MIT Libraries | DSpace@MIT

MIT Open Access Articles

Code Completion From Abbreviated Input

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Sangmok Han, D.R. Wallace, and R.C. Miller. "Code Completion from Abbreviated Input." Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on. 2009. 332-343. © 2010 Institute of Electrical and Electronics Engineers.

As Published: http://dx.doi.org/10.1109/ase.2009.64

Publisher: Institute of Electrical and Electronics Engineers

Persistent URL: http://hdl.handle.net/1721.1/59377

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.

Massachusetts Institute of Technology

2009 IEEE/ACM International Conference on Automated Software Engineering

Code Completion From Abbreviated Input

Sangmok Han, David R. Wallace, and Robert C. Miller
Massachusetts Institute of Technology
Cambridge, MA
{sangmok, drwallace, rcm}@mit.edu

***Abstract*—Abbreviation Completion is a novel technique to improve the efficiency of code-writing by supporting code completion of multiple keywords based on non-predefined abbreviated input—a different approach from conventional code completion that finds one keyword at a time based on an exact character match. Abbreviated input is expanded into keywords by a Hidden Markov Model learned from a corpus of existing code. The technique does not require the user to memorize abbreviations and provides incremental feedback of the most likely completions. This paper presents the algorithm for abbreviation completion, integrated with a new user interface for multiple-keyword completion. We tested the system by sampling 3000 code lines from open source projects and found that more than 98% of the code lines could be resolved from acronym-like abbreviations. A user study found 30% reduction in time usage and 41% reduction of keystrokes over conventional code completion.**

Code Completion; Hidden Markov Model; Abbreviation; Multiple Keywords; Code Assistants; Data Mining

I. Introduction

Many programmers use code completion to accelerate code-writing by avoiding typing the whole character sequence of keywords. This paper describes a method for accelerating code completion still further by completing multiple keywords at a time based on non-predefined abbreviated input, utilizing frequent keyword patterns learned from a corpus of existing code. For example, Figure 1 shows a user entering *ch.opn(n);*, which is translated into a list of code completion candidates that includes *chooser.showOpenDialog(null)* as the most likely candidate. Entering slightly different abbreviations such as *cho.opdlg(nl);* or *cs.sopd(nu);* should also lead to the same best candidate because the system accepts non-predefined abbreviations of keywords.

The system aims to address the following limitations of the conventional code completion systems:

First, conventional systems complete only one keyword at a time. Therefore, the number of extra keystrokes increases in proportion to the number of completed keywords. For example, a typical code completion interaction, like the one shown in Figure 2, might take 20 keystrokes to complete three keywords of 29 characters.

Second, conventional code completion systems find a keyword based on an exact match of leading characters. Because the leading parts of keywords are often identical among candidates while the ending parts are more distinguishable, as in *showOpenDialog* and *showSaveDialog*, programmers often need to type potentially lengthy sequences of leading characters before they type distinctive characters close to the end.

Third, the default selection of a code completion candidate is often far from ideal. Conventional code completion selects the first match in an alphabetically sorted candidate list, ignoring any context implied by surrounding keywords. This leads to additional Up/Down Arrow keystrokes to adjust the selection to a correct candidate.

Here are a few motivating examples the new code completion system is designed to handle:

what you type…		*what you get…*
pv st nm	→	private String name
gval(r,c)	→	getValueAt(row,col)
f(i i=0;i<ls.sz();i++)	→	for(int i =0; i<list.size();i++)
pb st v m(st[] ag)	→	public static void main(String[] args)

This paper presents a new model and algorithm, based on a Hidden Markov Model (HMM), which has been integrated into a new code completion system called *Abbreviation*

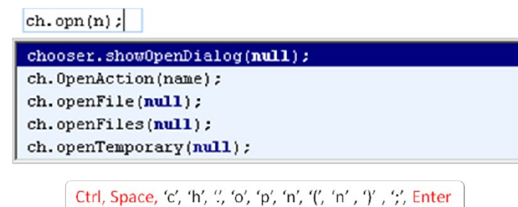


Figure 1. Abbreviation Completion can complete multiple keywords from abbreviated input in a single code completion dialog.

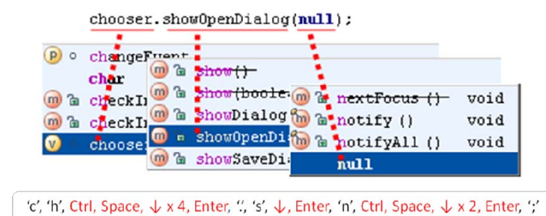


Figure 2. Code completion of multiple keywords requires many extra keystrokes for driving code completion dialogs.

This research was supported by Samsung Scholarship Foundation.

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.64

IEEE computer society

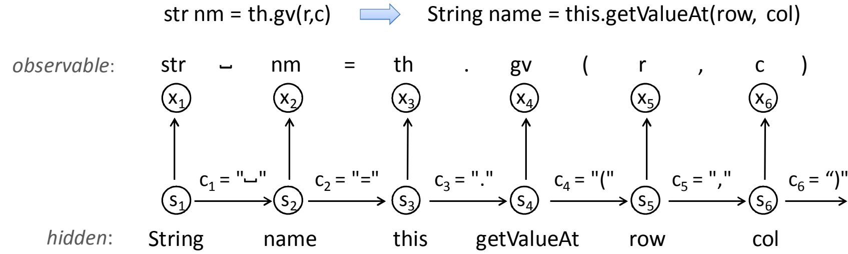


Figure 3. Resolution of multiple keywords is solved as a decoding problem of Hidden Markov Model.

Completion. An HMM has been applied to various engineering domains; however, Abbreviation Completion is believed to be its first application to the code completion domain. This paper also presents a new user interface for interactive multiple keyword completion because different usability concerns arise than in single keyword completion, such as manually overriding some of the keywords suggested by the system.

Compared to prior systems for completing multiple keywords, Abbreviation Completion does not require programmers to memorize predefined abbreviations, an improvement over code template systems [2,3], which convert a predefined token, such as *sysout*, into a predefined keyword sequence, such as *System.out.println()*, and provides incremental feedback of the most likely completions based on frequent keyword patterns, an improvement over Keyword Programming [4], XSnippet [5], and Prospector [6], all of which perform a single-pass search based on type constraints, optionally guided by heuristics based on keywords, frequencies, or paths. In addition, Abbreviation Completion is not limited to handling type-constrained expressions [4-6], but can generate any arbitrary, frequent sequence of keywords, including code for: declaration; loop construction; and a list of arguments. Previous work on improving the ordering of code completion candidates [7,8] can be related to Abbreviation Completion; however, our work focuses on prioritizing multiple-keyword code completion candidates using a difference source of information and a different algorithm.

Our key contributions are:

- A new application of an HMM to solve decoding of multiple keyword completion based on abbreviated input. An HMM has been extended in consideration of two distinctive characteristics of our problem domain.
- Development of a regression model for estimating a probabilistic distance between any given pair of an original keyword and an abbreviated keyword. This model is essential for the system to handle non-predefined abbreviated input.
- A user interface for interactive multiple keyword completion. A code editor supporting Abbreviation Completion has been implemented to demonstrate the new user interface. It can be downloaded from: http://cadlab6.mit.edu/sangmok/abbrcompletion/.
- Evaluation of the system's accuracy, time savings, and keystroke savings. Accuracy was measured by counting how many of the code lines sampled from open source projects could be reconstructed from abbreviated input. The system achieved average 98.9% accuracy against 3000 lines of code from six open source projects. Time savings and keystroke savings were measured by comparing time usage and the number of keystrokes of the system with those of a conventional code completion system. The system achieved average 30.4% savings in time and 40.8% savings in keystrokes in a user study with eight participants.

The next section describes models and algorithms for multiple keyword completion. We then present estimation methods to learn probability distributions required by the models. We describe a user interface for Abbreviation Completion. The setup and results of two evaluations follow. Finally, we discuss related work, future direction, and present conclusions.

II. Model and Algorithm

This section describes a model and algorithm to resolve the most likely keyword sequences from abbreviated input. Two structural extensions of an HMM and a modified backtracking mechanism of the Viterbi algorithm will be highlighted.

A. *Code Completion of Multiple Keywords As a Decoding Problem of a Hidden Markov Model*

An HMM is a graphical model that concerns two sequences. Only one of these sequences, called the observation symbol sequence, is observable while the other sequence, called the hidden state sequence, is the one of interest. Decoding of an HMM refers to a process of finding the most likely state assignment to a hidden state sequence based on an observation symbol sequence. The Viterbi algorithm [1] is an efficient dynamic programming algorithm for decoding an HMM.

Code completion of multiple keywords based on abbreviated input can be modeled as a decoding problem of a particular HMM. Figure 3 shows an example of the HMM for decoding *str nm = th.gv(r,c)*. Given two pieces of information, abbreviated input by a user and a set of keywords collected from a corpus of existing code, the two sequences of the HMM are created as follows:

First, an observation symbol sequence is created by splitting abbreviated input into a sequence of string tokens that have non-alphanumeric characters, such as spaces, dots or commas, between each one. For example, the result of splitting abbreviated input *str nm = th.gv(r,c)* is shown in Figure 3. Note that, in our implementation for Java language code completion, the underscore character was not treated as a splitting non-alphanumeric character because it is a valid character for the identifier name.

Second, a hidden state sequence is created by sequentially arranging keywords that have the same non-alphanumeric characters between each one, as in the observation symbol sequence.

Given the setup for creating the two sequences from abbreviated input and keywords collected from a corpus of existing code, finding the most likely code completion becomes a problem of finding the most likely assignment of keywords to a hidden state sequence.

B. An Extended Hidden Markov Model

We have extended a standard HMM [1] in two ways to take the following characteristics of our problem domain into account:

- Keyword transition always occurs through non-alphanumeric characters. For example, in Figure 3, transition from *String* to *name* occurs through a *space* character, and transition from *name* to *this* occurs through an *equal* character.
- The number of observation symbols is not finite because users are allowed to abbreviate keywords in their own non-predefined way.

The first extension is intended to exploit the keyword transition characteristics by modeling transition probability distributions in a more sophisticated way than conventional HMMs. Unlike conventional HMMs that employ a single transition probability distribution for any type of transition, the HMM for multiple keyword completion employs different transition probability distributions depending on the type of transition, identified by non-alphanumeric characters at the transition.

As a result, the transition probability distribution becomes dependent not only on a previous keyword but also on non-alphanumeric characters. Generally, allowing transition probabilities to be conditioned by a greater number of surrounding states results in better prediction, although more data is necessary for training. We assume that users can find training data that is large enough to take advantage of this extension because they have access to source code in their existing projects or in open source projects.

The second extension is to resolve issues with estimating emission probabilities, the probability of a hidden state generating an output symbol. Because the number of possible observation symbols is infinitely many, it is impossible to develop an estimation model that is guaranteed

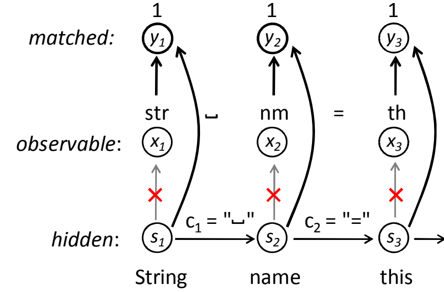


Figure 4. A modified HMM with match indicator nodes.

to produce a valid probability distribution—a valid emission probability distribution must sum to one when it is integrated over all possible observation symbols. However, it is possible to work around this issue by modifying the structure of an HMM as shown in Figure 4. The trick is to introduce a match indicator node, whose value is one if an observation symbol (an abbreviated keyword) is a correct match of a hidden state (an original keyword) and zero otherwise.

Then, we define a match probability as a probability of a match indicator becoming one given an observation symbol and hidden state pair. The match probability is equivalent to the emission probability in the sense that both measure a probabilistic distance between an abbreviated keyword and an original keyword. As we will see in the next section (II-C), the match probability is computationally equivalent to the emission probability, so it can replace the emission probability in the Viterbi algorithm. Once we have replaced the emission probability with the match probability, it is straightforward to develop a valid estimation model for the match probability because it is a probability of finite, binary events. For a formal description, we will use the following notation to characterize the HMM for multiple keyword code completion:

s_t : State at time *t* (original keyword).

S : A set of all possible states

x_t : Observation at time *t* (abbreviated keyword).

y_t : Match indicator at time *t*. The value of y_t is one if x_t and s_t are a correct abbreviation match. y_t is one at any time because x_t is assumed to be abbreviated input.

c_t : Non-alphanumeric characters between s_t and s_{t+1}, called a connector at time *t*.

$T(s)$: Start probability of state s.

$T_c(s'|s)$: Transition probability from state s to state s' through connector c.

$E(x|s)$: Emission probability of state s emitting observation symbol x.

$M(y|s,x)$: Match probability. $M(y=1|s,x)$ indicates the probability that state s and observation x are a correct match.

T : The length of an observation symbol sequence, which is always equal to the length of a hidden state sequence.

C. Modified Viterbi Algorithm

This section presents a modified version of the Viterbi algorithm for multiple keyword completion. Notably, two modifications have been made: First, the emission probability is replaced with the match probability; second, the backtracking part of the original Viterbi algorithm has been replaced with a new backtracking algorithm that retrieves *N*-best candidates, instead of the single best candidate of the original algorithm.

1) Match Probability

The Viterbi algorithm finds the most likely hidden state sequence by calculating joint probabilities of possible assignments of state values to a hidden state sequence. Because the joint probability distribution of the extended HMM is different from that of the original HMM (without match indicators), the Viterbi algorithm needs to be modified to accommodate the difference.

The modification just replaces the emission probability with the match probability. The difference is apparent comparing the joint probability distribution of the original HMM in equation (1) with the joint probability distribution of the extended HMM in equation (2). The only difference is that the emission probability $E(s_t \mid x_t)$ is replaced with the match probability $M(y_t \mid s_t, x_t)$.

$$P(s_1 \ldots s_T, x_1 \ldots x_T, c_1 \ldots c_T) = T(s_1) \prod_{t=2}^{T} T_{c_{t-1}}(s_t \mid s_{t-1}) \prod_{t=1}^{T} E(s_t \mid x_t) \quad (1)$$

$$P(s_1 \ldots s_T, x_1 \ldots x_T, c_1 \ldots c_T) = T(s_1) \prod_{t=2}^{T} T_{c_{t-1}}(s_t \mid s_{t-1}) \prod_{t=1}^{T} M(y_t \mid s_t, x_t) \quad (2)$$

2) Finding N-Best Candidates

The second modification to the Viterbi algorithm aims to enhance the way it handles backtracking, the final step used to construct the most likely state sequence using data structures calculated from a dynamic programming step. The original backtracking algorithm returns only one code completion candidate because it finds the single most likely sequence. However, a code completion system is expected to provide *N*-best code completion candidates.

Several backtracking approaches that find *N*-most likely sequences have been proposed [9-10]. We decided to take a tree-trellis based backtracking algorithm [9] because it is an exact algorithm that requires a negligible amount of computation when compared to computation for the dynamic programming step of the Viterbi algorithm: $O(T \cdot n^2 \log(n)) << O(T \cdot N^2)$. The lowercase *n*, the number of the *N*-best candidates, is generally much smaller than the capitalized *N,* the number of all possible states.

III. Parameter Estimation

Given a model and backtracking algorithm to compute the *N*-most likely code completions, this section describes a method to estimate model parameters. Three model parameters need to be estimated for the extended HMM with match indicators: start probability distribution, transition probability distribution, and match probability distribution. The first two probability distributions, which describe how keyword transition occurs at the beginning and in the remaining part of a sequence, are estimated from a corpus of existing code. The third match probability distribution, which describes how likely a pair of an abbreviated keyword and an original keyword to be a correct match, is estimated from examples of correct matches and incorrect matches.

A. Estimation of Start Probabilities and Transition Probabilities

1) Preparation of training examples

Generally, a set of known state sequences is used as training examples to estimate start probabilities and transition probabilities. In case of multiple keyword code completion, a corpus of existing code is used to generate training examples. We took the following two-step approach to convert a piece of source code into a set of state sequences:

In the first step, a lexical analyzer tokenizes source code into a sequence of lexical tokens that are one of the following types: identifiers, string literals, character literals, number literals, line breaks, non-line-breaking whitespaces, comments, and non-alphanumeric characters. For example, tokenizing *String name = null;<LF>* creates lexical tokens in Table I. This step allows us to identify keyword states: A lexical token of identifier type, string literal type, character literal type, or number literal type will form a keyword state in the state sequence.

In the second step, a state sequence generator scans through lexical tokens and selectively uses them to construct state sequences. Because keyword states were already identified in the first step, this step aims to identify connectors by concatenating non-alphanumerical characters and removing redundant whitespaces. A state sequence is extended by appending keyword states and connectors, one next to the other. The beginning of a new state sequence is signaled by a line or block separator, such as a semi-colon or curly bracket character in Java, followed by a line break. For example, lexical tokens of *String name = null;<LF>*

TABLE I. An Example of Lexical Analyzer Output

lexical token	type
String	identifiers
<space>	non-line-breaking whitespaces
name	identifiers
<space>	non-line-breaking whitespaces
equal	non-alphabetical characters
<space>	non-line-breaking whitespaces
null	identifiers
<semi-colon>	non-alphabetical characters
<LF>	line breaks

constructs the following state sequence in (3), in which ϕ indicates the end of a state sequence:

$$String \xrightarrow{\text{<space>}} name \xrightarrow{\text{<equal>}} null \xrightarrow{\text{<semi-colon>}} \phi \quad (3)$$

The implementation of the two-step approach is language-specific because it involves lexical analysis. However, it is often possible to customize a lexical analyzer to handle a different language simply by updating regular expressions so that they match language-specific lexical tokens.

2) Learning maximum likehihood estimates

Given state sequences extracted from a corpus of existing code, the maximum likelihood estimates of start probabilities and transition probabilities are calculated by counting the number of state transitions. Let us define $\text{COUNT}(i, s \xrightarrow{c} s')$ as the number of transitions from state s to state s' through connector c in the ith training example and $\text{COUNT}(i, s)$ as the number of occurrences of state s in the ith training example.

The maximum likelihood estimate of transition probabilities from state s to state s' through connector c is calculated as the ratio of the number of transitions from state s to state s' through connector c to the number of transitions from state s through connector c:

$$\hat{T}_c(s'|s) = \frac{\sum_i \text{COUNT}(i, s \xrightarrow{c} s')}{\sum_i \sum_{s'} \text{COUNT}(i, s \xrightarrow{c} s')} \quad (4)$$

The maximum likelihood estimate of start probabilities at state s is calculated from the ratio of the number of occurrences of state s to the number of occurrences of any state, as shown in (5). Notice that the number of occurrences is used instead of the number of beginning transitions. This is justified by the fact that a user can invoke code completion at any location in a code line. Therefore, the first keyword in a code completion does not necessarily represent its occurrence at the beginning of a code line, but can represent its occurrences at any location.

$$\hat{T}(s) = \frac{\sum_i \text{COUNT}(i, s)}{\sum_i \sum_s \text{COUNT}(i, s)} \quad (5)$$

In practice, transition probabilities based on exact counting, as in equation (4), may not generalize well because the number of training examples may not be sufficiently large to estimate transition probabilities of all possible state combinations. A popular technique to address this issue is to use the weighted sum of transition probabilities and occurrence probabilities as transition probabilities. A weight coefficient λ is set to 0.7 by following a common practice. Finally, the actual transition probability in the system is shown in equation (6).

$$\tilde{T}_c(s'|s) = \lambda \cdot \hat{T}_c(s'|s) + (1-\lambda)\hat{T}(s) \quad (6)$$

B. Estimation of Match Probabilities

1) An estimation model for match probabilites

The match probability indicates the probability of a match event, which occurs when a pair of an abbreviated keyword and an original keyword is a correct match. The estimation of match probabilities can be challenging because there are infinitely many combinations of abbreviated keywords and original keywords; therefore, should someone try to estimate match probabilities by directly counting the occurrences of match events, the person may need an infinite number of training examples.

We propose an approach based on a logistic regression model that predicts match probabilities from a set of similarity features between an abbreviated keyword and an original keyword. In this approach, an abbreviation pair—an abbreviated keyword and an original keyword—is represented as a feature vector, elements of which describe different aspects of similarity. Similarity features include the number of consonant matches or the percentage of matched letters in an original keyword. Table II shows a list of similarity features used in the latest implementation of Abbreviation Completion.

The feature-vector-based representation of abbreviation pairs allows us to use standard machine learning techniques to develop an estimation model of match probabilities. Notably, a logistic regression model fits well in our problem because it can learn probabilities of a binary event, like the match event.

2) Preparation of training examples

To train a logistic regression model, a set of positive training examples and a set of negative training examples are necessary. To prepare positive training examples, 400 abbreviation pairs were generated by two human volunteers. Some examples of the abbreviation pairs are *buffer→bf*, *getProperty→gppt*, and *moveCaretPosition→mvcarpos*. Then the abbreviation pairs were converted into a feature vector representation by calculating similarity features. 400 negative training examples were generated by pairing original keywords with randomly selected wrong abbreviations and converting the wrong pairs into a feature vector representation.

3) Learning maximum likelihood estimates

TABLE II. SIMILARITY FEATURES FOR ESTIMATING MATCH PROBABILITIES

feature name	feature description
$Sim_1(s,x)$	number of consonant letter matches
$Sim_2(s,x)$	number of capitalized letter matches
$Sim_3(s,x)$	number of non-alphabet character matches
$Sim_4(s,x)$	number of standard abbreviation matches
$Sim_5(s,x)$	number of letter matches with ordering ignored
$Sim_6(s,x)$	number of letter matches with ordering enforced
$Sim_7(s,x)$	percentage of matched capital letters in *s*
$Sim_8(s,x)$	percentage of matched consonant letters in *s*
$Sim_9(s,x)$	percentage of matched letters in *s*
$Sim_{10}(s,x)$	percentage of matched letters in *x*

Having collected 800 positive and negative examples, 200 examples were held for testing and 600 examples were used for training. The logistic regression model has 11 unknown parameters, denoted as $\beta_0, \beta_1, \cdots, \beta_{10}$, because 10 similarity features are included in the model. The logistic regression model for match probabilities is shown in equation (7):

$$M(y=1 \mid s,x) = g(\beta_0 + \beta_1 \cdot Sim_1(s,x) + \cdots + \beta_{10} \cdot Sim_{10}(s,x)) \quad (7)$$

$$\text{where } g(z) = \frac{1}{1+e^{-z}}$$

The maximum likelihood estimates of $\beta_0, \beta_1, \cdots, \beta_{10}$ were calculated by a generalized linear model regression function in a statistics package. The result is shown in equation (8). The train error and test error of the logistic regression model were recorded as 1.5% and 0.5%, respectively. Because the test error is slightly lower than the train error, the model is not expected to have an issue with overtraining.

$$[\beta_0, \beta_1, \cdots, \beta_{10}] = [-41.2, 1.2, 8.7, 7.5, 2.3, -1.3, -1.2, -0.07, -0.04, 0.29, 0.44] \quad (8)$$

4) Discussion of the training results

The training result reflects the relative importance of different similarity features, although it is specific to a certain style of abbreviation exhibited by our training data. We first notice that β_2 for the number of capitalized letter matches has the largest value among all number-of-matches parameters $\beta_1, \cdots, \beta_6$. It implies that an abbreviation pairs is likely to be a match if there are many capitalized letter matches. β_5 and β_6 for the number of letter matches with and without ordering are given negative values. Considering that β_1 for the number of consonant matches is positive, it implies that the larger number of vowel matches an abbreviation pair has, the less likely it is a correct match.

It is noteworthy that different users may have different preferred ways of abbreviating keywords, and even the same user may not abbreviate consistently over time. If such differences necessitate retraining of match probability parameters for each user or at any point of time, it would make the Abbreviation Completion system very expensive to use. However, the parameter values in equation (8) were found to work reasonably well in running an artificial corpus study and a user study in Section V and VI, without further retraining. This is promising because it indicates that the parameter values can serve as system defaults, which may attract users to the system. The default parameter values can later be updated from actual abbreviation examples collected from system usage. This updating is to be considered in future work.

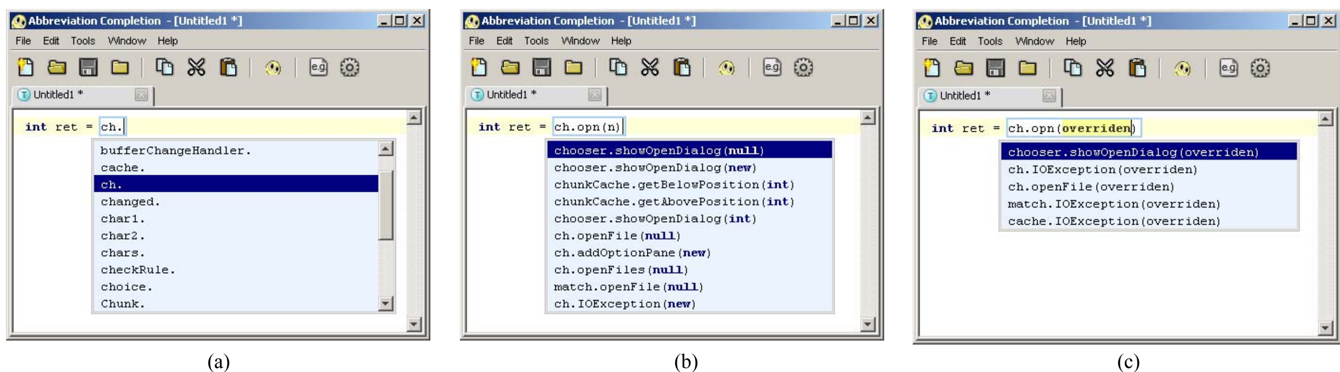
IV. User Interface And Implementation

This section describes a user interface for multiple keyword code completion. Three design requirements have been identified for the user interface: acceptance of abbreviated input that may include non-alphanumeric characters including spaces; allowance for users to override the system's suggestion; and display of code completion candidates in an effective way. The user interface has been implemented on a demonstrational code editor.

A. User Interface

It is important to support a scenario in which some part of a code line is typed, while another part is completed using Abbreviation Completion. For example, a user may first type *this.getValueAt()* and then try to complete *row, col* inside the parentheses from abbreviated input *r,c*. The demonstrational code editor utilizes an input popup that floats over the code editing area, shown as a light blue rectangle in Figure 5. The area accepts abbreviated input, including non-alphanumeric characters. The input popup appears at the current caret position when a user presses Ctrl+Space, so it can be used to insert completed code at any location in the code. Since the input popup tries to initialize its content from any highlighted text in the code editing area, a user may type abbreviated input in the code editing area and expand it by highlighting and pressing Ctrl+Space.

The second requirement is handled by a keyword-pinning capability, which is invoked using a keyboard shortcut Ctrl+B. The system highlights a pinned keyword in the input



(a) (b) (c)

Figure 5. A user interface for multiple keyword code completion: (a) the list is alphabetically sorted when there is one keyword, (b) the list is sorted by the likelihood when there are more than one keyword, and (c) a user can override the system's suggestion

popup by making it boldface with a light yellow background, as shown in Figure 5-(c). Internally, the system treats a pinned keyword as an observation symbol that matches all possible states with an equal probability. Once code completion candidates are generated using the match probabilities, they are displayed to users with keyword tokens at a pinned location replaced with overriding text.

Regarding the third design requirement, users are assumed to have two concerns when using the code completion candidate list. First, users want to know which candidates are more likely to be correct and should be examined first. A list sorted by the likelihood is useful for this purpose. Second, users want to navigate the list of candidates efficiently in a predictable manner. An alphabetically sorted list is useful for this purpose. We implemented the code completion candidate list such that it can be sorted one of the two ways depending on the number of keywords in the input popup. When there is only one keyword, the list is sorted alphabetically; the system's default suggestion can be often incorrect because the transition pattern cannot be utilized. When there is more than one keyword, the list is sorted by the likelihood; two keywords are often enough to locate the correct candidate in the top-10 list, so users may want to examine candidates from the most likely one. Figure 5-(a) and Figure 5-(b) demonstrates the behavior of the candidate list.

B. *Incremental feedback of Code Completions*

Responsive incremental feedback is essential for the usability of the multiple keyword code completion system. We applied a filtering technique to improve responsiveness of the system. The filtering technique effectively reduces the search space of the Viterbi algorithm by removing some of the states that are impossible to appear in the most likely candidates. Filtering based on characters and connectors (the non-alphanumeric characters between keywords) has been implemented. For example, given abbreviated input "*sys.",* the system first applies character-based filtering so that only the states that have both 's' and 'y' characters remain. Then the system applies connector-based filtering so that only the states that have a transition through '.' remain.

C. *Incremental Indexing of Source Code*

The demonstrational code editor supports incremental indexing of source code using a background thread. The code editor monitors changes of source code, which may occur inside the code editor or on the file system, and updates the HMM to reflect the changes. As a result, Abbreviation Completion can be used to complete code lines that may include recently introduced variable or method names. A full indexing of 400 source code files (3000 kilobytes in size) usually takes less than 3 seconds on a laptop computer with Intel Core 2 Duo P8400 CPU and 3 gigabyte ram. Therefore, an incremental indexing of a few changed entries can be processed in a negligible time compared to a normal lag between code edits. Note that users can specify the target directories for incremental indexing in a configuration file.

V. ARTICIFIAL CORPUS STUDY

The artificial corpus study aims to evaluate the accuracy of the Abbreviation Completion system. A total of 3000 frequent code lines was collected from 6 open source projects. The code lines were converted into acronym-like abbreviations by applying a particular text transformation rule. We measured how many of the original code lines could be completed from the abbreviated code lines. We report top-*N* accuracy, which refers to the rate of finding the original code line within the top-*N* candidates of code completion.

A. *Study Setup*

1) *Selection of Open Source Projects*

Six open source projects were selected from 14 open source projects that were used in a previous artificial corpus study of Keyword Programming in [4].

The six open source projects are: CAROL, a library for using different RMI implementations; DNSJava, an implementation of DNS in Java; JEdit, a source code editor implemented in Java; JRuby, a Java implementation of the Ruby programming language; RSSOwl, a news reader application for RSS feeds; and TV-Browser, a Java-based TV guide application.

2) *Preparation of Original Code Lines*

The 500 most frequent code lines, which are at least 20 characters long and include at least 2 keyword tokens, were collected from each open source project; a total of 3000 code lines was collected from six open source projects. Frequent code lines were selected since they are more likely to be a target of multiple keyword code completion. The minimum length requirement is introduced to exclude short code lines that are not likely to be targets of multiple keyword code completion. It also helps to include more of the challenging, longer code lines in the test set. Code lines are required to have at least 2 keyword tokens because we are interested in evaluating the code completion of multiple keywords.

3) *Preparation of Abbreviated Code Lines*

Given the large number of code lines, we decided to generate their abbreviations using an artificial abbreviation generator, implemented as a computer program. The artificial abbreviation generator creates acronym-like abbreviations with the maximum length of three characters, such as *bao* abbreviated from *ByteArrayOutputStream; th* abbreviated from *throw;* and *i* abbreviated from *if*. The artificial abbreviation generator takes the following steps to transform a keyword into an abbreviation:

- First, it determines the target length of an abbreviation to be created, which is always between one and three characters based on the following equation:

$$\text{Target length} \leftarrow \max(1, \min(3, \text{ceil}(\text{Keyword length} \times 40\%))) \quad (9)$$

- Second, it appends the first letter and as many capitalized letters (0, 1, or 2 letters) as possible to the abbreviation within the limit of the target length. All characters are appended in lower case.
- Third, if the abbreviation is still shorter than the target length, append letters following the first letter one by one until the abbreviation has the target length (for *Object*, *obj*).

Using this computer-based approach, we limit ourselves to testing the system against the particular style of abbreviation with potential biases. However, acronym-like abbreviation is believed to be one of popular ways of abbreviating keywords and therefore the result of this study may provide a reasonable estimate of the system's performance against human-generated abbreviations. We also imposed the maximum length limit of three characters as an effort to make a conservative estimate.

4) Test Procedure

To measure top-*N* accuracy of six open source projects, we repeated the following steps for each open source project:

- Train an HMM from the source code.
- Decode abbreviated code lines using the HMM. Count the number of successful decoding.
- Calculated top-*N* accuracy by dividing the occurrences of successful decoding within top-*N* candidates by the number of code completion invocations.

B. Results and Discussion

1) Top-N Accuracy

The top-10 accuracy against 3000 code lines from six open source projects was 98.9%. The top-5, top-3 and top-1 accuracies were 97.2%, 94.3%, and 80.0%, respectively. Table III shows accuracy values of individual open source projects.

There are two positive findings about the system's performance on accuracy. First, the accuracy itself is remarkably high, close to 99%. Although the accuracy is measured against a particular style of acronym-like abbreviations, such a high accuracy shows potential for achieving similarly high accuracy against human-generated abbreviations. Second, the system's accuracy is consistent across the six open source projects, which may involve different class libraries, use different naming conventions, and exhibit different code patterns. Figure 6 shows that there is no noticeable difference in top-*N* accuracy across the projects.

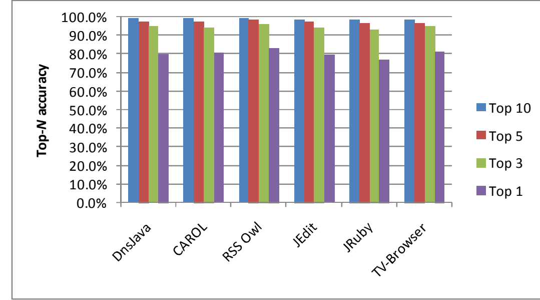


Figure 6. The system gives consistent top-*N* accuracy across the open source projects.

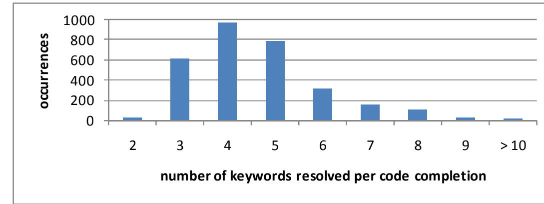


Figure 7. The histogram of the number of keywords resolved per code completion.

2) Time per code completion

Code completion of a code line took 0.41 seconds on average over 3000 code lines. The right-most column in Table III summarizes the average code completion time of the six open source projects. Code completion time varies considerably across the projects. It is because the time complexity of code completion increases in proportion to the number of keywords and the number of transitions.

Since none of the projects took more than one second for code completion on average, the responsiveness of the current implementation is considered acceptable. However, there is a room for improvement because two of the projects, JRuby and TV-Browser required more than 0.7 second per code completion, which is a noticeable lag.

3) Number of resolved keywords per code completion

TABLE III. THE SUMMARY STATISTICS OF HIDDEN MARKOV MODELS AND CODE COMPLETION RESULTS

	Source Code		HMMs			Code Completion Results				
project	**files**	**code lines**	**keywords (a)**	**transitions (b)**	**ratio (b)/(a)**	**top-10 accuracy**	**top-5 accuracy**	**top-3 accuracy**	**top-1 accuracy**	**time per code completion**
DnsJava	*123*	*35,658*	*2,962*	*15,790*	*5.3*	*99.0%*	*97.2%*	*94.8%*	*79.8%*	*0.10 sec*
CAROL	*157*	*44,166*	*2,729*	*12,810*	*4.7*	*99.2%*	*97.6%*	*94.2%*	*80.4%*	*0.11 sec*
RSS Owl	*412*	*147,162*	*7,579*	*40,565*	*5.4*	*99.0%*	*98.4%*	*96.0%*	*83.2%*	*0.25 sec*
JEdit	*394*	*233,908*	*13,803*	*80,769*	*5.9*	*98.8%*	*97.2%*	*94.0%*	*79.6%*	*0.43 sec*
JRuby	*677*	*320,124*	*21,157*	*123,115*	*5.8*	*98.8%*	*96.6%*	*93.0%*	*77.0%*	*0.70 sec*
TV-Browser	*852*	*348,942*	*20,358*	*112,830*	*5.5*	*98.6%*	*96.8%*	*94.8%*	*81.0%*	*0.88 sec*
Average	***436***	***188,326***	***11,431***	***64,313***	***5.4***	***98.9%***	***97.3%***	***94.5%***	***80.2%***	***0.41 sec***

The number of keywords in each code line was recorded to inspect how many keywords were resolved per code completion. A histogram in Figure 7 shows that resolution of 3 to 5 keywords was most frequent. The average number of keywords was 4.6.

4) *Statistics about Hidden Markov Models*

Table III shows statistics of HMMs trained by source code of six open source projects. The number of code lines is measured by counting effective code lines ignoring blank lines and comments. One interesting finding is that the ratio between the number of transitions and the number of keywords is about 5 in all six open source projects. This implies that a graph connecting keywords (nodes) through transitions (edges) is very sparse because a keyword is connected to just five keywords on average among many possible keywords.

5) *Inspection of Unsuccessful Code Completions*

There were 33 unsuccessful code completions among 3000 trials, in which none in the top-*10* candidates was the original code line. Two common failure types were identified by inspecting them.

Failure Type I: This type of failure is caused by the *new* keyword in Java language. Not only many keywords (states) make a transition to the *new* keyword (a state) but the *new* keyword also makes a transition to many keywords. Because transition probabilities of the first-order HMM like ours are conditioned by only one previous state, such a universally connected previous state cannot provide useful guidance in decoding. This failure type applies to 18 of 33 failures.

Failure Type II: This type of failure is caused by similar keywords making similar transitions. In this study, we have restricted the maximum length of an abbreviated keyword to three characters. The HMM tries to resolve ambiguity introduced by such a short abbreviation using a transition pattern. However, there are cases in which similar keywords make transitions in a similar fashion. Then it becomes difficult for the HMM to locate the original code line within the top-*N* candidates. For example, a code line *} catch (Exception e1)* could not be resolved from its abbreviation *} ca (exc e)* because there were other similar keywords making similar transitions such as *} catch (Exception e)*, *}* or *} catch (IOException e)*. This type of failure potentially applies to all failures.

VI. USER STUDY

The user study focuses on evaluating time savings and keystroke savings when a programmer uses the Abbreviation Completion system. The time usage and the number of keystrokes needed in the Abbreviation Completion system are compared with those needed in a conventional code completion system in *Eclipse*, a popular Java development tool. We report that time savings and keystroke savings were 30.4% and 40.8% and the difference in time and keystrokes was statistically significant.

```
JPanel content = new JPanel(new BorderLayout())
content.add(BorderLayout.CENTER, panel)
public void actionPerformed(ActionEvent evt)
label.setHorizontalAlignment(SwingConstants.CENTER)
GridBagLayout layout = new GridBagLayout()
cons.anchor = GridBagConstraints.WEST
label.setBorder(new EmptyBorder(0,0,0,12))
fireTableRowsUpdated(row,row)
SwingUtilities.invokeLater(new Runnable())
StringBuffer buf = new StringBuffer()
```

Figure 8. Code lines used in the user study to measure time usage and keystrokes needed for code-writing using two code completion systems.

A. *Paritipants*

Eight Java programmers were recruited using flyers and a mailing list in a college campus. They were informed that the user study would take about 30 minutes and one of the participants would be awarded a $25 gift certificate. There were six males and two females among the participants. The average age of the participants was 28.1.

All of them had a minimum of 5 years of general programming experience. Five people had used Eclipse for more than 3 years while three people had not used it or used it just briefly because they used different Java development tools, which they confirmed have a code completion capability similar to Eclipse.

B. *Usage Scenario and Assumptions*

We are interested in evaluating code completion systems in a particular test scenario, in which a programmer writes lines of code based on a concrete idea of what needs to be written. That is, a programmer can write multiple keywords without having to stop to ponder about the next keyword.

To simulate such a code-writing scenario, we decided to provide our subjects with a visual reference of code lines, which was always visible on the computer screen. We assumed that such a visual reference could work as an external memory, which would enable our subjects to type multiple keywords continuously as if they had what needs to be written in their minds. We also assumed that using a visual reference would not slow down code-writing significantly as long as subjects were familiar with the code lines in the visual reference.

C. *Study Setup*

The user study was performed at an office area in a college campus using a computer with a full-sized keyboard. One subject, assisted by one experiment facilitator, performed a set of code-writing tasks at each run of the user study.

The two code completion systems under investigation are called *Abbreviation Completion* and *Eclipse Code Completion.* To counterbalance the effect of trying one system first and the other later, we separated subjects into two groups. The first group, named *Abbreviation-First*, started using Abbreviation Completion first while the second

group, named *Eclipse-First*, started using Eclipse Code Completion first.

D. Tasks

The user study starts with the first task of learning two code completion systems. Let us assume that a subject from the Abbreviation-First group performs the first task. After the facilitator explains how to use Abbreviation Completion, the facilitator lets the subject practice using Abbreviation Completion. A subject is allowed to ask questions during the practice. For practice, the subject is required writes code lines in Figure 8 using Abbreviation Completion. Once the subject finishes writing the code lines, the facilitator explains how to use Eclipse Code Completion. A same practice session follows using Eclipse Code Completion.

The second task is a recording session to record time usage and keystrokes in writing code lines in Figure 8. Note that a subject is asked to write the same code lines that they already have written twice because it may help simulate the usage scenario of our interest. Let us assume that a subject from the Abbreviation-First group performs this task. The subject first writes the code lines using Abbreviation Completion. The time usage and keystrokes are unobtrusively recorded using custom-built instrumentation facilities in code editors. Once the subject finishes writing the ten code lines, the subject will have a short break and then repeat the same recording process using Eclipse Code Completion.

The ten code lines in Figure 8 were selected from *JEdit*, one of open source projects used in our artificial corpus study, through a random walk of its source code to find code lines that satisfy the following characteristics: each code line should appear at least four times in the whole project; each code line should be at least 30 characters long and at most 50 characters long; and selected code lines should demonstrate various styles of code-writing such as instantiations, assignments, declarations, member access, and parameters.

E. Results

1) Time savings

The overall time savings averaged for all subjects and for all code lines was 30.4%, as shown in Figure 9. The detailed time usage is presented in two ways, first by averaging for all code lines (Figure 9) and second by averaging for all subjects (Figure 10). Time savings were calculated by dividing the difference of time usage in two code completion systems by the time usage of Eclipse Code Completion. The standard deviation of time savings across subjects was 8.4%, indicating that there is a certain amount of variation in time savings depending on individual subjects. The standard deviation of time savings across code lines was 6.3%. The difference in the time usage between Abbreviation Completion and Eclipse Code Completion was statistically significant based on a paired t-test ($df = 79$, $p < 0.001$).

2) Keystroke savings

The overall keystroke savings averaged for all subjects and for all code lines was 40.8%, as shown in Figure 11, which is larger than the overall time savings. The number of keystrokes is presented in two ways, first by averaging for all code lines (Figure 11) and second by averaging for all subjects (Figure 12). The standard deviation of keystroke

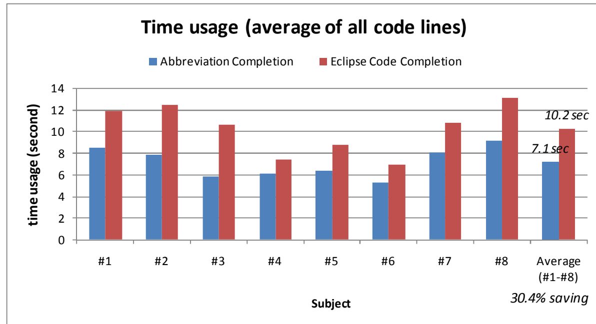


Figure 9. Time usage average of all code lines for each subject.

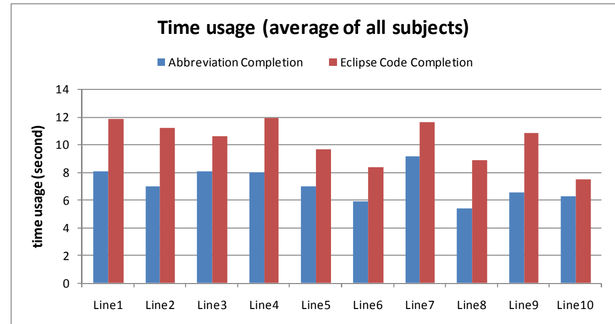


Figure 10. Time usage average of all subjects for each code line.

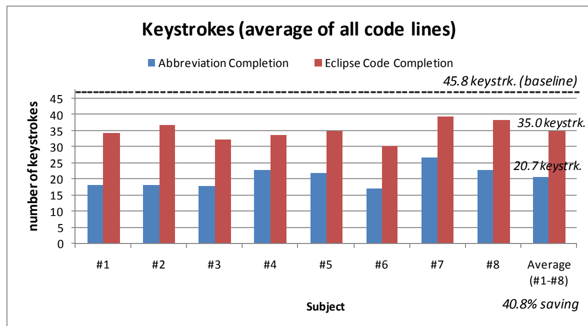


Figure 11. Keystrokes average of all code lines for each subject. The baseline keystrokes, also an average of all code lines, are shown as a dotted line.

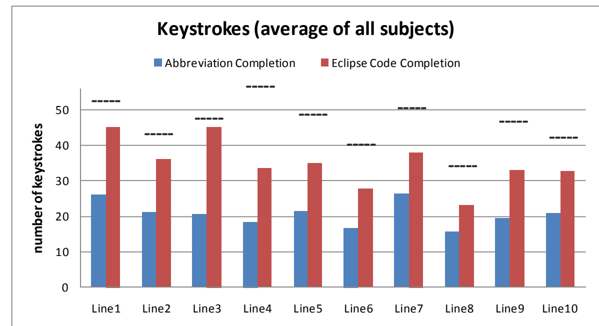


Figure 12. Keystrokes average of all subjects for each code line. The baseline keystrokes are shown as dotted lines.

savings was 6.7% across subjects and 7.5% across code lines. The difference in the number of keystrokes between Abbreviation Completion and Eclipse Code Completion was statistically significant based on a paired t-test ($df = 79$, $p < 0.001$).

Unlike the time usage, the number of keystrokes has a baseline value. A baseline value is the number of keystrokes when the whole character sequence in a code line is typed without using any code completion. Black dotted lines in Figure 11 and Figure 12 show the baseline values. Multiple dotted lines are shown in Figure 12 because each code line has its own baseline value. Comparing average keystrokes by Abbreviation Completion (20.7 keystrokes) with the baseline value (45.8 keystrokes), we see 54% of keystroke savings. Meanwhile, average keystrokes by Eclipse Code Completion (35.0 keystrokes) is just 24% less than the baseline value.

F. Discussion

In the user study, the Abbreviation Completion system achieved substantial savings in time and keystrokes. It is noteworthy that the keystroke savings were larger than the time savings. Obviously, the time usage is not a linear function of keystrokes; it is also a function of various mental operations, which could not be measured directly in the user study.

From our observation of subjects' behavior, one noticeably time-consuming mental operation was a validation of code completion candidates. After making some keystrokes, subjects stopped to check if the list of code completion candidates had the intended code line. Because the system showed ten code completion candidates, it could take seconds to scan through the list if the correct one did not appear near the top in the list.

One subject told us that validation was more difficult in Abbreviation Completion than in Eclipse Code Completion. The subject explained that it was because multiple keywords had to be validated all at once. Another subject told us that the subject had a desire to hit the Enter key right after typing abbreviated input without validating the candidates. He said that it was because the system's first suggestion seemed usually correct.

We think that both of subjects' comments point to a single usability issue of the Abbreviation Completion system. Information needed for validating code completion candidates is not clearly visible to users. Users only see abbreviated input and code completion candidates. Information about how well or why a code completion candidate matches the abbreviated input is not visible to users. Also, from the very nature of code completion, it is difficult to visually compare code completion candidates with the intended code line because the intended code line is in user's mind.

To enhance the visibility, we may improve the system in two ways. First, we may expose the system's confidence about code completion candidates. It can help users know when they need to be more careful about validation. The system's confidence may be calculated from two sources of information: a probabilistic model using the HMM and a history data of using the Abbreviation Completion system. Second, we may expose information about why a code completion candidate makes a match of abbreviated input. Instead of trying to explain a complicated probabilistic model, an approximated model that can be communicated easily may serve this purpose well. A simple visualization technique, such as highlighting which part in abbreviated input matches which part in a code completion candidate, should be useful as well.

VII. Related Work

Saving keystrokes and time for code-writing is one of the major design objectives of source code editors. Generating multiple keywords from a short character sequence is one way of achieving the objective. There have been two major approaches for supporting multiple keyword generation.

The first approach is based on a code template, a predefined code fragment that can be inserted into the code editor. Each code template is given an alias, such as *sysout*, so that programmers can insert a code fragment using the alias as a reference. Many code editors, including Emacs [2] and Eclipse [3], implement this approach.

The code template approach is effective at handling a handful of very frequently written code fragments. However, the burden of memorizing aliases can put a limit on the number of frequent code fragments a user can complete. The time-consuming process of adding new code templates may also be a limiting factor. Abbreviation Completion tries to overcome such limitation by supporting non-predefined abbreviations and by just requiring users to specify the locations of source code. Abbreviation Completion can automatically construct an equivalence of a library of code templates from a corpus of source code.

The second approach is based on a type-constrained search that can construct programming expressions containing multiple keywords. Keyword Programming [4], XSnippet [5], and Prospector [6] implement this approach, but they have differences in the type of input queries and output expressions and the kind of heuristics for guiding type-constrained search. Prospector takes two Java types as an input and outputs code lines for converting one type to the other type using heuristics based on the graph path. XSnippet also takes two Java types as an input, but optionally it can take additional Java types to specify a context. XSnippet uses heuristics based on snippet lengths, frequencies, and context matches to generate multiple lines of code for instantiation. Keyword Programming takes a set of keywords as input and outputs a code line of Java expression using heuristics based on the keyword matches.

Type-constrained search systems focus on serving a specific type of users who need help with choosing or using classes. They do not serve a different type of user who already has a good idea of what needs to be written and wants to write it more efficiently. Abbreviation Completion

demonstrates that some of the techniques used in type-constrained search systems —notably, code mining and text-based hints—can be used to serve the other type of users.

Abbreviation Completion can also be related to previous work on improving the ordering of code completion candidates [7,8] because in essence the Abbreviation Completion algorithm tries to solve the problem of prioritizing a large number of code completion candidates using an HMM. Prototype systems in [7] and Mylyn [8] explore the utility of the change history and the task context, respectively, as additional sources of information for prioritizing single-keyword candidates. Abbreviation Completion explores the utility of keyword sequences extracted from a corpus of source code for prioritizing multiple-keyword candidates.

VIII. Conclusion and Future work

This paper has presented Abbreviation Completion, a novel technique to complete multiple keywords at a time based on non-predefined abbreviated input. We presented an algorithm based on an HMM to find the most likely code completions. We presented a method to learn parameters of the HMM from a corpus of existing code and examples of abbreviations. A new user interface for multiple keyword code completion has been implemented on a demonstrational code editor. The accuracy of the Abbreviation Completion system is evaluated in an artificial corpus study, in which 3000 code lines from six open source projects were completed from their abbreviations. The system achieved average 98.9% accuracy. Time savings and keystroke savings were evaluated in a user study, in which the Abbreviation Completion system was compared with the Eclipse Code Completion system. The overall time savings and keystroke savings were 30.4% and 40.8%.

One of the important goals of future work is to further improve the usability of the system. A user interface for validating code completion candidates deserves investigation because the user study revealed that the validation can be time-consuming and difficult. Improving efficiency is another goal because the system tended to be less responsive when the number of keywords was over 20,000. Finally, because the key benefit of our approach is keystroke savings, application to programming environments with limited input capabilities, such as mobile devices, may be worthy of investigation.

Acknowledgment

We thank CAD Lab members who provided a creative environment, UID group members who provided helpful discussion, and study participants for their time and comments. We thank anonymous reviewers for their insightful comments.

References

[1] L. R. Rabiner, "Tutorial on hidden Markov models and selected applications in speech recognition," In Proc. IEEE, vol. 77, 1989.

[2] “Abbrevs,” in GNU Emacs Manual. http://www.gnu.org/software/emacs/manual/emacs.html.

[3] “Editor Template,” in Eclipse Ganymede Documentation. http://help.eclipse.org/ganymede/index.jsp.

[4] G. Little and R. C. Miller, "Keyword programming in Java," In Proc. ASE, vol. 16, pp. 37-71, 2007.

[5] N. Sahavechaphan and K. Claypool, "XSnippet: Mining For sample code," OOPSLA, pp 413-430, 2006.

[6] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," In Proc. PLDI, 2005.

[7] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion," In Proc. ASE, 2008.

[8] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," In Proc. FSE, 2006.

[9] F. K. Soong and E. F. Huang, “A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition,” In Proc. ICASSP, vol. 1, pp 705-708, 1991.

[10] D. Nilsson and J. Goldberger, “An efficient algorithm for sequentially finding the n-best list,” In Proc. IJCAI, 2001.