Mining Implicit Design Templates for Actionable Code Reuse

Yun Lin*, Guozhu Meng†, Yinxing Xue†, Zhenchang Xing‡, Jun Sun§,
Yang Liu†, Wenyun Zhao¶ and Jinsong Dong*‖
*National University of Singapore, Singapore, †Nanyang Technological University, Singapore, ‡Australia National University, Australia, §Singapore University of Technology and Design, Singapore, ¶Fudan University, China, ‖Griffith University, Australia

***Abstract*—In this paper, we propose an approach to detecting project-specific recurring designs in code base and abstracting them into design templates as reuse opportunities. The mined templates allow programmers to make further customization for generating new code. The generated code involves the code skeleton of recurring design as well as the semi-implemented code bodies annotated with comments to remind programmers of necessary modification. We implemented our approach as an Eclipse plugin called *MICoDe*. We evaluated our approach with a reuse simulation experiment and a user study involving 16 participants. The results of our simulation experiment on 10 open source Java projects show that, to create a new similar feature with a design template, (1) on average 69% of the elements in the template can be reused and (2) on average 60% code of the new feature can be adopted from the template. Our user study further shows that, compared to the participants adopting the copy-paste-modify strategy, the ones using *MICoDe* are more effective to understand a big design picture and more efficient to accomplish the code reuse task.**

I. Introduction

Programmers usually adopt copy-paste-modify practice when implementing similar features [1]. Once a feature involves several related classes, methods or fields, such practice causes more a design duplication than several pieces of duplicated code. Empirical studies have shown that design duplication frequently occurs among classes, packages, and subsystems [2].

Recurring (or duplicated) designs usually indicate project-specific programming conventions and future reuse opportunities. Programmers can efficiently implement a new similar feature if they are aware of a general skeleton of related recurring designs. However, such recurring designs are usually implicit and undocumented, and the programmers usually have to reinvent the wheel without knowing the potential reuse opportunities [1]. Moreover, even if the programmers discover such reuse opportunities, copy-paste-modify practice is hardly a sound and systematic approach for reuse [3].

Many approaches [4], [5], [6] have been proposed to extract recurring designs based on code clone. Basit et al. [4], [5] first proposed to extract *structural clone* as recurring design. Structural clone is extracted from simple cloned code with frequent item set mining technique [7]. They are reported as a set of files/modules sharing multiple clone sets[1]. Following their work, Qian et al. [6] proposed a technique to detect *logic clone*, which aims to capture similar business logics by code clone. Their logic clone is described as a graph in which a node represents a set of methods sharing code clones and an edge represents "abstract" invocation between the method sets. Potentially useful as the above approaches are, they still suffer from some insufficiencies for practical reuse tasks. First, they cannot describe a sophisticated design in practice for missing important concepts such as interface, class, association, etc. Second, these techniques are designed for comprehension, their reported designs cannot facilitate actionable code reuse tasks in an explicit way.

Alternatively, reverse SPL (Software Product Line) engineering techniques [8], [9], [10], [11], [12] have good potential to meet the reuse need. Fisher et al. [8] and Martinez et al. [9] proposed different techniques to recover a generalized software product line model from a set of software variants. However, the challenges to adopt their approaches lie in two-fold. First, these approaches assume the variant products as its input. However, in our case, the code for "variant design" is unknown in advance. Second, these approaches extract SPL model in the grain of component or product (e.g., feature model [13]) while we need the recurring designs to be extracted on the finer level (e.g., class and method) for facilitating actionable code reuse tasks.

In this paper, we propose an approach to detecting and extracting recurring designs in code base into a list of design templates to facilitate template-based code generation. In our approach, we first identify each set of "correspondent" program elements across the code base into a *program multiset*. Then, we construct a graph by building various relations (e.g., declare, invoke, etc.) between program multisets. Finally, we heuristically split such a graph into a set of subgraphs, each of which is abstracted into a design template. As a result, each design template captures various types of object-oriented program elements (and their relations) and can be visualized in the form of UML class diagram (some extracted templates can be checked at our website [14]). Programmers can manage the design templates and customize them to generate code skeleton for the reusable features. The generated code skeleton contains semi-implemented code that is annotated with hints and comments to remind programmers of necessary modification.

We implemented our approach as an Eclipse plugin called

[1]a clone set consists of multiple pieces of duplicated code.

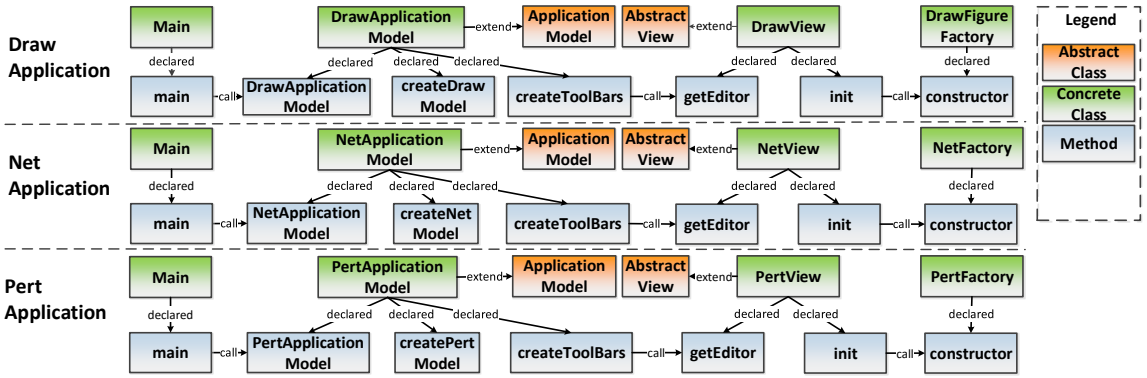

Fig. 1. A Recurring Design in JHotDraw Projects

MICoDe (**M**ining **I**mplicit **Co**de **De**sign). We evaluated our approach with a reuse simulation experiment and a user study involving 16 participants. The results of our simulation experiment on 10 open source Java projects show that, for creating a new similar feature with a design template, (1) on average 69% of the elements in the template can be reused and (2) on average 60% code of the new feature can be adopted from the template. Our user study further shows that, compared to the participants adopting the copy-paste-modify strategy, the ones using *MICoDe* are more effective to understand a big design picture and more efficient to accomplish the code reuse task.

The contributions of this paper are listed as follows. First, we propose an approach to detecting and extracting recurring design from code base as design templates and support template-based code generation; Second, we implement an Eclipse plugin *MICoDe* for practical use of our approach (The tool, demo video, and snapshots are now published at [14]); Third, we evaluate our approach with both simulation experiment and a user study to reveal the effectiveness of our approach. The results show that the extracted design templates are both accurate and practical for code reuse task.

II. Motivating Example

Fig. 1 shows a design recurring in the JHotdraw7.1 project [15], a drawing framework for Java language. All the designs showed in Fig. 1 aim to create a customized drawing application according to project convention. According to Fig. 1, when programmers intend to create a new drawing application, they need to (1) construct several Java classes (e.g., class `Main` and `*Factory`), (2) make some of them inherit existing class (e.g., class `*ApplicationModel` and `*View`), and (3) make them cooperate with each other by implementing various methods. If we do not know such recurring designs in Fig. 1 in code base, creating a customized application is effort-consuming.

Nevertheless, it is still a non-trivial task even when we know these recurring designs. One straightforward way is to copy all the code of one design (e.g., `Draw` Project) and modify it to satisfy our own need. However, we have to address the following challenges:

Q1: What are the code files to copy? Apart from the classes shown in Fig. 1, each application involves other classes (not shown in Fig. 1). Without a big picture of how the code is organized, we may either miss-copy some code files or copy additional unnecessary code files.

Q2: Which design shall we copy? Intrinsically, we can copy the most "similar" design as the draft code to save the follow-up customization effort. However, it is hard to know the similarity unless we go through all the code of those designs.

Q3: What parts of design do we need to customize? Even if we have copied the most similar design, we still need to consistently customize the copied classes. For example, after copying `Draw` project in Fig. 1, we have to consistently customize our own `ApplicationModel` class, `View` class, etc. Moreover, we may also need to remove the irrelevant classes, or check other designs to integrate their relevant code.

Q4: How do we customize the code? Table I shows the method body of `init()` methods of class `DrawView`, `PertView`, and `NetView` in Fig. 1. Their differences are very helpful hints for code customization. Suppose we copy and modify the `Draw` project, it is easy to miss modifying the `DrawFactory` (line 7) into a customized factory class or miss setting scale factor (line 8) when necessary, which leads to bugs. Nevertheless, comparing across code in multiple designs takes tremendous effort, if not impossible.

TABLE I
Corresponding init() Methods in JHotDraw Project

```
0  DrawView.java                PertView.java:               NetView.java:
1  void init() {                void init() {                void init() {
2    super.init()                 super.init()                 super.init()
3    ...                          ...                          ...
4    setEditor(new                setEditor(new                setEditor(new
5      DrawEditor())                DrawEditor())                DrawEditor())
6    view.setFactory(new          view.setFactory(new          view.setFactory(new
7      DrawFigFactory())            PertFactory())               NetFactory())
8                                   setScaleFactor(2.0)          setScaleFactor(1.0)
9    ...                          ...                          ...
10 }                            }                            }
```

In order to customize our own design with regard to existing program convention, we usually need to understand the general design structure, take different existing implementations as reference, and be aware of their differences as potential customization points. The copy-paste-modify practice can hardly meet this end.

We propose *MICoDe*, a tool-supported approach which can both *detect* and *summarize* the recurring designs in Fig. 1 into a design template. The template is visualized as a UML class

diagram (sample snapshots are available at our website [14]), which captures the structural commonalities of the instances of the recurring design. *MICoDe* supports further customization on the template to generate both code skeleton and the method body. The generated method bodies are annotated with comments to indicate what modification could be adopted.

III. Approach

Figure 2 shows an overview of our approach. Our approach takes the source code of a code base as input, and generates customizable design templates as output. The design templates can be instantiated and customized to generate code skeletons.

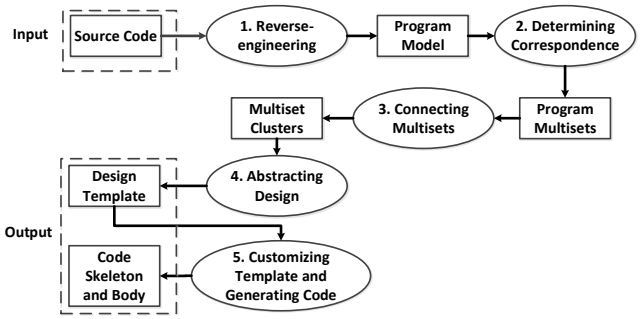

Fig. 2. Approach Overview

Given the source code, we first reverse-engineer it into a program model, which captures various program elements (e.g., *class* and *method*) and their relations (e.g., *declare* and *invoke*). Second, we identify "corresponding" program elements by clustering technique. We regard each set of corresponding program elements as a *program multiset*. Third, we build the relation (e.g., *declare* and *invoke*) among the program multisets so that they can be connected into a graph. Then, we heuristically split subgraphs of connected program multisets into a set of multiset clusters as potential designs. Fourth, we abstract each multiset cluster into design templates. Finally, the generated design templates will be manifested as UML class diagrams, and programmers can manage the templates and customize them to generate code skeleton and body. In the following, we present the details of each step.

A. Reverse Engineering

We reverse-engineer the source code of a code base into a program model consisting of program elements and their relations. Fig. 3 shows the meta model of program model, which describes the program elements (i.e., *class*, *interface*, *method* and *field*) and their relations (i.e., *declare*, *extend*, *implement*, *invoke*, and *access*). Each program element in the model is attached with its inherent program attributes. For example, a *method* program element will be attached with the attributes such as method name, return type, and parameters. Moreover, each mined design template is also represented as a program model.

B. Determining Correspondences

We determine corresponding program elements in order to identify the program elements playing the similar role in each

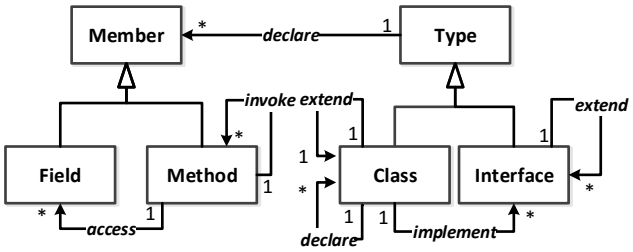

Fig. 3. The Meta Model of Program Model

instance of a recurring design. For example, in Fig. 1, the three classes, i.e., `DrawView`, `NetView`, and `PertView`, are considered as corresponding elements. We call a set of corresponding program elements as a **program multiset**. In order to generate a set of program multisets from the program model, we first construct a **declaration tree** over the model based on the *declare* relation defined in our meta model (see Fig. 3). In a declaration tree, each node represents an entity in the meta model and the parent-child relation represents a *declare* relation in the meta model. For example, a class can declare fields, methods and inner classes, and the inner class can further declare its fields, methods and sub inner classes.

The correspondence of program elements is determined in a top-down manner. We first determine correspondences of top-level program elements (i.e., type) as multisets. Next, for each of multiset, we correspond the declaration children of its elements to form new multisets in the next level.

1) Corresponding Top-Level Elements: We determine the correspondences of the top-level program elements by clustering them with regard to code similarity and heuristic rules. We adopt the hierarchical clustering strategy with complete linkage [16] to cluster all the top-level classes and interfaces.

a) Similarity: The similarity between two types (class or interface) is defined in terms of type name, shared super types, and type body, i.e., $sim = \sum(w_i \cdot sim_i)$, in which $i \in \{name, superType, body\}$. w_{name}, $w_{superType}$, w_{body} represents respective weight for the three factors, which requires:

- $w_{name} + w_{superType} + w_{body} = 1$
- $w_{name}, w_{superType}, w_{body} \in (0, 1)$

The similarity of type name, super types, and type body is computed as follows. The type name of t_1 and t_2 is split into token sequences ts_1 and ts_2 according to some program convention (e.g., camel convention). Let len_1 and len_2 be the length of ts_1 and ts_2, and let len_{lcs} be the length of the longest common subsequence of ts_1 and ts_2. Name similarity is $sim_{name} = 2 \times len_{lcs}/(len_1 + len_2)$. Let st_1 and st_2 be the set of super types of the type t_1 and t_2. Super type similarity is the Jaccard coefficient of the two sets of super types, i.e., $sim_{superType} = |st_1 \bigcap st_2| / |st_1 \bigcup st_2|$. Type bodies of the two types are compared at the textual level. Let len_1 and len_2 be the length of the two source code token sequences, and let len_c be the length of their shared cloned code, source code similarity is $sim_{body} = 2 \times len_c/(len_1 + len_2)$.

b) Heuristics: Similarity sometimes is not sufficient to ensure a correspondence relation. Thus, we enforce the following heuristic rules to improve the accuracy:

- The similarity between a class and an interface is 0.
- The similarity between a type and its super type is 0.

As a result, we can obtain some clusters such as {`DrawView`, `PertView`, `NetView`} and {`DrawApplicationModel`, `PertApplicationModel`, `NetApplicationModel`}. We regard each cluster as a top-level multiset.

2) Recursively Generating New Multisets: Given a multiset, $ms = \{e_1, e_2, ..., e_n\}(n \geq 2)$, we generate new multisets from the declaration children of $e_i(1 \leq i \leq n)$ so that each e_i contributes at most one declaration child to a new multiset. The recursive process stops when the new generated multiset has no declaration children. For the example in Fig. 1, given the multiset $class\_set$ = {`DrawView`, `PertView`, `NetView`}, we determine the correspondence of the declaration children of elements in $class\_set$, i.e., their inner classes, methods, and fields. When we generate a new multiset in which each element is an inner class, we will further determine the correspondence of the declaration children of its elements. When we generate a new multiset in which each element is method (e.g., {`DrawView.init()`, `PertView.init()`, `NetView.init()`}), the recursive process stops as the `init()` methods have no declaration child. The same case applies for the multiset of fields.

Algorithm 1 Corresponding Children

Require: a multiset set, entity similarity threshold t_{sim}
Ensure: a set of multisets $MSSet$
1: $dcSet \leftarrow$ retrieve declaration children sets of set;
2: $MSSet \leftarrow \emptyset$;
3: **for** each set of declaration children $dc \in dcSet$ **do**
4: **for** each not-yet-corresponded element $e \in dc$ **do**
5: $ms \leftarrow \emptyset$;
6: add e to ms;
7: $candSet \leftarrow dcSet \setminus \{dc\}$;
8: **while** $candSet \neq \emptyset$ **do**
9: find a not-yet-corresponded element $e_c \in cand$ $(cand \in candSet)$ with maximum $sim(e_c, ms)$
10: **if** $sim(e_c, ms) < t_{sim}$ **then**
11: break;
12: **end if**
13: add e_c to ms;
14: $candSet \leftarrow candSet \setminus \{cand\}$;
15: **end while**
16: mark all nodes in ms as corresponded;
17: if $|ms| > 2$ then $MSSet.add(ms)$;
18: **end for**
19: **end for**
20: **return** $MSSet$;

Algorithm 1 shows the details of corresponding the declaration children of elements in a given multiset set with threshold t_{sim}. In Algorithm 1, we mark an element as "corresponded" if it has been used to form a new multiset. We first construct a set $dcSet = \{e | e$ is a set of declaration children of an element (i.e., type) in $set\}$ (line 1). Then we iteratively process each dc in $dcSet$ (line 3). We start with processing each declaration child $e \in dc$. If e has not been used to generate any multiset (i.e., not-yet-corresponded), we add it as a seed entity to an empty multiset ms (line 5–6). Then, we try to add declaration children from the rest sets $candSet$ $(dcSet \setminus \{dc\})$ to ms.

For each set $cand \in candSet$, we attempt to find the most *similar* child $e_c \in cand$ to ms (line 8-15). If the similarity between e_c and ms (i.e., $sim(e_c, ms)$ at line 10) is above the threshold, we add it into ms. The similarity between a candidate element and elements in the multiset $sim(e_c, ms)$ is computed as the average similarity of e_c and each entity $e \in ms$, i.e.,

$$sim(e_c, ms) = \frac{\sum_{e \in ms} sim(e_c, e)}{|ms|} \quad (1)$$

Finally, we return all the multisets containing at least two elements (line 17).

C. Connecting Multisets

Given multisets of corresponding program elements, we connect the multisets by the relations of program elements across multisets. We regard each set of connected multisets as a potential design for abstracting design template. First, we build the relations (i.e., *declare*, *invoke*, *access*, *extend*, and *implement*, see Section III-A) between multisets. Second, we form a graph of multisets and detect its connected components as potential designs. Finally, we refine the results with frequent item mining technique [17].

1) Determining Multisets Relations: We determine the relations of two multisets with regard to the relations in meta model. Let MS_s and MS_t be two multisets of program elements. Let rel_t be a type of relations from an element $e_s \in MS_s$ to another element $e_t \in MS_t$. Then, MS_s, MS_t, and all the relations of type rel_t from MS_s to MS_t can form a bipartite graph $G_{bi} = (MS_s, MS_t, rel_t)$. Let $Match$ be the set of the maximum number of matchings [18] of G_{bi}. We define the **connectivity strength** from MS_s to MS_t of relation type rel_t as:

$$strength(MS_s, MS_t, rel_t) = \frac{|Match|}{min(|MS_s|, |MS_t|)}$$

Given a user specified threshold $t_{abrel}(0 \leq t_{abrel} \leq 1)$, we create a relation rel_t from MS_s to MS_t if the following requirements are satisfied:

- $strength(MS_s, MS_t, rel_t) \geq t_{abrel}$
- $|Match| \geq 2$

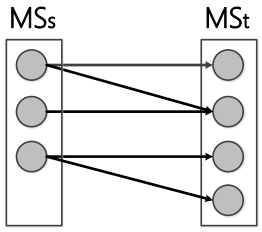

Fig. 4. An Example of Connecting Multisets

In Fig. 4, $|Match|$ = 3, $min(|MS_s|, |MS_t|)$ = 3. Thus, the connectivity strength from MS_s to MS_t is 3/3 = 1.

2) Forming Graph: After determining the relations among multisets, we build a graph of multisets $G_m = (V, E)$ in which V denotes all the multisets of *top-level* program elements (i.e., top-level class or interface) as its elements and E denotes the relations between the multisets in V.

In order to reflect the association such as *invoke* and *access* relation among top-level multisets, we further enhance the

edges in G_m by building *invoke* and *access* relations. If a multiset M_d is reachable by a top-level multiset M_{top} through *declare* relation, we call M_d as the **descendent multiset** of M_{top}. For example, in Fig. 1, the multiset {`DrawView.init()`, `PertView.init()`, `NetView.init()`} is a descendent multiset of multiset {`DrawView`, `PertView`, `NetView`}. Given two top-level multisets M_{top1} and M_{top2}, we build a relation of type rel_t from M_{top1} to M_{top2} if there exists a descendent multiset M_{d1} of M_{top1} has the relation of type rel_t with a descendent multiset M_{d2} of M_{top2}. By this means, we enhance the set of edges from E to E' and $E \subseteq E'$. Given the enhanced graph $G'_m =< V, E' >$, we regard each maximal *connected component* in G'_m as a potential design consisting of top-level multisets.

Fig. 5 shows the example of an extracted maximal connected component. In Fig. 5, each node represents a top-level multiset and each edge represents an *invoke* relation between two multisets. We can see that a maximal connected component could be over large and involve multiple duplicated designs. In Fig. 5, the multisets C={`MySQLDB, OracleDB`} and D={`MySQLConfig, OracleConfig`} can form a more independent design template, and so can the multisets E={`ExcelGen, XMLGen`} and G={`ExcelFormat, XMLFormat`}. We adopt a frequent-item based approach to refining the maximal connected components into more independent clusters of multisets.

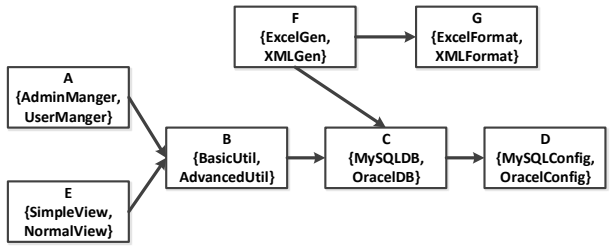

Fig. 5. An Example for Maximal Connected Components

3) Refining Connected Components: The rationale of refinement is to split commonly invoked multisets from the maximum connected components.

Given a maximal connected component $G = (V, E)$, we select $V_{src} = \{v | v \in V$ and v's in-degree is $0\}$. For each $v \in V_{src}$, we traverse G from v with the depth-first strategy through *invoke* and *access* edge and all the visited nodes in G can form a **traversing node set** $V_{tr} \subseteq V$, denoted as $V_{tr}(v)$. For example in Fig. 5, V_{src}={A, E, F}, and $V_{tr}(A)$={A, B, C, D}, $V_{tr}(E)$={E, B, C, D}, and $V_{tr}(F)$={F, G, C, D}.

Next, we regard each traversing node set as a *transaction* and each of its contained nodes (i.e., multiset) as an *item*, and apply the FP Growth algorithm [19] with support of 2. It means that, among all the transactions, once a set of items appears at least twice, it is considered as a frequent item set. Typical frequent item sets in our example are {C, D} and {B, C, D} which appear three times and twice among $V_{tr}(A)$, $V_{tr}(E)$, and $V_{tr}(F)$.

Among the frequent item sets, we use the following heuristics to filter trivial item sets and rank the rest in terms of their independence.

- **Filter**: Given a list of frequent item sets $list$, a frequent item set $set_1 \in list$ is removed if $\exists set_2 \in list$ which (1) is a super set of set_1 and (2) has the same support with set_1. For example, the set {C} is removed as there exists a set {C, D} with the same support.
- **Rank**: Given two frequent item sets, set_1 and set_2,
  - set_1 is ranked over set_2 if the support of set_1 is over that of set_2.
  - if set_1 and set_2 have a tie in terms of support (set_1 is not a super set of set_2, and vice versa), we rank set_1 over set_2 if $|set_1| > |set_2|$.

  Namely, the stronger support and the larger size a frequent item set has, the more independent we deem it is. In our example, the ranked list of frequent item sets is $<${C, D}, {B, C, D}$>$.

Algorithm 2 Refine Clustering Results

Require: a list of traversing node sets $traList$, an ranked list of frequent item set $fList$
Ensure: a list of refined clusters of multiset $clusters$

```
1:  tmpClusters ← ∅;
2:  for each item set is ∈ fList do
3:      if ist ≠ ∅ then
4:          add is into tmpClusters;
5:          for each traversing node set tSet ∈ traList do
6:              tSet ← tSet \ is;
7:          end for
8:          for each item set set ∈ fList do
9:              set ← set \ is;
10:         end for
11:     end if
12: end for
13: add all the non-empty traversing node set in traList into tmpClusters;
14: clusters ← ∅;
15: for each cluster cluster ∈ tmpClusters do
16:     clSet ← split cluster into a set of connected components;
17:     add all the clusters in clSet into clusters;
18: end for
19: return clusters;
```

Given a list of traversing node sets (i.e., "transaction") $traList$ derived from a maximum connected component $G =< V, E >$, and a ranked list of frequent item sets $fList$ we split G as described in Algorithm 2. In Algorithm 2, we go through the ranked item sets from the most independent item set to the least independent one (line 2). Each time when we process a non-empty item set $is \in fList$, we first add is into a cluster list $tmpClusters$, then we remove the elements in is from the set in $traList$ and $fList$ (line 3-11). After processing all the frequent item sets, we add all the non-empty traversing node sets in $traList$ into $tmpClusters$. Finally, in order to ensure the elements of each cluster can form a connected component, we further split each cluster in $tmpClusters$ if its elements (i.e., multisets) are not connected (line 15-18).

For the example in Fig. 5, we will have three traversing node sets (i.e., {A, B, C, D}, {E, B, C, D}, and {F, G, C, D}) and the ranked list of frequent item sets consisting of two elements set_1 = {C, D} and set_2 = {B, C, D}. Then, we first process set_1, which will create a new cluster {C, D}, and make all the three traversing node sets into {A, B}, {E, B}, and {F, G} and set_2 into {B}. Afterwards, we process set_2,

which will create a new cluster {B} and make the traversing node sets into {A}, {E}, and {F, G}. Therefore, we can have the clusters as: {A}, {E}, {F, G}, {B}, and {C, D}. Given all the clusters with its elements connected, these clusters are the final clusters we obtain.

D. Abstracting Design Templates

Given a list of clusters of multisets, we generate design templates consisting of template classes, template interfaces, template fields and template methods. Generating template entities involves abstracting name, parameters, and super types.

Given the names of a multiset of corresponding entities, we generalize these sequences by wildcards. For example, an abstract name **View* can be generalized from the multiset of three classes {*DrawView, NetView, PertView*}.

In addition, we further abstract the parameters of corresponded methods, and super types of corresponded classes and interfaces. We abstract the parameters of corresponded methods by computing the intersection of their parameter type sets. For example, given a multiset of methods as $mSet$ = {*m1*(), *m2*(), *m3*()} and the set of parameters are {String, int, float}, {String int}, and {String, int} respectively, the abstracted parameters for $mSet$ is {String, int}. In the same vein, we abstract the super types of corresponded classes or interfaces by computing the intersection of their super types.

E. Managing Templates and Generating Code

The mined templates can be further managed (i.e., refined and customized) and used to generate code.

1) Managing and Customizing Template: We support the following template refinements. Programmers are allowed to: (1) remove template entities (or relations) they consider irrelevant; (2) add template entities (or relations) and split a large template into several smaller ones or merge some templates into a larger one; (3) load design templates and select the template they wants to reuse. Customization involves fixing name placeholders, adding or removing super types, etc. Our tool provides a wizard that guides the programmer through all customizable entities and relations. Once the programmer customizes all the template entities, the tool prompts the programmer that the template is ready for code generation.

2) Generating Code: Apart from generating the code skeleton, we also generate semi-implemented method bodies if the text similarity of original method bodies is above a threshold. When the text similarity is above the threshold, we apply the *MCIDiff* technique [20] [21] on these similar method bodies to analyze their difference. It reports the difference as token-sequence-based differential multiset such as {ϵ, *setScaleFactor(1.0);*, *setScaleFactor(2.0);* }. We regard each differential multiset indicates a potential customization point in the body. We copy the body of the longest method into the generated method, and generates "TODO" comments for each of the copied statements involved in differential multisets. For example, assume the first method in Table I is the longest one, we will generate a comment such as "TODO: you may additionally have the code like *setScaleFactor(1.0);* or *setScaleFactor(2.0);*" before the empty line.

IV. Tool Support

A snapshot and a demo video of our tool *MICoDe* are available at our website [14]. *MICoDe* stores mined design templates as Eclipse Modeling Framework (EMF) models. It lists all mined design templates in *Template* view.

Template Visualization and Customization. To facilitate the refinement and customization of design templates, *MICoDe* visualizes design templates in a class-diagram-like template editor. Different types of template entities (i.e., class, interface, method, and field) in different status (i.e., configured and un-configured) are represented by rectangles with different colors. For example, abstracted super classes are represented by bright orange rectangles which cannot be configured; template classes are represented by the dark green rectangles, which are to be customized. Similarly, template method and template field are represented by dark blue and dark yellow rectangles which are to be customized respectively. Double-clicking a template entity opens a wizard dialogue which allows programmers to customize the entity by specifying its detailed information such as class name, package name, parameters, etc. After the user customizes a template entity, *MICoDe* will turn the color of the entity into a bright one (e.g., bright green, bright blue, etc.). In addition, the user can delete or add some entities or relations to the template. The newly added entities and relations will be used to generate code in the same way as the template entities mined from the code.

Code Detail Comparison. Right-clicking a template entity and choosing *Show Supporting Entities* menu item open the *MCIDiff* view that presents the differencing results of corresponding entities from which the template entity is abstracted. The *MCIDiff* view shows corresponding methods side by side and highlights their differences using different colors.

Code Generation. After a user finishes customizing the template, he or she can utilize *MICoDe* to validate the "instantiated" template by checking whether all the template entities are customized. If the customized template is valid, the user can click the *Generate Code* menu item to generate the code.

V. Simulation Experiment

We aim to answer the following questions in the experiment:

- **RQ1.** Whether the extracted design templates are reusable for a new similar feature?
- **RQ2.** How many duplicated designs are there in existing large code base?

A. Experiment Setup

We answer the above research questions by applying *MICoDe* on 10 open source Java projects (see Table II) to detect their recurring designs. We simulate code reuse tasks on those design templates supported by over three design instances. Suppose a design template T is supported by n ($n \geq 3$) design instances, we choose one of its design instances I and generate a new design template T' from the other $n-1$ instances. Then we compare the template T' and design instance I to see how similar they are. In other words, we regard the instance I as a design implementing a new similar feature, the similarity

TABLE II
RECURRING DESIGNS IN OPEN SOURCE JAVA PROJECTS

Project	Version	Project Profile				Extracted Templates				Evaluation Subject		Evaluation Result	
		#LoC	#Class	#Method	#Field	#Template	#Class	#Method	#Field	#Trial Template	#Trials	Precision	Recall
Apache-commons-math	3.4.1	182192	1596	13102	3924	191	853	5038	914	37	107	0.66	0.56
FreeHEP	2.0	61662	804	5551	2643	71	217	1274	299	5	10	0.98	0.64
JFreeChart	1.2.0	96478	676	8324	2978	119	456	4524	964	5	17	0.60	0.77
JHotDraw	7.0.6	32447	350	3146	868	52	189	1103	296	4	15	0.69	0.75
JMental	5.3	37370	518	2815	1579	68	252	790	571	3	6	0.77	0.67
JMeter	3.2	102407	1044	9553	6053	190	669	3087	1463	10	31	0.62	0.58
Log4j	2.8.2	10882	127	1452	370	13	54	519	103	2	6	0.54	0.51
Soot	2.5.0	186655	2228	14191	4688	275	1302	5231	550	10	25	0.79	0.61
XChart	3.3.0	6917	102	743	282	14	61	346	92	2	5	0.89	0.70
Xerces2-j	3.3.1	127764	943	9089	5397	124	518	3429	1241	5	10	0.72	0.47
Total	/	844774	8388	67966	28782	1117	4571	25341	6493	83	232	/	/
Average	/	84477.40	838.80	6796.60	2878.20	111.70	457.10	2534.10	649.30	8.30	23.20	0.69	0.60
Median	/	79070	740	6937.50	2810.50	95	354	2180.50	560.50	5	12.50	0.66	0.54

between T' and I indicates how reusable T' is to create I. In this experiment, we call each of such comparisons as a **trial**. Given a design template supported by n instances, we can have n trials.

We calculate the similarity between a design template T and a design instance I as follows. T can be considered as a declaration tree $G_T = (V_T, E_T)$ (defined in Section III-B) where each node in V_T corresponds to a program multiset; I can be considered as a declaration tree $G_I = (V_I, E_I)$ where each node in V_I corresponds to a program element. Therefore, a node $v_T \in V_T$ is considered reusable for creating design I if $\exists v_I \in V_I$ so that v_T *matches* v_I. Given a design template $G_T = (V_T, E_T)$, a design instance $G_I = (V_I, E_I)$, and let the set of matched nodes be $V_m (V_m \subseteq V_I)$, we calculate the precision as $|V_m|/|V_T|$ and the recall as $|V_m|/|V_I|$.

We use precision and recall to evaluate how reusable a design template for creating a new design instance. For a trial (i.e., simulated reuse task), precision means how many program elements in the template can be used for creating a new similar feature while recall means how many program elements in the new similar feature can be reused from the template. High precision indicates that we do not need to delete many program elements in the template. High recall indicates that, apart from the code adopted from the template, we do not need to additionally add much code to accomplish the similar feature.

We match the nodes in G_T and G_I in a top-down manner by progressively matching the nodes in the same layer in the declaration tree. For example, we first match the nodes in the top layer, then for each pair of matched nodes, we proceed to match their children. When matching the set of nodes in the same layer, i.e., $V'_T \subseteq V_T$ and $V'_I \subseteq V_I$, we construct a bipartite graph $G_b = (V'_T, V'_I, E)$ where an edge e exists between two nodes v_T ($v_T \in V'_T$) and v_I ($v_I \in V_I$) if the similarity between v_T and v_I (see Equation 1) is above a threshold. Then we use the Blossom algorithm [22] to get the best matching in the bipartite graph G_b. In this experiment, we set the field similarity threshold to be 0.6, method similarity threshold to be 0.8, and type similarity threshold to be 0.2.

B. Result

Table II shows the details of the Java open source projects in terms of project profiles (by lines of code, number of classes, methods, and fields), extracted template (by number of extracted templates, number of involved classes, methods, and fields), number of used templates for simulating reuse tasks (#T-Templ), number of trials (#Trials), average precision, and average recall. All the extracted designs are available on our website [14].

In this experiment, the recurring designs involve 4571 out of 8388 files, which is a considerable number. *MICoDe* reports 1117 templates among all 10 projects, of which 83 (i.e., 7.4%) involve over three design instances. In Table II, we can see that the summarized templates can be reused with a reasonably good accuracy. The average precision and recall are 0.69 and 0.60 respectively. It means that, on average, 69% of the elements in a template can be reused to accomplish the similar feature, and 60% code of new similar feature can be adopted from a template.

We further inspect the trials with low precision or recall.

a) Low Precision Trials: Low precision usually happens when the design template is enriched with more elements than a design instance needs. Fig. 6 shows a reuse trial with a low precision (i.e., 10%) in the *JMeter* project. Fig. 6 (a) shows a template captures the interaction of 4 template classes such as `*SamplerGui`, `*Sampler`, `*ConfigGui`, and `*Client`. Fig. 6 (b) shows a design instance used to evaluate the template.

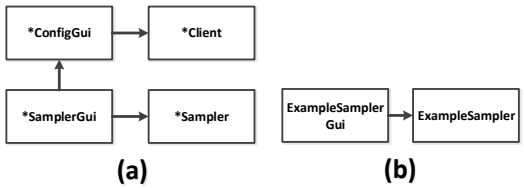

Fig. 6. An Example of Low Precision

In the *JMeter* project, some sampler GUIs such as `TCPSamplerGui` interact with a configure GUI while others such as `SMTPSamplerGui` do not. *MICoDe* aims to capture the design structure of all the design instances comprehensively. Thus, it reports a super set of four abstracted classes and their relations. For the trial in Fig. 6, the design instance {`ExampleSamplerGui`, `ExampleSampler`} only matches `*SamplerGui` and `*Sampler`; and `*ConfigGui` and `*Client` declare a large number of abstracted methods

and fields. In such a case, the design instance is only a "subset" of the template, which leads to a low precision.

Nevertheless, in practice, *MICoDe* allows programmers to manually customize the template for their own need. Moreover, once the programmers find the additional abstracted class not useful, they can simply delete it on template, which is not effort-consuming.

b) Low Recall Trials: Low recall lies in that some design instances involve additional functions other than that adopted from the template. Still in *JMeter* project, one design instance involves two classes `ProxyControlGui` and `ProxyControl`, which has a low recall (i.e., 9%) after compared to the design template of {`*ControlGui`, `*Control`}. In this trial, compared to template `*Control` class, `ProxyControl` class additionally declare a great number of new fields/methods (such as `CERT_ALIAS`, `CERT_ALIAS`, `startProxy()`, etc.) for implementing proxy relevant features. Nevertheless, the precision of this trial (i.e., 95%) is fairly acceptable. In practice, it is usual for programmers to enhance additional features based on existing template code. Therefore, we deem the performance of *MICoDe* (average 60%) is acceptable.

In conclusion, the experiment shows that recurring designs are frequent in open source code projects and the templates extracted by *MICoDe* are generally reusable.

C. Threats to Validity

One threat is that our simulation cannot be applied on the templates with only 2 instances. Without authoritative template benchmark of the open source projects, it is hard to avoid subjectiveness and bias when evaluating the reusability or meaningfulness of design templates. We will cooperate with industrial partners and deploy *MICoDe* in real developing environment to generalize our results. The other threat is that we can generate different templates with different similarity thresholds. In this experiment, we set the thresholds by our preliminary observation and experience. We will conduct a more comprehensive study for the impact of thresholds.

VI. User Study

MICoDe aims to enable easy and systematic reuse of existing code designs in code base. To evaluate whether *MICoDe* achieves this goal, we conducted a user study to investigate the programmers' efficiency in completing reuse-based development task when they are with and without the *MICoDe* tool.

A. Study design

We recruited 16 graduate students from Fudan University, China. We conducted a pre-study survey for these participants to understand their programming experience and capability. These participants were matched in pairs by experience and capability, and then randomly allocated into experimental or control group. The experimental group G_1 (participants P1-P8) used the *MICoDe* tool to perform the software development task, while the control group G_2 (participants P9-P16) used Eclipse IDE to perform the same task. We provided a 3-hour tutorial for the experimental group and training session of the *MICoDe* tool is conducted 3 hours before the experiment. The chosen training example in our tutorial is irrelevant to the task assigned to participants in the experiment so that we can make the participants familiarize themselves with the tool features while avoid introducing experimental bias. According to our survey, all the participants are familiar with Eclipse IDE, thus we did not provide tutorial and training on Eclipse IDE.

We chose a JHotDraw template of creating a new JHotDraw-based Web Applet as our subject system in the user study. The reason for choosing this template lies in two-fold. First, such a template is extracted from a demo package of JHotDraw showing users how to build their own drawing application based on JHotDraw API. Since none of the participants have experience in JHotDraw project, it is more comprehensible than other design templates involving more detailed JHotDraw framework knowledge. Second, the size of the template is appropriate (consists of 3 template classes, 20 template methods, and 14 template fields), which is neither too trivial (so that the task can be finished without sufficient thought), nor too complicated (so that everybody is exhausted to fail).

The demo package from which the selected template is extracted provides 6 examples of implementing a JHotDraw-based Applets for drawing application within web browser, namely, `Pert` applet, `Net` applet, `Draw` applet, `SVG` applet, `ODG` applet, and `Teddy` applet. In this user study, we asked the participants to develop a `Pert` applet based on the other five applet samples. The participants of the control group were given five existing applet samples (`Draw`, `Net`, `ODG`, `SVG`, and `Teddy`) along with their clone information. They were free to reuse any code example in these five samples that they deemed relevant. The participants of the experimental group were given the template mined from the JHotDraw project with `Pert` applet sample removed. They needed to customize the template to generate code for the `Pert` applet. They also have access to the original 5 applet samples.

We asked the participants to complete the task in two hours. The participants were required to run a full-screen recorder while they were working on the task. After the participants finished the task or the time run out, they submitted task videos and the Java Applets they developed. The task videos allow us to time the task completion process and analyze the participants' behaviors during the task. Furthermore, each participant was requested to fill in a post-experiment survey for us to collect their feedbacks on the task and tool usage.

We evaluated the participants' task efficiency by comparing time to complete basic features (e.g., applet lifecycle management and basic JHotDraw framework extension), time to complete advanced features (e.g., Pert-specific visualization), and total task completion time of the participants in the two study groups. Advanced features involve cross-cutting code and require good understanding of alternative solutions in different Applets. For the *MICoDe* group, total task time also includes template customization time. These time metrics were

estimated from the participants' task videos.

B. Results

Table III and Table IV show the results. Overall, the *MICoDe* users completed basic features, advanced features, and the whole task more efficiently. 7 *MICoDe* users completed the development task (i.e., implemented all basic and advanced features and passed all our test cases), while 6 Eclipse users completed the task. The *MICoDe* user P8 and the Eclipse user P12 failed to complete some advanced features. The Eclipse user P16 failed to complete the basic feature and did not submit a working applet. The relevant time metrics of these three participants were invalid for comparison because these participants would need more time to complete the task.

TABLE III
TASK EFFICIENCY OF *MICoDe* GROUP

Participant	ConfigTime(m)	BasicTime(m)	AdvanceTime(m)	TotalTime(m)	Success
P1	11.28	35.35	14.85	61.38	Yes
P2	9.00	38.80	15.2	63.00	Yes
P3	11.17	42.97	16.45	70.58	Yes
P4	14.00	44.67	27.33	86.00	Yes
P5	6.78	62.38	21.68	90.85	Yes
P6	4.33	65.13	30.38	99.85	Yes
P7	9.63	73.88	17.00	100.52	Yes
P8	16.00	66.45	/	/	No
Average	10.28	53.69	20.41	81.74	/
Std.Dev.	3.50	13.85	5.79	15.46	/

TABLE IV
TASK EFFICIENCY OF *Eclipse* GROUP

Participant	ConfigTime(m)	BasicTime(m)	AdvanceTime(m)	TotalTime(m)	Success
P9	/	41.00	14.67	55.67	Yes
P10	/	50.52	22.43	72.95	Yes
P11	/	87.00	21.60	108.60	Yes
P12	/	73.67	/	/	No
P13	/	60.17	41.33	101.50	Yes
P14	/	63.00	38.00	101.00	Yes
P15	/	64.32	43.98	108.30	Yes
P16	/	/	/	/	No
Average	/	62.81	30.34	91.35	/
Std.Dev.	/	13.81	11.18	19.98	/

We used Wilcoxon's matched-pairs signed-ranked tests to to evaluate the difference of the two groups in terms of time on basic features (BasicTime), time on advanced features (AdvanceTime), and total task time (TotalTime). The results are shown in Table V. At the 0.05 significance level, we reject the null hypothesis for AdvanceTime, i.e., there is a significant difference between the two study groups in terms of the time on accomplishing advanced features. Moreover, the *MICoDe* group outperformed the *Eclipse* group in AdvanceTime metrics. Thus, we conclude that the *MICoDe* users accomplished the advanced features of JHotDraw-based `Pert` applet in significantly shorter time.

C. Analysis

Based on the recorded video and interviews with the participants, we summarize the reason as follows. Some *MICoDe* users (e.g., P5, P6) have not significantly advantage in completing basic features, compared with the Eclipse users with similar program experience (e.g., P13, P14). Although the number of the basic features is large, the implementations of basic features are usually straightforward, which requires little comparison between multiple applet samples for reference. Thus, copy-paste-modify is an efficient way for reusing basic features. Furthermore, the Eclipse tools (such as text replacement and rename refactoring) can save the Eclipse users a part of time in modifying copied code.

TABLE V
SIGNIFICANCE TEST FOR USER STUDY

H	Var	Group	Samples	p	Decision
H0	BasicTime	MICoDe	7	0.237	Accept
		Eclipse	7		
	AdvanceTime	MICoDe	6	0.046	Reject
		Eclipse	6		
	TotalTime	MICoDe	6	0.075	Accept
		Eclipse	6		

Nevertheless, the *MICoDe* users outperformed the Eclipse users in advanced features. The Eclipse users had to spend more time to understand design structure and implementation alternatives of the reused code in order to properly develop the advanced features. In contrast, the design template presented by *MICoDe* tool shows the design structure of the to-be-reused feature. This makes it easier for the *MICoDe* users to compare, understand and customize the code of the advanced features. Note that the BasicTime accounts for a large portion of the total task time as the number of basic features is large, which reduces the differences of total task time between the *MICoDe* users and the Eclipse users. Nonetheless, *MICoDe* users accomplished the task about 10 minutes faster than Eclipse users on average. In addition, the difference of two groups in TotalTime is near-significant (p value is 0.075, slightly higher than the set 0.05 significance level). Thus, we deem that *MICoDe* users had better performance than Eclipse users in this task.

Thus, we conclude that *MICoDe* tool can help programmers more efficiently accomplish reuse-based development tasks when the tasks require understanding a big design picture and comparing implementation alternatives.

D. Threats to Validity

There are three threats to our user study. First, we assume that two study groups are "equivalent" despite that individual difference always exists. To address this threat, we carefully compared participants' experience and capabilities and randomly allocate comparable participants into the two groups. Second, our training session might introduce additional "warm-up" experience for experimental group on using *MICoDe*, which can introduce experimental bias. According to our pre-study survey, all the participants are familiar with Eclipse IDE (have Eclipse as their most frequent IDE). Therefore, we deem that such a threat is largely mitigated. Last, we conducted the controlled experiment only on one Java design as the experimental session had time limitation. Further studies are required to generalize our findings on more systems.

VII. DISCUSSION

In this section, we discuss (1) the difference between design template and design pattern [23] and (2) both reuse and refactoring opportunities conveyed by design templates.

Difference with Design Pattern. Our mined design template can also be regarded as a "pattern" of design which is, however, different from design pattern. A design pattern is usually intentionally created for code abstraction. In contrast, a design template is usually unintentionally created by code duplication. Therefore, the form of design patterns are more fixed while that of design templates are more dynamic. Hence, they must be detected with different techniques and convey different information. Technically, the design patterns can be detected by matching code with enumerable rules, while design templates should be mined with a more generalized way. Semantically, the design pattern describes general and project-independent design structure while design template describes recurring designs implementing similar project-specific feature. Despite of their difference, they are not exclusive. We observe that some of our mined design templates can involve design pattern such *Factory* pattern and *Strategy* pattern [14].

Reuse vs Refactoring Opportunities. In this work, design templates are largely regarded as reuse opportunities. The reason lies in as follows. Despite that design templates are usually caused by duplication, as showed in Fig. 1, abstraction has already been adopted. Thus, the duplication occurs in terms of code structure rather than code text. With the *well-organized* duplication, recurring designs usually convey project-specific convention for implementing a similar feature. Nevertheless, when the duplicated classes lack sufficient abstraction, they can also convey refactoring opportunities indeed. Therefore, we deem that it is important to detect recurring templates in the first place, so that we can decide the follow-up maintenance measure of either reuse or refactoring.

VIII. Related Work

A. Clone Detection

Both our work and clone detection techniques base on the code similarity measurement. Many researchers transfer code into an abstract form (such as string [24], token list [25] [26], AST [27] [28], and PDG [29]) to compare their similarity. A comprehensive survey of code clone reach can be found in [30]. Clone detection techniques aim to detect similar code fragments across software systems. In contrast, our approach detects and extracts correlated similar program elements, which unveils a bigger picture in terms of "similarity", i.e., recurring designs reusable for code generation.

B. Code Structure Pattern

Our approach starts at clustering or corresponding relevant program elements for abstraction, which is similar to a set of code structure pattern mining work. Basit et al. [4], [5] leverage frequent mining technique to mine structural clones indicating possible correlation between groups of code clones. Qian et al. [6] proposed a technique to mine logical clones to reveal similar high-level business logic in code base. Moreover, Lin et al. [31] proposed a technique to summarize syntactic patterns of code clones and aggregate the code clones with similar patterns. Our approach is different from these techniques in two folds. First, the patterns mined from above techniques are less expressive for the purpose of revealing recurring designs. In contrast, our approach is able to capture a rich set of program elements and program relations so that the mined template could represent sophisticated design in practice. Second, the patterns of the above techniques are generated as knowledge, meanwhile, our mined design templates are customizable for code generation, which can better facilitate programmers in real reuse-based software maintenance tasks.

C. Reverse Engineering and SPL Re-engineering

Many reverse-engineering methods [32], [33], [34], [35], [36], [37], [38] have been proposed to recover system design or architecture from the source code of a software system. We consider reverse-engineering methods as a basis and leverage it for the purpose of template mining. With similar purpose, some research work reverse-engineers source code for re-engineering legacy variant products into software product line [39], [40], [9], [10], [11], [12]. Haslinger et al. [39] extracted SPL feature models by analyzing configuration scripts of variant products. Valente et al. [41] proposed a semi-automatic approach to identify the code of optional features in SPLs. Martinez et al. [9] proposed an approach to migrate existing similar model variants into a software product line. In addition, Fischer et al. [8] developed a tool called *ECCO* to utilize reusable features from existing similar products to help programmers systematically compose a new variant product. These SPL re-engineering approaches often aim to recover a system-level SPL architecture for systematically developing and maintaining a set of variant products in a specific domain. Our work is similar with the above technique for the same purpose of code reuse. Nevertheless, our main differences with these approaches lie in that (1) we take code base instead of variant products or model as input and (2) the design template is extracted in more finer grain.

IX. Acknowledge

This research has been supported by the National Research Foundation, Singapore (No. NRF2015NCR-NCR003-003), the National Key Research and Development Program of China under Grant No. 2016YFB1000801, and the National Natural Science Foundation of China under Grant No. 61370079.

X. Conclusion and Future Work

The paper presents an approach to detecting and extracting implicit recurring code designs in code base into customizable design templates for code generation. We developed template editor to manage and customize design templates and code generator to generate code skeleton filled with semi-implemented code. Our simulation experiment shows that the design templates are useful to facilitate design-level reuse task. Our user study shows that the *MICoDe* tool helps the programmers to reuse recurring designs in code base more efficiently, compared with copy-paste-modify practice. In the future, we aim to apply our approach to crowdsourced code examples available online (e.g., Github) to build feature-oriented template libraries.

REFERENCES

[1] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proceedings of the International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.

[2] Z. M. Jiang and A. E. Hassan, "A framework for studying clones in large software systems," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 203–212.

[3] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 311–320.

[4] H. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 497–514, 2009.

[5] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," in *Proceedings of the European software engineering conference held jointly with ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 156–165.

[6] W. Qian, X. Peng, Z. Xing, S. Jarzabekz, and W. Zhao, "Mining logical clones in software: Revealing high-level business and programming rules," in *Proceedings of the International Conference on Software Maintenance*, 2013, pp. 40–49.

[7] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, 1993.

[8] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2014, pp. 391–400.

[9] J. Martinez, T. Ziadi, T. F. Bissyand, J. Klein, and Y. l. Traon, "Automating the extraction of model-based software product lines from model variants (t)," in *IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 396–406.

[10] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the European software engineering conference held jointly with ACM SIGSOFT international symposium on Foundations of software engineering*, 2013, pp. 81–91.

[11] Y. Xue, "Reengineering legacy software products into software product line," Ph.D. dissertation, National University of Singapore, 2012.

[12] G. Zhang, L. Shen, X. Peng, Z. Xing, and W. Zhao, "Incremental and iterative reengineering towards software product line: An industrial case study," in *Proceedings of the International Conference on Software Maintenance*, 2011, pp. 418–427.

[13] S. Apel, D. Batory, C. Kstner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[14] "MICoDe Website," http://linyun.info/micode/, [Online; accessed 2 Sep 2017].

[15] "Jhotdraw," http://www.jhotdraw.org/, accessed: April 20th, 2017.

[16] G. J. Szekely and M. L. Rizzo, "Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method," *Journal of classification*, vol. 22, no. 2, pp. 151–183, 2005.

[17] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the International Conference on Very Large Data Bases*, 1994, pp. 487–499.

[18] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., 2004.

[19] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Mining and Knowledge Discovery*, vol. 8, no. 1, pp. 53–87, 2004.

[20] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proceedings of International Conference on Software Engineering*, 2014, pp. 164–174.

[21] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *Proceedings of the European software engineering conference held jointly with ACM SIGSOFT international symposium on Foundations of software engineering*, 2015, pp. 520–531.

[22] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices," *Journal of Research of the National Bureau of Standards: Section B Mathematics and Mathematical Physics*, vol. 69B, no. 1-2, pp. 125–130, 1965.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, 1995.

[24] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Working Conference on Reverse Engineering*, 1995, pp. 86–95.

[25] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transaction on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[26] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner, "Clonedetective – a workbench for clone detection research," in *Proceedings of International Conference on Software Engineering*, 2009, pp. 603–606.

[27] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 368–377.

[28] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of International Conference on Software Engineering*, 2007, pp. 96–105.

[29] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the International Symposium on Static Analysis*, 2001, pp. 40–56.

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

[31] Y. Lin, Z. Xing, X. Peng, Y. Liu, J. Sun, W. Zhao, and J. Dong, "Clonepedia: Summarizing code clones by common syntactic context for software maintenance," in *Proceedings of the International Conference on Software Maintenance*, 2014, pp. 341–350.

[32] M. Lanza and S. Ducasse, "Polymetric views - a lightweight visual approach to reverse engineering," *IEEE Transaction on Software Engineering*, vol. 29, no. 9, pp. 782–795, 2003.

[33] M. Lungu, "Towards reverse engineering software ecosystems," in *Proceedings of the International Conference on Software Maintenance*, 2008, pp. 428–431.

[34] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *Proceedings of the International Conference on Software Maintenance*, 1999, pp. 50–59.

[35] K. Mobley, "Reverse engineering for software performance engineering," in *Proceedings of the Working Conference on Reverse Engineering*, 2007, pp. 302–304.

[36] A. Sanchez, N. Oliveira, L. S. Barbosa, and P. Henriques, "A perspective on architectural re-engineering," *Science of Computer Programming*, vol. 98, pp. 764–784, 2015.

[37] M.-A. D. Storey, K. Wong, and H. A. Müller, "Rigi: a visualization environment for reverse engineering," in *Proceedings of International Conference on Software Engineering*, 1997, pp. 606–607.

[38] J. Van Geet and S. Demeyer, "Reverse engineering on the mainframe: Lessons learned from "in vivo" research," *IEEE Software*, vol. 27, no. 4, pp. 30–36, 2010.

[39] E. Haslinger, R. Lopez-Herrejon, and A. Egyed, "Reverse engineering feature models from programs' feature sets," in *Proceedings of the Working Conference on Reverse Engineering*, 2011, pp. 308–312.

[40] C. Kastner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *IEEE Transaction on Software Engineering*, vol. 40, no. 1, pp. 67–82, 2014.

[41] M. Valente, V. Borges, and L. Passos, "A semi-automatic approach for extracting software product lines," *IEEE Transaction on Software Engineering*, vol. 38, no. 4, pp. 737–754, 2012.