Completion

Word completion predates code completion and is present in most text editors. Since the algorithms used for text completion are different from the ones used in code completion, we do not cover these, referring Fazly's work for a state of the art [6].

Code completion uses the large amount of information it can gather about the code base to significantly reduce the number of matches proposed to a user when he triggers it. For instance, when a Java-specific code completion engine is asked to complete a method call to a String instance, it 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 by modern IDEs.

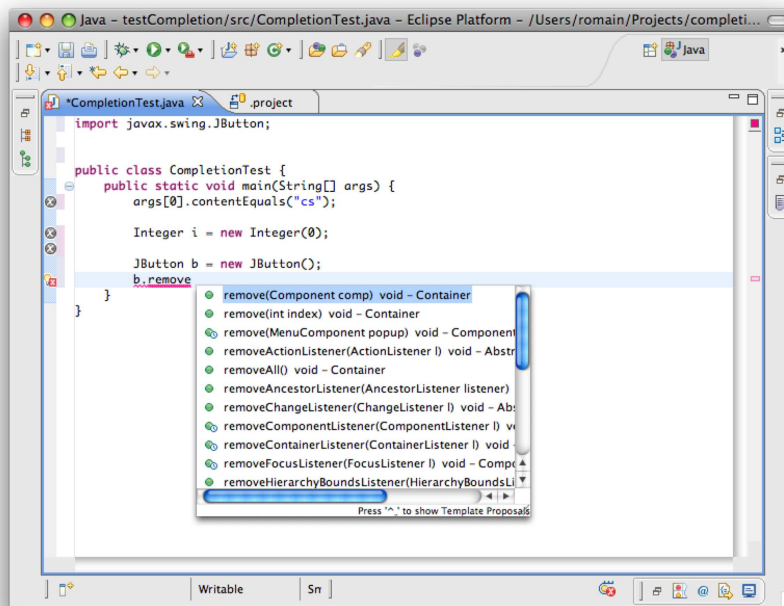


Fig. 1 Code completion in Eclipse

2.1 Code Completion in the Real World

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).

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 shows Eclipse code completion in action: The programmer typed “remove” and attempts to complete it via the completion engine, named “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 displaying roughly 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 be more of 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 prefixes: For instance, more than a hundred methods in JButton’s interface start with the prefix “get”.

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 cannot make any assumption about 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 2,600 classes already defined and accessible initially. These 2,600 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.

Code completion in Squeak. Squeak is an open-source Smalltalk IDE.² The completion system of Squeak 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 [20]. 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 (3,000 classes, 57,000 methods totalling 33,000 unique method names). The type inference engine finds the correct type for a variable roughly half of the time. Both systems sort matches alphabetically.

¹ <http://www.cincomsmalltalk.com>

² <http://www.squeak.org>

2.2 Classifying Code Completion Approaches

The algorithms we surveyed all share 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 classify completion algorithms, we reuse an analogy from Software Configuration Management. Versioning systems have two ways to handle conflicts during concurrent development [4]:

1. *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.
2. 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. The cognitive load associated to reading the list might incur a context switch from the coding task at hand.

In contrast, we want to introduce an optimistic completion algorithm, free of the obligation to sort matches alphabetically, under the following requirements:

- The number of matches returned with each completion attempt is small. 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 8) limits the number of matches returned to 3.
- 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.
- 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. It attempts to do so even for short completion prefixes to minimize the typing involved by the programmer.

3 Evaluating Code Completion

How can one accurately evaluate code completion? The problem applies to recommender systems in general (of which code completion is one), and is not trivial. Since these tools are ultimately used by humans, a direct user evaluation with a controlled experiment is a sensible choice. However these studies have shortcomings that we review before motivating our use of an alternative evaluation strategy. We believe that a combination of benchmarking, to fine-tune the recommendation algorithm, with user surveys after longer-term usage of the recommender, to be more suited to recommender systems in general and code completion in particular.

3.1 Human Subject Studies and Benchmarks

Human subject studies have a long tradition as an evaluation method in software engineering for methodologies and tools. They usually compare two treatments and hence involve two groups of people assigned to perform a given task, one using the methodology or tool under study, and a control group not using it. The performance of the groups are measured according to the protocol defined in the study, and compared with each other in order to determine whether the methodology or tool under study provides an improvement for the task at hand. To have confidence in the measure, a larger sample of individuals is needed to confirm a smaller increase in performance. Human subject studies are the “golden standard” to measure the effect of a treatment in a large number of cases. However some of their characteristics make them unsuited for recommender systems in software engineering:

- They are time-consuming and potentially expensive to set up. Dry runs must be performed first, so that the experiment’s protocol is carefully defined. Volunteers have to be found, which may also require a monetary compensation. The most extreme case in recent history is the pair programming study of Arisholm et al., which tested –and compensated– 295 professional programmers [1].
- The original authors need to document their experimental setup very carefully in order for the experiment to be reproduced. Lung et al. documented [10] the difficulties they encountered while reproducing a human subject study [5].
- These studies are unsuited for incremental refinement of an approach, as they are too expensive to be run repeatedly. In addition, a modest increment on an existing approach is harder to measure and must be validated on a higher sample size, increasing the complexity of the study.
- Comparing two approaches is difficult, as it involves running a new experiment pitting the two approaches side by side. The alternative is to use a common baseline, but variations in the setup of the experiment may skew the results.
- In the case of tools, they include a wide range of issues possibly unrelated to the approach the tool implements. Simple UI and usability issues may overshadow the improvements the new approach brings.

In short, controlled experiments involving human subjects give great confidence in the results they provide, but are hard to scale for a larger number of treatments, or small variations in the treatments.

Benchmarks are designed to automatically evaluate the performance of approaches on a dataset [18]. A benchmark delimits the problem to be solved in order to reliably measure performance against a baseline. The outcome of a benchmark is typically an array of measurements summing up the overall efficiency of the approach. An example is the CppETS benchmark for C++ fact extractors [19]: Its data corpus consists of several C++ programs exercising the various capabilities of fact extractors. A fact extractor run on the dataset returns the list of extracted facts, which can be compared with known results to produce a performance measurement. A benchmark has a set of characteristics that are suited to the evaluation of recommenders:

- Automated benchmarks can be run at the press of a button. This allows each experiment to be run easily, and re-run if needed. This considerably eases the replication of experiments done by other researchers.
- Automated scoring makes it trivial to compare approaches. The accuracy of the scoring allows one to evaluate the impact of incremental improvements.
- Benchmarks test a restricted functionality, and are impervious to usability issues.

In a nutshell, benchmarks are useful when one needs to compare a larger number of approaches which may feature low amounts of variations.

3.2 Our Evaluation Procedure

Recommenders are essentially algorithms that propose a set of recommendations. As such, they require a fair amount of fine-tuning. This involves an extensive number of replications, re-runs and comparisons of variants of the recommendation algorithm, which is the weak point of controlled experiments.

We advocate a two-step approach: We first determine through benchmarking the best performing recommender algorithm, and then evaluate its impact on users.

Given that we already have carefully measured the performance of the various recommendation algorithms, we perform a more qualitative study in the second step. After having programmers use the recommender for a period of time, we ask them through a survey if they perceived an improvement in their daily activities when using the recommender, and gather additional free-form feedback. Choosing a survey after a longer-term usage period of the tool, instead of a controlled user study, allows us to trade precision in the measured performance (which we have thanks to the benchmark in the first step), for impressions after real-life usage of the tool. This lets the users determine how well the recommender actually fit in their daily workflow.

This approach assumes that we have a benchmark at our disposal. However, creating the benchmark itself and the data corpus it uses represents a considerable amount of work. For the C++ fact extractor benchmark, it presumably involved a manual review of the C++ programs in the dataset to list the expected facts to be extracted. In the case of code completion, what is needed is a way to reproduce the usage of code completion by developers in order to retrospectively evaluate how well an alternate completion engine would have performed. Since 2005, we have set up and populated a change-based software repository that contains the data needed for this. Before detailing the benchmarking procedure, we first describe how a change-based software repository works.

4 Change-based Software Repositories

The benchmark and some of the algorithms presented here rely on our previous work on *Change-Based Software Evolution* (CBSE). CBSE aims at accurately modeling how software changes by treating change as a first-class entity. In our previous work we used this model to perform software evolution analysis [14, 15].

Model and Implementation. CBSE models software evolution as a sequence of changes that takes a system from one state to the next by means of syntactic (i.e., not 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 [16] for the Squeak Smalltalk IDE. SpyWare monitors the programmer’s activity, converts it to changes and stores them in a change-based repository. We also implemented a prototype for the Eclipse IDE and the Java language called EclipsEye [17].

Program Representation. CBSE represents programs as domain-specific entities, e.g., classes, methods, etc. rather than text files. We represent a software system as an evolving abstract syntax tree (AST) containing nodes which represent packages, classes, methods, variables and statements, as shown in Figure 2.

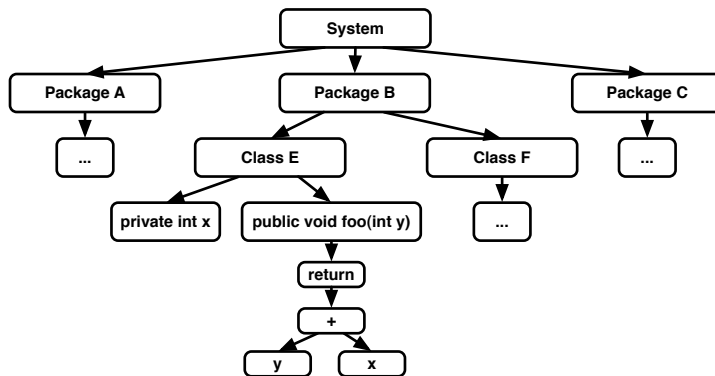


Fig. 2 An example program AST

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).

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*: Create and initialize a new node with id n of type t . The node is created, but is not added to the AST yet. The opposite of a creation change is a *Destruction*.
- *Destruction*: Remove node n from the system. Destructions only occur as undos of *Creations*, never otherwise (removed nodes are kept as they could be moving to a new branch).
- *Addition*: Add a node n as the last child of parent node p . This is the addition operation for unordered parts of the tree. The opposite of an addition is a *Removal*.
- *Removal*: Remove node n from parent node p . The opposite of the Addition change.
- *Insertion*: Insert node n as a child of node p , at position m (m is the node just before n , after n is inserted). Contrary to an addition, an insertion addresses the edition of ordered parts of the tree. The opposite change is a *Deletion*.
- *Deletion*: Delete node n from parent p at location m . The opposite of *Insertion*.
- *Change Property*: Change the value of property p of node n , from v to w . The opposite operation is a property change from value w to value v . The property can be any property of the AST node, and as such depends on the properties defined in the model.

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. Change operations can be abstracted into higher-level composite changes. The first of these levels is the *developer-level change*, which groups atomic changes in logical actions a developer performs. Examples are class additions (create a class, add it to a package, set its name and superclass, as well as creating and adding instance variables), or modifying a method (creating and adding a set of statements to a method, and removing another set of statements from the same method). Since we do not use higher-level composite changes in this article, we do not detail them further.

5 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 program statements while maintaining the system’s structure as an AST. While replaying the evolution of the program, we 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, e.g., the completion engine is able to locate in which class it is called, thus 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 occurring 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, name);
            Increment(results[length(prefix),index]);
        end
    end
    processChange(engine,ch);
end
return results;

```

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 a programmer inserted a method call to a method named “hasEnoughRooms”, we would query the completion engine first with “ha”, “has”, “hasE”, . . . , up to “hasEnoug”. 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 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:

- 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.
- 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 user 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 appears at a particular rank for a particular prefix length.

5.1 Evaluation Procedure

To compare algorithms, we need a numerical estimation of their 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 times 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.

Typed and Untyped Completion. As we have seen in Section 2, there are 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 Smalltalk 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 improve type-sensitive completion.

5.2 Benchmark Data

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 developer-level changes in several hundred development sessions.

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.

The number of tests for each system we used is shown in Table 1.

Table 1 Benchmark Data

Project	Completion Attempts
SpyWare	131,000
(SpyWare typed)	(49,000)
S1	5,500
S2	8,500
S3	10,700
S4	5,600
S5	5,700
S6	9,600
Total	176,600

In total, more than 175,000 method calls were inserted, resulting in the same number of tests for our algorithm, and more than a million individual calls to the completion engine.

6 Code Completion Algorithms

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 describe its principles and 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 beyond 10 is considered a failure since it would require scrolling to be selected. We provide the global accuracy score for each algorithm, computed from the results. At the end, we discuss all the algorithms and their performances on the six other projects.

6.1 Default Untyped Strategy (Score: 12.15)

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

Table 2 Results for the Default Untyped Strategy algorithm

Prefix	2	3	4	5	6	7	8
% 1st	0.00	0.33	2.39	3.09	0.00	0.03	0.13
% 2nd	2.89	10.79	14.35	19.37	16.39	23.99	19.77
% 3rd	0.70	5.01	8.46	14.39	14.73	23.53	26.88
% 4-10	6.74	17.63	24.52	23.90	39.18	36.51	41.66
% fail	89.63	66.20	50.24	39.22	29.67	15.90	11.53

Results (Table 2): 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: By the time the prefix reaches a length of 7, nearly 50% of the correct matches are in the second or third position. However these longer prefixes contribute little to the overall score.

6.2 Default Typed Strategy (Score: 47.95)

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

Table 3 Results for the Default Typed Strategy algorithm

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	3.20	1.94	2.00
% 4-10	16.29	12.54	12.24	8.12	6.29	4.14	2.79
% fail	37.30	33.11	29.83	28.24	27.08	28.91	30.18

Results (Table 3): Only the results where the type inference engine found a type were considered. The algorithm consistently achieves more than 25% of matches in the first position, which is much better than the untyped case. On 2-letter prefixes, it still has a less than 50% chance to get the right match in the top 3 positions.

6.3 Optimistic Structural Completion (Score: 34.15)

Principle: Local methods are called more often than distant ones (i.e., in other packages). The algorithm searches first in the methods of the current class, then in its package, and finally in the entire system.

Table 4 Results for the Optimistic Structural Completion algorithm

Prefix	2	3	4	5	6	7	8
% 1st	12.70	22.45	24.93	27.32	33.46	39.50	40.18
% 2nd	5.94	13.21	18.09	21.24	20.52	18.15	22.40
% 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	63.20	42.26	32.69	26.26	18.07	7.08	5.87

Results (Table 4): This algorithm does not use the history of the system, only its structure, but is still an optimistic algorithm since it does not order the matches alphabetically. This algorithm represents how far we can go without using an additional source of information. Its results are a definite improvement over the default algorithm, since even with only two letters it gets more than 10% of correct matches.

6.4 Recently Modified Method Names (Score: 36.57)

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

Table 5 Results for the Recently Modified Method Names algorithm

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	19.30	18.23	16.37	21.31
% 3rd	4.56	6.27	6.83	7.70	11.53	15.58	10.76
% 4-10	15.53	17.00	20.16	20.73	20.34	20.65	21.55
% fail	56.63	39.89	29.70	23.90	16.47	6.30	5.18

Results (Table 5): 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 optimistic algorithm slightly less than on par with the type-aware algorithm, albeit without the need for type information.

6.5 Recently Modified Method Bodies (Score: 70.14)

Principle: 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. 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.

Table 6 Results for the Recently Modified Method Bodies algorithm

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	5.60	3.33	2.81

Results: 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 more than two thirds of the cases (71.94%). With a six-letter prefix, in two-third of the cases it is the first one, and it is in the top three in close to 90% of the cases (87.85%). This level of performance is worthy of an optimistic algorithm.

6.6 Recently Inserted Code (Score: 62.66)

Principle: 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. 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.

Table 7 Results for the Recently Inserted Code algorithm

Prefix	2	3	4	5	6	7	8
% 1st	33.99	52.02	59.66	60.71	63.44	67.13	68.10
% 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	8.50	6.33	3.77	3.17

Results: In this case our intuition 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.

6.7 Per-Session Vocabulary (Score: 71.67)

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

Results: 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.

Table 8 Results for the Per-Session Vocabulary algorithm

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	13.80
% 3rd	7.97	5.73	4.64	4.30	3.45	3.00	3.40
% 4-10	14.66	8.18	6.50	6.19	5.44	4.53	4.16
% fail	13.56	8.12	6.58	5.32	4.25	2.39	2.17

6.8 Typed Optimistic Completion (Score: 76.79)

Principle: Merging optimistic completion and type information should give us the best of both worlds. This algorithm merges two previously seen algorithms. It uses the data from the session-based algorithm (our best optimistic 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.

Table 9 Results for the Typed Optimistic Completion algorithm

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	2.50
% Fail	12.31	8.51	6.03	4.75	3.40	1.54	1.44

Results: The result is a significant 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 the others, 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% of the time.

6.9 Discussion of the Results

Most of our expectations 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 performs 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.

Table 10 Scores of each algorithm, for all projects

Project	SW	S1	S2	S3	S4	S5	S6
Baseline (Section 6.1)	12.15	11.17	10.72	15.26	14.35	14.69	14.86
Structure (Section 6.3)	34.15	23.31	26.92	37.37	31.79	36.46	37.72
Names (Section 6.4)	36.57	30.11	34.69	41.32	29.84	39.80	39.68
Bodies (Section 6.5)	70.14	82.37	80.94	77.93	79.03	77.76	67.46
Inserted (Section 6.6)	62.66	75.46	75.87	71.25	69.03	68.79	59.95
Sessions (Section 6.7)	71.67	79.23	78.95	70.92	77.19	79.56	66.79
Typed (Section 6.2)	47.95	-	-	-	-	-	-
Typed Optimist (Section 6.8)	76.79	-	-	-	-	-	-

Considering type information, we saw that it gives a significant improvement on the default strategy. However, the score obtained by our optimistic algorithms – without using any type information – is still better. Further, our optimistic 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 optimistic algorithm based on type information, gives even better results.

7 Class-level Code Completion Algorithms

As we present later in Section 9, we released our tool to the developers of two open-source communities in order to gather feedback and improve the usability of our tool. One of the first requests we received after releasing our code completion tool to the Squeak and Pharo (a derivative of Squeak) communities was to make it support class-name completion. Our first version initially supported only the completion of method names, using the default pessimistic algorithm in the case of classes. Such a request is sound, since the number of classes available in an IDE is the second most numerous category of entities after methods: If methods number in the tens of thousands, classes number in the thousands. We hence applied the same evaluation strategy, with the following changes:

- Instead of testing the completion engine each time a method call was inserted, we test it when a reference to a class is added to the program.
- The completion algorithm is aware that classes are expected, and returns a list of candidate classes.

Before detailing the four algorithms we investigated, we make a few observations:

- The tests are much less numerous than for methods, since referencing a class directly is a much less common activity. All in all, we tested the completion engines for classes around 8,000 times - an order of magnitude less than for methods. The number of classes is lower as well (by an order of magnitude). This impacts the results and makes the algorithms score better overall.
- As Squeak lacks the concept of namespaces, classes have often –by convention– their names prefixed by a two letter abbreviation identifying the application they belong to. This may impact the results by lowering them for short prefixes.
- Type-aware completion does not help, since we are inserting a type itself.

7.1 Default Strategy for Classes (Score: 41.37)

Principle: The match we are looking for may be any class in the system. The algorithm searches through all the classes defined in the system whose name matches the prefix on which the completion is attempted. The algorithm sorts the resulting matches alphabetically.

Table 11 Results for the Default Strategy for Classes algorithm

Prefix	2	3	4	5	6	7	8
% 1st	5.02	27.8	58.76	57.92	60.41	72.0	76.15
% 2nd	7.75	14.39	14.57	15.86	15.97	8.66	6.93
% 3rd	1.59	3.73	3.35	3.94	3.58	3.19	2.45
% 4-10	2.64	9.83	3.6	1.97	1.7	1.5	1.51
% fail	82.96	44.21	19.69	20.27	18.31	14.61	12.92

Results: Due to the reduced number of entities, the pessimistic algorithm fares much better in the case of classes than methods. It reaches a very high probability of putting the right match on top if the prefix is long enough (4 letters or more). The score for a two-letter prefix is much lower, due to the convention of prefixing classes to indicate which package they belong to.

7.2 Structure-aware Completion (Score: 45.36)

Principle: Classes in the same package are more often used than classes outside of it. When using code completion in a given class, the algorithm prioritizes the classes that belong to the same package over those of the whole system.

Results: The algorithm is more precise than the default algorithm overall, but not by much. One reason for that is that the assumption behind the algorithm is sometimes invalidated: Developers often use base classes from libraries that are outside of the application, such as string, collection or file classes. In that case, this algorithm fares no better than the previous one. The accuracy for two-letter prefixes is improved, but remains still overall quite low.

Table 12 Results for the Structure-aware Completion algorithm

Prefix	2	3	4	5	6	7	8
% 1st	8.71	32.5	61.72	60.01	62.48	74.07	78.76
% 2nd	8.78	13.29	16.1	17.26	17.48	9.76	6.71
% 3rd	2.21	6.79	2.76	3.42	2.93	2.75	2.17
% 4-10	3.76	9.53	3.37	2.3	2.01	1.74	1.63
% fail	76.5	37.85	16.01	16.98	15.07	11.65	10.68

7.3 Recently Used Classes (Score: 79.29)

Principle: Classes used in methods the programmer change have more chances to be used again in the future. All the class entries have an associated date, initialized to January 1st 1970. Whenever the programmer changes a method, all the references to classes in its body are updated to the date of the change. Entries are ordered by date, with the most recent first.

Table 13 Results for the Recently Used Classes algorithm

Prefix	2	3	4	5	6	7	8
% 1st	58.78	77.66	85.04	84.01	84.97	86.85	88.0
% 2nd	7.55	9.65	6.63	7.04	6.76	5.56	4.95
% 3rd	6.34	2.97	1.23	1.39	1.28	1.0	0.81
% 4-10	5.95	2.25	1.21	0.93	0.69	0.59	0.47
% Fail	21.35	7.44	5.86	6.6	6.26	5.97	5.73

Results: This algorithm takes into account usage recency, and again the benefits are clearly visible, as its score is nearly double that of the previous best-performing algorithm. Scores are much higher for all prefix lengths, but the most contributing factor is the short prefixes. For a length of 2, the algorithm puts the right match in the right position nearly seven times as often as the previous best performer. Indeed, only a recency factor could help differentiating between a large number of classes which all share the same two-letter prefix.

7.4 Recently Inserted Classes (Score: 79.86)

Principle: Classes previously used the by the programmer have a higher probability of being reused. Whenever a reference to a class is added in the system, its entry's date is changed to the current time. Entries are ordered by date, with the last used first.

Results: This algorithm provides a slight improvement over the previous one. Nevertheless, favoring class references that were actually inserted gives a slight edge all over the board, as opposed to the method case, where it was slightly detrimental. We

Table 14 Results for the Recently Inserted Classes algorithm

Prefix	2	3	4	5	6	7	8
% 1st	60.14	77.94	85.34	84.17	85.22	86.88	88.29
% 2nd	7.62	9.49	6.16	6.65	6.32	5.41	4.5
% 3rd	6.73	2.94	1.39	1.64	1.56	1.21	1.1
% 4-10	5.02	2.18	1.16	0.87	0.58	0.47	0.34
% fail	20.46	7.41	5.91	6.63	6.29	6.0	5.73

are unsure about the causes. Since the improvement was so small, we are convinced we reached a point of diminishing returns, after which increases in accuracy will be minimal.

7.5 Discussion of the Results

The problem of class completion is simpler than the one of method completion, as the entities to choose from are less numerous by an order of magnitude. There is however space for valuable improvement, as we have shown: A history-aware completion algorithm can easily improve over the default or structure-aware algorithms. Moreover, the improvements are considerable for short prefixes.

Table 15 Scores for the class completion algorithms of all projects

Project	SW	S1	S2	S3	S4	S5	S6
Baseline	41.37	18.27	29.26	57.30	46.79	53.49	65.84
Structure	45.36	18.85	29.43	70.45	47.42	56.19	72.87
Used	79.29	86.22	88.47	92.51	92.76	95.46	93.35
Inserted	79.86	87.12	88.80	92.51	92.76	95.46	93.22

The overall results for each project are shown in Table 15. We observe variations in accuracy in each of the projects, as we saw previously, but the algorithms stay in the same order across projects. Of course, smaller projects also had a higher accuracy as the number of classes defined for each project was much lower. This makes completion of domain classes significantly easier. This did not affect the completion of library classes, which are still used a significant portion of the time. Depending on the usage patterns of classes in projects, the improvement given by the structural algorithm ranges from very significant (for project S4 that uses mainly classes it defined), to nearly insignificant (for project S3 that uses a large number of base classes). Project S1 is a bit of an outlier as the performance of the default algorithms is decidedly lower than other projects: It is using a restricted number of classes that happen to have rather ambiguous names. Of note, if the default and structural algorithms have relatively large variations, the history-aware algorithms are much more stable across all projects, suggesting that we reached a plateau of efficiency. Comforting this point, the difference between the algorithms is negligible to the point that on some case studies the algorithm perform identically, with very high scores.

8 A User Interface for Optimistic Completion

All user interfaces for completion tools suit pessimistic completion algorithms: the interface is a menu invoked by the programmer via a keyboard shortcut. Arrow keys are then used to select the right match. Thus code completion is used explicitly by the programmer, who may underuse it and still type entire methods and class names, even when the completion engine would have been successful. The alternative is to have the completion engine propose candidates constantly, without explicit intervention by the programmer. This maximizes the chances that the programmer uses code completion, but induces the risk of distractions when the propositions are too often inaccurate. Considering the accuracy of pessimistic code completion algorithms, having an explicit access to completion is reasonable. With the increased accuracy of optimistic completion algorithms, it may be time to revisit this choice.

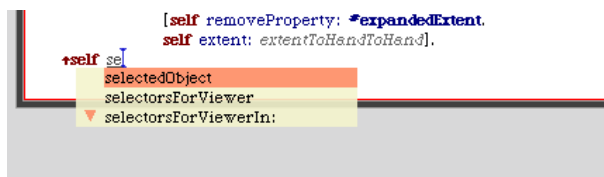


Fig. 3 Optimistic completion in action

We implemented OCompletion³ (see Figure 3), which uses optimistic completion together with an implicit invocation interface. As the programmer types the names of identifiers, a short list is permanently shown and interactively updated as the programmer types. We call the suggestions on this list “Automatic Suggestions” as it appears without explicit user interaction. Pressing tab inserts the first candidate in the list, while pressing the down arrow causes the menu to scroll down. Scrolling down also signifies interest in the matches, so the menu expands to show up to seven matches, a number that can still be easily processed by humans [11]. Initially, we show only up to three matches to minimize the reading effort. Based on our empirical data, this is sufficient to show the expected match close to 75% of the time.

As for the algorithm we implemented in the tool, the empirical results on our dataset allowed us to make an informed decision, taking other parameters than raw performance into account, such as the simplicity of implementation. We chose the “recently changed method bodies” algorithm for methods, as it has a performance close to the ideal without needing to store a lot of usage data. For classes, we chose “recently used classes”, which had a performance close to the best performing algorithm. Choosing these algorithms limits the amount of parsing we have to perform in the tool, as we do not have to differentiate two versions of a method to extract changes. This makes the released tool lighter overall. Since upon installation, OCompletion has not gathered usage data yet, it asks the user for a list of packages he or she is working on in order to have a reasonable initial set of matches.

³ Available at <http://www.squeaksource.com/OCompletion>

9 A Qualitative Evaluation of Optimistic Completion

We decided that the best course of action to evaluate how well our tool works in practice was to release OCompletion to the developer community, have them use it in their habitual work environment, and collect their feedback. We preferred this over a controlled experiment for the following reasons:

- The tool is used in a real-world setting. The tool was mature enough so that developers used it over several weeks in their daily activities. As such they could get familiar with it and give us detailed feedback, both of which would not have been possible over a short usage period as in a controlled experiment.
- Controlled experiments of recommenders are hard to set up. As we already mentioned in Section 3, the numbers of variables one has to account for in a programming task can be so large that isolating the variable of interest can be difficult. On the other hand, gathering feedback about the overall usefulness of OCompletion over time is much more straightforward.
- We already have solid empirical evidence that our algorithms significantly improve on current algorithms. Since we are able to precisely isolate and measure the accuracy of code completion over time using benchmarks, gathering subjective developer feedback is sufficient.
- The real-world impact is greater. By releasing the tool to the community, we had to take care of a number of usability issues to make sure people are willing to use it daily. We know of several people who have integrated OCompletion in their daily toolset. It is even included by default in one of the programming environments for which we released it.

Initial Release and Feedback. We publicly released OCompletion at the beginning of May 2009 to the Squeak and Pharo development communities. Squeak is an open source implementation of a Smalltalk programming environment aimed at research, teaching and multimedia activities. Pharo is a fork of Squeak aimed at a more professional development of applications. The feedback we received was very positive as quotes from the mailing lists show:

- “I like [OCompletion’s] non-intrusive smartness.”
- “I love OCompletion, first time I’ve found completion in Smalltalk actually helpful.”
- “In my opinion this could be in the Pharo dev image by default.”

As a consequence, the Pharo maintainers decided to include OCompletion in the standard development distribution of Pharo, Pharo-dev. Independently, OCompletion has been downloaded more than 100 times so far. OCompletion users also quickly reported bugs and feature requests, such as making OCompletion work also for class names (see Section 7). We addressed these issues and released a second version of OCompletion, at which point we decided to collect more formal feedback using a survey.

9.1 OCompletion User Survey





The goal of the survey was to assess with greater accuracy how people felt about OCompletion and how well it fares compared to its predecessor, eCompletion. eCompletion is the tool currently used by most users of the Squeak community. It mimics the way code completion in eclipse works, and as such uses a pessimistic completion algorithm. Our survey consisted of 10 questions (to keep the user's time investment low), separated into five categories:

1. The first question assessed the experience of the participants using the development environment;
2. the next three questions assessed how the respondents used the previous completion tool, eCompletion;
3. the following three questions compared OCompletion to eCompletion with similar questions;
4. the next two questions addressed the automatic suggestions of OCompletion and how the participants perceived them;
5. finally, a space for free-form feedback collected general impressions, requests for enhancements, etc.

We advertised the survey on the Squeak and Pharo mailing list, attracting 29 respondents, of which 20 had used OCompletion. 8 others had used eCompletion but did not try OCompletion, and one did not use code completion tools at all. In the following tables, questions marked with a star allowed multiple answers. The survey kept track of how many people answered each question (which may be distinct from the number of answers), and of how many people skipped it. We now comment on the results of each question.

9.2 Experience of the Respondents (Table 16)















Table 16 Experience of responders using Squeak

Answers	Responses	Percent	Count
Q1: How long have you been using Squeak?		29 answers, 0 skipped	
less than 3 months		3.4%	1
3 months to a year		13.8%	4
1 to 4 years		51.7%	15
5 years or more		31%	9

Q1. We can see that the respondents are overall quite experienced with the IDE, and as such are educated in the available tools in the IDE. More than 80% have been using the IDE for more than a year, and nearly a third of the respondents for 5 years or more.

9.3 Usage of eCompletion (Table 17)

Table 17 Experience of responders using eCompletion

Answers	Responses	Percent	Count
Q2: How often did you use eCompletion?		29 answers, 0 skipped	
I didn't use it		6.9%	2
I used it but eventually stopped		27.6%	8
Only when it would save me keystrokes		6.9%	2
Regularly, but I sometimes type full names		20.7%	6
As much as possible		37.9%	11
Q3: If you stopped using eCompletion, why did you?*		14 answers, 15 skipped	
Unclear benefits		14.3%	2
Imprecise		64.3%	9
Too slow		50%	7
Buggy		7.1%	1
Other		21.4%	3
Q4: Where would you find the match you needed?		28 answers, 1 skipped	
I rarely got the match I wanted		10.7%	3
After a lot of scrolling and typing		37.9%	11
In the first few menu items		46.4%	13
In the top position		3.6%	1

Q2. This question addresses the usage patterns of eCompletion users. Two persons did not use eCompletion: one was not using completion tools at all, and the second switched from no completion tool to OCompletion. More than 25% of the respondents ended up stopping to use eCompletion. A minority were wary of eCompletion's accuracy and deliberately typed longer prefixes to be sure that the tool would propose the right match. The majority used OCompletion fairly often.

Q3 investigates why people stopped to use eCompletion. The main reason for people stopping to use eCompletion was its lack of precision, followed by a perceived slowness. One respondent states that he switched to OCompletion, another does not remember exactly (—"It probably got on my nerves") and a third did not install it again when he changed environments (probably because the benefits were not important enough to him). A larger number of people responded to Q3 than the number of people who said that they stopped using eCompletion in Q2. This also applies to Q5 and Q6; we do not know the reason of the discrepancy.

Q4 gathers impressions about eCompletion's accuracy. We see that the curve is balanced, with small minorities choosing extreme choices. A small minority found it really imprecise, and an even smaller minority (one respondent) found it very precise. The two main blocks state that either using eCompletion required a lot of scrolling and typing (somewhat imprecise), or that it was satisfactory overall.

9.4 Usage of OCompletion (Table 18)

Table 18 Experience of responders using OCompletion

Answers	Responses	Percent	Count
Q5: How often do you use OCompletion?		26 answers, 3 skipped	
I didn't use it		23.1%	6
I used it, but eventually stopped		7.7%	2
I use it regularly		26.9%	7
If it is not installed, I install it		15.4%	4
It is installed by default in my environment		26.9%	7
Q6: If you stopped using OCompletion, why did you?*		4 answers, 25 skipped	
Unclear benefits		25%	1
Imprecise		0%	0
Too slow		25%	1
Buggy		0%	0
Other		50%	2
Q7: Where would you find the match you needed?		20 answers, 9 skipped	
I rarely got the match I wanted		0%	0
After a lot of scrolling and typing		10%	2
In the first few menu items		65%	13
In the top position		25%	5

Q5 is about OCompletion and its usage patterns. Since the user interface is different (OCompletion's user interface is always there and not summoned on demand), some of the possible answers differ from Q2. Beyond people not using OCompletion, we see that fewer people stopped using OCompletion (7.7%) than eCompletion (27.6%). This leaves us under the impression that its behavior is overall very satisfactory. Several users responded that they depended on it enough to install it if it is missing, while others (Pharo users) have it installed by default.

Q6. Few people stopped using OCompletion. Nobody stopped because of a lack of precision. Reasons cited include OCompletion not working with class names (this problem was fixed in the second version), and slowness. In the informal feedback we received, we associated slowness with the class name issue: Attempts to complete a class name in the first version of OCompletion fell back to eCompletion's algorithm, which was indeed slow. We suspect these respondents did not try the second version.

Q7. In terms of accuracy, we see a large difference with eCompletion. None of the respondents said that they had trouble finding the match they needed. On the other hand, people finding OCompletion extremely precise are now the second largest category. Overall, the shape is much more skewed towards accuracy, whereas eCompletion's was centered.

9.5 Impressions About OCompletion's Automatic Suggestions (Table 19)

Table 19 OCompletion's automatic suggestions

Answers	Responses	Percent	Count
Q8: How useful do you find automatic suggestions?		18 answers, 11 skipped	
I stopped using OCompletion because of them		0%	0
Somewhat annoying		0%	0
I don't mind them		22.2%	4
I find them useful		61.1%	11
I use them all the time		16.7%	3
Q9: What is the main improvement of OCompletion?		18 answers, 11 skipped	
Increased accuracy		16.7%	3
Automatic suggestions		11.1%	2
The combination of both		61.1%	11
Neither/No significant improvement		11.1%	2
Other		0%	0

Q8 investigates how people react to the automatic suggestions. If OCompletion gave inaccurate suggestions, people would find automatic suggestions annoying, or stop using the tool altogether. The respondents are overwhelmingly in favor of automatic suggestions, as nobody stated that they were annoyed by the suggestions.

Q9. This is also reflected in what people think are the advantages of OCompletion. Again, a strong majority find that the combination of an increased accuracy with automatic suggestions is worth more than both improvements considered on their own. A small minority of respondents did not see visible improvements with the tool. We hope to win them over with subsequent versions of OCompletion.

9.6 Detailed Feedback

Q10 was a detailed feedback form. Only 8 users chose to fill it, a bit below half of the respondents who browsed the last page of the survey (which contained Q8, Q9 and Q10). Common suggestions included:

- Add support for other language constructs beyond methods, with an emphasis on classes. This has been added in the second version of the tool.
- Restore one feature that eCompletion has, but OCompletion does not: the ability to explore the matches in detail. Since we propose a small list of matches, users cannot explore many methods in the list even if they would like to.
- Perform cosmetic changes such as a better integration with the IDE's look and feel.
- Make OCompletion work in all parts of the IDE. OCompletion right now works in the most common tools, but not all of them.

9.7 Conclusions on the Survey

This survey comforted our opinion that optimistic completion algorithms are more accurate than pessimistic ones, and that the improvement is perceptible in real-world usage. In addition, it validated our expectation that an alternative completion interface would benefit optimistic completion, as it maximizes completion opportunities without distracting the user with too many wrong suggestions. The respondents to our survey were overwhelmingly in favor of this. Finally, the user feedback we gathered by publicly releasing the tool pointed us to what users really wanted to see improved in subsequent versions of the tool (e.g., class name completion, and restoring the exploration possibilities), changes that were included in the second version of the tool.

10 Discussion

Despite the provably more efficient completion algorithms we presented –and their usefulness in practice–, 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 consistent among the different programs we tested. If an algorithm performs better in one, it tends to perform better on the others. Moreover, the respondents to our survey reported real-world improvements over the default completion algorithm.

Applicability to other languages. Our results are currently valid for Smalltalk only. However, the tests showed that our optimistic 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 IDEs 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 small number of matches. Indeed, two of the survey respondents specifically reported that they wished this behavior back. Programmers could use optimistic completion while typing (without explicit invocation), and still invoke the regular code completion algorithm using the old keyboard shortcut if they wish to explore the system. This would make the two approaches complementary.

Resource usage. Our benchmark in its current form is resource-intensive. Testing the completion engine several hundred thousands times in a row takes a few hours for each benchmark. We are looking at ways to make this faster. On the other hand, since the best performing algorithm uses a more limited number of matches, an optimistic code completion tool can actually be faster than a pessimistic one.

11 Related Work

We reviewed a number of completion approaches used in practice in Section 2. We review here the few academic contributions we are aware of in the domain of code completion. Beyond the classical completion algorithms, few works can compare with our approach, for the reasons we mentioned in Section 1: The lack of new data sources to improve code completion, the difficulty of evaluation without a benchmark-type approach such as ours, and the necessity of a large improvement to convince users.

Mylyn by Kersten and Murphy features a form of code completion based on task contexts prioritizing elements belonging to the task at hand [8], which is 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, which amounts to the “recently modified method names” algorithm). The data we recorded includes interactions only on a smaller period and could thus not be compared with the rest of the data. Mylyn’s completion is mentioned as a minor contribution in their paper, and is not evaluated separately.

Another completion mechanism is Keyword Programming by Little and Miller [9], 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. We see this approach as halfway between code completion and code search engines such as Google code search⁴, Koders⁵ or academic source code search engines such as Sourcerer by Bajracharya et al. [2] or S6 by Reiss [12].

A final completion mechanism is the one proposed by Bruch et al. [3]. They propose 3 completion algorithms based on the existing usage of classes in the code base, and compare their accuracy with the default completion mechanism by having their algorithms train on a part of the code base, and propose completions on the remaining part of the code base. There are several differences with our approach. First, the algorithms they propose learn from existing code, while the algorithms we experimented with use recent change information. Our algorithm are lighter-weight, and may adapt more quickly to the developer’s actions in the case where examples are not yet available to learn from. Second, their evaluation uses existing code bases that are divided in a training set and a testing set, while we use actual, recorded sequence of changes. Third, they evaluate their completion mechanism without prefixes, while we study the behavior of our algorithms for a varying length of prefixes. Similarly to us, they perform a second validation by having users perform a coding task with their best-performing mechanism and having them fill a questionnaire afterwards. A difference in the second step of evaluation is that we released the tool and let developers use it for a longer period of time during their actual coding activities, while they specified a task that developers worked on using one of their previously trained completion algorithms.

⁴ <http://www.google.com/codesearch>

⁵ <http://www.koders.com/>

12 Conclusion

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. We applied our approach to the two most numerous kind of entities in a software system, class and method names, and saw significant improvements in both areas.

Our optimistic completion algorithms return the correct match in the top 3 in close to 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 since 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.

We integrated our improved completion algorithm in a code completion tool named OCompletion, and we released it in two open-source communities. The informal feedback we received was very positive and guided us in our subsequent improvements of the tool. In addition, a survey of the users showed that they found it significantly more accurate than the previous tool they used, and that OCompletion's user interface, optimized for optimistic completion, was also a part of the increased usability of the tool. More telling to us is the fact that OCompletion is now included in the Pharo development environment and is as such used daily by its developers.

Acknowledgements

We thank D. Pollet and S. Krishnamurthi for discussions about this work. We acknowledge the financial support of the Swiss National Science foundation for the project "REBASE" (SNF Project No. 115990). We also thank the Squeak and Pharo users that downloaded OCompletion, gave us feedback, and answered our survey.

References

1. Arisholm, E., Gallis, H., Dybå, T., Sjøberg, D.I.K.: Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Trans. Software Eng.* **33**(2), 65–86 (2007)
2. Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 681–682. ACM, New York, NY, USA (2006). DOI <http://doi.acm.org/10.1145/1176617.1176671>
3. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: *ESEC/FSE'09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 213–222 (2009)
4. Conradi, R., Westfechtel, B.: Version models for software configuration management. *ACM Computing Surveys* **30**(2), 232–282 (1998)
5. Dehnadi, S., Bornat, R.: The camel has two humps (working title) (2006). URL <http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf>

6. Fazly, A.: The use of syntax in word completion utilities. Master's thesis, University of Toronto (2002)
7. Gail Murphy, Mik Kersten, Leah Findlater: How are java software developers using the eclipse ide? *IEEE Software* (2006)
8. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: *FSE '06: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 1–11 (2006)
9. Little, G., Miller, R.C.: Keyword programming in java. In: *ASE '07: Proceedings of the 22nd International Conference on Automated Software Engineering*, pp. 84–93 (2007)
10. Lung, J., Aranda, J., Easterbrook, S.M., Wilson, G.V.: On the difficulty of replicating human subjects studies in software engineering. In: *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pp. 191–200 (2008)
11. Miller, G.A.: The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review* **63**, 81–97 (1956). URL <http://users.ecs.soton.ac.uk/~{ }harnad/Papers/Py104/Miller/miller.html>
12. Reiss, S.P.: Semantics-based code search. In: *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pp. 243–253 (2009)
13. Robbes, R.: Mining a change-based software repository. In: *MSR '07: Proceedings of Fourth International Workshop on Mining Software Repositories*, p. 15. ACM Press (2007)
14. Robbes, R., Lanza, M.: An approach to software evolution based on semantic change. In: *FASE '07: Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, pp. 27–41 (2007)
15. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: *ICPC '07: Proceedings of the 15th International Conference on Program Comprehension*, pp. 155–164 (2007)
16. Robbes, R., Lanza, M.: Spyware: a change-aware development toolset. In: *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pp. 847–850 (2008)
17. Sharon, Y.: Eclipse — spying on eclipse. Bachelor's thesis, University of Lugano (2007)
18. Sim, S.E., Easterbrook, S.M., Holt, R.C.: Using benchmarking to advance research: A challenge to software engineering. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pp. 74–83 (2003)
19. Sim, S.E., Holt, R.C., Easterbrook, S.: On using a benchmark to evaluate c++ extractors. In: *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, p. 114. IEEE Computer Society, Washington, DC, USA (2002)
20. Wuyts, R.: Roeltyper: a fast type reconstructor for smalltalk. <http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/> (2007)