# Cookbook: In Situ Code Completion using Edit Recipes Learned from Examples

John Jacobellis, Na Meng, Miryung Kim
University of Texas at Austin, USA
jwjacobellis@gmail.com, mengna09@cs.utexas.edu, miryung@ece.utexas.edu

## ABSTRACT

Existing code completion engines leverage only pre-defined templates or match a set of user-defined APIs to complete the rest of changes. We propose a new code completion technique, called COOKBOOK, where developers can define custom *edit recipes*—a reusable template of complex edit operations—by specifying change examples. It generates an abstract edit recipe that describes the most specific generalization of the demonstrated example program transformations. Given a library of edit recipes, it matches a developer's edit stream to recommend a suitable recipe that is capable of filling out the rest of change customized to the target. We evaluate COOKBOOK using 68 systematic changed methods drawn from the version history of Eclipse SWT. COOKBOOK is able to narrow down to the most suitable recipe in 75% of the cases. It takes 120 milliseconds to find the correct suitable recipe on average, and the edits produced by the selected recipe are on average 82% similar to developers' hand edit. This shows COOKBOOK's potential to speed up manual editing and to minimize developers' errors. Our demo video is available at `https://www.youtube.com/watch?v=y4BNc8FT4RU`.

## Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Interactive environments

## General Terms

Experimentation, Performance

## Keywords

code completion, edit recipe

## 1. INTRODUCTION

Automated code completion is a feature provided by many source editors and it has become an essential arsenal of modern software development. Existing code completion engines either provide simple "quick fix" for API method calls and object instantiation values, or perform complex edit operations based on predefined templates. Several code completion approaches [1, 5, 9] rank certain method targets higher than others based on the patterns found in the codebase or version histories. Omar et al. [8] suggest possible values to use based on a predefined value set. However, these tools cannot perform complex edit operations customized to individual targets. Ge et al. and Forster et al. [3, 4] monitor a user's editing actions to predict which type of refactoring the user intends to apply and then complete the rest of refactoring operations. However, they are limited to predefined refactoring operations supported by IDEs. Nguyen et al. [7] support code completion of complex API usage adaptations in client applications but require developers to manually input API usage patterns to extend code completion capability.

We propose a new code completion technique, called COOKBOOK. Developers can define custom *edit recipes*—a reusable template of complex edit operations—by specifying change examples. When a user provides one or more method-level change examples by selecting their old and new versions, COOKBOOK generates an abstract edit recipe that shows the most specific generalization of changes demonstrated by the change examples. This edit recipe then can be applied to different target contexts where control and data flow contexts match but use different type, method, and variable names. Specifically, for each edit recipe, COOKBOOK first extracts the common edit operations among a set of changed methods. It then creates a general representation for the extracted edit operations. Next, it identifies context relevant to the common edit operations in each method based on control and data dependences. Fourth, it extracts the largest common context between the identified contexts. Finally, by combining the common edit operations and common context, COOKBOOK creates an *edit recipe*—an abstract program transformation script that is capable of transforming similar code fragments. Once users have a collection of recipes in COOKBOOK, their edit operations are matched with the recipes frequently so that any potentially applicable recipes are found as early as possible and listed for developers to select and automatically complete the rest of editing. COOKBOOK is built on our prior work LASE [6]. While LASE focuses on matching *codebase* against a single recipe to perform similar edits to all matching code locations, COOKBOOK matches an *incoming edit stream* against multiple recipes to find the most suitable recipe. COOKBOOK therefore includes a new algorithm of matching incoming edits to AST-level edits in the recipe library and an algorithm to rank suitable recipes. In this



*ICSE Companion'14*, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591076

paper, we present a COOKBOOK Eclipse plug-in with the following features:

- A user can define a new *edit recipe* by specifying more than one exemplar changed method. Once an edit recipe is defined, a user can review the edit recipe with a graphical viewer. More information on recipe creation can be found in [6].
- A user can save an edit recipe in an XML format and add it to the library of edit recipes. A user can also load a library of edit recipes defined by other users.
- COOKBOOK performs real time matching of an incoming edit stream against the library of edit recipes. It then presents a ranked list of suitable recipes with a confidence score.
- When a user selects an edit recipe and invokes code completion, COOKBOOK concretizes template edits to the current context and then performs AST-level edits to apply the rest of changes.

We evaluate COOKBOOK using 68 exemplar changed methods drawn from the version history of Eclipse SWT. Each method contains one or more statement changes. Methods containing similar changes are grouped together, leading to 28 method groups. Each group of methods is used both as the exemplar edits for COOKBOOK to learn edit recipes from and the expected edits (the test oracle) when developers are editing the corresponding methods' original versions. In this way, we evaluate COOKBOOK's accuracy in selecting a correct recipe and completing the expected changes. When a user simulates typing each exemplar change, we check whether COOKBOOK ranks the correct recipe as the most suitable recipe and measure the time taken for the recipe to rank top. In our empirical evaluation, COOKBOOK is able to narrow down to the most suitable recipe in 75% of the cases and it spends 120 milliseconds to find the correct recipe on average. We also measure the accuracy of COOKBOOK by comparing the auto-completion result with the expected change. The edits produced by the selected recipe are on average 82% similar to developer's hand edit. This shows COOKBOOK's potential to speed up manual hand editing and minimize developers' errors.

## 2. RELATED WORK

API code completion tools [1, 5, 9] improve existing IDE's code completion features by integrating information mined from version control systems. Robbes et al. [9] rank most recently used API methods higher than other APIs. Bruch et al. [1] rank methods which are always called together with those that are already present in the code under editing. Lee et al. [5] use refactoring history information to suggest correct, updated APIs. These tools do not perform complex edit operations such as statement, if-condition, and while loop insertions or deletions. Graphite [8] associates all possible values or expressions of an object with colors on a pop-up palette menu. Each time when a developer wants to insert a certain value to the code, she can simply click the corresponding color in the menu. Graphite does not handle complex edit operations either. In addition, it does not scale when there are hundreds of values, because developers need to memorize all bindings between values and colors.

```
UPDATE:  if(this.v$_0_!=null){
TO:      if(this.v$_1_ >= 0)
INSERT:    int[] range=null;
INSERT:    for(int i = 0; range == null && {
INSERT:      if(this.v$_2_[i]==node){
INSERT:        range=this.v$_3_[i];
DELETE:    range=(int[])this.v$_0_.get(node);
```

**Figure 1: Comment Mapper Recipe**

BeneFactor [4] and WitchDoctor [3] model common editing patterns for certain refactoring tasks. By matching a developer's edits at runtime with the predefined refactoring edit patterns, the tools infer the refactoring task in the developer's mind and suggest automated refactoring completion accordingly. The tools are limited to pre-defined refactoring operations and cannot handle complex, semantic-changing edit operation recipes handled in COOKBOOK. On the other hand, COOKBOOK allows users to define edit recipes easily using code change examples.

GraPacc [7] is the most relevant work to COOKBOOK. It extracts API usage patterns together with relevant contexts based on data/control dependencies and models them as graphs. When a user queries the tool for code completion suggestions, GraPacc extracts context-sensitive features from the code under editing, ranks patterns that best match the features, and fills in missing code. However, the tool requires developers to manually input API usage patterns to extend its code completion capability. In contrast, COOKBOOK only needs developers to specify more than one example change to define a new edit recipe. In COOKBOOK, such examples can be either demonstrated by developers or specified from version histories.

## 3. MOTIVATING EXAMPLE

Suppose Jon wants to refactor comment processing logic code by updating an if-condition check and an element retrieval from a collection. The refactoring is complex and tedious because it involves similar yet not identical changes to multiple methods. Therefore, Jon refactors two of the methods on his own and uses them as exemplar edits for COOKBOOK to generate a `CommentMapper` recipe. Jon saves the recipe for later use, as shown in Figure 1. Meanwhile, Alice creates and saves another recipe when she refactors a widget event handler (see Figure 2). Both recipes are saved to the developers' shared recipe library in COOKBOOK.

Later Jon starts to perform a similar change to the `getExtendedStartPosition` method by updating `"!= null"` to `">= 0"` (see line 155 in Figure 3). COOKBOOK runs in the background to identify a suitable recipe for Jon's edit stream. COOKBOOK ranks the `CommentMapper` recipe higher than the `EventHandler` recipe, as Jon's edit matches the UPDATE operation in `CommentMapper` and the line 156 also matches the recipe's context. Since Alice's recipe does not match the change in line 155, it is not suggested to Jon. When Jon invokes code completion using `CommentMapper`, COOKBOOK performs all entailed operations, but excluding the one already done by Jon to complete the rest of the change. After applying the recipe, Jon manually fills in some generalized identifiers (e.g. `v$_2`) that COOKBOOK could not fill in. The reason why COOKBOOK cannot fill in `v$_2` is that the identifier is newly introduced by the recipe and COOKBOOK gets no clue about how to concretize it.

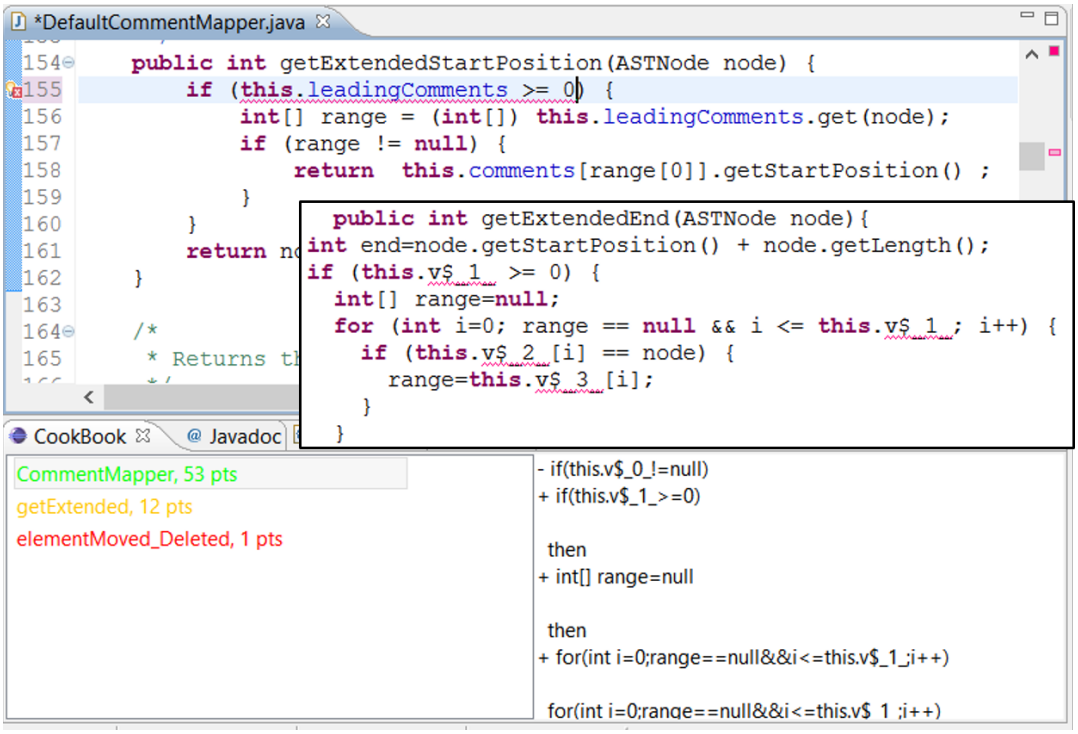


**Figure 3: Eclipse plug-in for CookBook. The method before and after recipe completion are shown.**

```
UPDATE:  if(focusIndex==1){
TO:      if(focusItem==null){
DELETE:    expandItem item=items[focusIndex];
           event event = new Event();
UPDATE:    event.item=item;
TO:      event.item=focusItem;
```

**Figure 2: Event Handler Recipe**

## 4. APPROACH

With CookBook, a user can create a new recipe by specifying exemplar edits and save the recipe for later use. The user can also run CookBook in the background of Eclipse while editing code so that CookBook matches the incoming edit stream against all saved recipes to identify and rank recipes suitable to the current editing context. When a user selects a recipe and invokes code completion, CookBook fills out the rest of edit operations in the selected recipe.

**Recipe Creation.** When a user provides multiple exemplar changed methods, CookBook uses an AST differencing algorithm to create an edit script for each method, extracts the common edits, and only abstracts identifiers which are used differently in different methods in order to create the most specific generalization of all inferred edit scripts. The recipe is then saved to an XML file and added to the recipe library of CookBook.

**Recipe Matching with Edit Stream.** When a user activates CookBook's matching feature and starts editing code, CookBook first searches for suitable recipes solely based on context matching. If the edited code contains AST matching nodes for all context nodes in a recipe based on their content's string similarity, the recipe is considered as a candidate. As the user makes edits, CookBook performs a line-level diff and interprets the differences as potential AST node insertions, deletions, updates, or moves, as shown in Table 1, to make sure that it does not incorrectly guess the user's intent. By comparing these inferred operations with the edit operations of each suitable recipe, CookBook calculates a matching score for each recipe and ranks them accordingly.

**Recipe Ranking.** A recipe's rank depends on its matching score for the method under editing. A higher score means a higher rank. Scores are affected by both context matching and edit matching. Intuitively, a successful context matching initializes the matching score to 50 points. Each time a user edits a line, CookBook infers the edit operation and matches it to every edit operation in each candidate recipe based on edit types and content. When edit types match, an exact match for content between the inferred edit operation and an edit operation of a certain recipe increases the recipe's matching score by 50 points, while a partial match gives points based on the matching percentage.

**Recipe Application** When a user selects a suggested recipe to complete editing, CookBook applies all edit operations in the recipe to the method's original version, if CookBook's matching feature was activated. CookBook manipulates the method's AST according to the edit operations in the recipe and generates source code from the new AST.

**Table 1: AST edit operation types corresponding with a user's line-level edits**

| User's edit | Potential AST edit type |
|---|---|
| Insert line | INSERT, MOVE |
| Delete line | DELETE, MOVE |
| Change line | UPDATE, INSERT, DELETE, MOVE |

# 5. EVALUATION

To evaluate COOKBOOK, we use 68 exemplar changed methods drawn from the version history of Eclipse SWT. This data set is used in the evaluation of LASE for automating similar changes [6]. Methods containing similar changes are grouped together, leading to 28 groups. In each group, methods have at least 40% similar content according to ChangeDistiller [2] and experience at least one similar AST edit operation. However, these methods do not necessarily experience the same number of identical AST edits. Therefore, this experiment demonstrates a realistic scenario of applying similar but different edits to different methods. Each group of methods is used to generate an edit recipe, resulting 28 recipes in COOKBOOK's library. Evaluation results are shown in Table 2. Column 1 lists recipe names. Column 2 shows the number of examples in each group. The other columns show our evaluation for COOKBOOK in terms of response time, effectiveness in saving developers' manual effort (convergence), and quality of code completion (accuracy). These tests aim to provide a baseline performance measurement, and are not exhaustive.

**Response Time.** In our experiment, the first author simulated by hand the original refactoring operations from the version history on each of the 62 test methods. We timed how fast COOKBOOK could find applicable recipes and how fast the user could complete the refactorings with COOKBOOK. Columns 3 and 4 show the average matching times for each group of test runs. The longest delay, context matching, took under 200 ms in 85 percent of cases, and under 500 ms in 95 percent of cases. Column 5 shows the group average total runtime for the user to completely refactor a method, with a total average of 6 seconds.

**Convergence.** We measure *convergence* in terms of how many keystrokes and syntactic edit operations a user has to make before COOKBOOK has significant confidence in its top ranked recipe, meaning that the top one has at least ten more points of matching score than its followers. In 60% of the cases, COOKBOOK converges within 10 keystrokes (and within a single AST edit operation).

**Accuracy.** We measure accuracy by comparing the auto-completed version by COOKBOOK with the expected version using an AST differencing algorithm [2] to compute similarity between them. On average, COOKBOOK generates code 82% similar to the expected one.

**Table 2: CookBook Evaluation Results**

| Recipe | $\sum$ | Matching (ms) Ctxt | Matching (ms) Edit | Task (sec) | Convergence Key. | Convergence Synt. | Accu. (%) |
|---|---|---|---|---|---|---|---|
| Average Per Changed Method in Each Group | | | | | | | |
| win32 | 2 | 160 | 2.7 | 3.8 | 1 | 0 | 72 |
| selectListener | 2 | 8 | 0 | 2.6 | 1 | 0 | 75 |
| compListener | 2 | 14 | 1.1 | 8.6 | 23 | 1 | 83 |
| mergeSrc | 2 | 55 | 0 | 3.3 | 1 | 0 | 100 |
| CompareRun | 2 | 17 | 1.5 | 8.7 | 21 | 1 | 100 |
| getColor | 2 | 230 | 6.9 | 8.3 | 1 | 0 | 100 |
| srcLocator | 2 | 177 | 11 | 4 | 1 | 0 | 80 |
| migration | 2 | 52 | 0.1 | 6.9 | 21 | 0.5 | 90 |
| setList | 2 | 78 | 0.5 | 4.5 | 1 | 0 | 85 |
| elmtMoved | 2 | 23 | 10.5 | 4.2 | 1 | 0 | 80 |
| getNewRange | 2 | 41 | 2.5 | 4.7 | N/A | N/A | 100 |
| fixFocus | 3 | 119 | 6.7 | 2.8 | 1 | 0 | 100 |
| foreground | 6 | 123 | 1.4 | 11.5 | 35 | 1 | 100 |
| verifyText | 2 | 85 | 1 | 2.5 | 1 | 0 | 100 |
| diffViewer | 2 | 19 | 1.5 | 5.7 | 1 | 0 | 100 |
| strokeColor | 2 | 144 | 4.5 | 4.7 | 1 | 0 | 100 |
| paintSides | 2 | 144 | 5.5 | 3.4 | 1 | 0 | 100 |
| addNewRange | 2 | 48 | 3 | 5.5 | N/A | N/A | 100 |
| saveAdapter | 2 | 41 | 0.8 | 3.7 | 2 | 0 | 58 |
| paintCenter | 2 | 828 | 12 | 4.2 | 1 | 0 | 100 |
| compareIpt | 2 | 13 | 1 | 3.9 | 1 | 0 | 50 |
| opCompare | 2 | 82 | 1 | 4.6 | 1 | 0 | 74 |
| setDIrty | 2 | 14 | 0.9 | 4.3 | 3 | 0 | 14 |
| getExtended | 2 | 32 | 0.6 | 7.4 | N/A | N/A | 42 |
| commentMap | 2 | 133 | 2.8 | 5.7 | N/A | N/A | 73 |
| ExpandBar18 | 2 | 455 | 2.1 | 12.6 | N/A | N/A | 58 |
| ExpandBar14 | 2 | 126 | 1.8 | 11.7 | N/A | N/A | 54 |
| ExpandBar10 | 2 | 61 | 1 | 5.9 | N/A | N/A | 54 |
| Per Changed Method of All Groups | | | | | | | |
| Average | | 120 | 3.0 | 6.0 | 8.2 | 0.2 | 82.0 |
| Min | | 0 | 0 | 1.6 | 1 | 0 | 14 |
| Max | | 1144 | 21 | 15 | 43 | 1 | 100 |
| Median | | 54 | 1.1 | 5.2 | 1 | 0 | 92 |

# 6. SUMMARY

While existing code completion in popular IDEs leverages pre-defined templates to complete changes, COOKBOOK lets developers define custom, reusable templates of complex edit operations by specifying change examples. It allows users to define new recipes and store them in an XML file for easy sharing of the library. COOKBOOK matches a developer's edit stream to the individual recipes' context and edit operations and ranks suitable recipes in the background. Our evaluation shows that the overhead of recipe matching in real time is imperceptible, and COOKBOOK is able to narrow down to a single most suitable recipe within 8.2 keystrokes on average. COOKBOOK is highly accurate, producing results 82% similar to the expected edits in the evaluation data set. In the future, we would like to perform a comprehensive user study and improve the usability of our current UI.

# 7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, CCF-1018271, CCF-0811524, CNS-1239498, and a Google Faculty Award.

# 8. REFERENCES

[1] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC/FSE '09*, 2009.

[2] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *TSE '07*, 2007.

[3] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *ICSE '12*, 2012.

[4] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *ICSE '12*, 2012.

[5] Y. Y. Lee, S. Harwell, S. Khurshid, and D. Marinov. Temporal code completion and navigation. In *ICSE '13*, 2013.

[6] N. Meng, M. Kim, and K. McKinley. Lase: Locating and applying systematic edits. In *ICSE '13*, 2013.

[7] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE '12*, 2012.

[8] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *ICSE '12*, 2012.

[9] R. Robbes and M. Lanza. How program history can improve code completion. In *ASE '08*, 2008.