SCIENCE CHINA
Information Sciences

Reflective feature location: knowledge in mind meets information in system

Xin PENG[1,2]*, ZhengChang XING[3], Sen PAN[1,2], WenYi QIAN[1,2],
Václav RAJLICH[4] & WenYun ZHAO[1,2]

[1]*School of Computer Science, Fudan University, China;*
[2]*Shanghai Key Laboratory of Data Science, Fudan University, China;*
[3]*Research School of Computer Science, Australian National University, Australia;*
[4]*Department of Computer Science, Wayne State University, USA*

Abstract Locating code entities relevant to a feature (i.e., feature location) is an important task in software maintenance. Feature location is challenging due to the information gap between the developer's knowledge about a feature and the feature's implementation in the system.In this paper, we present a reflective feature location approach *ReFLex* to bridge this gap. *ReFLex* automatically computes a reflexion model between the developer's logical view of a feature and the feature's implemented view reverse-engineered from the code.It provides interactive reflection support for the developer to refine the logical view and the feature location results such that the logical view and the implemented view converge gradually in an iterative feature location process. We have implemented our approach as an Eclipse plugin and investigated the benefits and challenges in reflective feature location through a user study.

Keywords feature location, information seeking, reflexion model

Citation Peng X, Xing Z C, Pan S, et al. Reflective feature location: knowledge in mind meets information in system. Sci China Inf Sci, 2016, 59(1): xxxxxx, doi: xxxxxxxxxxxxxx

1 Introduction

A software feature is defined by requirements and accessible to users [9]. Software maintenance tasks are usually specified by features, such as adding new features, extending existing features, and fixing bugs in features. To accomplish a software maintenance task, developers must identify code entities relevant to the feature. This activity is termed as *feature* or *concept location* [9, 38–40].

Among many techniques proposed to support feature location, Information Retrieval (IR) based code search has been widely used in practice [9] . With IR-based code search, developers describe a feature using a set of keywords (i.e., a feature query). The keywords can be from feature descriptions (e.g., change request, bug report) or domain knowledge [25]. Code search engines use IR techniques to compute the relevance score between the feature query and the code entities. Relevant code entities are ranked in a list for inspection.

* Corresponding author (email: pengxin@fudan.edu.cn)

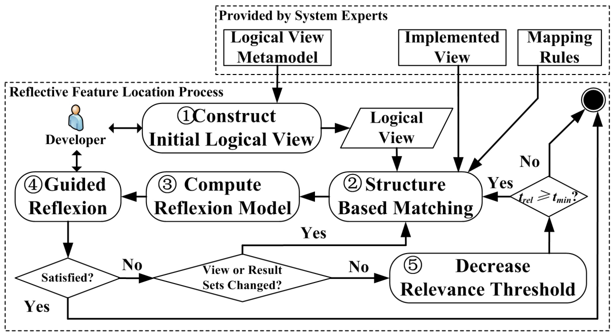

Provided by System Experts
Logical View Metamodel
Implemented View
Mapping Rules
Reflective Feature Location Process
Developer
①Construct Initial Logical View
Logical View
②Structure Based Matching
③ Compute Reflexion Model
④Guided Reflexion
⑤ Decrease Relevance Threshold
Satisfied?
View or Result Sets Changed?
$t_{rel} \geqslant t_{min}$?
Yes
No

Figure 1 Reflective Feature Location Process

Using IR-based code search the developers are often faced with two difficulties. First, the developer may infer from the feature description that the feature should involve several correlated logical entities. However, it is difficult to express this understanding in a set of keywords. Second, the ranked list of code entities provides no ways to identify and reconcile the discrepancies between the developer's high-level understanding of the feature and the feature implementation, for example, whether the code entities have the dependencies they are or are not supposed to have.

Query reformulation techniques can help developers rewrite the query based on the user-relevance feedback on search results [18, 23] or the linguistic properties of feature query and code entities [32, 33]. Concept lattice [11] or multiple syntactic and semantic facets [23] can be used to facilitate the inspection of query results. Although useful, these techniques lack systematic ways to bridge the gap and reconcile the discrepancies between the developer's high-level understanding of the feature and the feature implementation in the system.

In this paper, we propose a reflective feature location approach (*ReFLex*) to tackle the above two difficulties. *ReFLex* allows developers to express their understanding and knowledge about a feature in a high-level logical view. It uses a structure-based matching algorithm to map the logical view of the feature and the implemented view of the system. Based on the mapping results, *ReFLex* identifies and highlights the discrepancies between the two views in a reflexion model. It provides interactive reflection support for developers to refine the logical view and/or the mapping results. As a result, the logical view and the implemented view of the feature converge gradually. Code entities relevant to the feature emerge from this reflective feature location process.

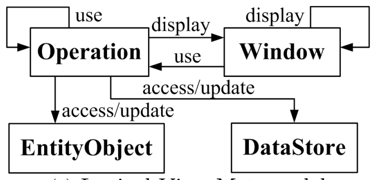
We have implemented our approach as an Eclipse plugin. To evaluate the benefits and challenges in the *ReFLex* process and the tool support, we conducted a user study in which two groups of nine developers performed three feature location tasks using our *ReFLex* tool and the standard Eclipse IDE respectively. Our results show that the *ReFLex* process and tool work better for coarse-grained features involving several logical entities and complex business logic. The *ReFLex* users demonstrated feature location behaviors that were different from those of the Eclipse users. We observed effective strategies for reflective feature location.

2 The Approach

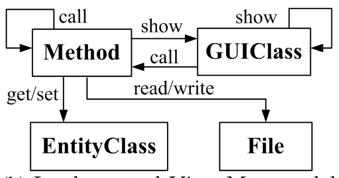
In this section, we present an overview of our *ReFLex* approach and then detail the underlying techniques.

2.1 Overview

Figure 1 presents an overview of our *ReFLex* approach. *ReFLex* assumes that a system expert can provide the metamodel of logical view, the metamodel of the implemented view of the system, and a metamodel mapping of types of entities and relations between the two views. It assumes that the implemented view can be reverse-engineered from the system source code according to the implemented-view metamodel.



(a) Logical-View Metamodel



(b) Implemented-View Metamodel

Figure 2 Examples of Metamodels

The *ReFLex* process consists of 5 steps. Step 1 and Step 4 involve interaction with the developer. To start a feature location process, *ReFLex* asks the developer to construct an initial logical view [28] (i.e., an instance of logical-view metamodel) of the feature (Step 1). This logical view captures a set of correlated logical entities of the feature. The initial logical view of the feature may not agree with the feature implementation. During feature location process, the developer can refine the logical view based on the reflexion model.

Given the logical view of the feature, and the implemented view of the system and the metamodel mapping between the two views, *ReFLex* uses a structure-based matching algorithm to determine the relevance of logical entities and code entities based on their lexical and structural similarities (Step 2). Based on the relevant code entities identified for each logic entity, *ReFLex* computes a reflexion model (Step 3) that highlights the discrepancies between the logical view of the feature and the implemented view of the system. The developer inspects the reflexion model, and refines the logical view of the feature and/or the current mapping results to reduce the discrepancies in the reflexion model (Step 4). In Step 4 the developer can freeze some logical entities. *ReFLex* will not update the relevant-entities set of these frozen logical entities. Freezing logical entities allows the developer to focus on certain aspects of a feature by freezing other aspects. If the developer is satisfied with current feature location results (e.g., when the logical view and the implemented view converge), he can finish the process. The relevant-entities set of all the logical entities constitute the code entities relevant to the feature. Otherwise, *ReFLex* proceeds to the next iteration. If the logical view of the feature and/or the relevant-entities set of some logical entities have been changed in Step 4, *ReFLex* calls the structure-based matching algorithm to remap the logical view of the feature and the implemented view of the system, and update the reflexion model accordingly. Otherwise, *ReFLex* decreases the relevance threshold t_{rel} (Step 5) and then calls the structure-based matching algorithm.

2.2 Metamodels and Metamodel Mapping

ReFLex maps the logical view of a feature and the implemented view of the system and computes a reflexion model [20] between the two views. We define here the metamodel of the logical view, the metamodel of the implemented view, and the mapping between the two metamodels.

A software reflexion model maps a high-level model of interest and a source model from the source code based on a declarative mapping between the two models. The high-level model represents a kind of logical structure of the software system. For example, it can be a design model described in design documents or reflect the developers understanding of the logical structure of the current implementation. The reflexion model schema [20] defines *HLMENTITY* and *SMENTITY*, which define the types of high-level model entities and source model entities respectively, *HLMRELATION* and *SMRELATION*, which define the types of relations over high-level model entities and source model entities respectively, and *HLMTUPLE* and *SMTUPLE*, which define the types of tuples in the *HLMRELATION* and *SMRELATION*. The declarative mapping defines the correspondences between *HLMENTITY* and *SMENTITY*, and between *HLMTUPLE* and *SMTUPLE*. We focus on object-oriented software in this paper. However, our approach is not limited to object-oriented software because reflexion model schema is essentially an entity-relation schema that can be applied to many types of software.

The logical view of a feature reflects the developer's high-level understanding and knowledge about a feature. Depending on the feature location needs, a system expert may define the metamodel of logical

Table 1 Metamodel Mapping

Entity of Logical View	**Entity of Implemented View**
Operation	Method
Window	GUIClass
EntityObject	EntityClass
DataStore	File
Relation of Logical View	**Relation of Implemented View**
Operation *use* Operation	Method *call* Method
Operation *display* Window	Method *show* GUIClass
Operation *access/update* EntityObject	Method *get/set* EntityClass
Operation *access/update* DataStore	Method *read/write* File
Window *use* Operation	GUIClass *call* Method
Window *display* Window	GUIClass *show* GUIClass

view as shown in Figure 2(a). A logical view of the feature may consist of four types of *HLMENTITY*. A *Window* represents a boundary object (e.g., a form or a dialog) that the user interacts with. An *Operation* performs certain application logic. An *EntityObject* represents a piece of data, while a *DataStore* is a subtype of *EntityObject* that represents a piece of persistent data. An entity has two attributes: identifier (id) and description (desc). The description stores the natural language description of the entity.

Different types of entities in the logical view have different types of usage relations (i.e., *HLMTUPLE*). An *Operation* can *use* another *Operation*, *display* a *Window*, and *access/update* an *EntityObject* or a *DataStore*. A *Window* can *use* an *Operation* and *display* another *Window*.

The implemented view of the system describes a source model (such as a call graph or an inheritance hierarchy) from the source code. Depending on the feature location needs and the knowledge of system implementation, the system expert may define the metamodel of implemented view as shown in Figure 2(b). The implemented view of the system may consist of four types of *SMENTITY*. A *GUIClass* represents a GUI widget (e.g., a subclass of *JComponent* of Java Swing). A *Method* represents a method of a class. An *EntityClass* represents a class with only attributes or getter/setter methods. A *File* represents a piece of external data the system uses. An entity has two attributes: *name* and *code*, which are reverse-engineered from the source code.

Different types of entities in the implemented views have different types of usage dependencies (i.e., *SMTUPLE*). A *Method* can *call* another *Method*, *show* a *GUIClass*, and *get/set* an *EntityClass* or *read/write* a *File/Database*. A *GUIClass* can *call* a *Method* and *show* another *GUIClass*.

Given the metamodels of the logical view and the implemented view, the system expert can define the mapping between the two metamodels. Table 1 shows the mapping between the two metamodels in Figure 2(a) and Figure 2(b). According to this mapping rule, *ReFLex* locates user interface (*Window*) and data object (*EntityObject*) implementations at class level (*GUIClass* and *EntitiClass*), while it locates application logic (*Operation*) implementations at method level (*Method*).

2.3 Constructing Initial Logical View (Step 1)

To locate a feature's implementation, the developer constructs an initial logical view of the feature using a graphical modeling tool. For example, the feature "Reloading From Disk" of JEdit (http://www.jedit.org) checks if an opened file was modified by another application when the view of the file is brought to the foreground. Even with limited knowledge of the feature implementation, a developer can identify some key abstractions for this feature by analyzing the feature's use case description or trying out the feature with the software. For example, an initial logical view of the feature "Reloading From Disk" may consist of an operation checking whether any opened buffer was modified, a user interface (UI) object prompting whether to reload a file, and an operation reloading the file.

Figure 3 presents an initial logical view of the JEdit feature "Reloading From Disk" that one of the

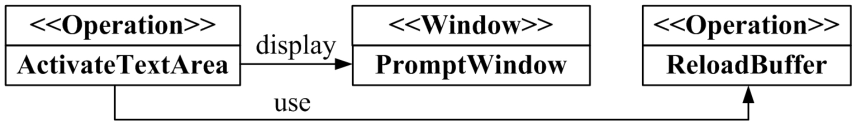


Figure 3 A Logical View of "Reloading From Disk"

developers defined in our user study. Based on the feature description and the use of JEdit software, the developer believed that the implementation of the feature should include an *ActivateTextArea* operation to handle the activation of the text area that is brought to front; *ActivateTextArea* should display a *PromptWindow* for user confirmation, and use *ReloadBuffer* operation to reload the file.

The initial logical view of the feature may not agree with the system implementation, especially in early iterations of feature location. The discrepancies were later identified and fixed through reflective feature location process.

2.4 Structure-Based Matching (Step 2)

ReFLex's structure-based matching algorithm is based on the computation of lexical and structural similarities.

2.4.1 *Similarity Metrics*

Given a logical entity d and a code entity p, *ReFLex* encodes the description of d and the code fragment of p in the TF/IDF (Term Frequency/Inversed Document Frequency) vectors V_d and V_p, based on the vector space model computed from the code entities in the implemented view of the system. It computes the lexical similarity $S_{lex}(d,p)$ as the cosine similarity between the TF/IDF vectors V_d and V_p, i.e., $S_{lex}(d,p) = \frac{\sum_{i=1}^{n} V_d[i] \times V_p[i]}{\sqrt{\sum_{i=1}^{n} V_d[i]^2} \times \sqrt{\sum_{i=1}^{n} V_p[i]^2}}$ $(0 \leqslant S_{lex}(d,p) \leqslant 1)$. We remark that our approach is independent of any specific computation methods of lexical similarity. Other lexical similarity computation methods, such as those based on the words extracted from class and method names as well as from class comments [7], can also be used in our approach.

Given a logical entity d and a code entity p, *ReFLex* computes their structural similarity $S_{stru}(d,p)$ based on the established mappings between the neighboring logical entities of d (denoted as $NGB(d)$) and the neighboring code entities of p (denoted as $NGB(p)$). Let $RS(d)$ be the relevant-entities set of d. We define $CVG(d,p)$ as the convergent neighboring logical-entity set of d, i.e., $CVG(d,p) = \{d' \in NGB(d) | \exists p' \in NGB(p) \& p' \in RS(d') \& RT(d,d') \leftrightarrow RT(p,p')\}$. $RT(d,d') \leftrightarrow RT(p,p')$ indicates the relation between d and d' and the relation between p and p' can be mapped according to the metamodel mapping. The structural similarity $S_{stru}(d,p)$ is defined as the percentage of d's convergent neighboring logical entities in all the d's neighboring logical entities, i.e., $S_{stru}(d,p) = \frac{|CVG(d,p)|}{|NGB(d)|}$ $(0 \leqslant S_{stru}(d,p) \leqslant 1)$.

The overall similarity between a logical entity d and a code entity p is defined as the average of their lexical and structural similarities, i.e., $S(d,p) = (S_{stru}(d,p) + S_{lex}(d,p))/2$. In the first iteration of the *ReFLex* process $S(d,p)$ is set to $S_{lex}(d,p)$ as the relevant-entities sets of all the logical entities are empty.

2.4.2 *Matching Algorithm*

The matching algorithm $identifyRelevant$ (see Algorithm 1) takes four parameters: logical view lv, implemented view iv, candidate-entity threshold t_{cand}, and relevant-entity threshold t_{rel}. During the feature location process, each logical entity maintains a set of relevant code entities ($d.rels$) identified in previous iterations and a set of candidate code entities ($d.cands$) from which $identifyRelevant$ identifies new relevant entities for the logical entity. The implemented view maintains a set of yet-unmapped entities ($iv.unmappeds$) which initially contains all the code entities in the implemented view.

For each logical entity $d \in lv.entities$ whose description ($d.desc$) has been changed in the previous iteration (the description of the newly introduced logical entities is considered as changed), $identifyRelevant$ invalidates the relevant-entities set $d.rels$ and the candidate-entities set $d.cands$ of d (lines 2-5). It adds

Algorithm 1 Identify relevant code entities

```
Require: $identifyRelevant(lv, iv, t_{cand}, t_{rel})$
 1: for each $d \in lv.entities$ & $d.descchanged$ do
 2:     $iv.unmappeds = iv.unmappeds \bigcup d.rels$
 3:     $d.rels = \{\}$; $d.cands = \{\}$
 4: end for
 5: for each $d \in lv.entities$ & $d.descchanged$ do
 6:     for each $p \in iv.unmappeds$ do
 7:         if $S_{lex}(d, p) \geqslant t_{cand}$ then
 8:             $d.cands = d.cands \bigcup \{p\}$
 9:         end if
10:     end for
11: end for
12: for each $(d, p) : d \in lv.entities$ & $p \in d.rels$ do
13:     $(d, p).sim = S(d, p)$
14: end for
15: for each $d \in lv.entities$ & $d.unfronzen$ do
16:     $d.newrels = \{\}$
17:     for each $p \in d.cands$ do
18:         if $S(d, p) \geqslant t_{rel}$ then
19:             $d.newrels = d.newrels \bigcup \{p\}$
20:         end if
21:     end for
22:     $iv.unmappeds = iv.unmapped - d.newrels$
23:     $removeCandidateEntities(lv.entities, d.newrels)$
24: end for
25: for each $d \in lv.entities$ & $d.unfronzen$ do
26:     $d.rels = d.rels \bigcup d.newrels$
27: end for
```

code entities in *d.rels* back to the set of yet-unmapped entities of the implemented view (*iv.unmappeds*), and set *d.rels* to empty set. It also set *d.cands* to empty set.

Then, $identifyRelevant$ refinds candidate entities for the logical entity whose description has been changed (lines 6-12). If the lexical similarity of a logical entity d and a yet-unmapped code entity p (i.e., $S_{lex}(d, p)$) is above the candidate-entity threshold (t_{cand}), $identifyRelevant$ adds the code entity p to *d.cands*. Candidate-entities set of different logical entities may overlap. The union of candidate entities of all the logical entities is a subset of all the yet-unmapped entities in the implement view.

Next, for each existing mapping between a logical entity and a relevant code entity (d, p), $identifyRelevant$ recomputes its overall similarity (lines 13-15). This is because the structural reflexion may increase or decrease the overall similarity of existing mappings. $identifyRelevant$ does not remove the existing relevant code entities of a logical entity, even if the updated overall similarity of these code entities and the logical entity is below t_{rel}. The *ReFLex* tool support can highlight the similarity changes for the developer to make the decision.

Finally, $identifyRelevant$ identifies new relevant entities for each unfrozen logical entity (lines 16-25). If the overall similarity between the d and a candidate code entity $p \in d.cands$ (i.e., $S(d, p)$) is above the relevant-entity threshold (t_{rel}), $identifyRelevant$ adds the code entity p to a temporary relevant-entities set of d (*d.newrels*). If a code entity is identified as relevant to a logical entity, the code entity will be removed from the candidate-entities set of all the logical entities and from the yet-unmapped-entities set of the implemented view. Therefore, $identifyRelevant$ produces a one-to-many mapping between the logical entities and the code entities. After processing all the unfrozen logical entities, *identifyRelevant* adds the newly identified relevant-entities of these logical entities to their relevant-entities set (lines 26-28).

2.5 Computing Reflexion Model (Step 3)

ReFLex computes a reflexion model between the logical view of the feature and the implemented view of the system by comparing the relations between the logical entities defined in the logical view (i.e., expected relations) and the relations between the logical entities induced from their relevant code entities (i.e., induced relations). The reflexion model consists of three kinds of edges [20], i.e., *Convergences*, *Divergences*, and *Absences*. *Convergences* identify the set of relations where the logical view agrees with the implemented view (i.e., expected relations match induced relations). *Divergences* identify the set of induced relations from the relevant code entities that are missing in the logical view (i.e., no corresponding expected relations). *Absences* identify the set of expected relations in the logical view that are missing in the relevant code entities (i.e., no corresponding induced relations).

2.6 Guided Reflection (Step 4)

The reflexion model can be visualized for the developer to refine the logical view and/or the current mapping results.

2.6.1 *Structural Reflexion*

Structural reflexion allows the developer to reduce the structural discrepancies (i.e., *divergences* and *absences* edges) between the logical view and the implemented view. The developer can refine the logical view by adding, removing, merging or splitting logical entities. He can also add, remove, or change relations between logical entities. These refinements of the logical view can remove *Divergences* and *Absences* in the reflexion model.

The developer can select one or more *Divergences* edges and remove the code entities in the relevant-entities set of a logical entity that cause *Divergences* edges. He can also select one or more *Convergences* edges and remove all the code entities in the relevant-entities set of a logical entity that have no contributions to *Convergences* edges.

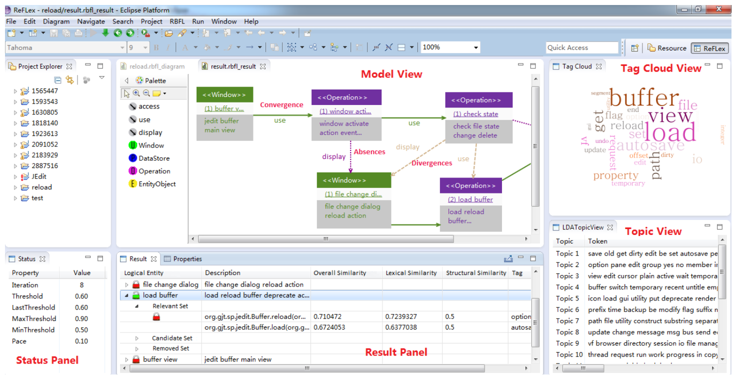
The developer can examine the source code of relevant code entities of a logical entity. If he believes a code entity is irrelevant, he can remove the code entity from the relevant-entities set of the logical entity. This helps remove *Divergences*. The removed code entity will be added back to the candidate-entities set of other logical entities and the yet-unmapped-entities set of the implemented view. The developer can also examine the source code of candidate code entities of a logical entity. If he believes a candidate code entity is relevant, he can add the candidate code entity to the relevant-entities set of the logical entity. This helps remove *Absences*. The added code entity will be removed from the candidate-entities set of other logical entities and the yet-unmapped-entities set of the implemented view.

2.6.2 *Descriptive Reflexion*

Descriptive reflexion allows developers to refine the description of a logical entity by examining terms and topics involved in its current relevant code entities. Tag cloud and topics (mined from code using Latent Dirichlet Allocation technique [13]) allow the developer to identify important terms/topics in the relevant code entities of a logical entity. Investigating the terms/topics of a logical entity and its neighboring logical entities can also give the developer insights into the roles and relations of a set of logical entities. The developer can refine the description of a logical entity using terms/topics from its neighboring entities.

2.7 Decreasing Relevance Threshold (Step 5)

The $identifyRelevant$ algorithm considers a logical entity and a code entity as relevant if their similarity is above the relevant-entity threshold t_{rel}. *ReFLex* uses an incrementally-decreasing-threshold strategy during reflective feature location. This strategy has been proven effective in improving feature location results [8,43]. *ReFLex* allows the developer to set a high initial t_{rel} to ensure the high-quality of mappings between the logical entities and the code entities in the first few iterations. The t_{rel} will be decreased by a *pace* in subsequent iterations to locate more code entities with lower similarities. Once the t_{rel} is lower than the minimal relevance threshold t_{min}, *ReFLex* ends feature location process with the current relevant-entities set of the logical entities as the final results.



Property	Value
Iteration	8
Threshold	0.60
LastThreshold	0.60
MaxThreshold	0.90
MinThreshold	0.50
Pace	0.10

Logical Entity	Description	Overall Similarity	Lexical Similarity	Structural Similarity	Tag
file change dialog	file change dialog reload action				
load buffer	load reload buffer deprecate ac...				
Relevant Set					
	org.gjt.sp.jedit.Buffer.reload(or...	0.710472	0.7239327	0.5	option
	org.gjt.sp.jedit.Buffer.load(org.g...	0.6724053	0.6377038	0.5	autosa
Candidate Set					
Removed Set					
buffer view	jedit buffer main view				

Topic	Token
Topic 1	save old get dirty edit be set autosave pe
Topic 2	option pane edit group yes no member ir
Topic 3	view edit cursor plain active wait tempora
Topic 4	buffer switch temporary recent untitle emp
Topic 5	icon load gui utility put deprecate render
Topic 6	prefix time backup be modify flag suffix n
Topic 7	path file utility construct substring separat
Topic 8	update change message msg bus send e
Topic 9	vf browser directory session io file manag
Topic 10	thread request run work progress in copy

Figure 4 The *ReFLex* Tool

3 Tool Implementation

Figure 4 presents our *ReFLex* tool. We implemented the *ReFLex* tool using Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF). The GUI of the *ReFLex* tool consists of: model view, result panel, tag cloud view, topic view, and status panel.

The model view implements a GEF-based graphical modeling tool for the developer to construct the initial logical view and refine the reflexion model. *Convergences*, *Divergences*, and *Absences* edges in the reflexion model are shown as solid, dashed, and dotted lines with different colors. In the model view, each logical entity is represented as a node with three compartments showing its type, identifier and description. The number before the identifier indicates the size of relevant-entities set. When a logical entity is selected, a tag cloud is generated using WordCram[1)] for all its current relevant code entities. A set of topics are also mined using JGibbsLDA[2)].

The result panel shows in a tree list view of relevant-entities set, candidate-entities set, and removed-entities set of all the logical entities. The developer can expand a logical entity and inspect its result sets. The view displays similarity metrics of the relevant and candidate code entities of the logical entity. The developer can double-click a code entity to open its source code. The developer can select a logical entity and inspect the sub-reflexion-model of the selected entity and its neighboring entities in the pop-up "Filter by Interaction" window.In this window, he can remove irrelevant code entities of the selected logical entity by *Convergences* or *Divergences* edges.

During the feature location process, a set of status parameters are shown in the status panel, including the number of iterations and various threshold values.

4 Evaluation

We conducted a user study to investigate the benefits and challenges inherent in reflective feature location process, and gather user impressions and observations on our tool prototype. We investigated the following questions:

Q1 What types of features are supported well by *ReFLex*, and which types are not?

Q2 How does the *ReFLex* tool change the developers' behaviors during feature location tasks?

Q3 What are effective strategies for reflective feature location?

4.1 Experiment Design

We first describe the design of our user study.

1) WordCram Website: http://wordcram.org/
2) JGibbsLDA Website: http://jgibblda.sourceforge.net/

4.1.1 *Participants*

Our study used matched-subjects design. We recruited 18 participants (16 master students and two PhD students) from School of Computer Science, Fudan University. These participants were matched in pairs based on their programming capability and experience and then randomly allocated to experimental group or control group. The experimental group G_1 used the *ReFLex* tool to perform feature location tasks. The control group G_2 used standard Eclipse IDE to perform the same set of tasks. We chose standard Eclipse IDE as baseline tool to compare with the *ReFLex* tool because we are interested in behavioral changes enabled by *ReFLex* during feature location tasks, compared with "normal" behavior in commonly used IDEs.

4.1.2 *Tasks*

In this study, we used feature location tasks from the JEdit benchmarks [9]. The benchmark consists of 34 features location tasks for the JEdit text editor and the ground-truth code entities of these features. These 34 features were extracted from issue tracking system of the JEdit. The ground-truth was established by analyzing the commits fixing the relevant issues. We chose JEdit as the subject system because all the participants have general knowledge about text editing features, even though they do not know the implementation details of JEdit. We chose three feature-enhancement tasks from the benchmark:

Task 1: HyperSearch reports the occurrences of the string: The result of HyperSearch will be shown in a window, including the number of occurrences of the string;

Task 2: Save the editmode for recent files: In the Buffer Options window, the edit mode of current opened file could be set and saved;

Task 3: Provide a way of reloading changed buffers without prompting: Reloading settings could be changed under the General tag in the Global Options window. And when the editor pane is activated again, the opened file will be reloaded with or without prompting window according to reloading settings.

These three tasks have different characteristics. Task 1 involves only internal application logic. The ground-truth of Task 1 includes 4 classes and 8 methods. Task 2 involves both application logic and data object. The ground-truth of Task 2 includes 6 classes and 9 methods. Task 3 involves user interface, application logic and data object. The ground-truth of Task 3 includes 9 classes and 18 methods.

The description of these tasks was written in a way that a developer with limited knowledge of JEdit could understand. The relevant-entities of the features are medium-sized in scope that could be identified within limited time.

4.1.3 *Metamodels and Reverse-Engineering Tool*

Acting as the system expert, we defined the metamodels of the logical view and the implemented view for Java Swing-based applications (see Figure 2) as well as the mapping between the two metamodels (see Table 1). We spent a half day on the discussion and definition of the metamodels and mapping rules based on our knowledge of Java Swing and understanding of the design of JEdit. We tested our metamodels and mapping rules based on some features and made several rounds of changes of the granularities of the entities. We think the efforts required to create the metamodels and mapping rules is acceptable if the system experts have sufficient understanding of the subject system, since the metamodels and mapping rules apply to most of the features of the system in a long time.

We developed an Eclipse JDT/DOM API based tool to reverse-engineer a class model from the source code of a Java Swing-based application. The tool labels JComponent subclasses as *GUIClasses* and classes with only attributes or getter/setter methods as *EntityClasses*. The tool reverse-engineers *Method call Method* and *Method read/write Attribute* relations from the source code. It aggregates the elementary relations into method-class and class-class relations. The tool extracts the name and code fragment of a code entity. It converts code fragment into a bag of words in three steps: tokenization, stop word (e.g., if, while) removal, and stemming. Given bags of words of all the code entities, the tool builds a Vector Space Model (using Lucene API [1]) in which a code entity is represented as a TF/IDF vector [30].

Table 2 Performance of the *ReFLex* Group (G_1)

	Task 1			Task 2			Task 3		
	P	R	F	P	R	F	P	R	F
P1	.50	.63	.60	.50	.78	.70	.48	.61	.58
P2	.50	.38	.39	.62	.89	.82	.73	.44	.48
P3	.04	.25	.13	.41	.78	.66	.83	.56	.60
P4	.24	.63	.47	.38	.67	.58	.52	.72	.67
P5	.11	.13	.12	.54	.78	.71	.50	.61	.59
P6	.13	.25	.21	.39	.78	.65	.17	.28	.25
P7	.29	.25	.26	.12	.22	.19	.55	.61	.60
P8	.00	.00	.00	.33	.33	.33	.53	.50	.51
P9	.22	.25	.24	.00	.00	.00	.56	.50	.51
Ave.	.23±.17	.31±.20	.27±.18	.36±.19	.58±.30	.52±.26	.54±.17	.54±.12	.53±.11

4.1.4 *Procedure*

In this study, the *ReFLex* tool uses the following parameters: the initial relevance threshold $t_{init} = 0.9$, the pace for decreasing the relevance threshold $pace = 0.1$, the minimal relevance threshold for stopping the feature location process $t_{min} = 0.5$, the threshold for selecting candidate elements $t_{cand} = 0.2$. This set of parameters have been proven effective in the evaluation of our Iterative Context-Sensitive Feature Location algorithm [43].

Before the experiment we provided a 30-minute tutorial of the JEdit software in which we introduced and demonstrated the main features of the JEdit and its package structure. Furthermore, we provided a one-hour tutorial on the *ReFLex* tool to the experimental group. Because all the participants are familiar with Eclipse IDE, we did not provide tutorial and training on Eclipse IDE. We explained the concept of reflexion model and demonstrated the main features of the *ReFLex* tool. The tutorial did not introduce any specific style of feature location process using the *ReFLex* tool.

Both groups (G_1 and G_2) were given the same set of tasks and were requested to accomplish the tasks in the same order from Task 1 to Task 3. For each task, the participants were given a description of the feature. They were requested to identify as many relevant-entities as possible in 30 minutes. They were required to run a full-screen recorder throughout the experiment. We instrumented the *ReFLex* tool to collect tool usage statistics during reflective feature location process. After the experiment, we interviewed the two groups to collect their feedback on the tasks and the tool usage.

4.2 Results: Performance Improvement (Q1)

We evaluated the participants' performance on the three tasks by precision, recall and F-measure. Given a feature f, precision (P_f) is the percentage of correctly reported relevant code entities for the feature; recall (R_f) is the percentage of actually relevant code entities of the feature being reported. The F-measure of the feature f is computed as $F_f = (1 + b^2)/(1/P_f + b^2/R_f)$ to reflect a weighted average of precision and recall. Following the treatment in [24], we set b to 2, i.e., recall is considered four times as important as precision, because finding missing results is more difficult than removing incorrect results. Table 2 and Table 3 summarize the participants' performance.

We introduced the following null and alternative hypotheses to evaluate how different the performance of experimental group (G_1) and control group (G_2) is.

H0: The primary null hypothesis is that there is no significant difference between the performance of G_1 and G_2.

H1: An alternative hypothesis to H0 is that there is significant difference between the performance of G_1 and G_2.

We used Wilcoxon's matched-pairs signed-ranked tests to evaluate the null hypothesis H0 in terms of precision, recall and F-measure for the three tasks with $p = 0.05$. Table 4 presents the results of these tests. Based on the results, we accept the null hypothesis H0 for the precision, recall and F-measure of Task 1. That is, there is no significant difference between the performance of G_1 and G_2 in Task1.

Table 3 Performance of the Eclipse IDE Group (G_2)

	Task 1			Task 2			Task 3		
	P	R	F	P	R	F	P	R	F
P10	.50	.50	.50	.29	.22	.23	.50	.28	.30
P11	.33	.25	.26	.50	.22	.25	.57	.22	.25
P12	.50	.13	.15	.33	.11	.13	.50	.06	.07
P13	.38	.38	.38	.00	.00	.00	.56	.28	.31
P14	.33	.25	.26	.13	.11	.11	.40	.33	.34
P15	.33	.13	.14	.10	.11	.11	.31	.22	.24
P16	.07	.13	.11	.13	.11	.11	.33	.22	.24
P17	.20	.13	.14	.00	.00	.00	.50	.11	.13
P18	.17	.13	.13	.00	.00	.00	.33	.11	.13
Ave.	.31±.14	.22±.13	.23±.13	.16±.16	.10±.08	.11±.09	.44±.10	.20±.09	.22±.09

Table 4 Results of Wilcoxon Tests of the null hypothesis H0, for precision, recall, and F-measure. The following measurements are reported: minimum value, maximum value, median, means (μ), variance (σ^2), Degrees of freedom (DF), Pearson correlation coefficient (PC), Z statistics, and statistical significance (p).

Task	Var	Approach	Samples	Min	Max	Median	μ	σ^2	DF	PC	Z	p	Decision
Task 1	P	ReFLex	9	.00	.50	.22	.23	.03	8	.148	-1.120	.156	Accept
		Eclipse	9	.07	.50	.33	.31	.02	8				
	R	ReFLex	9	.00	.63	.25	.31	.04	8	.816	-1.732	.074	Accept
		Eclipse	9	.13	.50	.13	.22	.02	8				
	F	ReFLex	9	.00	.60	.24	.27	.04	8	.811	-0.770	.248	Accept
		Eclipse	9	.11	.50	.15	.23	.02	8				
Task 2	P	ReFLex	9	.00	.62	.39	.36	.04	8	.630	-2.380	.008	Reject
		Eclipse	9	.00	.50	.13	.16	.03	8				
	R	ReFLex	9	.00	.89	.78	.58	.10	8	.637	-2.585	.004	Reject
		Eclipse	9	.00	.22	.11	.10	.01	8				
	F	ReFLex	9	.00	.81	.65	.52	.08	8	.655	-2.521	.004	Reject
		Eclipse	9	.00	.25	.11	.11	.01	8				
Task 3	P	ReFLex	9	.17	.83	.53	.54	.03	8	.557	-1.599	.064	Accept
		Eclipse	9	.31	.57	.50	.44	.01	8				
	R	ReFLex	9	.28	.72	.56	.54	.02	8	.296	-2.675	.002	Reject
		Eclipse	9	.06	.33	.22	.20	.01	8				
	F	ReFLex	9	.25	.67	.58	.53	.02	8	.138	-2.666	.002	Reject
		Eclipse	9	.07	.35	.24	.22	.01	8				

Table 2 and Table 3 show that the *ReFLex* users achieved better recall but worse precision in Task1 than the Eclipse IDE users. The performance of the two groups is very close in terms of F-measure. We reject the null hypothesis H0 for the precision, recall and F-measure of Task 2 and for the recall and F-measure of Task 3. Therefore, we accept the alternative hypothesis H1, i.e., there is significant difference between the performance of G_1 and G_2 for the precision, recall and F-measure of Task 2 and the recall and F-measure of Task 3. Table 2 and Table 3 show that the *ReFLex* users achieved better precision, recall, and F-measure in Task2 and Task3 than the Eclipse IDE users. Thus, the *ReFLex* users achieved significantly better performance than the Eclipse IDE users in Task2 and Task3, except that the precision improvement in Task3 is not significant.

The results show that *ReFLex* can help the developers improve the recall of feature location results. In both Task 2 and Task 3 the *ReFLex* users archived significantly better recall than the *Eclipse* users. In Task 1 the *ReFLex* users archived better (but not significantly) recall than the *Eclipse* users. The results show that the impact of *ReFLex* on the precision is mixed. In Task 2 the *ReFLex* users archived significantly better precision than the *Eclipse* users. In Task 3 the *ReFLex* users archived better (but not significantly) precision than the *Eclipse* users. In Task 1 the *ReFLex* users archived worse (but not significantly) precision than the *Eclipse* users.

Considering the granularity and complexity of the three tasks, the results suggest that *ReFLex* may work well on coarse-grained features (such as Task 2 and Task 3) that involve two or more types of logical entities and complex business logic. However, *ReFLex* may not work well on fine-grained features (such as Task 1) that involve mainly fine-grained application logic. Such fine-grained features often cannot be well broken down into a set of correlated logical entities. For example, the feature of the Task 1 was implemented in 8 methods. The logical boundary among these methods is not clear. Therefore, it becomes challenging in constructing proper logical view for the feature. This challenge is evident in the statistics of the *ReFLex* tool usage (e.g., the number of restarts) and the quality of the final reflexion model in the Task 1.

4.3 Results: Behavioral Changes (Q2)

We qualitatively compared behavioral differences between the experimental group and the control group by watching the participants' task-completion videos and by analyzing the field notes of interviews. We focused on behavioral differences in how a developer starts feature location, what he does during feature location, and when he finishes feature location.

We observed that the Eclipse users usually started feature location by identifying keywords from feature descriptions (i.e., IR-based search pattern in [22]) They then tried to find entry points by examining source code of code entities in the search results and/or exploring related entities from the search results. In contrast, the *ReFLex* users had to think about the logical entities and their relations involved in the feature. Some *ReFLex* users mentioned in the post-experiment interview that *ReFLex* requires them to consider not only entry points but also other related aspects required for the conceptual integrity of the feature.

Some *ReFLex* users mentioned that sometimes it was hard to construct the initial logical view due to the lack of understanding of the feature. In such cases, they used the *ReFLex* tool to learn the logical structure of the feature in a trial-and-error manner from the reflexion model. Once they had deeper understanding of the feature, they discarded the initial logical view and restarted with a clearer logical view. The *ReFLex*'s usage statistics support this qualitative observation. The performance metrics and the quality of final reflexion models suggest that the difficulty in constructing the proper logical view seems to affect the *ReFLex* users' feature location performance.

During feature location process, the Eclipse users refined queries based on hints collected from the result list or reading the code. They often selected some methods and examined other methods declared in the same class. They also explored code entities by following type hierarchy or call relations, but they usually can explore only two or three layers before they were lost.

In contrast, the *ReFLex* users interpreted the reflexion model and performed the structural and descriptive reflexion. They focused on macro issues such as *Convergences* and *Absences* edges in the reflexion model, the number of relevant code entities identified, and the terms and topics of relevant code entities. The *ReFLex* users usually only examined source code to confirm the relevant code entities or remove the irrelevant ones when the relevant-entities set of the logical entity is small and the edges of this logical entity in the reflexion model become *Convergences*.

The Eclipse users usually ended their feature location when they felt they cannot find any more relevant code entities. They may have confidence in the precision of their feature location results, but they often had no idea about the recall. Because they lacked an overall understanding of the feature. In contrast, 63% of *ReFLex* users ended their feature location tasks when the reflexion model became convergent and the relevant-entities set of the logical entities had a reasonable size. The *ReFLex* users often had high confidence in their results due to the overall understanding of the feature and its implementation obtained from the reflexion model. They usually had confidence in the recall of their feature location results when the reflexion model became convergent, as they knew that all the logical constituents of the feature had their corresponding implementation entities found in source code and these code entities implement all the expected relations between the logical constituents.

4.4 Results: Reflexion Strategy (Q3)

We also observed a few effective reflexion strategies in the three tasks. First, in the first few iterations of reflective feature location process the relevant-entities set of some logical entities can be large (e.g., $\geqslant 50$ elements in our study). Such a number of relevant entities can make the analysis of *Divergences* or *Absences* edges meaningless, as there may be hundreds of edges. In such situations, an effective strategy is to refine the description of these logical entities by descriptive reflexion in order to reduce the size of their relevant-entities sets.

Second, when a logical entity has a medium-sized relevant-entities set (e.g., 20-30 elements), the developer can focus on reducing *Divergences* edges in the reflexion model by removing irrelevant code entities of the logical entities. This is much easier than reducing *Absences* edges, as the developer only needs to examine each of the *Divergences* edges and consider to remove code entities that are related to the edge.

Third, when a logical entity has a small relevant-entities set (e.g., $\leqslant 10$ elements), the developer should try to reduce *Absences* edges in the reflexion model by refining the structure of the logical view or adding candidate code entities to the relevant-entities set. This is usually harder than reducing *Divergences* edges, as the developer may need to examine a lot of candidate code entities to select one from them to reduce a *Divergences* edge. The developer can also proceed to the next iteration to identify more relevant code entities with lower relevant-entity threshold. Furthermore, the developer can examine the relevant code entities in details (e.g., by reading its code) to confirm their relevance to the logical entity. He can freeze the logical entity if he has high confidence in its relevant-entities set and does not want it to be changed.

4.5 Threats to Validity

A major threat to the internal validity of our study is the "equivalence" of the experimental group and the control group. To address this threat, we carefully compared participants' experience and capabilities and randomly allocated comparable participants into the study groups. Another threat to the internal validity is "ground-truth" feature location results provided by the JEdit benchmark. As the benchmark has been used for evaluating several feature location approaches [9], the quality of the benchmark should be high. Furthermore, we manually examined and confirmed "ground-truth" results of the features used in our study. The third threat to the internal validity is the insufficient training of the tools (*ReFLex* and Eclipse IDE). This factor has greater influence on the experimental group using *ReFLex*, since the participants use Eclipse IDE regularly while *ReFLex* is a completely new tool for the participants.

A major threat to the external validity of our study lies in the fact that we only conducted one experimental study. The study was limited by the following factors: a relatively small Java system (i.e., JEdit), three features, graduate students as participants, and familiar application domain. Another threat to the external validity is that we only compared *ReFLex* with Eclipse. Other IDEs with more advanced search mechanisms such as IntelliJ IDEA or other advanced feature location tools may weaken the advantage of *ReFLex*. Therefore, our findings may not be applicable to other systems with millions lines of code or developed in other programming languages. And the assumed metamodels and mapping rules in Section 2.2 may not be applicable for other kinds of features with different characteristics. Further studies are required to generalize our findings in different kinds of systems, with different kinds of features, with professional developers, with unfamiliar application domain.

5 Case Study

We illustrate the process of applying *ReFLex* for feature location with a case from our user study. In this case the user is told to complete **Task 1**, i.e., locating relevant classes and methods for the feature "HyperSearch reports the occurrences of the string" of JEdit.

To start feature location with *ReFLex*, the user first runs JEdit and experiences the feature "HyperSearch reports the occurrences of the string". Based on his understanding of the feature, he constructs

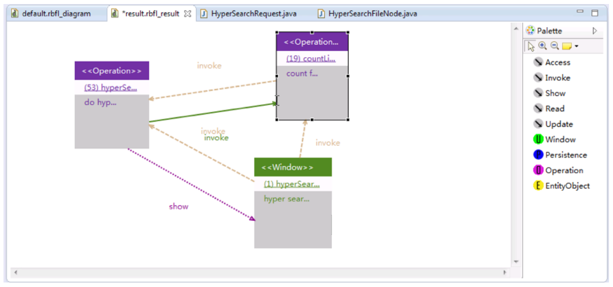


Figure 5 The Reflexion Model of the Third Iteration

an initial logical view with three logical entities: a *HyperSearch* operation (with the description "hyper, search") invokes a *CountLineNumber* operation (with the description "count, line, number") and shows a *HyperSearchWindow* window (with the description "hyper, search, result").

Iteration 1. Based on this initial logical view, *ReFLex* runs an automatic mapping process and returns an initial mapping result: 53 code entities mapped for *HyperSearch*, 9 mapped for *CountLineNumber*, and 2 mapped for *HyperSearchWindow*. The user then inspects the mapping results of the three logical entities using the "Filter by Interaction window. As there are many code entities mapped for *HyperSearch* and *CountLineNumber*, he does not filter out their results. For *HyperSearchWindow*, he confirms that a candidate code entity *HyperSearchResults* class matches the logical entity well while the other one is not relevant, so he removes the irrelevant one and freezes the results of *HyperSearchWindow*. Moreover, the user decides to revise the description of *CountLineNumber* to "count, line, file, node", as he sees the terms "file" and "node" when inspecting the candidate results and thinks that hyper search is implemented by accessing file nodes.

Iteration 2. Based on the filtered result set and updated description, *ReFLex* returns an updated mapping result: 53 code entities mapped for *HyperSearch*, 2 mapped for *CountLineNumber*, and 1 mapped for *HyperSearchWindow*. The user inspects the candidate results of the three logical entities and revises the descriptions of *HyperSearch* and *CountLineNumber* to "do, hyper, search" and "count file node" respectively.

Iteration 3. Based on the updated description, *ReFLex* returns an updated mapping result: 53 code entities mapped for *HyperSearch*, 19 mapped for *CountLineNumber*, and 1 mapped for *HyperSearchWindow*. The resulted reflexion model is shown in Figure 5. As the numbers of candidate results for *HyperSearch* and *CountLineNumber* are large, the user decides to filter the candidate results by interaction. He chooses all the candidate results of *CountLineNumber* that match the interaction from *HyperSearch* to *CountLineNumber* (a *Convergences* edge) by clicking the arrow and removes all the other candidate results. He inspects the rest candidate results of *CountLineNumber* by examining their similarity and source code. Based on the examination, he further filters out some candidates and freezes the results of *CountLineNumber*. For *HyperSearch*, he decides to keep all the candidate results of it that match the interaction from *HyperSearch* to *CountLineNumber* (a *Convergences* edge). On the other hand, there are a *Divergences* edge from *HyperSearchWindow* to *HyperSearch* and an *Absences* edge from *HyperSearch* to *HyperSearchWindow*. Based on the implementation knowledge revealed by the reflexion model, the user thinks that maybe the edge from *HyperSearchWindow* to *HyperSearch* is correct, so he decides to keep all the candidate results of *HyperSearch* that match this edge. All the other candidate results of *HyperSearch* are then removed. Moreover, the user removes the *Absences* edge from *HyperSearch* to *HyperSearchWindow* in the logical view.

Iteration 4. Based on the updated result sets and logical view, *ReFLex* returns an updated mapping result: 17 code entities mapped for *HyperSearch*, 5 mapped for *CountLineNumber*, and 1 mapped for *HyperSearchWindow*. The user thinks that the only problem now is further filtering the result set of *HyperSearch*. To this end, he examines the overall similarity and source code of the candidate results of *HyperSearch* and manually removes some of them. On the other hand, he adds an *invoke* edge from *HyperSearchWindow* to *HyperSearch* in the logical view.

Iteration 5. Based on the updated result sets and logical view, *ReFLex* returns an updated mapping result: 5 code entities mapped for *HyperSearch*, 5 mapped for *CountLineNumber*, and 1 mapped for *HyperSearchWindow*. And the reflexion model now contains two *Convergences* edges: *HyperSearchWindow* invokes *HyperSearch*, *HyperSearch* invokes *CountLineNumber*. Based on the reflexion model, the user thinks he has done with task. The final results make a precision of 50% and a recall 62.5%.

6 Discussion

The effectiveness of *ReFLex* highly relies on the two predefined metamodels of logical view and implemented view and the mapping rules between them. Different metamodels may lead to different feature location results of different quality. For example, if the developers of a system have the knowledge that a feature is implemented by message-based communication, their knowledge cannot be used for feature location based on the metamodels shown in Figure 2. The implemented-view metamodel in Figure 2 cannot reveal the implementation of message-based communication in source code. In this case more complete and fine-grained metamodels such as the one defined by Spoon [31] should be used to enable the definition of message-based communication in source code. The metamodels usually can be defined by technical experts considering the characteristics of specific business domains and technical platforms and the convention of the development team. For example, the metamodels shown in Figure 2 reflect the characteristics of Java Swing-based applications. The definition of logical view metamodel should allow the developers to fully express their knowledge of the implementation of features, i.e., the logical constituents of a feature and their interactions. The definition of implemented view metamodel, on the other hand, should consider the following two aspects. First, the implementation entities and relationships can be recovered from source code. Second, the implementation entities and relationships can be well mapped to the entities and relationships in logical view according to carefully defined mapping rules.

Therefore, the problem usually lies in the lack of clear and consistent mappings between high-level model entities in the logical view and source model entities in the implementation view. One possibility is that the implementation lacks modularity, for example, different kinds of responsibilities (user interface, business logic, technical details) are mixed in the same methods or classes. The other possibility is that some relations in the implemented view cannot be identified by static analysis, for example some functional dependencies implemented by asynchronous messaging. We remark that, although these problems may influence the accuracy of feature location for some features, our approach can generally improve the current feature location practice. Our approach prompts the developers to consider the logical constituents of a feature, which essentially reduces the gap between features and code entities by introducing an intermediate layer. On the other hand, we can ensure the positive influence of relation mapping by only involving high-confident relation mappings. In this way, the feature location results of some features with correct relation mappings can be improved, while those of the other features will not be influenced by noises introduced by relation mapping.

ReFLex asks the developers to construct a logical view of the feature and refine the reflexion model. This seems to be an additional effort for feature location, compared with IR-based code search. However, this seemingly additional effort may bring in benefits to not only feature location in particular but also software maintenance in general. *ReFLex* provides to the developers a systematic way to evaluate the feature location results and refine their search. This can improve the developers' performance (especially recall) in feature location tasks (especially for coarse-grained features). Furthermore, the resulting convergent reflexion model emerged from reflective feature location process can increase the developers' confidence in their feature location results and their overall understanding of the feature and its implementation. Feature location has been recognized as the prerequisite for many software maintenance tasks [9, 40]. The overall understanding of the feature and its implementation is necessary to accomplish a software maintenance task. *ReFLex* makes the development of this understanding an integral part of reflective feature location process. Therefore, *ReFLex* may have longer-term impact on the maintenance of a software system, because it puts feature location back into the context of software maintenance tasks in which it is required.

7 Related Work

IR-based code search can be enhanced by query reformulation techniques [32, 33], better presentation of search results [23,29], and interactive exploration tools [6,23,35]. Beck et al. [17] propose a feature location approach with improved user interface to better support developers during browsing and inspecting the retrieved code entities. Feature location can be facilitated by grouping source artifacts that use similar vocabulary by using topic mining techniques such as the semantic clustering technique proposed by Kuhn et al. [2]. It can also be facilitated by extracting conceptual descriptions from source code by using source code summarization techniques [34]. The user studies by Wilson et al. [25] and Fritz et al. [36] suggest that developers form high-level concept map or mental model of the feature to assist their feature location and code navigation tasks. However, these approaches do not provide systematic ways that allows developers to express their high-level view of the feature and to reconcile the discrepancies between the developers' high-level understanding of the feature and the feature's implementation in the system.

Techniques have been proposed to exploit structural information of software system to improve feature location [15, 27, 44]. For example, Robillard's approach [27] identifies relevant code entities based on structural relevance of candidate entities to previously identified relevant entities. Hill et al. [15] explore lexical and structural information to suggest code entities based on a set of entrance points. Portfolio [10] uses network analysis of code structure to improve the ranking of the search results. None of these techniques explore the structural information at both logical and code levels in an iterative feature location process.

Our study on how developers perform feature location tasks suggest that feature location is a human-centered, iterative information seeking process [22]. Incremental feature location approaches allow developers to interactively investigate candidate results, provide feedbacks, and update feature queries. iFL [35] iteratively improves relevance scores of candidate elements based on a relevance feedback mechanism that examines method invocations of code entities. Lucia et al. [6] propose an incremental process for IR-based traceability recovery, which allows developers to tune the threshold in an iterative process and control the number of validated correct links. In [8], Lucia et al. further compare the performances of "one-shot" approach and incremental approach with an empirical study. Their results indicated that incremental process can improve the performances of traceability link recovery and reduce required effort. Poshyvanyk et al. [11] propose to organize query results in a labeled concept lattice to facilitate the exploration of relevant categories in the lattice. Our previous work [23] supports multi-faceted interactive feature location. It allows developers to interactively group, sort, and filter feature location results with multiple syntactic and semantic facets automatically mined from candidate code entities. These approaches do not capture and use logical design structure of a feature in the exploration process.

Our Iterative Context-aware Feature Location algorithm (*ICFL*) [43] compares both the lexical description and structural context of features and code entities to determine their relevance. The *ReFLex*'s structure-based matching algorithm adopts structure matching concept similar to that of *ICFL*. However, *ICFL* takes as input a feature model (i.e., multiple features and their dependencies), while the *ReFLex*'s matching algorithm takes as input a logical view of one feature. More importantly, *ICFL* is just an algorithm, while *ReFLex* supports reflective feature location process. Structure-based matching is only one step of this process.

Hill et al. [14] empirically investigate how feature role information, i.e., the role a code element plays in the larger feature, can be utilized to strengthen evaluation studies of feature location techniques. Their study provides empirical support for the assumption of our approach, i.e., feature location can be improved by considering the logical constituents (roles) of features.

In contrast, our *ReFLex* approach explicitly supports developers to express their understanding of the feature in a high-level logical view. It computes the reflexion model [20] to help developers to reconcile the discrepancies between the high-level logical view of the feature and the implemented view of the system through a reflexion process. The usefulness of software reflexion model has been demonstrated in various software maintenance tasks such as architecture compliance checking [21], change impact analysis [37], and reengineering [19]. Our approach is the first attempt to adopt reflexion model for feature location.

8 Conclusion and Future Work

In this paper we presented our reflective feature location approach and the tool support. Our approach provides a systematic way for the developers to express their high-level understanding of the feature and to identify and reconcile the discrepancies between their high-level understanding of the feature and the feature implementation. Our user study yielded several promising observations of the reflective feature location. First, developers can express their high-level understanding of the feature in a logical view through reflective feature location process. Second, reflexion on the discrepancies between the logical view of the feature and the system implementation can improve feature location results, especially for coarse-grained features. Third, reflective feature location may lead to longer-term impact on software maintenance as it can help developers gain a deeper understanding of the feature and its implementation. In the future, we will investigate more reflexion mechanisms to better support developers to refine logical views and/or feature location results.

Acknowledgements This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000801, National Natural Science Foundation of China under Grant No. 61370079.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 http://lucene.apache.org/.

2 Kuhn A, Ducasse S, and Gîrba T. Semantic clustering: Identifying topics in source code. *Inform & Software Tech*, 49(3):230–243, 2007.

3 Marcus A, Sergeyev A, Rajlich V, and Maletic J I. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223.

4 Marcus A and Maletic J I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135.

5 Eisenberg A D and Volder K D. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 337–346.

6 Lucia A D, Fasano F, Oliveto R, and Tortora G. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM T on Softw Eng Meth*, 16(4), 2007.

7 Lucia A D, Penta M D, Oliveto R, Panichella A, and Panichella S. Labeling source code with information retrieval methods: an empirical study. *Empir Softw Eng*, 19(5):1383–1420, 2014.

8 Lucia A D, Oliveto R, and Tortora G. IR-based traceability recovery processes: An empirical comparison of "one-shot" and incremental processes. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 39–48.

9 Dit B, Revelle M, Gethers M, and Poshyvanyk D. Feature location in source code: A taxonomy and survey. *J Softw-evol Proc*, 25(1):53–95, 2013.

10 McMillan C, Grechanik M, Poshyvanyk D, Xie Q, and Fu C. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120.

11 Poshyvanyk D and Marcus A. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension*, ICPC '07, pages 37–48.

12 Poshyvanyk D, Guéhéneuc Y, Marcus A, Antoniol G, and Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE T Software Eng*, 33(6):420–432, 2007.

13 Blei D M, Ng A Y, and Jordan M I. Latent dirichlet allocation. *J Mach Learn Res*, 3:993–1022, 2003.

14 Hill E, Shepherd D, and Pollock L L. Exploring the use of concern element role information in feature location evaluation. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 140–150.

15 Hill E, Pollock L L, and Vijay-Shanker K. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 14–23.

16 Thomas E, Rainer K, and Daniel S. Locating features in source code. *IEEE T Software Eng*, 29(3):210–224, 2003.

17 Beck F, Dit B, Velasco-Madden J, Weiskopf D, and Poshyvanyk D. Rethinking user interfaces for feature location. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 151–162.

18 Gay G, Haiduc S, Marcus A, and Menzies T. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th IEEE International Conference on Software MaintenanceM*, ICSM '09, pages 351–360.

19 Murphy G C and Notkin D. Reengineering with reflexion models: A case study. *IEEE Comput*, 30(8):29–36, 1997.

20 Murphy G C, Notkin D, and Sullivan K J. Software reflexion models: Bridging the gap between design and implementation. *IEEE T Software Eng*, 27(4):364–380, 2001.

21 Knodel J and Popescu D. A comparison of static architecture compliance checking approaches. In *Proceedings of the Sixth Working IEEE / IFIP Conference on Software Architecture*, WICSA '07, pages 12–12.

22 Wang J, Peng X, Xing Z, and Zhao W. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *Proceedings of the IEEE 27th International Conference on Software Maintenance*, ICSM '11, pages 213–222.

23 Wang J, Peng X, Xing Z, and Zhao W. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 762–771.

24 Hayes J H, Dekhtyar A, and Sundaram S K. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE T Software Eng*, 32(1):4–19, 2006.

25 Wilson L A, Petrenko M, and Rajlich V. Using concept maps to assist program comprehension and concept location: An empirical study. *Journal of Information & Knowledge Management*, 11(3), 2012.

26 Trifu M. Using dataflow information for concern identification in object-oriented software systems. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pages 193–202.

27 Robillard M P. Topology analysis of software dependencies. *ACM T Softw Eng Meth*, 17(4):18:1–18:36, 2008.

28 Kruchten P. The 4+ 1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

29 Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension*, ICPC '07, pages 37–48.

30 Baeza-Yates R and Ribeiro-Neto B. *Modern Information Retrieval*. 1st ed. New Jersey: Addison Wesley, 1999.

31 Pawlak R, Monperrus M, Petitprez N, Noguera C, and Seinturier L. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, page na, 2015.

32 Haiduc S, Bavota G, Marcus A, Oliveto R, Lucia A D, and Menzies T. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 842–851.

33 Haiduc S, Bavota G, Oliveto R, Lucia A D, and Marcus A. Automatic query performance assessment during the retrieval of software artifacts. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, pages 90–99.

34 Haiduc S, Aponte J, and Marcus A. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 223–226.

35 Hayashi S, Sekine K, and Saeki M. iFL: An interactive environment for understanding feature implementations. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–5.

36 Fritz T, Shepherd D C, Kevic K, Snipes W, and Bräunlich C. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 7–18.

37 Kim T, Kim K, and Kim W. An interactive change impact analysis based on an architectural reflexion model approach. In *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference*, COMPSAC '10, pages 297–302.

38 Biggerstaff T J, Mitbander B G, and Webster D E. Program understanding and the concept assignment problem. *Commun. ACM*, 37(5):72–82, 1994.

39 Rajlich V. *Software Engineering: The Current Practice*. 1st ed. BocaRaton: CRC Press, 2012.

40 Rajlich V and Wilde N. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–278.

41 Zhao W, Zhang L, Liu Y, Sun J, and Yang F. SNIAFL: Towards a static non-interactive approach to feature location. *ACM T Softw Eng Meth*, 15(2):195–226, 2006.

42 Wong W E, Gokhale S S, Horgan J R, and Trivedi K S. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, ASSET '99, pages 194–203.

43 Peng X, Xing Z, Tan X, Yu Y, and Zhao W. Improving feature location using structural similarity and iterative graph mapping. *J Syst Software*, 86(3):664–676, 2013.

44 Wang X, Lo D, Cheng J, Zhang L, Mei H, and Yu J X. Matching dependence-related queries in the system dependence graph. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 457–466.