Mining Logical Clones in Software: Revealing High-Level Business and Programming Rules

Wenyi Qian*, Xin Peng*, Zhenchang Xing†, Stan Jarzabek‡, and Wenyun Zhao*
*Software School, Fudan University, Shanghai, China
Email: {11212010025, pengxin, wyzhao}@fudan.edu.cn
†School of Computer Engineering, Nanyang Technological University, Singapore
Email: zcxing@ntu.edu.sg
‡School of Computing, National University of Singapore, Singapore
Email: dcssj@nus.edu.sg

***Abstract*—Software systems contain many implicit application-specific business and programming rules. These rules represent high-level logical structures and processes for application-specific business and programming concerns. They are crucial for program understanding, consistent evolution, and systematic reuse. However, existing pattern mining and analysis approaches cannot effectively mine such application-specific rules. In this paper, we present an approach for mining logical clones in software that reveal high-level business and programming rules. Our approach extracts a program model from source code, and enriches the program model with code clone information, functional clusters (i.e., a set of methods dealing with similar topics or concerns), and abstract entity classes (representing sibling entity classes). It then analyzes the enriched program model for mining recurring logical structures as logical clones. We have implemented our approach in a tool called MiLoCo (Mining Logical Clone) and conducted a case study with an open-source ERP and CRM software. Our results show that MiLoCo can identify meaningful and useful logical clones for program understanding, evolution and reuse.**

***Keywords*-logical clone; semantic clustering; program comprehension; evolution; reuse;**

I. Introduction

The development and maintenance of a software system often follows many business and programming rules. Take the diagram in Figure 1 as an example. This diagram represents a logical business and programming rule implemented in a publication management system. It consists of various kinds of code entities such as methods, code clone sets, entity classes, functional clusters, and invoke/contain/access relationships between these code entities.

This rule shows business process as well as program convention for adding a publication into the system. The system currently deals with three types of publications, i.e., conference papers, journal articles, and books. To add a specific publication, the system invokes the corresponding `add` operation of `PublicationProcess`. The `add` operation first accesses information from the `Publication` entity object representing the publication to be added. It then invokes `User.checkAuthority()` to examine if the user has the access right to add a new publication. After that, the `add` operation checks the integrity of publication information by invoking the corresponding `check` operation of `PublicationProcess`. Next, it invokes

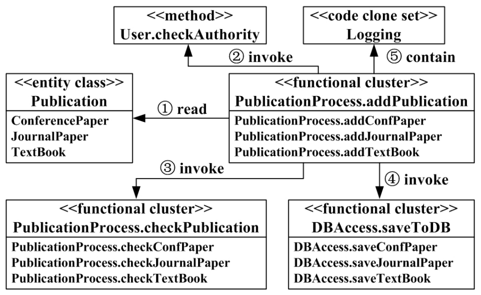

<<method>>
User.checkAuthority
<<code clone set>>
Logging
② invoke
⑤ contain
<<entity class>>
Publication
ConferencePaper
JournalPaper
TextBook
① read
<<functional cluster>>
PublicationProcess.addPublication
PublicationProcess.addConfPaper
PublicationProcess.addJournalPaper
PublicationProcess.addTextBook
③ invoke
④ invoke
<<functional cluster>>
PublicationProcess.checkPublication
PublicationProcess.checkConfPaper
PublicationProcess.checkJournalPaper
PublicationProcess.checkTextBook
<<functional cluster>>
DBAccess.saveToDB
DBAccess.saveConfPaper
DBAccess.saveJournalPaper
DBAccess.saveTextBook

Fig. 1. An Example of Logical Clone

`DBAccess.saveToDB()` to store publication information into database. Finally, the `add` operation logs the transaction.

Such logical rules are crucial for program understanding, consistent evolution, and systematic reuse. For example, suppose the system needs to be extended with a new kind of publication (e.g., CD). Given the above logical rule, a developer is clear about what he needs to do. He will introduce a new `Publication` entity class, a new `add` and `check` operation in `PublicationProcess` for CD, and also a new `save` operation to `DBAccess`. He can duplicate logging code from existing `add` operations. Furthermore, he can also learn the proper business process for adding a CD from the ordering of operations captured by the logical rule. Without this explicit logical rule that can be followed, the developer may face the risk of introducing inconsistent evolution or even bugs. For example, he may forget to check information integrity before storing a publication into database.

This logical rule may also provide useful hints for reengineering decisions because it provides a high-level overview of several related business and entity objects. For example, a developer may consider moving information integrity checking operation from `PublicationProces` to corresponding `Publication` entity classes, because these checking operations suffer from feature envy [1] in the overall business process.

In spite of the importance of logical business and programming rules, they are often not well documented during software development and maintenance. Instead, they are only implicitly present in system implementation. Making such rules explicit and understood by developers improves developers' efficiency in software maintenance and also quality of the software system. However, detecting business and programming rules such as the one in Figure 1 poses unique challenges to existing pattern mining and analysis approaches.

The application of design patterns [2] may result in similar design structures. As design patterns provide clear descriptions about static structures and dynamic behaviors of the documented patterns, certain structural or behavioral templates [3], [4] can be defined to identify design structures following design patterns. However, implicit business and programming rules are application-specific. They are essentially open-ended and emerging from system implementation. No templates can be predefined. Furthermore, design patterns describe good design in an abstract manner. Detecting instances of design patterns is not concerned with application-specific semantics. In contrast, business and program rules reflect application-specific semantics, for example the meaning of a given method invocation sequence.

Software clones refer to similar code fragments or code structures. Business and programming rules are implemented as high-level similar structures and processes across several classes and methods. They are at a much higher level of abstraction than code clones and their instances may consist of one or more code clones (such as `Logging` in our example). Structural clones [5] can reveal high-level duplications in a system which may consist of multiple cross-cutting simple code clones. However, the elements in a business or programming rule that deal with the same or similar business concerns may not be similar enough to be detected as structural clones. For example, the three `add` operations of `PublicationProcess` are very different in implementation because they deal with different kind of publication. However, they share similar topics and play the same role in the business processes of the publication management system.

In this paper, we present an automatic approach for mining implicit business and programming rules in a software system. We call mined rules logical clones. A logical clone represents a high-level logical structure and process for an application-specific business or programming concern. Our approach takes as input source code of a software system. It mines logical clones in two steps. First, it identifies various kinds of similar program elements. Our approach analyzes methods, entity classes, and persistent data objects. It groups similar methods into functional clusters using semantic clustering technique [6], and detects similar code fragments inside methods (i.e., code clones) using clone detectors (such as Simian [7]). The first step generates a large graph whose nodes represent methods, entity classes, persistent data objects, and the identified functional clusters and code clones, and whose edges represent invoke/access/contain relations between program elements. Next, our approach uses subgraph pattern mining technique to mine logical clones in the system. The mined logical clones consist of functional clusters (i.e., a set of similar methods), single methods, code clone sets, entity classes (concrete or abstract), persistent data objects, and their invoke/access/contain relations (see Figure 1 for an example).

We have implemented our approach in a tool called MiLoCo (Mining Logical Clone). To evaluate the effectiveness and usefulness of logical clones that MiLoCo reports, we conducted a case study with Opentaps [8], an open-source Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) software. In this study, MiLoCo reported 1,690 logical clones that involve more than 24% classes and 5% methods in the subject system. These logical clones reveal recurring program conventions, design structures, business tasks, and business processes in Opentaps. We also surveyed five experienced developers about the usefulness of the mined logic clones. Our survey suggests that most of the mined logical clones were classified as meaningful and useful for program understanding and evolution.

The remainder of the paper is organized as follows. Section II presents some related work and compares them with ours. Section III describes the proposed approach for mining logical clones. Section IV presents the supporting tool MiLoCo. Section V reports the evaluation based on the case study with Opentaps. Section VI discusses some related issues. Finally, Section VII concludes the paper and outlines the future work.

II. Related Work

Related work of our research spans five aspects, i.e., simple clone detection, high-level clone detection, design pattern detection, model clone detection, and API usage pattern detection.

Simple clone means similar or identical code fragments reflecting duplication at the code level. They can be introduced by copy-paste-modify practice, implementation of similar requirements, or following usage specification of APIs. Simple clone detection has been a well-researched area in recent years and a number of clone detection techniques have been proposed. These techniques detect simple code clones by analyzing program text [9], program tokens [10], and code metrics [11]. Marcus et al. [12] proposed an approach that uses an information retrieval technique (i.e., latent semantic indexing) to identify identify implementations of similar high-level concepts (e.g., abstract data types). Some techniques have been proposed to detect clones by comparing Abstract Syntax Tree (AST) [13], or Program Dependency Graph (PDG) [14]. These techniques can find code fragments that are lexically different but structurally similar. However, clones being reported are still at the code level within methods.

Different from simple code clones, structural clones [5], [15] are larger duplicated program structures consisting of multiple fragments of duplicated code. Such high-level clones can also represent important domain or design concepts. Basit et al. [5] proposed a data mining approach to detect structural clones at different levels, including method clones, file clones and directory clones. Their follow-up study [15] showed that over 50% of simple clones in the subject systems can be captured by structural clones. Structural clones are recurring configurations of simple clones. In contrast, logical clone detection combines several techniques, such as simple clone detection, semantic clustering and so on, to get more complex clones. Besides simple clones, the elements in logical clone can also be methods sharing similar topics or entity classes inherited from the same superclass. Furthermore, the elements in a logical clone can be distributed in different classes and connected by invocation and access relations.

Some researchers focus on detecting similar design structures caused by application of design patterns. Tsantalis et al. [3] proposed an approach that uses the similarity score between graph representation of a system and a design pattern to detect instances of design patterns. Romano et al. [4] presented an approach that applies text clustering on classes of a system and then uses existing tools such as DPR [16] and Pattern4 [3] to identify design pattern instances based on obtained clusters. These design pattern detection approaches are usually based on predefined descriptions about the structure and behaviors of a design pattern, while logical clones represent application-specific business and programming rules that are implicitly applied in system implementation. Furthermore, design pattern detection is not concerned with application-specific semantics. In contrast, logical clones reveal application-specific semantics, for example what a set of similar methods do.

Model clone detection is focused on detecting similar or identical fragments in software models. Alalfi et al. [17] proposed an approach to identify structurally meaningful subsystem clones in graphical models such as Simulink models. Different from model clone detection, our approach on logical clone detection is concerned with application-specific semantics such as topics of program elements.

In recent years, there has been some research [18], [19], [20] focusing on detecting API usage patterns, which reflect similar API call sequences and pre-conditions when using a set of APIs. Compared with API usage patterns, logical clones reveal high-level system structures involving elements from multiple classes and richer structural relations such as accessing entity classes in addition to invoking APIs. Moreover, logical clone detection can identify invocations of similar methods rather than invocations to exactly the same APIs.

III. Approach

In this section, we first give an overview of our approach for mining logical clones. Figure 2 shows an overview of our approach. Our approach takes as input source code of a software system and reports a set of logical clones. It consists of two steps, i.e., model extraction and graph mining. Model extraction is to build a program model from source code for the subsequent graph mining step. The initial program model consists of methods, entity classes, persistent data objects, and their invoke/access relations. Similar methods are then clustered into functional clusters, code clones are detected by clone detection tools, and sibling entity-classes are grouped by their inheritance relations. Based on the extracted program model, a subgraph pattern mining algorithm is used to mine logical clones. This mining algorithm first generates initial logical clones with only one functional cluster and then incrementally extends current logical clones by including one neighboring node at a time.

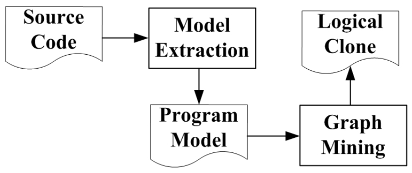


Fig. 2. Approach Overview

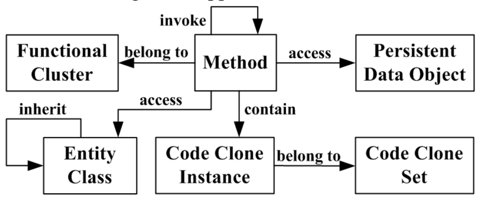


Fig. 3. Metamodel of Extracted Program Models

A. Model Extraction

Figure 3 presents the metamodel of program model being extracted in our approach. The program model is a directed graph. The initial program model contains methods, entity classes, and persistent objects. A method can invoke other methods, and access (create, read, or update) entity classes and persistent data objects. An entity class can inherit other entity class. Given the initial program model, individual methods that share similar topics can be grouped into functional clusters based on their lexical descriptions using semantic clustering [6]. The cloned code fragments (i.e., clone instances) in methods are identified by a clone detector (such as Simian [7]). Clone detector usually reports code clone instances in clone sets. Sibling entity classes that are the descendant classes of the same classes are grouped together as an abstract entity class.

1) Methods and Functional Clusters: Methods of a software system and their invocation relations can be easily extracted from source code by static analysis. We exclude simple getter and setter methods from the program model, because our logical clone analysis is focused on meaningful operations for high-level business or programming concerns. However, it is important to note that we exploit getter and setter methods to discover data access relations between methods and entity classes. For example, we consider that a method reads information from an entity class if the method invokes getter methods of the entity class.

To reveal application-specific topics of several methods, we use semantic clustering [6] to group methods that share similar vocabulary into functional clusters. Our approach first transforms all the methods in the program model into a corpus of documents for clustering. For each method, its method name (including package and class name) and parameter names are used together as document title. Method body is transformed into content of document. The identifier names and comments in method body are extracted and transformed by standard Information Retrieval (IR) preprocessing steps including tokenization, stop-words filtering and word stemming.

Given the generated corpus of documents, our approach then generates a Term-Frequency/Inverse-Document-Frequency (T-F/IDF) vector for each method document. It uses a bisecting K-means clustering algorithm to group generated document vectors into different clusters. Bisecting K-means clustering [21] is an extension of K-means clustering, which does not require a predefined K value. We implemented a bisecting K-means clustering algorithm based on the K-means clustering algorithm provided by Weka [22] (see Algorithm 1).

The algorithm takes as input a set of document vectors V and produces a set of functional clusters. It performs an iterative clustering process with a queue initialized with V. In each iteration, it dequeues a set V' of document vectors from the queue and divides it into two subsets (i.e., clusters) by K-means clustering with K=2. For each of produced clusters, we use $Diff$ function (see the following formula) to measure the overall difference between the document vectors in the cluster and the mean vector of the cluster, where $distance$ function returns Euclidean distance between a vector x_i in the cluster and the mean vector $\bar{x}$.

$$Diff(Cluster) = \sqrt{\frac{1}{|Cluster|}\sum_{i=1}^{|Cluster|} distance(x_i, \bar{x})^2}$$

If this overall difference of the cluster is higher than a given threshold δ, it is added to the queue for further division. Otherwise, it is added to $Clusters$ as a functional cluster. The iterative process ends when the queue is empty.

Algorithm 1 Bisecting K-means Clustering Algorithm

```
1:  function BISECTINGCLUSTERING($V$)
2:      $Clusters = [\,], queue = [\,]$
3:      $queue.enQueue(V)$
4:      while $queue.length > 0$ do
5:          $V' = queue.deQueue$
6:          $C = KmeansClustering(V', 2)$
7:          for $(i = 1; i \leq 2; i++)$ do
8:              if $Diff(C[i]) > \delta$ then
9:                  $queue.enQueue(C[i])$
10:             else
11:                 add $C[i]$ to $Clusters$
12:             end if
13:         end for
14:     end while
15:     return $Clusters$
16: end function
```

2) Entity Classes: An entity class encapsulates information of a business entity and corresponding getter and setter methods. The access (i.e., create, read, update) relations between a method and an entity class can be identified as follows. If a method invokes constructors of an entity class, it creates instances of the entity class. If a method reads properties or invokes getter methods of an entity class, it reads information from the entity class. If a method modifies properties or invokes setter methods of an entity class, it updates the entity class.

Inheritance relations between entity classes are also extracted by static analysis. We consider entity classes inheriting from the same superclass defined in the system as playing the same role in a logical clone.

Access relations between a method and an entity class will be generalized as access relations between the method and the abstract entity class. In our running example, three `Publication` subclasses are grouped into an abstract entity class representing `Publication` entity classes. The read relations from different `add` operations to sibling entity classes `ConferencePaper`, `JournalPaper`, and `TextBook` can be generalized as a read relation to the abstract entity class `Publication`.

3) Code Clones: In our approach we detect code clones using existing clone detection tools. Our current implementation uses text-based tool (e.g., Simian) for clone detection. Clone detectors usually report code clones in clone sets. Each clone set contains two or more code clone instances (i.e., similar code fragments) in several methods. For example, three `add` operations of `PublicationProcess` contain code clones in a code clone set. These cloned code fragments implement the same feature, i.e., logging transaction for adding publication.

4) Persistent Data Objects: Persistent data objects represent data tables in database or data entries in files (e.g., XML files). They can usually be identified from data dictionaries or data schemas. However, different applications may use different methods to access persistent data objects. For example, in some applications, methods may directly access database tables with SQL queries. In other applications, database access may be implemented using Object-Relational (O/R) mapping frameworks (such as Hibernate [23]). Therefore, access relations between methods and persistent data objects need to be analyzed depending on application-specific persistent data access strategy. For example, for the first case mentioned above, we can intercept SQL queries issued in each method to determine persistent data objects that the method accesses. For the second case, we can analyze XML configuration files of O/R mapping frameworks.

B. Subgraph Pattern Mining

Figure 4 shows metamodel of logical clones mined using our subgraph pattern mining algorithm. A logical clone is represented as a graph that consists of functional clusters that invoke other functional clusters and/or methods, access entity classes (concrete or abstract classes), and contain code clone sets. For example, the logical clone shown in Figure 1 consists of six graph nodes. The functional cluster `PublicationProcess.addPublication` consists of three similar `add` operations. It reads abstract entity class `Publication` that consists of three sibling subclasses of `Publication`. It invokes method `User.checkAuthority` and two other functional clusters `PublicationProcess.checkPublication` and `DBAccess.saveToDB`. Finally, it logs transaction using cloned code fragments found in all `add` operations.

Based on the program model produced by model extraction step, we then use a subgraph pattern mining algorithm to detect logical clones (see Algorithm 2). The algorithm takes as input a program model $Model$, which includes a set of functional

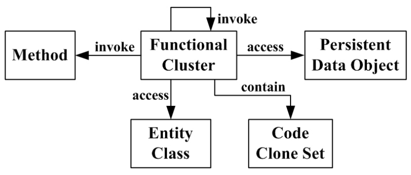


Fig. 4. Metamodel of Logical Clones

clusters $FCSet$, a set of methods $MESet$, a set of entity classes $ECSet$, a set of code clone instances $CISet$, a set of code clone sets $CSSet$, and a set of persistent data objects $DOSet$. It produces a set of logical clones, each of which can be represented as a graph as defined by the logical clone metamodel (see Figure 4).

The algorithm first generates an initial set of logical clones that consists of only one node and then incrementally extends existing logical clones by including neighboring nodes one at a time. Two thresholds are used to eliminate logical clones with too few nodes (lower than $threshold_{node}$) or too few instances (lower than $threshold_{instance}$). An instance of a logical clone is a program structure consisting of methods, code clone instances, entity classes, persistent data objects, each of which is an instance of the corresponding node in the logical clone.

The algorithm first initiates a set of logical clones with only one node (Line 3-9). For each functional cluster with the number of methods greater than $threshold_{instance}$, $newLCNode$ function creates a logical clone node $node$ using all of the methods in the functional cluster as the instances of the logical clone node. This logical clone node is then added to a queue.

Next, an incremental analysis process is conducted until the queue is empty (Line 10-25). In each iteration, the algorithm dequeues a candidate logical clone $candLC$ and considers possible extensions to it. Each functional cluster in $candLC$ is considered as an extension point (Line 13-21). For each extension point, $Extend$ function (see Algorithm 3) generates a set of extended logical clones based on $candLC$. These extended logical clones are added to the queue for further extension.

For each $candLC$, a flag is used to indicate whether it is contained by another logical clone extended from it. The $contain$ function returns whether an extended logical clone $newLC$ contains $candLC$, i.e., all instances of $candLC$ are contained by the instances of the extended logical clone $newLC$. If the flag is false (i.e., $candLC$ cannot be contained by any extended logical clones) and the size of $candLC$ (i.e., the number of nodes in the candidate logical clone) is greater than $threshold_{node}$, this $candLC$ is added to $LCSet$ as a detected logical clone (Line 22-24).

$Extend$ function (see Algorithm 3) generates a set of extended logical clones based on a given logical clone $candLC$. Each of the generated logical clone extends the original $candLC$ with one more node from the given extension point $node$. The newly included node el can be: a method that some instances (i.e., methods belonging to the functional cluster) of $node$ invoke (Line 4-6); a functional cluster that some

Algorithm 2 Logical Clone Mining Algorithm

```
1: function LOGICALCLONEDETECT($Model$)
2:     $LCSet = \{\ \}, queue = [\ ]$
3:     for each $fc \in Model.FCSet$ do
4:         $methods = \{m|m \in Model.MESet \wedge m$ belong to $fc\}$
5:         if $methods.size \geq threshold_{instance}$ then
6:             $node = newLCNode(fc, methods)$
7:             $queue.enQueue(\{node\})$
8:         end if
9:     end for
10:    while $queue.length > 0$ do
11:        $candLC = queue.deQueue()$
12:        $flag = False$
13:        for each $node \in candLC \wedge node.type = FC$ do
14:            $newLCS = Extend(Model, candLC, node)$
15:            for each $newLC \in newLCS$ do
16:                $queue.enQueue(newLC)$
17:                if $contain(newLC, candLC)$ then
18:                    $flag = True$
19:                end if
20:            end for
21:        end for
22:        if $\neg flag \wedge candLC.size \geq threshold_{node}$ then
23:            $LCSet = LCSet \cup candLC$
24:        end if
25:    end while
26:    return $LCSet$
27: end function
```

instances of $node$ invoke (Line 7-9); an entity class whose subclasses (including itself) are accessed by some instances of $node$ (Line 10-12); a persistent data object that some instances of $node$ access (Line 13-15); a code clone set whose clone instances are contained in some instances of $node$ (Line 16-18). Note that not all the instances of the extension point $node$ are required to have certain types of relations with el. Therefore, the instances of a logical clone extended from $candLC$ may be less than those of $candLC$. Only those extended logical clones whose instance numbers are greater than $threshold_{instance}$ are returned (Line 19-22).

C. Representation of Logical Clones

To help developers understand mined logical clones, we generate meaningful labels and natural language summary for each logical clone.

1) Labeling: Our approach generates a meaningful label for each node of a logical clone using different strategies for different kinds of nodes. For method node or persistent data object node, method name or object name is used as its label. For functional cluster node, a label is generalized from names of all the methods in the cluster. Our approach splits method names into word sequences. It keeps longest common subsequence of method names and replace differences in method names using wildcards. For example, the names of two methods $updateImportServices$ and $updateExportServices$ can be generalized to a label $update * Services$, which can be interpreted as "update different kinds of services". For code clone set node, a set of tags are generated as its label using

Algorithm 3 Logical Clone Extension Algorithm

```
1: function EXTEND($Model, candLC, node$)
2:     $ExLCSet = \{\ \}, queue = [\ ]$
3:     for each $el \in Model.elements \wedge el \notin candLC$ do
4:         if $el \in Model.MESet$ then
5:             $INS = \{n | n \in node.instances \wedge n$ invoke $el\}$
6:         end if
7:         if $el \in Model.FCSet$ then
8:             $INS = \{n | n \in node.instances \wedge \exists m \in \{m' | m' \in Model.MESet \wedge m'$ belong to $el \wedge n$ invoke $m'\}\}$
9:         end if
10:        if $el \in Model.ECSet$ then
11:            $INS = \{n | n \in node.instances \wedge \exists m \in \{m' | m' \in Model.ECSet \wedge m'$ inherit $el \wedge n$ access $m'\}\}$
12:        end if
13:        if $el \in Model.DOSet$ then
14:            $INS = \{n | n \in node.instances \wedge n$ access $el\}$
15:        end if
16:        if $el \in Model.CSSet$ then
17:            $INS = \{n | n \in node.instances \wedge \exists m \in \{m' | m' \in Model.CISet \wedge m'$ belong to $el \wedge n$ contain $m'\}\}$
18:        end if
19:        if $INS.size \geq threshold_{instance}$ then
20:            $newLC = candLC \cup \{el\}$
21:            $ExLCSet = ExLCSet \cup \{newLC\}$
22:        end if
23:    end for
24:    return $ExLCSet$
25: end function
```

topic mining techniques. For entity-class node, the name of the common superclass that all entity classes inherit is used as its label. Furthermore, our approach generates labels for different types of data access relations (i.e., create, read or update).

To partially reflect the sequence of method invocations and data accesses, our approach produces numbers indicating the order of invoke/access by analyzing the position of the invoke/access relations in source code. As a functional cluster node can have multiple instances, the ordering number is only produced when the orders in all its instances are consistent.

2) Topics and Natural Language Summary: For each logical clone, our approach generates a set of topics from source code of all its instances using topic mining techniques. Furthermore, it generates a natural language summary based on a template defined according to the logical clone metamodel. This natural language summary provides an overview of the logical clone. For example, our approach generates a natural language summary for the logical clone shown in Figure 1 as follows.

The addPublication method, read information of Publication entities, invoke User.method, invoke checkPublication methods, invoke saveToDB methods, contain clone for Logging.

IV. Tool Implementation

We have implemented our approach in a tool called MiLoCo (Mining Logical Clone). Figure 5 shows a screen shot of the tool as it has been used to analyze logical clones in Opentas [8] in our case study.

MiLoCo currently supports logical clone analysis in Java-based software systems. Our current implementation uses Simian [7], a text-based clone detector, for clone detection in code. In our current implementation of bisecting K-means clustering algorithm, we set the maximum number of iterations to be executed to 30 in K-means.

MiLoCo displays mined logical clones in a list view (the upper part). The list can be sorted by the number of nodes in logical clones or the number of instances in logical clones. For each logical clone, the list shows its ID (a unique identifier generated by our approach), the number of nodes, the number of instances, topics, and labels of the logical clone.

A user can inspect the details of a logical clone by double-clicking the logical clone in the list. The detail panel (the lower part) provides three views for inspecting logical clone, i.e., summary view on the left, graph model view in the middle, and source code view on the right. The summary view provides natural language description of the selected logical clone and its basic information such as topics, the number of nodes, and the number of instances. The graph model view shows nodes of the logical clone and their relations. For example, the logical clone shown in Figure 5 contains eight graph nodes. Different kinds of nodes are shown in different colors, for example, blue nodes for functional clusters, red nodes for code clone sets, pink nodes for persistent data objects, green nodes for entity classes. For each functional cluster or code clone set node, the user can expand or shrink the node to inspect its instances in the cluster or clone set by double clicking the node on the graph model. The source code view shows source code of all the instances of a node. When a node is selected on the graph model, the source code of all its instances is shown in different tab views in source code view. For example, the source code view in Figure 5 shows the source code of all the instances of a code clone set. The yellow highlight shows cloned code clone fragments.

V. Evaluation

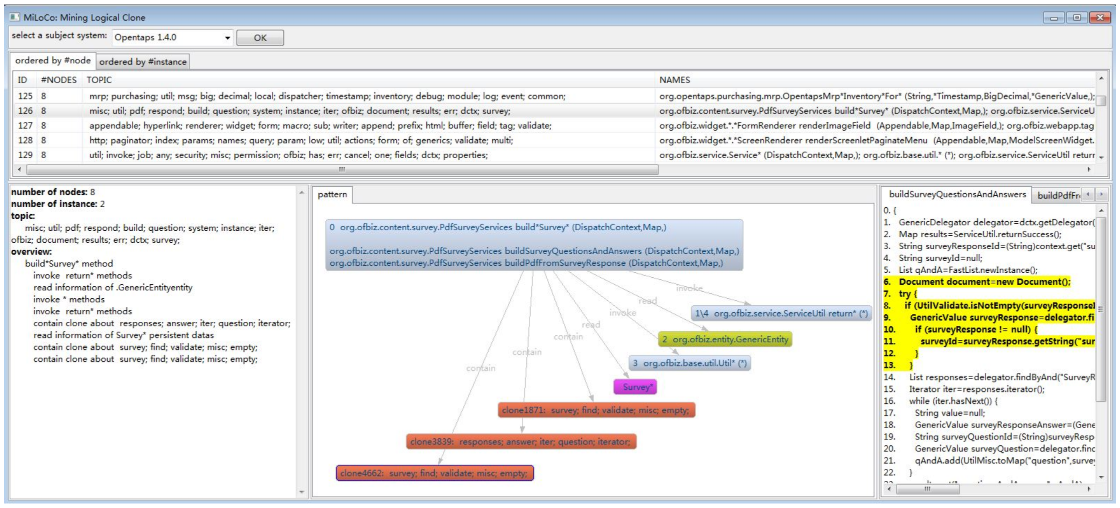
To evaluate whether the proposed approach and MiLoCo can identify meaningful logical clones that are useful for program understanding, evolution, and reuse, we conducted a case study with Opentaps [8] to investigate the following three research questions:

RQ1 What kinds of logical clones can be mined from the system? What knowledge do these logical clones represent?

RQ2 How well does MiLoCo mine high-level clones compared with existing approaches?

RQ3 How do developers evaluate mined logical clones in terms of their usefulness for program understanding, evolution and reuse?

In the following subsections, we first introduce the basic results of the case study and then present our answers to the three research questions.


MiLoCo: Mining Logical Clone

select a subject system: Opentaps 1.4.0

ordered by #node | ordered by #instance

ID	#NODES	TOPIC	NAMES
125	8	mrp; purchasing; util; msg; big; decimal; local; dispatcher; timestamp; inventory; debug; module; log; event; common;	org.opentaps.purchasing.mrp.OpentapsMrp*Inventory*For* (String,*Timestamp,BigDecimal,*GenericValue,);
126	8	misc; util; pdf; respond; build; question; system; instance; iter; ofbiz; document; results; err; dctx; survey;	org.ofbiz.content.survey.PdfSurveyServices build*Survey* (DispatchContext,Map,); org.ofbiz.service.ServiceU
127	8	appendable; hyperlink; renderer; widget; form; macro; sub; writer; append; prefix; html; buffer; field; tag; validate;	org.ofbiz.widget.*.*FormRenderer renderImageField (Appendable,Map,ImageField,); org.ofbiz.webapp.tag
128	8	http; paginator; index; params; names; query; param; low; util; actions; form; of; generics; validate; multi;	org.ofbiz.widget.*.*ScreenRenderer renderScreenletPaginateMenu (Appendable,Map,ModelScreenWidget.
129	8	util; invoke; job; any; security; misc; permission; ofbiz; has; err; cancel; one; fields; dctx; properties;	org.ofbiz.service.Service* (DispatchContext,Map,); org.ofbiz.base.util.* (*); org.ofbiz.service.ServiceUtil returr

number of nodes: 8
number of instance: 2
topic:
misc; util; pdf; respond; build; question; system; instance; iter; ofbiz; document; results; err; dctx; survey;
overview:
build*Survey* method
invoke return* methods
read information of .GenericEntityentity
invoke * methods
invoke return* methods
contain clone about responses; answer; iter; question; iterator;
read information of Survey* persistent datas
contain clone about survey; find; validate; misc; empty;
contain clone about survey; find; validate; misc; empty;

pattern
0 org.ofbiz.content.survey.PdfSurveyServices build*Survey* (DispatchContext,Map,)
org.ofbiz.content.survey.PdfSurveyServices buildSurveyQuestionsAndAnswers (DispatchContext,Map,)
org.ofbiz.content.survey.PdfSurveyServices buildPdfFromSurveyResponse (DispatchContext,Map,)
invoke
read
invoke
read
contain
contain
contain
1\4 org.ofbiz.service.ServiceUtil return* (*)
2 org.ofbiz.entity.GenericEntity
3 org.ofbiz.base.util.Util* (*)
Survey*
clone1871: survey; find; validate; misc; empty;
clone3839: responses; answer; iter; question; iterator;
clone4662: survey; find; validate; misc; empty;
buildSurveyQuestionsAndAnswers
buildPdfFr

```
{
  GenericDelegator delegator=dctx.getDelegator(
  Map results=ServiceUtil.returnSuccess();
  String surveyResponseId=(String)context.get("su
  String surveyId=null;
  List qAndA=FastList.newInstance();
  Document document=new Document();
  try {
    if (UtilValidate.isNotEmpty(surveyResponseI
      GenericValue surveyResponse=delegator.fi
        if (surveyResponse != null) {
          surveyId=surveyResponse.getString("sur
        }
    }
    List responses=delegator.findByAnd("SurveyR
    Iterator iter=responses.iterator();
    while (iter.hasNext()) {
      String value=null;
      GenericValue surveyResponseAnswer=(Gene
      String surveyQuestionId=(String)surveyResp
      GenericValue surveyQuestion=delegator.finc
      qAndA.add(UtilMisc.toMap("question",survey
    }
```

Fig. 5. MiLoCo Layout

A. Basic Results

Optentaps is a large open-source ERP and CRM system developed in Java. The subject system we used was Opentaps 1.4.0, which has 14,351 classes and interfaces, 253,743 methods, and about three million lines of code. The whole logical clone mining process, including model extraction, subgraph pattern mining and representation, took about four hours and no more than 170MB disk space with a 2.9GHz dual-core CPU and 8GB RAM.

In the case study, MiLoCo reports 1,690 logical clones with the settings of $threshold_{node} = 3$ and $threshold_{instance} = 2$, i.e., having at least three nodes and two instances. These logical clones involve 3,553 (24.8%) classes and interfaces and 14,053 (5.5%) methods of the system.

These 1,690 logical clones contain 3 to 38 nodes (11.2 on average) and 2 to 2020 instances (7.9 on average). The distribution of logical clones with different numbers of nodes and instances is shown in Figure 6 and Figure 7.

All 1,690 logical clones include 19,005 nodes in total, out of which 5,760 are functional clusters (30.3%), 955 are methods (5.0%), 7,527 are code clone sets (39.6%), 4,681 are entity classes (24.6%), 82 are persistent data objects (0.5%).

After analyzing the results, we found that a large part (1,050) of the detected logical clones have very similar structures as shown in Figure 8. These logical clones reflect the programming convention of creating new service instances: a service instance is created, and then the service user is obtained from input and set to the created service instance. Therefore, we merged these similar logical clones into one and obtained 641 logical clones for further analysis and evaluation.

B. RQ1: Categories of Mined Logical Clones

Having analyzed all 641 logical clones, we found that they can be categorized into the following four types based on abstraction level, scope and complexity of duplicated logical structures. The former two types (i.e., programming convention and design structure) reflect duplicated logical structures

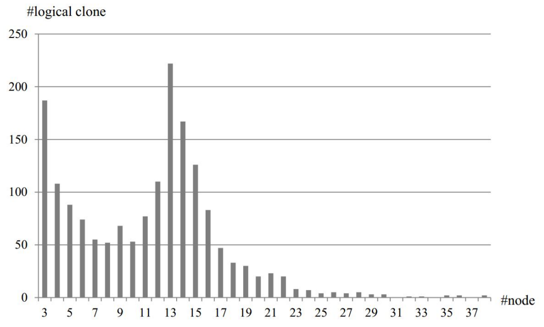


Fig. 6. Distribution of Logical Clones with Different Numbers of Nodes

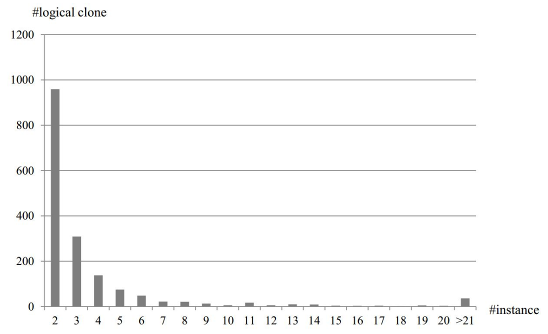


Fig. 7. Distribution of Logical Clones with Different Numbers of Instances

on technical issues. The latter two types (i.e., business task and business process) reflect duplicated logical structures on business issues. Due to the limitation of space, the examples used in this paper may only show a part of logical clone nodes.

1) Programming Convention: A programming convention conveys similar ways of using a set of related operations when implementing similar functions. It is similar to API usage pattern, but may involve multiple classes and similar methods playing the same role. An example of programming convention is shown in Figure 8. In this example, $fromInputWith*Service$ method in a $Service$ class gets a $Service$ instance as input and invokes $inputMap$ method to transfer data of the

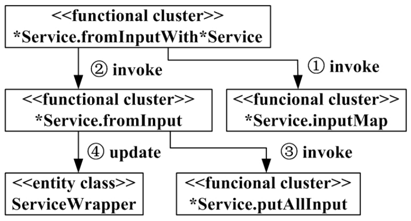


Fig. 8. An Example Logical Clone of Programming Convention

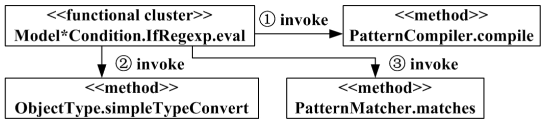


Fig. 9. An Example Logical Clone of Design Structure

instance into a Map, and then invokes $fromInput$ method to process Map, including storing the data by $putAllnput$ and updating the entity class $ServiceWrapper$. This specific logical clone conveys how to use a set of operations to process a service instance.

2) Design Structure: A design structure reflects similar interaction structures in different parts of a system. It differs from programming convention in that it involves more complex interaction structures among multiple classes. An example of design structure is shown in Figure 9. This example indicates the interaction structure among four classes, i.e., $Model * Condition$, $ObjectType$, $PatternMatcher$, $PatternCompiler$, for the purpose of evaluating a regular expression. Furthermore, logical clones representing program conventions appear more frequently than those representing design structures.

3) Business Task: A business task conveys similar ways of using a set of related operations when implementing similar business tasks. An example of business task is shown in Figure 10. This example tells how to combine a set of operations to implement the tasks of showing some information on POS (Point Of Sells) screen: it first invokes a $PosScreen.show*$ method, and this method in turn reads information from $PosScreen$ and invokes another method to $refresh$ the $PosScreen$ or $showDialog$ on $PosDialog$.

4) Business Process: A business process reflects similar business processes or sub-processes in different parts of a system. It differs from business task in that it involves multiple business tasks or steps An example of business process is shown in Figure 11. This example reflects the steps involved in POS logout or shutdown: it first reads information from $PosScreen$, then shows $PosScreen$, and finally invokes $PosTransaction.closeTx$ to close POS transaction.

The distribution of different types of logical clones is shown in Figure 12. Among all the detected logical clones, 37% are programming conventions, 24% are design structures, 23% are business tasks, 16% are business processes.

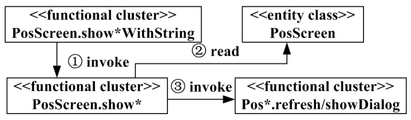


Fig. 10. An Example Logical Clone of Business Task

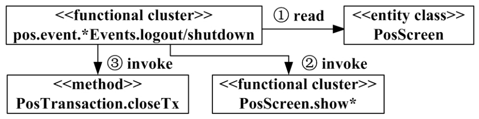


Fig. 11. An Example Logical Clone of Business Process

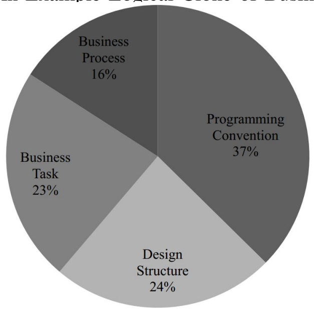


Fig. 12. Distribution of Different Types of Logical Clones

In summary, logical clones detected by our approach can reveal software maintenance knowledge spanning different levels of abstraction, including programming convention, design structure, business task, and business process.

C. RQ2: Mining High-Level Clones

To evaluate the capability of our approach in mining high-level clones, we compared the logical clones detected by MiLoCo and the structural clones detected by CloneMiner. CloneMiner [5] is a structural clone detection tool developed based on token-based simple clone detector. It can detect structural clones at the method, file, and directory levels. We compared the detected logical clones with file-level structural clones, since basic elements of both logical clones and structural clones are methods. In this comparative evaluation, we chose a package of Opentaps for analysis, which has 129 Java files and 812 methods. We configured CloneMiner with the following token threshold settings: simple clone 30, method clone 50, and file clone 50.

From the package, CloneMiner detected 15 file clones and MiLoCo detected 96 logical clones. It can be seen that MiLoCo can mine much more high-level clones, since it can use semantic clustering to identify methods that share similar topics but are not similar enough in their source code. For example, the two functional clusters of the logical clone shown in Figure 13 were not detected as method clones by CloneMiner, but they were grouped by semantic clustering. Moreover, MiLoCo can identify logical clones with various kinds of nodes distributed in different classes (files).

For the 15 file clones detected by CloneMiner, we found that 7 of them can be roughly covered by the detected logical clones. All the method clones in these file clones are included in the detected logical clones, but may distribute in different logical clones. For example, two source files $RemoveList$

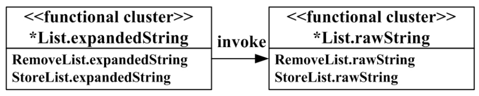


Fig. 13. Functional Clusters not Detected as Method Clones by CloneMiner

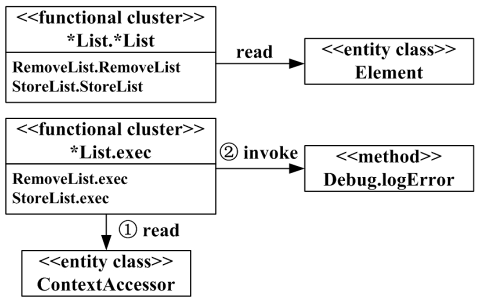


Fig. 14. Method Clones Distributed in Different Logical Clones

and $StoreList$ were detected as file clones with two method clones, i.e., constructor and $exec$ method. These two sets of similar methods distribute in two different logical clones shown in Figure 14. These logical clones together provide more information about duplicated logical structures (e.g., accessing entity classes $Element$ and $ContextAccessor$, invoking method $Debug.logError$) than the detected file clones. For the other 8 file clones, some of their method clones were not covered by the detected logical clones.

In summary, MiLoCo can detect much more high-level clones than CloneMiner, but in some cases method clones detected by CloneMiner cannot be identified by MiLoCo. Therefore, it may be helpful to integrate method clones detected by CloneMiner as another kind of duplicated elements in program models. In fact, method clones can be easily integrated into logical clone mining process, because we only need to extend our metamodel to include not only simple code clones but also method clones.

D. RQ3: Developer Evaluation

This question concerns developers' evaluation on the usefulness of the detected logical clones. To answer this question, we selected five senior graduate students from our school of computer science as the developers for survey. These students have 3 to 5 years (3.8 years on average) of experience on Java development and experience of developing enterprise software. All of them described themselves as "Java experts" and had rich experience with ERP software.

Before the survey, we gave them a two-hour tutorial about background, business, and architecture of Opentaps. We also provided them with useful documents (e.g., user manual) obtained from Opentaps website. After the tutorial, the participants had a general understanding of the requirements and design of Opentaps. We then randomly selected 100 logical clones from all the 641 logical clones and surveyed the developers' evaluation on the usefulness of these logical clones. For each logical clone, the developers were asked to answer the following two questions:

Programming understanding: Do you think this logical clone is meaningful and useful for you to understand the system?

Reuse/Evolution: Do you think it is helpful or even necessary to follow this business or programming rule when implementing similar functions or maintaining existing implementations?

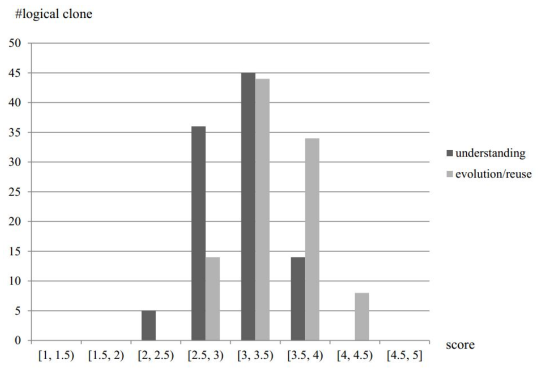


Fig. 15. Developers' Evaluation on the Logical Clones

For each question, the developers were asked to give a score from 1 to 5 (1 being useless/helpless, 3 being useful/helpful, and 5 being very useful/helpful). The survey was finished in three hours, during which the developers can check source code and documents of Opentaps to help them better understand the meaning of the logical clones.

The results of the survey are presented in Figure 15. It can be seen that most of the logical clones are thought to be useful and helpful ($score \geq 3$) for program understanding (59%) and reuse/evolution (86%).

It is interesting to note that developers' answers to the second question are much more positive than their answers to the first question. After discussing the results with them, we found this reflects the real situation. Due to the lack of domain knowledge, they cannot fully understand those logical clones in a short time. Therefore, their answers to the first question (i.e., the usefulness for program understanding) were cautious. In contrast, they were more confident that these logical clones reflect programming and business rules that need to be followed when implementing similar functions or maintaining existing implementations. Therefore, their answers to the second question (i.e., the helpfulness for reuse and evolution) were more positive.

VI. Discussion

From Section V-D, it can be seen that 41 out of the 100 logical clones under evaluation were thought to be not so meaningful for program understanding ($score < 3$). In contrast, only 14 out of the 100 logical clones were thought to be not so helpful ($score < 3$) for software evolution and reuse. According to our discussion with the participants after the survey, the main reason that they evaluated a logical clone to be not so meaningful for program understanding is that they felt it hard to fully understand what logical clones reveal about the system. In contrast, they often thought a logical clone to be helpful for software evolution and reuse even though they could not fully understand it. Their explanation was that they can recognize the value of the revealed logical structures for consistent implementations for similar topics or concerns. Therefore, they thought that when implementing new similar topics or concerns or maintaining existing implementations it is necessary to follow the rules captured by the logical clones.

The main factors that caused insufficient understanding of some of the detected logical clones include lack of domain knowledge and less intuitive representation mechanisms for logical clones. The domain knowledge required for the understanding of logical clones include the meaning of business vocabulary and the understanding about business requirements and technical framework. Due to the lack of domain knowledge, the participants may feel it hard to understand the meaning of a logical clone even when some meaningful labels and tags are available. In some cases, they can realize the meaning of a logical clone soon after we explain the meaning of some logical clone nodes to them. Therefore, we think for professional developers with necessary domain knowledge it should be easier for them to understand the detected logical clones. For the problem of insufficient representation mechanisms, the UI (user interface) of MiLoCo can be improved from several aspects, including better visualization of logical clones, more intuitive summary, and more interactive navigation.

To make full use of the benefit of logical clones for software reuse and maintenance, it is necessary to integrate MiLoCo with IDEs (integrated development environments) like Eclipse and extend MiLoCo to support the management and reuse of logical clones in addition to mining and representation. With proper management mechanism, a developer can organize logical clones and their instances in a structured way, monitor their evolution, and discover potential problems. For example, when inconsistent changes to the instances of a logical clone are made, or a newly developed functionality does not follow the logical clones involved in the implementations of similar functionalities, the developer can be notified about the problem and take necessary actions. The reuse mechanism can be combined with code completion mechanisms supported by IDEs to generate skeleton code when the IDE detects that the developer is implementing a functionality that needs to follow certain logical clones.

VII. Conclusion

In this paper, we have proposed the concept of logical clones, i.e., similar logical structures that involve similar or relevant concerns and topics but are not necessarily similar at the code level. We have presented an approach for mining logical clones. The approach first extracts from source code a program model consisting of methods, entity classes, persistent data objects, and invoke/access relations between them. To identify implicit duplications, we use semantic clustering to group methods into functional clusters and detect code clones using code clone detectors. Then the approach uses a graph mining algorithm to mine logical clones by detecting duplicated logical structures. Additional representation mechanisms are provided to help developers understand mined logical clones.

We have implemented our approach in a tool called MiLoCo to support logical clone mining and representation of logical clones. We conducted a case study with an open-source ERP and CRM system. The results show that MiLoCo can detect a rich set of logical clones spanning different levels, including programming convention, design structure, business task, and business process. A developer evaluation shows that the detected logical clones are useful for program understanding, reuse, and maintenance.

In the future work, we will conduct more case studies with industrial and open-source systems of different types to discover more representative logical clones. We will further investigate how detected logical clones can be used by developers in software maintenance tasks and develop tools to support the management and usage of logical clones.

Acknowledgment

This work is supported by National High Technology Development 863 Program of China under Grant No.2012AA011202, and by NTU SUG M4081029.020.

References

[1] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *ICSM*, 2007, pp. 519–520.
[2] R. J. J. V. Erich Gamma, Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[3] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, 2006.
[4] S. Romano, G. Scanniello, M. Risi, and C. Gravino, "Clustering and lexical information support for the recovery of design pattern in source code," in *ICSM*, 2011, pp. 500–503.
[5] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 497–514, 2009.
[6] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, 2007.
[7] "Simian." [Online]. Available: http://www.harukizaemon.com/simian/
[8] "Opentaps." [Online]. Available: http://opentaps.org/
[9] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *ICSM*, 1999, pp. 109–118.
[10] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.
[11] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM*, 1996, pp. 244–253.
[12] A. Marcus and J. Maletic, "Identification of high-level concept clones in source code," in *ASE*, 2001, pp. 107–114.
[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.
[14] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008, pp. 321–330.
[15] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in *ICSM*, 2012, pp. 275–284.
[16] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177 – 1193, 2009.
[17] M. Alalfi, J. Cordy, T. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for simulink models," in *ICSM*, 2012, pp. 295–304.
[18] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending API usage patterns," in *ECOOP*, 2009, pp. 318–343.
[19] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *ASE*, 2009, pp. 283–294.
[20] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *ASE*, 2011, pp. 456–459.
[21] G. Hamerly and C. Elkan, "Learning the k in k-means," in *NIPS*, 2003.
[22] "Weka 3: Data mining software in java." [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/
[23] "Hibernate." [Online]. Available: http://www.hibernate.org/