Weekly Meeting 10/6/2015

Revise Problem and Related Work: find examples from informal documentations

1. PROBLEM STATEMENT

Developers frequently use source code examples as the basis for interacting with an application programming interface (API) to obtain the needed functionality [28]. In this case, the source code example serves as the explicit origin for a reuse task which transforms a sequence of APIs with corresponding control structure to the target context [11]. During the process of reuse task such as 'add undo and redo actions in my Graphics Editor', developers often query for a set of examples considering the task, and integrate examples that match the task into their system [25].

Both location and integration of reusable examples are not trivial. For the phase of location, existing tools that search for specific API usage examples [3, 4, 24, 33] might not be able to support such reuse tasks that always involve in multiple classes and related methods. As a result, developers always query general search engines like Google or expert sites like StackOverflow to investigate potential tutorials for desired features. These search engines will return a list of informal documentations and users have to search for source code elements in these resources. Although some linking recovery tools [9, 27] can help identify a partial program in the form of code snippet, they are not sufficient to create a pragmatic reuse plan as the main functionality is always interleaving with auxiliary ones and it is unlikely to find a 'perfect' example that provides all desired features for the reuse task [12]. Considering that a single example is not enough for a given task, developers need to investigate multiple example variants manually for the desired features (i.e., common API calls). For the phase of integration, developers should not only measure structural and semantic similarity between the example and target context for the main API calls that provide desired features [7], but also determine which unnecessary code should be eliminated to save maintenance cost [14, 15] and which related elements should be transformed with structural correspondence [6] to the target context. Unfortunately, for any selected example in non-trivial system, the number of structural dependencies to follow is much too large to be completely covered by developers. The partial programs [8] extracted from the location phase make it even harder to tease out irrelevant elements without knowing a full scenario of precise type information and method implementation details. Moreover, the location and integration phase can be iterative as there is no guarantee that a selected example will be appropriate until the integration results are examined [12].

To overcome these challenges, we propose to automatically identify common API calls that provide the main feature across different partial program examples from informal documentation, and integrate this reusable example to the target context with a set of related elements that are necessary to implement the main feature. We assume that the overlapping API calls are the main features that the user intends to reuse based on the user query, and the related elements should be transformed to the target together with main features if they are data dependent and control dependent on the common functionalities in a common way across different examples. Figure 1 illustrates the process of our system. Given a free-form query from users, our tool invokes search engine (e.g., Google) to obtain a list of partial programs by recovering links between code elements and informal resources, clusters these examples and extracts common structural facts (e.g., jQuery logical facts) with corresponding control structures for each cluster, identifies a set of related elements that interact with main elements in a common way, constructs reuse task with both main elements and related elements and ranks them based on user's context, and finally integrates the reuse task selected by users to the target context.

Our approach helps identify and integrate reusable examples in three phases:

- **Example Collection Phase.** We identify the main features from multiple partial programs derived from informal resources. Without confined to an established code corpus, we leverage web search engine to obtain a list of code examples in the form of partial programs [9, 27]. We use partial program analysis [8] to infer partial type and resolve syntactic ambiguity.

- **Example Clustering Phase.** We take three steps in this phase. Based on the type facts extracted from partial program analysis, we first cluster similar code examples based on the number of shared class-level facts to reduce the duplicated code snippets. Next, for each example in each cluster, we identify main facts that are shared among all examples, and perform slicing to identify related elements. We ignore the specific

related elements that appear in a single example. Finally, we group the related elements based on their common interaction with main elements.

- **Example Integration Phase.** Our approach investigates the user's contexts to recommend the best-fit reuse task considering the number of matched structural facts in target context. After user's selection, our approach integrates the reuse task to the target context by transforming the unmatched structural facts based on matched facts in the context.

2. MOTIVATING EXAMPLE

Consider that user wants to add undo and redo actions for their Java Swing text editor application with undo/redo buttons. Without knowing any APIs, developer first queries our tool with a free-form query `'java undo redo example'`. Our tool investigates top 10 web pages returned from Google, because empirical evidence indicates that developers rarely look beyond this limit when searching [31]. Our tool uses 'expert' heuristic to filter and rank web pages: it favors pages that have complete code snippet and text description because these pages are more likely to be written by experts. Thus it removes 3 pages without explicit code snippets, and performs the initial parsing for the informal documentations to identify code-like terms.

Example collection phase. With 7 partial programs at hand, it first extracts class-level structural facts for each code example. Based on a relative match threshold to measure the relative occurrence of the number of matched facts out of all class-level facts, our tool clusters these examples in three groups using complete linkage technique: 5 examples use classes in Java Swing like 'AbstractAction' and 'UndoableEditListener', another example uses a 'Node' data structure to illustrate how to implement undo/redo actions using command design pattern, and the last example is the javadoc for 'UndoManager'. Our tool starts from the first cluster as it has the highest relative occurrence out of all examples. Based on partial program analysis, we generate a list of type facts to infer types and bind methods.

The common facts our tool extracts include:

Table 1: Common Structural Facts Extracted from the Examples

Level	Structural Facts
class	subType(Undo*Action, AbstractAction), subType(Redo*Action, AbstractAction), subType(*Listener, UndoListener)
method	override(actionPerformed(), Undo*Action, AbstractAction), override(actionPerformed(), Redo*Action, AbstractAction),
statement	init(*,javax.swing.undo.UndoManager), invoke(UndoManager.undo, actionPerformed(), Undo*Action), invoke(UndoManager.redo, actionPerformed(), Redo*Action)

We simplify some full names due to space limitation:
AbstractAction: javax.swing.AbstractAction;
UndoListener: javax.swing.event.UndoableEditListener
actionPerformed():actionPerformed(ActionEvent)

Example clustering phase. After extracting facts for each example, our tool identifies main facts shared in all 5 examples shown in Figure 2 part A. It leverages slicing to group related facts. In this example, 3 of them add `Undo*Action` to `JMenuItem` by invoking `JMenuItem.add(ActionEvent)` and support undo and redo actions for Text Editor shown in Figure 2 part B. The other two of them invoke `JButton.addActionListener(ActionEvent)` to add these two actions to `JButton`, and support `JApplet` Shape drawing shown in Figure 2 part C. After extracting both main facts and related facts, we generate two reuse tasks by combining main facts with each related fact group.

(A) Common facts

```
import javax.swing.undo.UndoManager;
import javax.swing.event.UndoableEditListener;
import javax.swing.AbstractAction;

public class Foo {
  public void run() {
   UndoManager undoManager = new UndoManager();
  }}
  class UndoListener implements UndoableEditListener {
  public void undoableEditHappened(UndoableEditEvent e) {
    undoManager.addEdit(e.getEdit());
  }}
class UndoAction extends AbstractAction {
   public void actionPerformed(ActionEvent e) {
     undoManager.undo();
  }}
class RedoAction extends AbstractAction {
  public void actionPerformed(ActionEvent e) {
     undoManager.redo();
  }}
```

(B) Auxiliary facts: Menu

```
import com.sun.java.swing.*;

public class Foo {
  public void run() {
   UndoManager undoManager = new UndoManager();
   new TextComposite().getDocument().addUndoableEditListener(new UndoListener())
   new JMenuItem(new UndoAction());
   new JMenuItem(new RedoAction());
  }}
```

(C) Auxiliary facts: Button

```
import javax.swing.undo.UndoableEditSupport;
import com.sun.java.swing.Button;

public class Foo {
  public void run() {
   UndoManager undoManager = new UndoManager();
   new Button.addActionListener(new UndoAction());
   new Button.addActionListener(new RedoAction());
   new UndoableEditSupport().addUndoableEditListener(new UndoListener());
  }}
```

Figure 2: Common features and auxiliary features extracted from informal resources

Example integration phase. Given a context like Figure 3 part D, our tool identifies the best-fit reuse task based on the number of matched facts in the user's context. If our tool fails to identify any matched facts in the user's context, it will simply rank the reuse task based on its occurrence in the code examples. Our tool asks for user's selection by presenting the generated main facts and related facts to the user in the form of code snippet. After confirmed by users, our tool automatically fills the rest facts that do not exist in the context shown in Figure 3 part E.

3. RELATED WORK

Extract partial program from informal documentation. Lightweight regular expressions and information retrieval techniques have been widely used to resolve the links between source code elements and documentations. Bacchelli et al. [2] use regular expressions to identify code ele-

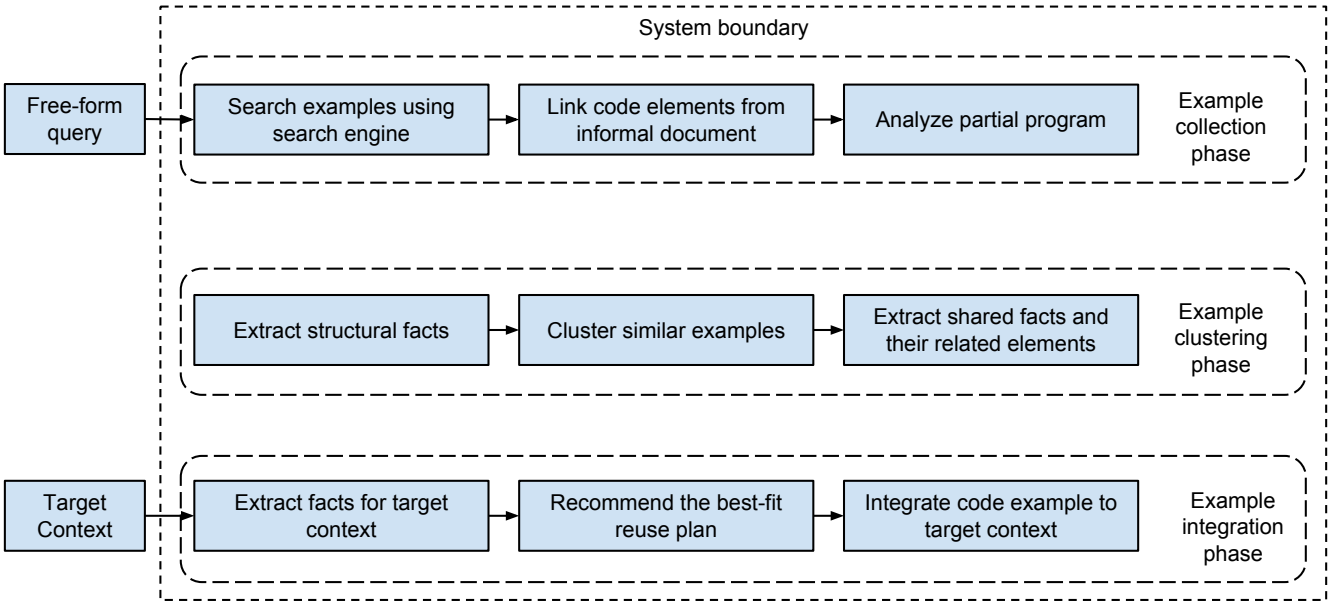

Figure 1: System process overview

(D) User's context

```
import com.sun.java.swing.*;

public class MyTextEditor {
  public void init() {
    JFrame frame = new JFrame("Undo Sample");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JTextArea textArea = new JTextArea();
   JButton undoBtn_;
   //add undo and redo action to text editor
  }}
```

(E) Suggested reuse plan based on user's context

```
import com.sun.java.swing.*;
import javax.swing.undo.UndoManager;
import javax.swing.event.UndoableEditListener;
import javax.swing.AbstractAction;

public class MyTextEditor {
    JFrame frame = new JFrame("Undo Sample");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JTextArea textArea = new JTextArea();
   JButton undoBtn_;
   JButton redoButton; //added
  public void init() {
   //add undo and redo action to text editor
   textArea.getDocument().addUndoableEditListener(new UndoListener());
   undoBtn_ = new Button.addActionListener(new UndoAction());
   redoButton  = new Button.addActionListener(new RedoAction());
  }}
```

Figure 3: User's context and best-fit reuse plan based on the context

Table 2: Comparison of representative code search tools, code reuse tools, and code transformation tools

Tool	I	O_m	Related	Cluster	Partial	Transfer
RecoDoc [9]	D				✓	
ACE [27]	D				✓	
MAPO [33]	M			✓	✓	
Buse et al. [4]	M			✓	✓	
SNIFF [5]	Q			✓	✓	
MUSE [23]	M		✓	✓		
Keivanloo [16]	Q		✓	✓		
PRIME [22]	Q		✓		✓	
Gilligan [15]	E	✓	✓			
Jigsaw [7]	E		✓			✓
LASE [21]	E		✓	✓		✓
Ours	Q	✓	✓	✓	✓	✓

The **I** column specifies the input format (Informal **D**ocument, **M**ethod, free-form **Q**uery, **E**xample); The O_m column specifies if the output is across the method boundary; The **Related** column specifies if the tool considers related methods when recommending example or performing transformation; The **Cluster** column specifies if the output results are clustered to reduce redundancy; The **Partial** column specifies if the tool supports partial program as input. The **Transfer** column specifies if the tool is able to transfer the code to another context.

ments in email discussions based on programming naming conventions. Antoniol et al. [1] apply a probabilistic Vector Space Model to resolve terms while Marcus et al. [19] use Latent Semantic Indexing to recover links. However, both regular expression and information retrieval techniques suffer from a low precision and recall around 0.5. RecoDoc [9] leverages term context to extract code-like terms and use partial program analysis [8] and a set of structural/lexical heuristic filters to resolve ambiguous terms. ACE [27] extends RecoDoc with island parser to identify code-like terms and reparses each document to resolve ambiguous terms using term's context. We leverage ACE to extract partial program from informal documentations and use partial program analysis [8] to infer type declaration and resolve method binding ambiguity. We advance these tools in the sense that we not only identify partial program as code example, but also help cluster similar examples and integrate reusable examples to the target context.

Cluster code examples. Our example clustering approach is similar to some prior works that extract representative examples for specific APIs or user query. MAPO [33] leverages

frequent call sequences to cluster the usage of specific APIs and rank abstract usage patterns based on the context similarity. Buse et al [4] propose to generate abstract API usages by synthesizing code examples using symbolic execution for a particular API. Different from these works that generate abstract usage patterns for specific API or data type, SNIFF [5] performs type-based intersection of code chunks based on the keywords in the free-form query and cluster the common part of the code chunks for concrete code examples. However, these works only focus on providing code examples based on the popularity or textural relevance while developers have to manually resolve structural dependencies before reusing the examples. MUSE [23] addresses this limitation using slicing to generate concrete usage examples and selects the most representative ones based on the popularity and readability while Keivanloo et al [16] uses clone detection to cluster examples involving loops and conditions. We make it one step further to support partial code example clustering and identify structural correspondence to identify both common features and alternative features. PRIME [22] supports code search over partial programs based on type state transition, yet it requires users to provide partial temporal specification for generalized *typestate*. There exists a number of code example suggestion tools that recommend call chains [18, 29, 32] or related functions [20, 24] based on queries [3] or contexts [13, 26]. But these tools can only recommend code examples in the method level which make them insufficient for reuse tasks across multiple classes.

Identify structural correspondence for code reuse Although some approaches advocate refactoring code rather than reuse code [10], recent researches have found that these kind of 'clone' cannot be easily refactored [17] and have to be modified to meeting requirements in new context [30]. Jigsaw [7] supports small-scale integration of source code into target system between the example and target context. Based on its ancestor [6] that identifies structural correspondence based on AST similarity, it greedily matches each element between two contexts, transforms correspondent elements to the target context, and simply copies the source element to the target if it does not correspond with any element in the target. Unfortunately, developer has to provide source and target to enable a one-to-one transformation and resolve all dependencies when pasting code to the target. Our approach overcomes these two limitations: we extract common functionalities from multiple examples and identifies how related elements interact with main features in a common way to resolve dependencies based on the mapping from the source to the target. Our idea of leveraging multiple examples to discover commonality and eliminate specificity is similar to LASE [21], which applies similar but not identical changes to multiple code locations based on context similarity. Our approach works in a similar manner of Programing-by-Example, but focuses on a task-based code reuse across different methods or even different classes, while LASE is confined to the systematic edit within a single method and requires users to specify all input examples. Other related works on code reuse include Gilligan [15] and Procrustes [14] which try to address the problem of source code integration in the context of medium or large-scale reuse tasks. They automatically suggest program elements that are easy to reuse based on structural relevance and cost of reuse in the source context, and guide users to investigate and plan a non-trivial reuse task. They assume that developers have a perfect example at hand, and they can finish the reuse task by resolving all dependency conflicts and integrating the example to the desired context. However, we note that it is not easy to identify a good example as an example is always interleaving with other auxiliary features that should not be integrated. We observe that it is equally difficult, if not more so, to distinguish the major functionality and auxiliary ones from multiple examples than to identify related elements in a pragmatic reuse plan. We target the problem to identify the major features across different reusable examples and leverage Procrustes to evaluate the cost of reuse when recommending the best-fit reusable plan.

4. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.

[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 375–384, 2010.

[3] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259, 2014.

[4] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 782–792, 2012.

[5] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 385–400, 2009.

[6] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 165–174, 2007.

[7] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 214–225, 2008.

[8] B. Dagenais and L. J. Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 313–328, 2008.

[9] B. Dagenais and M. P. Robillard. Recovering

traceability links between an API and its learning resources. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 47–57, 2012.

[10] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

[11] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Software Eng.*, 31(7):529–536, 2005.

[12] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 555–558, 2009.

[13] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 117–125, 2005.

[14] R. Holmes, T. Ratchford, M. P. Robillard, and R. J. Walker. Automatically recommending triage decisions for pragmatic reuse tasks. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 397–408, 2009.

[15] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 447–457, 2007.

[16] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 664–675, 2014.

[17] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 187–196, 2005.

[18] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 48–61, 2005.

[19] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 125–137, 2003.

[20] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 111–120, 2011.

[21] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 502–511, 2013.

[22] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 997–1016, 2012.

[23] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus. How can I use this method? In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 880–890, 2015.

[24] E. Moritz, M. L. Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing API usages in large source code repositories. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 646–651, 2013.

[25] J. Parsons and C. Saunders. Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse. *IEEE Trans. Software Eng.*, 30(12):873–888, 2004.

[26] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the IDE into a self-confident programming prompter. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 102–111, 2014.

[27] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 832–841, 2013.

[28] C. Sadowski, K. T. Stolee, and S. G. Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 191–201, 2015.

[29] N. Sahavechaphan and K. T. Claypool. Xsnippet: mining for sample code. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 413–430, 2006.

[30] R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Trans. Software Eng.*, 31(6):495–510, 2005.

[31] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 157–166, 2009.

[32] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE*

2007), November 5-9, 2007, Atlanta, Georgia, USA, pages 204–213, 2007.

[33] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: mining and recommending API usage patterns. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 318–343, 2009.