2014 IEEE International Conference on Software Maintenance and Evolution

# Clonepedia: Summarizing Code Clones by Common Syntactic Context for Software Maintenance

Yun Lin[1,2], Zhenchang Xing[3], Xin Peng[1,2], Yang Liu[3], Jun Sun[4], Wenyun Zhao[1,2] and Jinsong Dong[5]

[1]School of Computer Science, Fudan University, China
[2]Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China
[3]School of Computer Engineering, Nanyang Technological University, Singapore
[4]Singapore University of Technology and Design, Singapore
[5]School of Computing, National University of Singapore, Singapore

***Abstract*—Code clones have to be made explicit and be managed in software maintenance. Researchers have developed many clone detection tools to detect and analyze code clones in software systems. These tools report code clones as similar code fragments in source files. However, clone-related maintenance tasks (e.g., refactorings) often involve a group of code clones appearing in larger syntactic context (e.g., code clones in sibling classes or code clones calling similar methods). Given a list of low-level code-fragment clones, developers have to manually summarize from bottom up low-level code clones that are relevant to the syntactic context of a maintenance task. In this paper, we present a clone summarization technique to summarize code clones with respect to their common syntactic context. The clone summarization allows developers to locate and maintain code clones in a top-down manner by type hierarchy and usage dependencies. We have implemented our approach in the *Clonepedia* tool and conducted a user study on JHotDraw with 16 developers. Our results show that *Clonepedia* users can better locate and refactor code clones, compared with developers using the *CloneDetective* tool.**

## I. Introduction

Code clones are identical or similar code fragments in software systems. They are introduced by copy-paste-modify practices, implementation of similar requirements, application of design patterns, or usage patterns of APIs. Previous studies show that industrial systems have a large number of code clones [9], [17]. Although whether code clones are harmful or not is still an open research question, it is agreed on that code clones must be made explicit so that they can be consistently managed and maintained.

Many clone detection tools [1], [7], [10], [11], [12], [17] have been developed. These clone detectors fulfill the developers' quest to answer "*where code clones are in software systems?*". Clone detectors can report various types of code clones with a good precision and recall [8]. Code clones are usually reported as a list of clone sets of similar code fragments in source files. To effectively manage and maintain code clones, this low-level clone report must be examined and understood in the larger context in which code clones occur.

To help developers understand code clones, researchers have proposed visualization [23], query-based analysis [29], and categorization [13] approaches for inspecting detected code clones. These approaches help developers answer such questions as "*which parts of the system contain more code clones?*" or "*which files are most similar to this file?*". These clone analysis approaches can help developers identify clone-based reengineering opportunities [2].

Our empirical studies on code clones in several software systems (JDK, JFreeChart, JEdit, JHotDraw) show that syntactic contexts (e.g., type hierarchy and usage dependencies) in which clones occur often share high commonalities. For example, code clones of several clone sets can appear in subclasses of a specific class, or they can call different implementations of an interface. Making explicit common syntactic contexts in which several related code clones occur can facilitate developers locating and maintaining these related code clones as a whole. However, existing clone analysis tools cannot answer the developers' questions "*what code clones are relevant to a specific syntactic context, such as a type hierarchy or a usage dependency?*".

In this paper, we present an automatic clone summarization technique that 1) extract commonalities of syntactic contexts in which code clones occur as syntactic patterns and 2) cluster code clones with regard to these patterns for developers to locate and maintain code clones relevant to certain syntactic contexts in a top-down manner. Our approach first builds an ontology model that captures code clones, program elements, and their relations. Based on this ontology model, our approach then uses sequence matching and clustering techniques to cluster code clones by their common syntactic contexts. Finally, it abstracts commonalities of syntactic contexts in which code clones occur as syntactic patterns.

We have developed a proof-of-concept tool called *Clonepedia* which has been integrated with Eclipse IDE. The *Clonepedia* tool implements a Graphical User Interface (GUI) to support developers to interactively explore and analyze code clones by their syntactic patterns. We conducted a user study involving 16 developers to evaluate our approach and the tool. The results show that syntactic patterns of code clones enable developers to more efficiently locate the code clones relevant to the maintenance tasks and allow developers to make more informed refactoring decisions on relevant clones, compared with developers using the *CloneDetective* tool.

The remainder of the paper is structured as follows. Sec-

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.56

tion II presents a motivating example. Section III discusses our clone summarization approach. Section IV describes our *Clonepedia* tool. Section V reports empirical evaluation of our approach. Section VI reviews related work. Finally, we conclude and discuss our future plan.

## II. Motivating Example

Table I shows code snippets of a clone set consisting of three clone instances in JHotDraw (a Java drawing framework for developing graphics applications). Manual investigation of these cloned code fragments reveals that these code clones share three pieces of syntactic commonalities beyond apparent code fragment similarity.

1) These code clones *reside in* the methods *init()* of three classes *PertProject*, *NetProject* and *SVGProject*. These three classes share similar class names and are all subclasses of the class *AbstractProject*.
2) These code clones *call* the constructors of three different types of factory objects (line 4) *PertFactory*, *NetFactory* and *SVGFactory*. These three classes share similar class names and are all subclasses of *DefaultDOMFactory*.

In our approach, the above syntactic commonalities of clone instances of a clone set are called *intra clone set patterns*. These patterns reveal latent commonalities in the syntactic context in which several clone instances appear. It is important to note that such syntactic patterns cannot be directly noticed by just reading cloned code fragments. Developers have to explore type hierarchy and call dependencies from each clone instance, and summarize their syntactic commonalities.

In the similar vein, we can further identify latent syntactic commonalities *across* clone sets (i.e., *inter clone set patterns* in our approach) based on intra clone set patterns. For example, there are 11 more clone sets in JHotDraw whose clone instances exhibit the same syntactic commonality as the first syntactic commonality of the clone set shown in Table I. That is, clone instances of these 11 clone sets reside in in the methods (e.g. *setEditor()*, *read()*) of *PertProject*, *NetProject* and *SVGProject*, respectively. There are 9 more clone sets whose clone instances exhibit the same syntactic commonality as the second one in above list. That is, clone instances of these 9 clone sets call constructors of different types of factory objects (i.e., *PertFactory*, *NetFactory* and *SVGFactory*). Inter clone set patterns group separate clone sets in the low-level clone report based on common syntactic contexts that these clone sets share.

In addition to JHotDraw system, we also studied code clones in three other open source projects, including JDK, JFreeChart and JEdit. Our studies suggest that syntactic patterns of code clones are not unique to JHotDraw. It is a common phenomenon in all the subject systems we studied.

## III. Approach

Our approach takes as input system source code and code clones in the system reported by an existing clone detector, such as *CloneDetective* [11]. *Clonepedia* builds an ontology model that captures clone sets, clone instances, relevant program elements, and various relations among code clones and program elements. Based on this ontology model, *Clonepedia* then abstracts *Intra Clone Set Patterns* that represent commonalities of syntactic context in which several clone instances of a clone set appear. Based on intra clone set patterns, *Clonepedia* further abstracts *Inter Clone Set Patterns* that represent commonalities of syntactic context in which several clone sets appear.

### A. Building Ontology Model

To mine commonalities of syntactic contexts in which code clones occur, we first build an ontology model to capture clone sets, clone instances, program elements, and their relations. Table II shows the schema of our ontology model. The types of ontology elements include clone set, clone instance, class, interface, method, field and variable. The types of ontology relations include regular program relations (i.e., extend, implement, declare, and data type association). In addition, our ontology schema introduces four clone-specific types of relations, i.e., *contain*, *reside_in*, *diff_use*, and *common_use*. These relations describe location and usage characteristics of cloned code fragments.

*contain* and *reside_in*. A clone set *contains* two or more clone instances (i.e., cloned code fragments). Each of the clone instances *resides in* a method. The *reside_in* relations connect code clones with type hierarchy (through declaring method, declaring class, and supertypes) in which they appear.

*diff_use* and *common_use*. We employ our *MCIDiff* algorithm [18] to build *diff_use* and *common_use* relations between clone instances and program elements. *MCIDiff* is designed to detect common parts and differences across multiple clone instances. It considers the code fragments of $n$ clone instances in a clone set as $n$ token sequences. It reports a list of multisets of corresponding tokens in these clone instances. We only consider identifier-tokens for *diff_use* and *common_use* relation because we are only interested in those tokens representing program elements. For example, {*view*, *view*, *view*} and {*PertFactory*, *NetFactory*, *SVGFactory*} are two examples of the multisets of corresponding tokens that *MCIDiff* reports given the code instances shown in Table I (see line 5). If all the tokens in a multiset are identical (e.g., {*view*, *view*, *view*}), the multiset is a *matched multiset*, otherwise, the multiset is a *differential multiset* (e.g., {*PertFactory*, *NetFactory*, *SVGFactory*}). We build a *common_use* relation from the clone instance to the program element that the token in a matched multiset represents. We build a *diff_use* relation from the clone instance to the program element that the token in a differential multiset represents. *diff_use* and *common_use* relations connect code clones with program elements that code clones use.

Figure 1 depicts a partial ontology model of JHotDraw. Each box represents an ontology element while each direct edge represents a relation between a subject and an object. For example, *CloneSet1* consists of four clone instances (#1-#4). These four instances reside in the method *init()* declared

TABLE I
CODE CLONE EXAMPLE IN JHOTDRAW SYSTEM

```
0  public PertProject extends AbstractProject{
1   public void init() {
2    super.init();
3    ...
4    view.setDOMFactory(new PertFactory());
5    undo = new UndoRedoManager();
6    view.setDrawing(
7      new DefaultDrawing());
8    ...
9   }
10 }
```

```
public NetProject extends AbstractProject{
 public void init() {
  super.init();
  ...
  view.setDOMFactory(new NetFactory());
  undo = new UndoRedoManager();
  view.setDrawing(
    new DefaultDrawing());
  ...
 }
}
```

```
public SVGProject extends AbstractProject{
 public void init() {
  super.init();
  ...
  view.setDOMFactory(new SVGFactory());
  undo = new UndoRedoManager();
  view.setDrawing(
    new SVGDrawing());
  ...
 }
}
```

TABLE II
ONTOLOGY SCHEMA

| *Subject* | *Relation* | *Object* |
|---|---|---|
| clone set | contain | clone instance |
| clone instance | reside_in | method |
| clone instance | diff_use | method, field, variable, class, interface |
| clone instance | common_use | method, field, variable, class, interface |
| class | extend | class |
| class | implement | interface |
| class | declared_in | class |
| interface | extend | interface |
| method | declared_in | class/interface |
| method | has_return_type | class/interface |
| method | has_param_type | class/interface |
| field | declared_in | class/interface |
| field | has_type | class/interface |
| variable | has_type | class/interface |

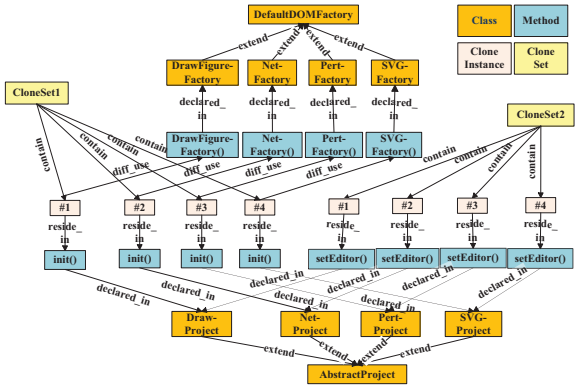


Fig. 1. Ontology Example

in four different classes, *DrawProject*, *NetProject*, *PertProject*, and *SVGProject*, respectively. These four project classes extend *AbstractProject*. Furthermore, the four clone instances in *CloneSet1 diff_use* constructor of four different classes, *DrawFigureFactory*, *NetFactory*, *PertFactory*, and *SVGFactory*. These four factory classes extend *DefaultDOMFactory*. In fact, our motivating example shown in Table I is derived from the *CloneSet1*.

### B. Mining Intra Clone Set Patterns

In our ontology model, a path is called a *clone path* if it starts from a *clone instance* node and ends at a node without any outgoing edges, for example, "*clone instance #1 of Cloneset1 reside_in init() declared_in DrawProject extend AbstractProject*". Two or more clone instances in a clone set may be located in similar clone paths, for example the four clone paths starting from the four clone instances of *CloneSet1* and ending at *AbstractProject*. We call a clone path an *abstract clone path* if some of its nodes are generalized in terms of their attributes such as name, types and etc. For example, the abstract clone path "*clones reside_in init() declared_in *Project extend AbstractProject*" indicates that code clones appear in the method *init()* of the subclasses of the *AbstractProject* class and these subclasses' names start with different words but all end with a word *Project*. We call an abstract clone path whose generalized clone instance origins from exactly one clone set as *intra clone set pattern*. *Clonepedia* first clusters these similar clone paths starting from clone instances in same clone sets. Given a cluster of similar clone paths, *Clonepedia* abstracts an abstract clone path representing commonality of syntactic context in which several clone instances appear.

*a) Clustering similar clone paths:* *Clonepedia* considers a clone path as a sequence of ontology elements and relations, and uses absolute Levenshtein distance [16] to measure distance of two clone paths in terms of a minimum cost to transform one path into another. It uses hierarchical clustering technique [19] to cluster clone paths. *Clonepedia* accepts a user-specified cost threshold. Only clone paths with distance below the cost threshold will be considered for clustering.

In our work, we are interested in similar clone paths across several different clone instances in a clone set. Therefore, we set distance of two clone paths originated from the same clone instance at $\infty$. Furthermore, as discussed in Section III-A, clone paths starting with *reside_in*, *diff_use* or *common_use* relations represent different types of connections between clone instances and syntactic context. Therefore, for two clone paths starting with different *reside_in*, *diff_use* or *common_use* relations, we set their distance at $\infty$. As such, *Clonepedia* will not attempt to cluster such clone paths.

Given two comparable clone paths, we set transformation cost between two identical nodes at 0. We set transformation cost between two identical edges at 0. That is, no transformation is needed for such nodes or edges. *Clonepedia* supports three types of sequence transformation operations, *insert*, *delete*, and *replace*. The cost of inserting or deleting a node or an edge is set at 1; the cost of replacing an edge with a node or vice versa is set at $\infty$; the cost of replacing one edge with a different-type edge (e.g., extend versus declared_in) is set at $\infty$; the cost of replacing one

TABLE III
FACTORS FOR COMPUTING REPLACE COSTS OF SAME-TYPE NODES

| *Node Type* | *Factors* |
|---|---|
| class | name, methods, fields |
| interface | name, methods, fields |
| method | name, return type name, parameters |
| parameter | name, parameter type name |
| field | name, field type name |
| variable | name, variable type name |

node with a different-type node (e.g., class versus method) is also set at $\infty$; the cost of replacing one node with a same-type node, for example, replacing class *DrawProject* with class *NetProject*, is computed by a cost function $replaceCost()$ that returns a value ranging from 0 to 1.

Given two same-type nodes, $replaceCost()$ computes their replacing cost based on different factors associated with node type (see Table III). $replaceCost()$ computes replacing costs of each factor recursively. Recursion stops when computing replacing cost of two names (i.e., strings). After obtaining replacing costs of all factors, $replaceCost()$ takes the average of these replacing costs as replacing cost of the two given nodes. For example, to compute replacing cost of two classes, $replaceCost()$ computes replacing cost of their class names, replacing cost of two sets of methods declared in the two classes, and replacing cost of two set of fields declared in the two classes. It then averages three replacing costs as the replacing cost for the two classes.

To compute replacing cost of two names (e.g., class names), *Clonepedia* splits the names into two sequences of words, using case switching, underscores, dashes as delimiters. It then computing Levenshtein distance [16] between the two word sequences. To compute replacing cost of two sets of elements (e.g. methods declared in two classes), $replaceCost()$ first computes replacing cost pairwisely between elements of the two sets. And then, it builds a bipartite graph whose disjointed nodes represent elements in the two sets respectively. The edges of bipartite graph represent replacing cost between an element of one set and an element of the other set. $replaceCost()$ then computes a minimum weight bipartite matching [25] to determine an optimal matching between two sets of elements with minimum replacing costs.

*b) Abstracting intra clone set patterns:* A cluster of similar clone paths represents a potential common syntactic context in which two or more clone instances appear. *Clonepedia* extracts this commonality by merging clone paths into an abstract clone path. Given a cluster of clone paths, the merging process adopts a progressive sequence alignment strategy. It starts with two most similar clone paths and then iteratively compares the current abstract clone path with less similar clone paths to update the abstract path. The process continues until all clone paths in a cluster have been processed.

Given two clone paths, they are merged into an abstract clone path based on their Levenshtein distance transformation results. The merging process scans the two paths from the beginning of the paths. For nodes or edges that do not need transformation, they are copied directly to the abstract path. For several consecutive nodes or edges that are inserted or deleted, a placeholder node or edge is added to the abstract path. For the two nodes that need replace transformation, for example *DrawProject* and *NetProject* in the two clone paths as shown in Figure 1, *Clonepedia* merges the two nodes using the same recursive process as defined in Table III for $replaceCost()$ computation. For example, it will first merge the name of class *DrawProject* and *NetProject*, and then recursively merge the methods and fields of the two classes. The recursive merging is to retain necessary information of the nodes to be merged, because the resulting abstract path needs to be compared for mining inter clone set patterns.

To merge two names, *Clonepedia* splits the names into a sequence of words, using case switching, underscores, dashes as delimiters. It then computes a longest common subsequence between the two word sequences. For those unmatched words, *Clonepedia* inserts "*" as placeholders to the proper places in the longest common subsequence. For the two class names *DrawProject* and *NetProject*, *Clonepedia* merges them into an abstract class name **Project*.

Given the four clone paths starting from the four clone instances of *CloneSet1* and ending at *AbstractProject* (see Figure 1), *Clonepedia* will generate an abstract clone path "*clones reside_in init() declared_in *Project extend AbstractProject*". This abstract path indicates that clone instances in *CloneSet1* appear in the method *init* of the subclasses of the *AbstractProject* class and these subclasses' names start with different words but all end with a word "*Project*". This abstract path reveals common type hierarchy in which the four clone instances appear.

### C. Mining Inter Clone Set Patterns

We call an abstract clone path whose generalized clone instance origins from more than one clone sets as *inter clone set pattern*. An inter clone set pattern presents common syntactic contexts in which several clone sets appear. We mine inter clone set patterns by 1) representing each intra clone set pattern by its abstract clone path, 2) clustering them by the similarity between abstract clone paths representing different intra clone set patterns, and 3) abstracting each cluster into a new abstract clone path representing inter clone set patterns. Therefore, mining inter clone set patterns follows the same steps for mining intra clone set patterns technically. The difference is that mining intra clone set patterns analyzes concrete clone paths of clone instances in a clone set, while mining inter clone set patterns analyzes abstract clone paths of several clone sets.

Take *CloneSet1* and *CloneSet2* in Figure 1 as an example. The *CloneSet1* has an abstract clone path "*clones reside_in init() declared_in *Project extend AbstractProject*", while *CloneSet2* has an abstract clone path "*clones reside_in setEditor() declared_in *Project extend AbstractProject*". These two abstract clone paths can be further clustered together from which a more general syntactic pattern can be abstracted, i.e., "*clones reside_in *() declared_in *Project extend AbstractProject*". This more general pattern indicates that clones appear in several methods of the subclasses of the *AbstractProject* class.

We call such patterns that several clone sets share *inter clone set patterns*. The inter clone set patterns group together several related clone sets based on their common syntactic contexts.

## IV. Tool Support

We have implemented our approach in the *Clonepedia* tool which has been integrated with Eclipse development environment. Figure 2 shows a screenshot of our tool. Currently *Clonepedia* supports three views for the developer to explore and analyze code clones in a top-down manner through syntactic patterns of code clones.

*Syntactic Pattern view* allows developers to explore syntactic patterns that *Clonepedia* abstracts in type hierarchies. Syntactic patterns are organized in three categories, i.e., *reside_in*, *diff usage*, and *common usage*. Developers can sort syntactic patterns based on different criteria (e.g., the number of clone sets sharing a syntactic pattern). They can also search patterns by names. Double-clicking a syntactic pattern opens a tab in the detail pane of the *Syntactic Pattern view* showing a summary of the selected pattern and the relevant clone sets. Developers can select a pattern to view the clone sets sharing this pattern in *Clone Set view*.

*Clone Set view* shows a table of selected clone sets. Double-clicking a clone set opens a tab in the lower pane of the *Clone Set view* that provides descriptions about different aspects of a clone set, as well as links to clone instances and intra/inter clone set patterns. Clicking a clone instance link opens the clone instance in a Java editor. Clicking an intra/inter clone set pattern reveals and shows the selected pattern in *Syntactic Pattern view*.

Double-clicking a clone set also shows *Code Snippet view*. *Code Snippet view* shows the *MCIDiff*'s differencing results of the clone instances of the selected clone set. Differences across the clone instances are highlighted in different colors and fonts [18]. This allows developers to see and examine differences across multiple clone instances as a whole.

## V. Evaluation

Our approach to clone summarization aims to help developers better locate and maintain code clones in software maintenance tasks. To evaluate if our *Clonepedia* tool achieves this goal, we conducted a user study to investigate the following three questions:

- **Q1:** How do developers use syntactic patterns when they are available?
- **Q2:** Can the *Clonepedia* tool help developers locate relevant code clones in software maintenance tasks more easily?
- **Q3:** Can the *Clonepedia* tool help developers better determine what to do with code clones in refactoring tasks?

### A. Experiment Design

We first introduce our experiment design, including baseline clone analysis tool, subject system, participants, maintenance tasks, and ground-truth answers.

*1) Baseline clone analysis tool:* In this study, we compared our *Clonepedia* tool with *CloneDetective* [11]. We chose *CloneDetective* as the baseline clone analysis tool for three reasons. First, *CloneDetective* has also been integrated with Eclipse. This allows us to make a fair comparison between *Clonepedia* and *CloneDetective* in the same development environment. Second, *CloneDetective* can detect Type I, Type II and Type III clones. This allows us to evaluate the effectiveness of *Clonepedia* with different types of code clones. To make a fair comparison, *Clonepedia* takes as input clone sets reported by *CloneDetective*. Third, *CloneDetective* is not just a clone detector. It is also equipped with a Graphical User Interface, including package explorer, keyword-based file search, AspectJ-based visualization of code clones across multiple files, and pairwise code comparison and difference highlighting. These features represent state-of-the-practice for exploring and analyzing code clones.

*2) Subject system:* We used JHotDraw version 7.0.6 as the subject system. This version contains 368 classes, 3340 methods, and 32435 lines of code. *CloneDetective* reports 334 clone sets in JHotDraw, among which 184 clone sets contain two clone instances, 140 contain 3-6 clone instances, the rest 10 contain more than 7 clone instances. These code clones are scattered in 499 methods of 164 classes.

*3) Participants:* We recruited 16 participants (15 graduate students and 1 senior undergraduate student) from the School of Computer Science in Fudan University. Based on our pre-experiment survey, two participants worked as professional developers in industry before they entered the graduate program; Six students participated in the development of industrial projects (e.g., internships) during their study. All participants used Eclipse regularly. Six students described themselves as above-average Java expertise and two described themselves as Java experts. None of 16 participants had experience with the subject system JHotDraw.

We used between-subjects design in this study. The 16 participants were matched in pairs based on their programming experience and capability and then were randomly allocated to the experimental group or control group. Group $G_1$ was the experimental group using *Clonepedia*. Group $G_2$ was the control group using *CloneDetective*.

*4) Procedure:* The participants were asked to complete a set of clone-related maintenance tasks described below. We gave a tutorial of *Clonepedia* and *CloneDetective* to the two groups three hours before the experiment and asked the participants to familiarize themselves with important concepts that the tools introduce as well as tool features. Participants were required to run a full-screen recorder while they were working on the tasks. The recorded task videos allows us to time the completion of the tasks and also observe and analyze the participants' behaviors in detail.

*5) Maintenance tasks:* We designed a maintenance scenario to simulate the maintenance of code clones reported in our industrial study [28]. This scenario consists of a set of clone location and refactoring tasks on JHotDraw.

**Maintenance scenario**: *You are a new team member of*

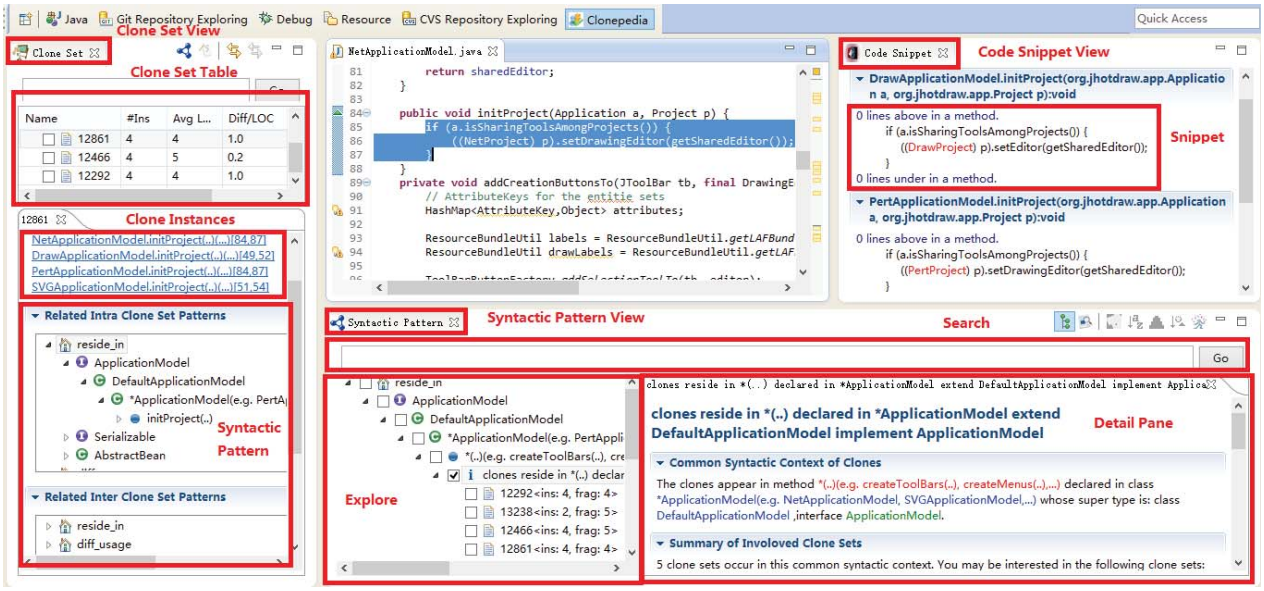


Fig. 2. The Screenshot of Clonepedia

*JHotDraw project. You take over John's code who already left the JHotDraw project. John's change logs indicate that he did copy-paste-modify many times in his work, for example, when he developed application-model classes (i.e., subclasses of the class DefaultApplicationModel), and DOM-related features (i.e., code using implementations of the DOMInput interface). Your manager asks you to examine code clones in John's code and determine whether they can be refactored and how.*

**Clone Location Task**: Participants were asked to locate code clones based on the above hints gathered from John's change logs. These clone location tasks examine how fast and accurate a participant can locate relevant code clones he is interested in. Participants were asked to complete task 1.1 within 10 minutes and task 1.2 within 15 minutes. They were asked to write down the IDs of clone sets (clone data is generated by *CloneDetective* and shared by both tools) that they deemed relevant to these tasks.

- **Task 1.1** *Please locate code clones in subclasses of the class DefaultApplicationModel.*
- **Task 1.2** *Please locate code clones using different implementations of the DOMInput interface.*

**Refactoring tasks**: Participants were given seven clone sets identified in the ground truth of clone location tasks (see Section V-A6). These seven clone sets differ in the number of clone instances and their complexity (see Table IX). The complexity of these clone sets was determined by experts based on whether potential refactorings require mainly localized information inside clones or more global information beyond clones (e.g, type hierarchy or usage dependency information). Three clone sets were considerd as high complexity (H), three as medium (M), and one as low (L).

There was no time limitation for participants to complete refactoring tasks. They were asked to decide whether the given clone sets can be refactored, if so what refactorings can be applied, and what has to be done (e.g., reconcile naming inconsistencies) before clones can be refactored. Participants were asked to write down their refactoring decisions and why they reached such decisions as detailed as possible.

TABLE IV
SCORING CRITERIA FOR REFACTORING REASONS AND DETAILS

| *Score* | *Criteria* |
|---|---|
| 0 | Irrelevant, wrong or non-sensible reason |
| 1 | General or vague reason |
| 2 | Partly specific and right reason |
| 3 | Specific and right reason |

*6) Ground-truth answers:* To evaluate the performance of participants, we need to build ground-truth answers for the tasks. To that end, we invited two experts to complete the above clone location tasks and refactoring tasks. One expert is a Ph.D student in our lab who has used JHotDraw in his work for about one year. The other expert is a senior software architect from Alcatel who has nine years industrial experience and is an expert on software clones and related maintenance issues. The experts did not have time limits for completing the tasks. After they completed the tasks, the two experts together examined and combined their task results to produce ground-truth answers for the tasks.

The experts also documented the reasons for their refactoring decisions and what has to be done before clones can be refactored. After the experiment, we asked the two experts to mark the participants' task results. We asked them to check two items: proposed refactorings and supporting reasons (i.e., refactoring reasons), and things to do before applying the proposed refactorings (i.e., refactoring details). For each item, a score from 0 to 3 (see Table IV) will be given to quantify the quality of refactoring decisions. To avoid experimenter expectancy effects, the two experts did not know which group a participant belongs when they graded the participant's task results.

TABLE V
USAGE STATISTICS OF *Clonepedia* AND ECLIPSE FEATURES

| Task Type | Task | Syntactic Pattern View | | | Clone Set View | | | Code Snippet View | Eclipse Feature | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | search | explore | detail pane | clone set table | clone insts | synt pattern | snippet | editor | type hierarchy | search |
| Clone Location | Task 1.1 | 1.00 | 3.13 | 1.00 | 8.50 | 9.63 | 0.000 | 3.00 | 7.00 | 1.00 | 0.00 |
| | Task 1.2 | 1.00 | 5.63 | 2.25 | 11.5 | 12.00 | 1.13 | 9.75 | 11.13 | 2.00 | 0.38 |
| Refactoring | #1 (H) | 0.00 | 0.25 | 0.13 | 1.63 | 3.88 | 1.75 | 3.13 | 3.00 | 0.38 | 0.00 |
| | #2 (M) | 0.00 | 0.00 | 0.00 | 1.25 | 2.00 | 1.13 | 2.75 | 2.38 | 0.00 | 0.00 |
| | #3 (L) | 0.00 | 0.00 | 0.13 | 1.38 | 1.38 | 0.50 | 1.38 | 1.25 | 0.00 | 0.00 |
| | #4 (H) | 0.00 | 0.00 | 0.13 | 1.63 | 2.00 | 1.25 | 2.25 | 2.00 | 0.38 | 0.00 |
| | #5 (M) | 0.13 | 0.13 | 0.00 | 2.50 | 2.00 | 1.25 | 3.25 | 3.00 | 0.13 | 0.00 |
| | #6 (H) | 0.00 | 0.13 | 0.26 | 3.25 | 2.88 | 0.50 | 2.75 | 1.88 | 0.00 | 0.00 |
| | #7 (M) | 0.00 | 0.00 | 0.13 | 4.88 | 1.75 | 0.63 | 2.75 | 1.88 | 0.00 | 0.00 |

### *B. Results: Using Clone Summarization (Q1)*

By analyzing screen-recorded task videos, we obtained the average number of times that the *Clonepedia* users used *Clonepedia* features (see Section IV) and Eclipse features (such as Java Editor, Type Hierarchy, Text Search) during their tasks. Table V summarizes our analysis results. We observed that participants used *Clonepedia* in a different way when tasks vary in nature and complexity.

*1) Clone location tasks: Syntactic Pattern view* was frequently used to locate clones. Our video analysis shows that participants usually started their clone location tasks by searching relevant syntactic patterns with keywords in task descriptions. This allows them to quickly focus on a small number of patterns that may be relevant to their tasks. *Clonepedia* presents syntactic patterns using a concise label, but this pattern label seems not sufficient for clearly understanding what a syntactic pattern means. This can be partially due to the learning curve of the concept of syntactic pattern. As a result, the participants had to check the details of syntactic patterns in the detail pane to understand syntactic patterns.

Once the participants found a plausible pattern, they usually inspected relevant clone sets in the clone set table of *Clone Set view* and explored the clone instances of the clone sets. Furthermore, they also frequently used *Code Snippet view* to investigate common parts and differences of the clone instances. They also frequently used Eclipse features (especially Java Editor) to investigate and validate the relevance of code clones (e.g., reading the code).

*2) Clone refactoring tasks:* The usage statistics of *Clonepedia* and Eclipse features in refactoring tasks show distinct characteristics than those in the clone location tasks.

Table V shows that when the participants investigated a given clone set they mainly used *Clone Set view* and Java editor. Furthermore, "synt pattern column" shows that syntactic patterns provided in the lower pane of *Clone Set view* were inspected more frequently than syntactic patterns in *Syntactic Pattern view*. This is because the participants' goal in refactoring task is to determine what they can do with a specific clone set. Therefore, they are more interested in the patterns relevant to the clone set under refactoring.

Our results also show that the complexity of a given clone set affects the usage of *Clonepedia* features. The complexity of a clone set is indicated in H/M/L (see Table IX for details). If program structures of code clones are simple, e.g., all cloned code fragments are the same and located in the same class, most participants did not further investigate syntactic patterns, for example in Task#3. However, when program structures of code clones become complicated (e.g., in Task#1), participants began resorting to syntactic patterns to better understand the syntactic context in which clones appear. That is why the frequencies of investigating syntactic patterns in *Clone Set view* vary.

### *C. Results: Locating Relevant Clones More Easily (Q2)*

We evaluated the participants' performance in clone location tasks by comparing precision, recall, F-measure and completion time of the experimental group ($G_1$) and control group ($G_2$). Given a clone location task, let $S_a$ be the set of relevant clone sets in our ground-truth answer. Let $S_r$ be the set of clone sets reported by a participant. Precision, $P$, represents the percentage of correctly reported clone sets, i.e., $P = |S_r \bigcap S_a|/|S_r|$. Recall, $R$, is the percentage of relevant clone sets reported, i.e., $R = |S_r \bigcap S_a|/|S_a|$. The F-measure is computed as $F = (1+b^2)/(1/P+b^2/R)$ to reflect a weighted average of the precision and recall. In our study, we set $b$ to 2, which means recall is considered four times as important as precision. That is, we consider that finding missing relevant clone sets is more difficult than removing irrelevant clone sets.

We introduced the following null and alternative hypotheses to evaluate the difference of the two groups' performance.

- **H0:** The primary null hypothesis is that there is no significant difference between the performance of the two groups.
- **H1:** An alternative hypothesis to H0 is that there is significant difference between the performance of the two groups.

Table VI and Table VII present the participants' performance (in terms of each participant's average precision, recall, F-measure, and completion time) in the two clone location tasks. Overall, the *Clonepedia* users achieved better precision, recall, and F-measure in these two tasks. The *Clonepedia* users completed the tasks faster than Clone-Detective users.

We used Wilcoxon's matched-pairs signed-ranked tests to evaluate the null hypothesis H0 in terms of precision, recall, F-measure, and task completion time at a 0.05 level of significance. Table VIII shows the results of these four tests. Based on the results we reject the null hypothesis H0 for all the performance metrics, i.e., Recall, Precision, F-measure, and task completion time. Therefore, we accept the alternative hypothesis H1 for all the performance metrics, i.e., there is

TABLE VI
PERFORMANCE OF *Clonepedia* GROUP

| Participant | Precision | Recall | F-Measure | Time(s) |
|---|---|---|---|---|
| P1 | 1.000 | 1.000 | 1.000 | 329.000 |
| P2 | 1.000 | 0.909 | 0.926 | 405.000 |
| P3 | 0.938 | 1.000 | 0.987 | 447.500 |
| P4 | 1.000 | 1.000 | 1.000 | 229.000 |
| P5 | 1.000 | 0.955 | 0.963 | 270.000 |
| P6 | 1.000 | 1.000 | 1.000 | 285.000 |
| P7 | 1.000 | 0.883 | 0.904 | 425.000 |
| P8 | 1.000 | 0.701 | 0.746 | 276.500 |
| Average | 0.992 | 0.931 | 0.941 | 370.875 |
| Std.Dev. | 0.021 | 0.097 | 0.081 | 107.051 |

TABLE VII
PERFORMANCE OF *CloneDetective* GROUP

| Participant | Precision | Recall | F-Measure | Time(s) |
|---|---|---|---|---|
| P9 | 0.928 | 0.416 | 0.467 | 546.500 |
| P10 | 1.000 | 0.603 | 0.655 | 611.000 |
| P11 | 0.875 | 0.487 | 0.534 | 736.000 |
| P12 | 0.333 | 0.364 | 0.357 | 738.000 |
| P13 | 1.000 | 0.532 | 0.587 | 463.000 |
| P14 | 1.000 | 0.377 | 0.430 | 640.500 |
| P15 | 0.600 | 0.325 | 0.357 | 592.500 |
| P16 | 0.300 | 0.273 | 0.277 | 696.000 |
| Average | 0.755 | 0.422 | 0.458 | 627.938 |
| Std.Dev. | 0.281 | 0.104 | 0.120 | 89.415 |

significant difference between the precision, recall F-measure, and task completion time of the *Clonepedia* users and the *CloneDetective* users in the two clone location tasks.

As the *Clonepedia* users achieved better precision, recall, F-measure and completed the tasks in shorter time than the *CloneDetective* users, our results show that the *Clonepedia* users achieved significant better performance than the *CloneDetective* users in the two clone location tasks.

### *D. Results: Better Refactoring Decision (Q3)*

We evaluated the participants' performance in seven refactoring tasks using refactoring reason scores and refactoring details scores. Table IX summarizes the results. The first row shows task id. Each task involves one clone set. The second row shows the number of clone instances in the corresponding clone set. The third and forth rows show ground-truth answers and task complexity judged by experts. "Y" indicates clones can be refactored, while "N" indicates clones cannot be refactored. "H", "M" and "L" represents high, medium and low complexity. The fifth and sixth rows summarize the number of participants that made correct decisions on whether clones can be refactored. The seventh to tenth rows summarize average refactoring reason and details scores of the two study groups.

Regarding whether clones can be refactored or not, most of the participants made correct decisions in 5 out of 7 tasks, except for *CloneDetective* group in Task#2 and both groups in Task#4. Our analysis of the *CloneDetective* users' Task#2 videos did not suggest a clear reason for their worse performance in this task. It seems that the performance of *CloneDetective* users certain randomness, because they often did not fully explore information needed for making a rational refactoring decision. In fact, a challenge for *CloneDetective* users was that they often did not know what they did not know. As a result, their decisions may depend on their intuition. This is also evident in their lower scores for refactoring reasons and details. Intuition-based decisions may result in their worse performance in Task#2.

TABLE IX
EXPERIMENTAL RESULTS FOR CLONE REFACTORING TASKS

| | ***TaskID*** | #1 | #2 | #3 | #4 | #5 | #6 | #7 |
|---|---|---|---|---|---|---|---|---|
| | Ins.No | 4 | 4 | 3 | 6 | 4 | 6 | 3 |
| | Refact | Y | Y | Y | N | Y | Y | Y |
| | Comp | H | M | L | H | M | H | M |
| Refactor or not | $G_1$ | 7 | 7 | 8 | 0 | 7 | 8 | 6 |
| | $G_2$ | 6 | 4 | 7 | 2 | 5 | 8 | 7 |
| Refactoring reason | $G_1$ | 2.50 | 2.38 | 2.88 | 0.00 | 2.25 | 2.00 | 2.63 |
| | $G_2$ | 1.13 | 1.38 | 2.38 | 0.38 | 1.00 | 1.63 | 2.00 |
| Refactoring details | $G_1$ | 2.25 | 2.13 | 2.38 | 0.00 | 2.00 | 1.63 | 2.25 |
| | $G_2$ | 1.38 | 1.63 | 2.00 | 0.25 | 1.38 | 0.63 | 1.63 |

Task#4 involves 6 clone instances. The cloned code fragments use different fields (e.g., *rectangle, ellipse, roundrect*) in the same program context. The type of these fields are inner class *Double* declared in different classes such as *Rectangle2D*, *Ellipse2D*. *Clonepedia* summarized this commonality as a diff usage pattern. This pattern makes the *Clonepedia* users propose to replace separate *Double* with one class for reducing code duplication. However, these *Double* classes are JDK classes that cannot be modified by the JHotDraw users. We will enhance the implementation of *Clonepedia* to make it explicit whether code elements are from source code or from imported libraries. For *CloneDetective* users, the high usage complexity results in their worse performance, even though they spent doubled time on the task.

For the rest five refactoring tasks, refactoring reason and details scores of the two groups are clearly different. The *CloneDetective* users usually provided vague or general explanations for their decisions, for example, "clones can be refactored because the code is very similar" or "clones cannot be refactored because refactoring them seems too troublesome". In contrast, the *Clonepedia* users can often offer more specific reasons and details for their refactoring decisions. This is because *Clonepedia* summarizes relevant syntactic patterns which provide concrete "hints" leading its users to further investigation of relevant program elements and structures.

Finally, the score gap between the two groups varies by task complexity. For simpler tasks (e.g., Task#3 involving small number of clone instances from a single file), the score gap is small. However, for more complex tasks the *Clonepedia* users exhibited better performance than the *CloneDetective* users by providing higher-quality reasons and details within shorter time (e.g., in Task#6). Such a phenomenon implies that complexity of refactoring task can be reduced by syntactic patterns provided by *Clonepedia*.

### *E. Discussion*

*Clonepedia* allows developers to locate and maintain code clones in a way different from existing clone analysis tools. We discussed insights we obtained in our user study.

*1) Top-down exploration of code clones: CloneDetective* provides rich features for analyzing code clones. However, *CloneDetective* provides little help in locating code clones relevant to a syntactic context. Our analysis of task videos reveals that the *CloneDetective* users had to manually collect necessary syntactic information by intensively using Eclipse IDE features (e.g., Type Hierarchy, Call Hierarchy). They then

TABLE VIII
RESULTS OF WILCOXON'S TEST OF HYPOTHESES, FOR THE VARIABLE RECALL, PRECISION, F-MEASURE AND COMPLETION TIME. MEASUREMENTS ARE REPORTED IN THE FOLLOWING COLUMNS: MINIMUM VALUE, MAXIMUM VALUE, MEDIAN, MEANS ($\mu$), VARIANCE ($\sigma^2$), DEGREES OF FREEDOM ($DF$), PEARSON CORRELATION COEFFICIENT ($PC$), Z STATISTICS ($Z$) AND STATISTICAL SIGNIFICANCE ($p$)

| H | Var | Approach | Samples | Min | Max | Median | $\mu$ | $\sigma^2$ | $DF$ | $PC$ | $Z$ | $p$ | Decision |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H0 | Precision | *Clonepedia* | 8 | 0.938 | 1.000 | 1.000 | 0.992 | 4.272E-04 | 7 | -0.147 | -2.203 | 0.043 | Reject |
| | | *CloneDetective* | 8 | 0.300 | 1.000 | 0.901 | 0.776 | 0.064 | 7 | | | | |
| | Recall | *Clonepedia* | 8 | 0.701 | 1.000 | 0.977 | 0.931 | 0.009 | 7 | 0.560 | -2.521 | 0.012 | Reject |
| | | *CloneDetective* | 8 | 0.273 | 0.604 | 0.451 | 0.446 | 0.012 | 7 | | | | |
| | F-measure | *Clonepedia* | 8 | 0.746 | 1.000 | 0.975 | 0.941 | 0.007 | 7 | 0.615 | -2.521 | 0.012 | Reject |
| | | *CloneDetective* | 8 | 0.278 | 0.656 | 0.501 | 0.482 | 0.014 | 7 | | | | |
| | Time | *Clonepedia* | 8 | 229.000 | 447.500 | 307.000 | 333.375 | 5881.797 | 7 | 0.049 | -2.521 | 0.012 | Reject |
| | | *CloneDetective* | 8 | 463.000 | 738.000 | 625.500 | 627.813 | 7998.00 | 7 | | | | |

had to use the collected syntactic information to search for code clones potentially relevant to the syntactic context under investigation. This analysis process requires many context switchings in order to interrelate scattered information. As such, the performance of *CloneDetective* users became sensitive to the complexity of tasks. Their performance degraded as task complexity increases.

*Clonepedia* enables a different analysis process by categorizing and presenting code clones in terms of their syntactic patterns. The *Clonepedia* users usually started with the syntactic patterns relevant to the maintenance tasks, and then zoomed into a set of potentially relevant clone sets or instances. These clone sets often cross-cut several source files.

Our results show that the *Clonepedia* users still needed to use Eclipse IDE features (e.g., type hierarchy). However, they used these IDE features to further investigate or confirm patterns that *Clonepedia* summarizes, or to further confirm the relevance of code clones under investigation. This is completely different from the use of Eclipse IDE features by the *CloneDetective* users. The *CloneDetective* users had to resort to these IDE features in order to gather syntactic information for exploring code clones. With *Clonepedia*, such syntactic information has been automatically summarized and is readily available for use. As such, *Clonepedia* users can locate relevant code clones in a more organized way, which resulted in stable precision, recall and task completion time in different tasks.

*2) Informed maintenance decisions on code clones:* In addition to support top-down exploration of code clones, *Clonepedia* also provides three levels of information for developers to make informed decisions (e.g., refactoring strategies) in clone-related maintenance tasks.

At *system level*, inter clone set patterns group related clone sets by their common syntactic contexts. They provide developers with a good overview about related clone sets, in terms of how they are present in different type hierarchies and how they use similar or different program elements.

At *clone set level*, intra clone set patterns summarize common syntactic contexts in which clone instances occur. These patterns remind developers of investigating important syntactic information so that they can make more informed maintenance decisions, for example whether to apply refactoring at method level or class level.

At *clone instance level*, differences across multiple clone instances of a clone set are detected and highlighted using our clone differening algorithm *MCIDiff* [18]. These differences help developers understand variation points across clone instances. Understanding these variation points can lead to more detailed decisions on concrete refactorings.

### F. Threats to Validity

In this user study, there are four major threats. First, the differences in the capabilities of the two groups of participants may threaten our assumption of the "equivalence" of two groups. To address this threat, we had tried our best to allocate participants with comparable capabilities into different groups based on our pre-study survey and evaluation. Second, refactoring decision scores are subject to human bias. To reduce this bias, we asked two experts to cross-check their scores and reach consensus. To avoid experimenter expectancy effects, the two experts blindly scored the results of the study groups. Third, both groups may have insufficient training of the tools used in the study. We believe this factor has much greater impact on *Clonepedia* users due to syntactic pattern concept introduced and more complex UI. Even in such a disadvantage situation, *Clonepedia* users still significantly outperformed those using *CloneDetective*. Forth, code clones used in this study are from one Java software and may not represent all possible cases in real-world clone-related maintenance tasks. Further studies with more clones, tasks, and subject systems are required to generalize our findings.

## VI. RELATED WORK

*Clone Detection*: Researchers have proposed many clone detection tools to detect code clones in software system. Baker et al. [1] proposed to detect code clones using string matching algorithm. Kamiya et al. developed CCFinder [12] and Juergens et al. [11] developed *CloneDetective*. Both tools support token-based clone detection. Mayrand et al. [20] proposed code metrics based detection technique. CloneDR [7] analyze and compare Abstract Syntax Tree (AST) for clone detection. Komondoor and Horwitz [15] developed a technique to detect code clones by Program Dependency Graph (PDG) isomorphism. DECKARD [10] detects clones using locality sensitive indexing. A comprehensive survey of code clone research can be found in [24]. Our approach does not make specific assumption about clone detectors. It can work with various types of clones reported by different clone detectors.

*Code Clone Analysis and Management*: Balazinska et al. [2] proposed an approach to measure reengineering opportunities of cloned code by combining difference parts of code clones

with AST. Based on this work, they proposed an approach to automatically refactor code clones by applying design patterns [3][4]. Kapser et al. [13] proposed to filter false positive clones reported by clone detectors by categorizing code clones. Their work does not analyze detailed syntactic contexts in which clone occur. Basit and Jarzabek [5][6] proposed a data mining approach to cluster simple token-based code clones by applying frequent item set mining. Their approach does not examine syntactic context of code clones. Our recent work [21] used graph mining technique to discover logical clones that represent common business and programming rules in a software system. Different from these approaches, *Clonepedia* mines common syntactic context in which code clones occur.

*Data Summarization*: Faced with large amount of software data, researchers have proposed summarization techniques to transform raw data into a more concise form. Miryung and Notkin [14] proposed to infer systematic structural changes as logic rules. Rastkar et al. [22] presented a technique to summarize crosscutting concerns in software system when developers search code base. Sridhara et al [26][27] proposed to summarize high-level action of a Java method by investigating its signature and method body. Different from existing work, *Clonepedia* summarizes a large amount of code clones by abstracting their common syntactic contexts and categorizing code clones by their syntactic patterns.

## VII. Conclusions and Future Work

This paper presented a clone summarization approach and the tool *Clonepedia*. *Clonepedia* abstracts syntactic patterns of code clones. The patterns allow developers to locate and maintain code clones relevant to certain syntactic context. Compared with state-of-the-practice, syntactic patterns of code clones enable a different analysis process of code clones in clone-related maintenance tasks. Our user study shows that the *Clonepedia* users were able to effectively use syntactic patterns to locate and refactor code clones with higher quality and shorter time, compared with the *CloneDetective* users.

*Clonepedia* provides a solid infrastructure for our research on practical use of code clones in software maintenance. Based on *Clonepedia*'s summarization, we will investigate several practical applications of code clones, including on-the-fly clone detection and refactoring recommendation, simultaneous editing of multiple clone instances, clone-based code generation and auto-completion.

## VIII. Acknowledge

This work is supported by National High Technology Development 863 Program of China under Grant No.2012AA011202 and National Natural Science Foundation of China under Grant No.61370079.

## References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, 1995.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, pages 292–303, 1999.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, pages 326–336, 1999.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE*, pages 98–108, 2000.

[5] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE*, pages 156–165, 2005.

[6] H. A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Trans. Softw. Eng.*, 35(4):497–514, 2009.

[7] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[11] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Clonedetective – a workbench for clone detection research. In *ICSE*, pages 603–606, 2009.

[12] 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.*, 28(7):654–670, 2002.

[13] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE*, pages 85–94, 2004.

[14] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[15] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[16] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[18] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting differences across multiple instances of code clones. In *ICSE*, pages 164–174, 2014.

[19] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.*, 33(11):759–780, 2007.

[20] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–253, 1996.

[21] W. Qian, X. Peng, Z. Xing, S. Jarzabek, and W. Zhao. Mining logical clones in software: Revealing high-level business and programming rules. In *ICSM*, pages 40–49, 2013.

[22] S. Rastkar, G. C. Murphy, and A. W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *ICSM*, pages 103–112, 2011.

[23] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.

[24] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University, Canada, 2007.

[25] P. Sankowski. Maximum weight bipartite matching in matrix multiplication time. *IEEE Trans. Service Comput.*, 410(44):4480–4488, 2009.

[26] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE*, pages 43–52, 2010.

[27] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, pages 101–110, 2011.

[28] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *ICSM*, pages 285–294, 2012.

[29] Y. Zhang, H. Basit, S. Jarzabek, D. Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. In *ICSM*, pages 376–385, 2008.