Does Automated Refactoring Obviate Systematic Editing?

Na Meng* Lisa Hua* Miryung Kim† Kathryn S. McKinley‡
*The University of Texas at Austin †University of California, Los Angeles ‡Microsoft Research
mengna09@cs.utexas.edu lisahua@utexas.edu miryung@cs.ucla.edu mckinley@microsoft.com

***Abstract*—When developers add features and fix bugs, they often make *systematic edits*—similar edits to multiple locations. Systematic edits may indicate that developers should instead refactor to eliminate redundancy. This paper explores this question by designing and implementing a fully automated refactoring tool called RASE, which performs *clone removal*. RASE (1) extracts common code guided by a systematic edit; (2) creates new types and methods as needed; (3) parameterizes differences in types, methods, variables, and expressions; and (4) inserts return objects and exit labels based on control and data flow. To our knowledge, this functionality makes RASE the most advanced refactoring tool for automated clone removal.**

We evaluate RASE with real-world systematic edits and compare to method based clone removal. RASE successfully performs clone removal in 30 of 56 method pairs (n=2) and 20 of 30 method groups (n≥3) with systematic edits. We find that scoping refactoring based on systematic edits (58%), rather than the entire method (33%), increases the applicability of automated clone removal. Automated refactoring is not feasible in the other 42% cases, which indicates that automated refactoring does not obviate the need for systematic editing.

I. INTRODUCTION

Developers often apply similar changes to multiple locations. Systematic editing tools automate this task to reduce the programming burden of these tedious, error-prone changes. For instance, simultaneous text editing automates edit actions in one context by replicating them in other pre-selected contexts [25]. Linked Editing [30] and Clever [27] keep track of code clones and propagate changes from one clone to other clones. Sydit [24] and Lase [23] infer context-aware abstract transformations from user-selected examples, and then use the transformations to apply custom changes to user specified or automatically suggested locations. These systematic editing tools help developers to make coordinated changes in multiple locations. However, this practice may encourage developers to create or maintain duplicated code, when perhaps programs would be easier to maintain and understand if developers instead refactored their code. If programmers should always refactor, then systematic editing tools may be encouraging poor practices. This paper examines this question, that is, if systematic edits are obviated by using automated clone removal refactoring.

We first design and implement a new completely automated refactoring approach RASE, which takes as input two or more methods with systematic edits to scope target code, and then performs *clone removal*. RASE combines *extract method* (pg. 110 in [7]), *add parameter* (pg. 275 in [7]), *introduce exit label*, *parameterize type*, *form template method* (pg. 345 in [7]), and *introduce return object* refactorings to extract and remove similar code. It creates an *abstract refactoring template* that abstracts differences in types, methods, variables, and expressions from the multiple locations. Based on this template, as well as control and data flow, RASE creates new types and methods; inserts and assigns return objects and exit labels; adds parameters to the new extracted method; and introduces customized calls to it.

To our knowledge, RASE implements state-of-the-art refactoring with respect to its capability to factor and generalize code. Existing clone removal refactoring tools only implement some, but not all of the refactoring techniques in RASE [3, 12, 14, 21, 29]. Nor do they combine and study the effectiveness of automated clone removal. Furthermore, prior work that does study clone removal *did not* actually construct an automated refactoring tool to investigate the refactoring of systematically changed code [1, 5, 8, 16, 17]. The lack of automation in prior work introduces the possibility of subjectivity bias. By automating refactoring, our study substantially improves on prior methodology for determining the feasibility of clone removal refactoring.

We evaluate RASE on 56 real-world systematically edited method pairs (n=2) from prior work [23, 24] and 30 systematically edited method groups (n≥3) drawn from two open source projects. RASE automatically refactors 30 of 56 method pairs (54%) and 20 of 30 (67%) method groups when scoping with systematic edits. RASE applies sophisticated refactorings with all six techniques and in multiple different combinations of up to four techniques at once. On average, RASE automatically applied 41 lines of edits in our examples, ranging from 6 to 285, with modest code size increases of up to 18 lines of code, and reductions of up to 149 lines. Not surprisingly, RASE is most effective at reducing code size for multiple methods. Manual transformation to attain the same results would require inserting, deleting, or modifying up to 285 lines of code. Our evaluation results add to the evidence that removing common code with variations is challenging in practice and needs automated tool support.

We compare RASE scoped by systematic edits to RASE scoped by entire methods: scoping with systematic edits improves the feasibility of automatic clone removal compared to method-level scoping. With method-level scoping, RASE only refactors 34% method pairs and 30% method groups. However, with systematic edit scoping, RASE refactors 54% method

pairs and 67% method groups. Systematic edits thus are a good clue for refactoring, rather than being obviated by method-level refactoring. However, RASE cannot automate refactoring in 46% of pairs and 33% of groups mainly because of language limitations, semantic constraints, and lack of common code. We manually checked software version histories after systematic edits and found that in many cases, systematically edited methods are not refactored. They either co-evolve, diverge, or stay unchanged. Our tool evaluation and software repository observations indicate that both automated systematic editing and refactoring are necessary to support software evolution.

This paper designs and implements an automated clone removal refactoring algorithm and demonstrates refactoring *feasibility*. Predicting refactoring *desirability* is a hard problem because it depends on complex factors, such as code readability, the frequency and types of changes, future changes in requirements, and code size. Since RASE automates the feasibility step and quantifies code size impact, it should help developers determine refactoring desirability [4, 28, 33] and help with cost and benefit analysis [22, 26, 34], but we leave that investigation to future work.

In summary, this paper makes the following contributions.

- We design and implement RASE, an advanced automated clone removal tool. It takes methods with systematic edits as inputs and fully automates refactoring to extract common code with variations in types, methods, variables, and expressions.
- Evaluation on real-world pairs and groups of methods shows that RASE effectively automates clone removal in many cases. This tool evaluation together with our manual software repository examination reveals that refactoring is not always applicable or actually applied to every systematically edited method. Thus, automated refactoring is unlikely to obviate systematic editing.
- Previous studies find that clone refactoring is not necessary or feasible, but they did not construct an automated refactoring tool [1, 5, 8, 16, 17]. The lack of automation introduces potential subjectivity bias. By automating refactoring, our study improves on the prior methodology and shows that refactoring is often feasible.

II. MOTIVATING EXAMPLE

This section overviews our approach with an example based on `org.eclipse.compare.CompareEditorInput` revisions v20061120 and v20061218. Figure 1 shows a systematic edit on two methods. The unchanged code is in black, added code is in **blue** with '+', and deleted code is in **red** with '−'. The two methods perform very similar input processing and experience similar edits: adding a variable declaration and updating statements. However, the changes involve using different type, method, and variable names: `IActionBars` vs. `ISLocator`; `getActionBars` vs. `getServiceLocator`; `findActionBars` vs. `findSite`; `offset` vs. `offset2`; and `actionBars` vs. `sLocator`.

Given two changed methods, our refactoring tool (RASE) first invokes LASE [23], which creates an abstract edit script.

```
1. public class CompareEditorInput {
2.   private ICompareContainer fContainer;
3.   private boolean fContainerProvided;
4.   private Splitter fComposite;
5.   public IActionBars getActionBars (int offset) {
6.     if (offset == -1)
7.       return null;
8. -   if (fContainer == null) {
9. +   IActionBars actionBars = fContainer.getActionBars();
10.+   if (actionBars==null&&offset!=0&&!fContainerProvided){
11.      return Utilities.findActionBars(fComposite, offset);
12.    }
13.-   return fContainer.getActionBars();
14.+   return actionBars;
15.  }
16.  public ISLocator getServiceLocator (int offset2) {
17.-   if (fContainer == null) {
18.+   ISLocator sLocator = fContainer.getServiceLocator();
19.+   if(sLocator == null&&offset2!=0&&!fContainerProvided){
20.      return Utilities.findSite(fComposite, offset2);
21.    }
22.-   return fContainer.getServiceLocator();
23.+   return sLocator;
24.  }
25.}
```

Fig. 1. An example of systematic changes based on org.eclipse.compare.-CompareEditorInput from revisions v20061120 and v20061218

```
1. … …method_declaration(… …) {
2.    … …
3.    INSERT: T$0 v$0 = fContainer.m$0();
4.    UPDATE: if (fContainer == null) {
5.       TO: if (v$0==null && v$1!=0 && !fContainerProvided){
6.      … …
7.    }
8.    UPDATE: return fContainer.m$0();
9.       TO: return v$0;
10.}
```

Fig. 2. Abstract edit script inferred by LASE

```
1.   T$0 v$0 = fContainer.m$0();
2.   if (v$0==null && v$1!=0 && !fContainerProvided) {
3.     return Utilities.m$1(fComposite, v$1);
4.   }
5.   return v$0;
```

Fig. 3. Abstract refactoring template of common code created by RASE

The script describes abstractly the edit applied to both methods. It represents edit operations with AST node inserts, updates, moves, and deletes. Figure 2 shows the inferred abstract edit script for this example.

Given an edit script, RASE identifies edited statements related to the systematic changes. It uses the ranges of edits to scope its automated factorization and generalization, extracting the maximum common contiguous clone which encompasses all systematically edited statements. If similar edits are surrounded by cloned statements, RASE expands the refactoring scope to the entire method. For our example, in Figure 1, RASE selects lines 9-12, 14, 18-21, and 23 to refactor. Note that RASE includes the unchanged lines 11-12 and 20-21 in order to extract syntactically valid `if` statements.

Next, RASE creates an abstract refactoring template for the selected code snippets by matching expressions and identifiers between them, as shown in Figure 3. It uses the original code when identifiers or expressions are identical and otherwise abstracts them (e.g., `offset` vs. `offset2`). It records a map of abstract names to their original concrete identifiers and expressions to use later.

Based on the template, RASE creates an executable refactoring plan and applies it to transform code, as shown in Figure 4. RASE performs a *parameterize type* refactoring because the type variation `T$0` must be handled to work correctly for the different type variables. The method variations `m$0` and `m$1` require a *form template method* refactoring to invoke the correct methods depending on the callers. The variation in the use of a variable name `v$1` requires the corresponding variable to be passed as a parameter to the extracted method. The variable wildcard `v$0` does not need such processing, because the variable is locally defined and used, and thus invisible to the extracted method's callers. RASE performs static analysis that determines identifier scopes to differentiate these cases.

III. OPPORTUNISTIC REFACTORING

RASE takes systematically edited methods as input. It works in two phases. Phase I scopes code regions for refactoring, analyzes variations between them, and outputs an *abstract refactoring template*. Phase II constructs and then applies an executable refactoring plan by handling type, method, variable, and expression variations in the template and by analyzing control flow, data flow, and the class hierarchies of original methods. RASE uses a combination of six different refactoring operations to extract common code and parameterize differences while preserving semantics.

A. Phase I: Abstract Template Creation

We use LASE to create an abstract edit script that describes the input systematic changes [23]. LASE represents the difference between before and after versions with AST node inserts, deletes, updates, and moves. It then extracts the common changes among all methods, and creates a generalized program transformation. We call this transformation an *abstract edit script*. It contains a code pattern describing the context where the edit is applicable, and a list of edit operations describing how to make the edit. It also abstracts identifiers used in the exemplar edits to generally represent those edits even though they manipulate different identifiers. Figure 2 shows an exemplar edit script.

To refactor code undergoing systematic edits, RASE identifies the maximum code clone enclosing the edit in each method's new version. RASE requires contiguity in the code clones, thus it identifies a single AST node (and all its child subtrees) or a set of contiguous subtrees under the same parent node. These restrictions guarantee that there is only one entry to the code region and the cloned code can be extracted as a method. The maximum code clone identification algorithm consists of three steps: merge, abstract, and expand.

1) Merge: RASE creates an initial subtree set by identifying all trees rooted at the edited code. For instance, if a return-statement is edited, RASE selects the return-statement itself. However, given an edited condition in an if-statement, RASE selects the if-statement, which includes the conditional and the subtrees rooted at the then-branch and else-branch. It then creates a contiguous region of AST nodes by merging trees until there is a single tree left or a sequence of adjacent ones under the same parent node in the set. The merging algorithm picks two subtrees, T_1 and T_2 with the longest paths from the root, such as $path(Root, N_1, N_2, T_1)$ and $path(Root, N_1, N_3, T_2)$. Note that N_1, N_2, and N_3 represent parent and ancestor nodes of $T1$ or $T2$. RASE identifies the lowest common ancestor where the two paths diverge, N_1 in this case. Next, it adds all subtrees of N_1, such as trees rooted at N_2 and N_3, into the set of extractable code. For conciseness, it also moves the original T_1 and T_2 out of the set because they are now covered by the newly added trees. In this way, RASE makes the code regions to extract closer to each other. By merging subtrees iteratively, RASE finally forms the minimum contiguous code region involved in the edit.

RASE relies on systematic edits to scope refactoring. If a systematic edit only deletes code or it cannot find edited code in any method's new version, RASE will not refactor.

2) Abstract: RASE then tries to create an abstract template to guide further refactoring. To successfully create such a template, RASE requires that (a) the code regions extracted from different methods have the same number of statements,

Newly created classes and methods through clone removal

```
1. public abstract class TemplateClass<T0>{
2.   public T0 extractMethod(int v1, Splitter fComposite,
3.       ICompareContainer fContainer,
4.       boolean fContainerProvided){
5.     T0 v0 = m0(fContainer);
6.     if (v0 == null && v1 != 0 && !fContainerProvided){
7.       return m1(fComposite, v1);
8.     }
9.     return v0;
10.  }
11.  public abstract T0 m0(ICompareContainer fContainer);
12.  public abstract T0 m1(Splitter fComposite, int v1);
13.}
14.public class ConcreteTemplateClass0 extends
15.    TemplateClass<IActionBars>{
16.  public IActionBars m0(ICompareContainer fContainer){
17.    return fContainer.getActionBars();
18.  }
19.  public IActionBars m1(Splitter fComposite, int v1){
20.    return Utilities.findActionBars(fComposite, v1);
21.  }
22.}
23.public class ConcreteClass1 extends
24.    TemplateClass<ISLocator>{
25.  public ISLocator m0(ICompareContainer fContainer){
26.    return fContainer.getServiceLocator();
27.  }
28.  public ISLocator m1(Splitter fComposite, int v1){
29.    return Utilities.findSite(fComposite, v1);
30.} }
```

Modifications to the original methods

```
1. public class CompareEditorInput {
2.   private ICompareContainer fContainer;
3.   private boolean fContainerProvided;
4.   private Splitter fComposite;
5.   public IActionBars getActionBars (int offset) {
6.     if (offset == -1)
7.       return null;
8.     return new ConcreteTemplateClass0().extractMethod(
9.         offset, fComposite, fContainer,
10.        fContainerProvided);
11.  }
12.  public ISLocator getServiceLocator (int offset2) {
13.    return new ConcreteTemplateClass1().extractMethod(
14.        offset2, fComposite, fContainer,
15.        fContainerProvided);
16.} }
```

Fig. 4. Code refactoring based on systematic edits

and (b) the statements are either identical or differ only in their use of types, methods, variables, and expressions. Requirement (a) guarantees that the template reflects the skeleton of all extracted code. Requirement (b) guarantees that we extract syntactically similar code.

RASE abstracts any differences in type names, method invocations, variable names, and expressions in the target methods by using wildcards `T$`, `m$`, `v$` and `u$` respectively. It attempts to establish a mapping between each concrete identifier and the abstract version, making sure all methods consistently use and define these identifiers. If not, RASE does not refactor them. This analysis checks for syntactic equivalence and consistent def-use relations between the methods.

3) Expand: To extract as much common code as possible between similarly changed methods, RASE expands the identified code clones by tentatively including the subtrees' parent nodes or siblings. For instance, if the identified code clones from different methods have similar parent nodes as well as siblings, RASE expands the refactoring scope to the tree rooted at the parent node, and then updates the abstract template without invalidating any established concrete-abstract mappings. RASE applies Steps 3) and 2) iteratively until no more common code is appended.

B. Phase II: Clone Removal Refactoring

Based on the abstract template and identifier mappings described in the previous section, RASE leverages control and data flow analysis to determine how to extract common code and parameterize differences without altering semantics. It creates and applies an executable refactoring plan, which consists of one or more of these six refactoring operations:

- **extract method** extracts common code into a method.
- **add parameter** handles variations in variables and expressions.
- **parameterize type** handles variations in types.
- **form template method** handles variations in method calls.
- **introduce return object** handles multiple output variables of extracted code.
- **introduce exit label** preserves control flow in the original code.

Type Variations: Given a type wildcard (`T$`) in the abstract template, RASE applies a **parameterize type** refactoring. It declares a generic type for the newly created class and modifies each original location to call the extracted method with type parameters. We define this new term because Fowler's catalog does not include it and current refactoring engines, such as Eclipse, do not support it. Figure 5 shows an example. When the target code differs in terms of type identifiers, RASE adds explicit type parameters to the new extracted method. The applicability of this refactoring is affected by language support for generic types. In our implementation for Java, the refactoring is not applicable, when any parameterized type creates an instance by calling its constructors (e.g., `new T()`), performs an `instanceof` check (e.g., `v instanceof T`) or gets the type literal (e.g., `T.class`), because Java does not support these cases. Even if developers may handle such cases manually with smart tricks, the resulting refactored code would have poor readability.

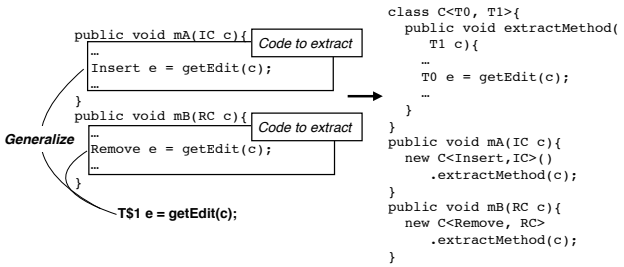

Fig. 5. Parameterize type refactoring

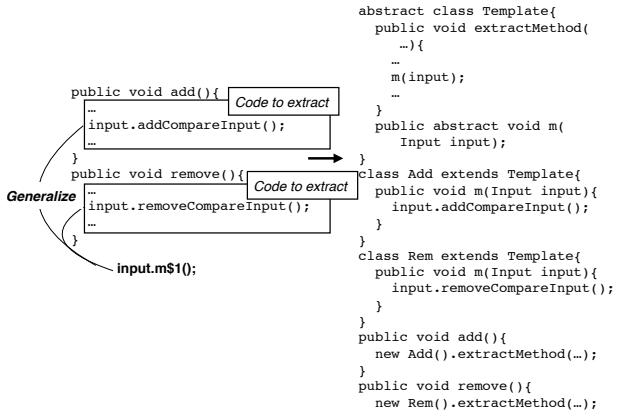

Fig. 6. Form template method refactoring

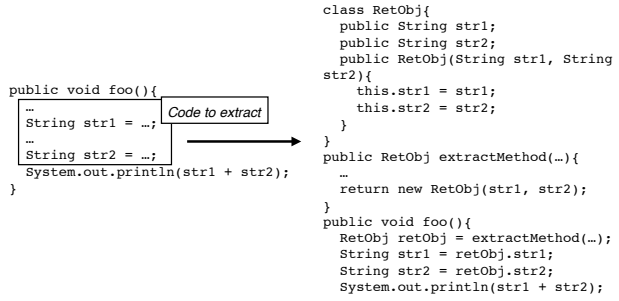

Fig. 7. Introduce return object refactoring

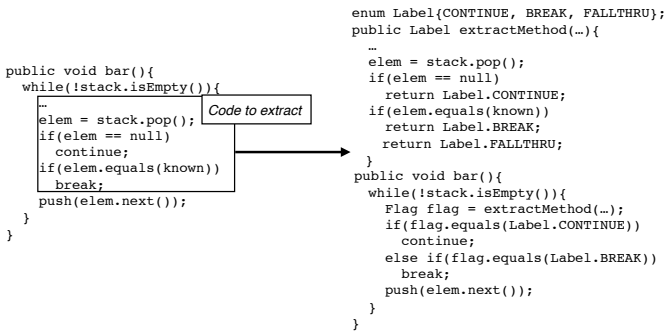

Fig. 8. Introduce exit label refactoring

Method Call Variations: Given a method wildcard (`m$`) in the abstract template, RASE applies the **form template method** (pg. 345 in [7]) refactoring. It creates uniform APIs that encapsulate the variations and changes the extracted method to invoke these APIs instead. Figure 6 shows an

example. RASE declares an abstract class which contains the extracted method and a sequence of abstract methods. Each abstract method corresponds to a method wildcard. For each original location, the refactoring declares a concrete class extending the abstract class so that all abstract methods are implemented to call the correct corresponding concrete methods. Each original location is modified to invoke the extracted method with the corresponding concrete class.

When a method wildcard represents a non-static method and is invoked via an object (e.g., `input.m$1()`), the corresponding method is declared to place the receiver object (e.g., `input`) as an argument (e.g., `m1(Input input)`). Then the actual method (e.g., `input.addCompareInput()` or `input.removeCompareInput()`) is invoked correctly inside each newly defined method. If any of the variant methods does not have a modifier `public` in its method declaration, the refactoring is not applied, because the method is not accessible by newly defined methods in the template class. If variant methods have different numbers of parameters, e.g., `foo(int offset)` vs. `bar(Object obj, boolean flag)`, the refactoring is not applied. Although it is possible to create a long method signature by merging different input signatures, we believe the resulting code is too hard to read.

Variable and Expression Variations: Given variations in variable names and expressions, RASE uses an **add parameter** (pg. 275 in [7]) refactoring. We use data dependence analysis to identify variables which have local uses but no local definitions in the extracted code. We consider variable wildcards (`v$`) as candidates for input arguments of the extracted method. For each variable wildcard, we check whether it is purely local to the extracted code, meaning that it is declared, defined, and used only in the extracted code. If so, RASE assigns it a concrete identifier and does not include the variable as an input parameter, since it is invisible to caller methods. For example, RASE declares the variable `v$1` in Figure 3 as a parameter but does not include `v$0` because it is purely local.

We consider expression wildcards (`u$`) as candidates for input arguments of the extracted method. Since the wildcards map to different AST node type expressions in different methods, each caller can pass appropriate expressions as arguments. For instance, if `u$` is mapped to `getConfiguration()` in one method, but is mapped to `fCompareEditorInput.getConfiguration()` in another method, RASE compares the types of both expressions. If the types are the same, it declares an input parameter with the common type. If the types are different, RASE records the type mapping and later applies **parameterize type** refactoring to accommodate the variation. Compared with prior work [3, 14], which solves expression variations by declaring new methods, our approach creates cleaner code by avoiding extra method declarations and invocations.

Return Value: RASE uses data dependence analysis to determine the variables that have local definitions and external uses. It converts these variables to output variables of the extracted method. When there is more than one such variable, RASE applies **introduce return object** refactoring. As shown in Figure 7, RASE encapsulates all return values into one object and inserts code at each call site to read appropriate fields of the returned object. Fowler's catalog does not include this refactoring and current refactoring engines do not support it.

Control Flow: RASE uses control flow analysis to determine the statements that exit the code in addition to the fall-through exit in the extracted code, such as a `return`, `break`, or `continue`. These non-local jump nodes either terminate execution or jump execution from one location to another. Naively putting them in the extracted method may cause compiler errors or incorrect control flow. RASE applies **introduce exit label** refactoring to correctly implement non-local jumps. It replaces non-local jumps with exit label return statements and modifies each original location to interpret the return labels. Fowler's catalog does not include this refactoring, nor is it implemented in current refactoring engines, such as Eclipse. We borrow the approach from Komondoor and Horwitz's prior work on automated procedure extraction [19]. Figure 8 shows an example with multiple exits. RASE replaces a non-local jump statement with a return statement to terminate the execution of the current method and adds a return label indicating the exit type. RASE inserts code at each call site to handle non-local jumps correctly.

Placing Extracted Code: RASE uses class hierarchy analysis to discover the relationship between classes declaring the originally edited methods. The relationships help RASE decide where to put the extracted method, and which input parameters or output variables to add. For instance, if the systematically changed methods are in the same class and there are no method or type wildcards in the abstract template, RASE places the extracted method in the same class. If the methods are in sibling classes extending the same super class, RASE puts the extracted method into their common super class. If the methods are in classes which do not have any type hierarchy relation, RASE must put the extracted method into a newly declared class. All fields that the extracted code reads from or writes to should be passed as input parameters and output variables separately, since they may not be accessible to the extracted method defined by the newly defined class. For correctness, RASE checks that (1) all methods invoked by the extracted method are declared as `public`, and (2) none of the method calls are unmovable, such as `super()`.

IV. EVALUATION

This section evaluates RASE with systematic editing tasks. It explores if automated refactoring eliminates the need for systematic editing and if systematic editing guides the scope of refactoring better or worse than method clones. It also takes a first look at whether automated refactoring is desirable when it is feasible.

Our data set consists of 56 similarly changed method pairs and 30 similarly changed method groups. These real-world systematic editing tasks are drawn from version histories of `JEdit`, Eclipse `compare`, `jdt.core`, `core.runtime`, `debug`, `JFreeChart` and `elasticsearch`. The method pairs are drawn

from prior evaluation of systematic editing [23, 24]. Each pair of methods have at least 40% syntactic similarity and share at least one common AST edit operation. Most are multi-line edits and require identifier abstraction. Each method group contains at least three similarly changed methods.

We use four variants of refactoring for our evaluation. The default RASE refactors as much code as possible given a systematic edit. RASE$_{min}$ chooses the smallest amount of code that includes the systematic edit. RASE$_{MA}$ refactors the entire method after the edit and RASE$_{MB}$ refactors the entire method before the edit. We apply RASE and its variants to the test suites. Table I and Table II present the results.

In the tables, each task has a unique identifier **ID**. If a task is automatically refactored, we characterize the refactoring with edit operations (**edits**), refactoring **types**, and resulting code size change (Δ**code**). RASE applies the following six refactoring types: E: extract method, R: introduce return object, L: introduce exit label, T: parameterize type, F: form template method, and A: add parameter. N/A means refactoring is not automated. We omit pairs with no refactoring in any configuration. In Δ **code size** column, a positive number means that refactoring increases the code size and a negative number means that code size decreases.

A. Method Pairs

Default RASE (columns 2-4) automatically refactors 30 out of 56 cases. RASE$_{MA}$ (columns 5-7 in the middle) is restricted to the method scope after edits and automates refactoring for nineteen cases, a strict subset of those refactored by RASE. RASE$_{MB}$ (columns 8-10 on the right) automates refactoring of methods before editing for eighteen cases, all of which are refactored by both RASE and RASE$_{MA}$. These 18 method pairs are clones, which experience similar changes and produce clones. Case 2 is not handled by RASE$_{MB}$, because the original version has no statements in the method and thus no clones can be extracted. This comparison shows that systematic edits better scope refactoring and increase refactoring opportunities compared to applying clone removal to the entire methods either before or after edits.

RASE$_{min}$ extracts the minimum common code enclosing systematic edits, as opposed to the maximum common code in default RASE. If we mark the minimum common code for extraction, we may have fewer variations between counterparts, which may cause less extra code added as necessary for specialization. On the other hand, we may extract less code than the actual commonality shared between changed methods, leaving redundant code after refactoring. The comparison between RASE$_{min}$ and RASE shows that RASE$_{min}$ performs differently from RASE in eight cases. In seven of the eight cases, RASE$_{min}$ is less effective at reducing code size because less common code is extracted. However in case 9, RASE$_{min}$ reduces code size more, because the extracted method does not include control flow jumps, which eliminates the need for code to interpret various flow jumps.

In 6 out of the 30 cases, RASE only uses the *extract method* to perform its refactoring tasks. All the other cases need a combination of different types of refactoring. The code size change varies between an increase of 11 lines and a decrease of 47 lines. RASE's automated refactoring reduces the code size in eight cases (27%) for the method pairs.

TABLE III
REASONS RASE DOES NOT REFACTOR 26 CASES: METHOD PAIRS

Reason	number of cases
Limited language support for generic types	7
Unmovable methods	5
No edited statement found	8
No common code extracted	6

TABLE IV
REASONS RASE DOES NOT REFACTOR 10 CASES: METHOD GROUPS

Reason	number of cases
Limited language support for generic types	2
Unmovable methods	0
No edited statement found	2
No common code extracted	6

B. Method Groups

To explore whether our conclusions based on method pairs generalize to multiple similarly changed methods, we apply RASE to 30 systematically edited method groups. Each group contains at least three methods and at most nine methods. We apply RASE$_{MA}$, RASE$_{MB}$, and RASE$_{min}$ to the same data set and compare refactoring capabilities. The results are mostly similar comparing Table I and Table II. The column labeled **#** in Table II shows the number of changed methods in each group. RASE refactors 20 out of 30 cases. Similar to Table I, we observe that RASE automates refactoring more than RASE$_{MA}$ and RASE$_{MB}$. RASE produces more concise code than RASE$_{min}$ in 6 out of 30 cases. One difference from the method pair results is that RASE decreases code size more consistently and frequently, reducing code size in 14 of the 20 refactored cases (70%) and on average reducing code by eight lines. This result is expected because the refactored code appears in just two methods with method pairs, whereas for method groups, the refactored code originally appears in three or more methods.

C. Reasons for Not Refactoring

We examined by hand the 26 method pairs that RASE did not refactor and found four reasons, which Table III summarizes. For seven cases, RASE failed to refactor due to Java's limited support for generic types. It is very difficult to convert some generalized statements like `v instanceof T$`, `T$.m$()`, and `v = new T$()`, into code that compiles.

For five cases, RASE did not refactor because some statements cannot be moved correctly into an extracted method. For instance, the super constructor `super(...)` is only valid in constructors and cannot be moved to any other method. Another example is, when attempting to put an extracted method into a newly declared class, calls to `private` methods

TABLE I
METHOD PAIRS: CLONE REMOVAL REFACTORINGS

ID	RASE edits	RASE types	RASE Δcode	RASE $_{min}$ edits	RASE $_{min}$ types	RASE $_{min}$ Δcode	RASE $_{MA}$ edits	RASE $_{MA}$ types	RASE $_{MA}$ Δcode	RASE $_{MB}$ edits	RASE $_{MB}$ types	RASE $_{MB}$ Δcode
2	15	E, A	-1	15	E, A	-1	15	E, A	-1		N/A	
4	6	E, A	2	6	E, A	2	6	E, A	2	6	E	2
6	14	E, F	10	14	E, F	10	14	E, F	10	31	E, F	11
9	77	E, R	-7	61	E	-15		N/A			N/A	
10	24	E	-4	20	E, L	8	24	E	-4	15	E	-1
11	20	E, F	8	20	E, F	8	20	E, F	8	14	E, F	10
12	31	E, F	11	31	E, F	11	31	E, F	11	14	E, F	10
13	38	E, F	2	32	E, F	4	38	E, F	2	29	E, F	5
18	42	E	-10	7	E	3		N/A			N/A	
19	61	E	-15	21	E, R	13	61	E	-15	61	E	-15
22	285	E, F	-47	285	E, F	-47	285	E, F	-47	288	E, F	-48
29	56	E, L, R	4	45	E, L, R	9		N/A			N/A	
32	9	E, A	1	6	E	2		N/A			N/A	
34	24	E, A	-4	24	E, A	-4		N/A			N/A	
35	9	E	1	9	E	1		N/A			N/A	
36	36	E, A	-8	36	E, A	-8		N/A			N/A	
38	16	E	0	12	E	0		N/A			N/A	
40	20	E, L	8	20	E, L	8		N/A			N/A	
45	6	E	2	6	E	2		N/A			N/A	
46	6	E, A	2	6	E, A	2	6	E, A	2	6	E	2
47	20	E, L, R	8	20	E, L, R	8		N/A			N/A	
48	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
49	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
50	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
51	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
52	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13	25	E, F, T	13
53	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
54	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
55	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
56	31	E, F	11	31	E, F	11	31	E, F	11	28	E, F	12
Average	35.5		2.4	31.5		4.1	39.4		4.0	38.9		4.9
Total automated	30			30			19			18		

TABLE II
METHOD GROUPS: CLONE REMOVAL REFACTORINGS

ID	#	RASE edits	RASE types	RASE Δcode	RASE $_{min}$ edits	RASE $_{min}$ types	RASE $_{min}$ Δcode	RASE $_{MA}$ edits	RASE $_{MA}$ types	RASE $_{MA}$ Δcode	RASE $_{MB}$ edits	RASE $_{MB}$ types	RASE $_{MB}$ Δcode
1	6	137	E, A, F, T	-7	137	E, A, F, T	-7	137	E, A, F, T	-7	137	E, A, F, T	-7
2	4	30	E	-10	30	E	-10	30	E	-10	10	E	2
4	3	17	E	-1	10	E, A	4		N/A			N/A	
5	7	36	E, T	-6	16	E	2		N/A			N/A	
6	8	42	E, T	-6	42	E, T	-6		N/A			N/A	
8	3	44	E, A, F	-4	44	E, A, F	-4	44	E, A, F	-4	32	E, A, F	2
9	5	58	E, L, R	18	58	E, L, R	18		N/A			N/A	
10	3	38	E, F	14	38	E, F	14	38	E, F	14	19	E, A, F	13
11	4	20	E	-4	10	E	2		N/A			N/A	
13	3	9	E	3	9	E	3		N/A			N/A	
15	3	32	E, A	-10	28	E	-8		N/A			N/A	
17	3	21	E	-3	9	E	3		N/A			N/A	
18	3	37	E	-11	37	E	-11	37	E	-11	25	E	-5
19	3	96	E, F	6	48	E, F, R	24	96	E, F	6	29	E, A, F, T	13
24	3	59	E, R	-1	59	E, R	-1	59	E, R	-1	8	E	2
25	3	26	E, R	14	26	E, R	14		N/A			N/A	
27	4	20	E, A	-4	20	E, A	-4		N/A			N/A	
28	3	24	E, T	0	24	E, T	0	24	E, T	0	12	E, A, T	0
29	9	211	E	-149	211	E	-149		N/A			N/A	
30	4	26	E, A	-6	26	E, A	-6	26	E, A	-6	15	E, A, T	7
Average		49.2		-8.4	44.1		-6.1	54.5		-2.1	31.9		3.0
Total automated		20			20			9			9		

Cases IDs from Meng et al. RASE by default includes as much code as possible. RASE $_{min}$ chooses the smallest scope that includes the systematic edit. RASE $_{MA}$ refactors the entire method after the edit and RASE $_{MB}$ refactors the entire method before the edit. The **edits** column is AST statement edit operations for refactoring. Refactoring **types** are: E: extract method, R: introduce return object, L: introduce exit label, T: parameterize type, F: form template method, and A: add parameter. RASE uses all the refactoring types in many combinations. Both tables show RASE automates refactoring in many cases: 30 out of 56 pairs, and 20 out of 30 method groups, but not all. Systematic edits scope clone removal opportunities better (RASE and RASE $_{min}$) than methods (RASE $_{MA}$ and RASE $_{MB}$).

by the extracted method are not semantically valid because `private` methods are only accessible for methods defined in the same class.

For eight cases, no edited statement is identified in the new version of each changed method. RASE depends on LASE to create an abstract edit script representing the input systematic changes. If LASE fails to create such an edit script or the edit script only deletes statements from old versions, RASE cannot locate code to extract, nor can it automate refactorings.

In six cases, the marked code snippets in different methods are not generalizable to create an abstract template. Four possible reasons explain this result. First, the code snippets contain different numbers of statements. Although some existing clone removal refactoring techniques leverage program dependence analysis and heuristics to shift irrelevant variant code and put together extractable code, these techniques cannot handle all the cases in this category either, indicating the difficulty of fully automated refactoring [12, 21]. Second, the AST node types of some extracted statements do not match, such as `ExpressionStatement` vs. `ReturnStatement`. Third, the number of parameters in method calls do not match, such as `foo(v)` vs. `bar(a, b, c)`. Although we may create a single method by merging input signatures of different methods, the resulting code may have poor readability. Fourth, there is at least one identifier mapping conflict. For instance, identifier `v` is mapped to an identifier `a` in one statement, but mapped to another identifier `x` in another statement. Ignoring these conflicts to apply refactoring would incorrectly modify the program semantics. Similar to Table III, Table IV shows that limited language support for generic types, no edited statement, and no common code extracted are the three main reasons.

In summary,

- Clone removal refactoring does not eliminate the need for systematic editing.
- Scoping refactoring based on systematic edits improves refactoring applicability over refactoring the entire methods, either before or after the edits.
- Extracting the maximum common code, instead of the minimum common code, usually creates a refactored version with a smaller code size.

D. Software evolution after systematic edits

To understand how RASE's refactoring recommendations correlate with developer refactorings, we manually examine how developers evolved methods after these systematic edits by going through the version histories. For our test suite, the average time interval in the version history, starting at the systematic editing version and ending at the latest version, is 1.3 years. Table V shows the results for method pairs and Table VI for method groups. The **Feasible** column corresponds to cases when RASE can automate clone removal refactoring and **Infeasible** corresponds to the rest. The **Refactored** row shows cases when developers either by hand or with the help of some other tool refactored code later in the version history. The other rows break down cases without developer refactoring. **Co-evolved** means the methods are systematically edited at least one more time in later versions and may indicate that refactoring is desirable. **Divergent** means the methods evolved in divergent ways and may indicate that refactoring is undesirable. For example, one method was deleted or only one method changed. Refactoring **Unchanged** methods may not be worthwhile because they are quite stable or it is premature to judge desirability due to lack of information.

TABLE V
MANUAL EVALUATION OF VERSION HISTORY AFTER SYSTEMATIC EDITS: METHOD PAIRS

		Feasible	Infeasible
Refactored		4	0
Unrefactored	**Co-evolved**	2	6
	Divergent	3	10
	Unchanged	21	10

TABLE VI
MANUAL EVALUATION OF VERSION HISTORY AFTER SYSTEMATIC EDITS: METHOD GROUPS

		Feasible	Infeasible
Refactored		1	0
Unrefactored	**Co-evolved**	2	1
	Divergent	4	0
	Unchanged	13	9

Table V shows that developers only refactored 4 out of 56 cases. RASE automates refactoring for the same 4 cases. Additionally, RASE refactors 26 cases which were not refactored by developers. In this test suite, 47 cases have version histories and 9 do not because they were specially crafted to test systematic editing [24]. Among these 26 cases not refactored by developers, 21 had no code changes, including the 9 without version histories. When code does not need to change to fix bugs or add features, developers are unlikely to aggressively remove clones. In 3 of 26 cases, developers evolved code differently by either changing both methods differently or deleting only one of the two methods. In 2 of 26 cases, developers did not refactor code for some reason, but similarly changed code once again. Such repetitive systematic edits on method pairs may indicate refactoring is desirable.

There are six cases in which methods were co-evolved by developers but are not automatically refactored by RASE. The major reason is the code invokes certain methods which are not accessible by an extracted method, contain non-contiguous cloned code, or have conflicting identifier mappings. It is not easy to automatically refactor these cases. If developers want to refactor them, they need to first apply some tricks to make the common code extractable.

Table VI summarizes the version history for systematically edited method groups and shows similar results to the method pairs in Table V. Developers refactored one case, which RASE also handles. Methods co-evolved in two cases and diverged in four cases, all of which RASE can refactor. There is one case where methods were co-evolved by only deleting code, but RASE does not refactor in this case.

Manually observing the version history reveals that there is no obvious correlation between the feasibility of RASE enabled refactoring and manual refactorings performed by developers on systematically edited code, although there are cases when automatic and manual refactoring overlaps. Whether developers refactor or not, they do not base their decision solely on code similarity and similar edits. They consider other factors, such as readability, code size, future plans for features, and bug fixes. Although RASE cannot decide for developers whether to refactor or not, by creating an executable refactoring plan, when developers decide to refactor, it helps reduce developer burden when applying code transformations.

To explore the reasons developers do not refactor while performing systematic edits, we randomly pick several examples and ask project owners for their expert opinion.

One developer is conservative about aggressive refactoring and merging commonality between methods: *"(I will not refactor because) this pair of methods is not a pain point during maintenance/evolution of JDT. That particular class is very stable, and the readability of the code as it is now outweighs potential benefits of refactoring. We have other duplications, that are more likely to cause pain, e.g., by being forgotten during maintenance. ... In these classes, potential gain might be greater, but then a refactoring to avoid redundancy would certainly introduce a significant amount of additional complexity. We don't typically refactor unless we have to change the code for some bug fix or new feature."*

Another developer refactors more proactively to reduce cloned code, but prefers reducing four duplicated methods to two, instead of the single method that RASE suggests to simplify the class hierarchy.

The feedback from developers illustrates that the decision to remove clones depends on many criteria including code similarity, co-evolution events, the effectiveness of cloned code in bug fixing and feature additions, the software architecture, readability, and maintainability of the resulting refactored code.

Based on our experience with software version history and communication with developers, we envision RASE as a refactoring recommendation tool when developers think about refactoring duplicated code. RASE will help further research on recommendations and cost/benefit analysis of clone removal. RASE should also serve to complement existing systematic editing tools because developers do not always aggressively reduce duplicated code but often maintain redundant code for various reasons.

V. THREATS TO VALIDITY

Our results are based on 86 systematic editing examples. Further evaluation with more subject systems, longer version histories, and larger scope of systematic edits beyond the method level remains as future work.

The refactoring capability of RASE is affected by the systematic editing tool—LASE—it uses. Given multiple similarly changed methods, if LASE fails to generalize an abstract edit script for them, RASE cannot provide any refactoring suggestion. The six types of refactorings implemented in RASE do not cover all possible code transformations applicable to clone removal. However, it is the state-of-the-art in terms of the number of clone removal refactorings it automates.

When handling variations in expressions, we promote expressions as input parameters of an extracted method. If the promoted expressions cause side effect, such as `i++`, we may alter semantics in some cases, although the alteration is not observed in our test suites.

RASE determines concrete refactoring transformations based on an abstract template without considering the global context such as the extent of code duplication across the entire codebase or the class relationship among methods. Therefore, RASE may not suggest the best possible transformation. However, RASE is the first automated tool that mechanically examines if systematic edits in multiple locations indicate refactoring opportunities.

Our results focus on automated refactoring *feasibility* instead of *desirability*. We leave developers to decide whether to refactor or not. We believe that it is difficult to choose between systematically editing methods and reducing clones to edit a single copy. To assess refactoring desirability, the refactoring cost/benefit analysis should account for factors such as how frequently future systematic edits may occur to fix bugs and add features, whether complexity increases, whether it is worthwhile to reduce future edits, and whether the related methods are likely to change and/or diverge in the future.

VI. RELATED WORK

Systematic Editing. Systematic editing tools automatically apply similar changes to multiple locations. Simultaneous text editing tools replicate the exact same users actions in one code region to other user-selected code regions [25, 30]. CloneTracker takes the output of a clone detector as input, maps corresponding lines in the clones, and then echoes edits in one clone to another upon a user's request [6]. The Clever version control system monitors code clones, detects changes to them, and then recommends edit propagation among clones [27]. SYDIT infers an abstract AST edit script from an exemplar changed method and applies it to user-selected methods [24]. LASE generalizes a partially abstract, context-aware edit script from multiple examples, uses the edit script to find additional edit locations, customizes the edit script to each new location, and then applies the result [23]. All these systematic editing tools automate repetitive changes to multiple locations, which may encourage the bad practice of creating and maintaining code duplications.

Clone Removal Refactoring. Based on code clones detected by various techniques [13, 15, 20], many tools identify or rank refactoring opportunities [2, 9]–[11, 32]. For instance, Balazinska et al. [2] define a clone classification scheme based on various types of differences between clones and automate the classification to help developers assess refactoring opportunities for each clone group. Higo et al. and Goto et al. rank clones as refactoring candidates based on coupling or cohesion metrics [9, 11]. Others integrate evolution information in

software history to rank clones that have been repetitively or simultaneously changed in the past [10, 32]. While these tools detect refactoring opportunities for clones, they do not automatically refactor code.

A number of techniques automate clone removal refactorings by factorizing the common parts and by parameterizing their differences using a *strategy* design pattern or a *form template method* refactoring [3, 12, 14, 21, 29]. Similar to RASE, these tools insert customized calls in each original location to use newly created methods. Juillerat et al. automate *introduce exit label* and *introduce return object* refactorings supported by RASE [14]. However, for variable and expression variations, Juillerat et al.'s approach and CloRT [3] define extra methods to mask the differences, while RASE passes these variations as arguments of the extracted method. CloRT was applied to JDK 1.5 to automatically reengineer class level clones. Similar to our results, they find this reengineering effort led to an increase in the total size of code because it created numerous simple methods. Hotta et al. use program dependence analysis to handle gapped clones—trivial differences inside code clones that are safe to factor out and such that they can apply the *form template method* refactoring to the code [12]. Krishnan et al. use PDGs of two programs to identify a maximum common subgraph so that the differences between the two programs are minimized and fewer parameters are introduced [21] . Unlike RASE, none of these tools handle type variations when extracting common code.

Automatic Procedure Extraction. Komondoor et al. extract methods based on the user-selected or tool-selected statements in one method [18, 19]. The *extract method* refactoring in the Eclipse IDE requires contiguous statements, whereas these tools handle non-contiguous statements. Program dependence analysis identifies the relation between selected and unselected statements and determines whether the non-contiguous code can be moved together to form extractable contiguous code. Similar to RASE, Komondoor et al. apply *introduce exit label* refactoring to handle exiting jumps in selected statements [19]. Tsantalis et al. extend the techniques by requiring developers to specify a variable of interest at a specific point only [31]. They use a block-based slicing technique to suggest a program slice to isolate the computation of the given variable. These approaches are only focused on extracting code from a single method. Therefore, they do not handle extracting common code from multiple methods and resolving the differences between them as RASE does.

Empirical Studies of Code Clones. Many empirical studies on code clones find that removing clones is not necessary nor beneficial [1, 5, 8, 16, 17]. Bettenburg et al. report that only 1% to 3% of inconsistent changes to clones introduce software errors, indicating that developers are currently able to effectively manage and control clone evolution [5]. Kim et al. observe that many long-lived, consistently changed clones are not easy to refactor without modifying public interfaces [17]. These empirical studies show that removing code clones is not always necessary nor beneficial. While these studies use longer version histories or larger programs than our evaluation, none of these studies, automatically refactor code to remove clones, as we do in this paper. Our work thus improves over their methodology by eliminating human judgment when determining the feasibility of edits.

VII. CONCLUSIONS

Similar edits in similar code may indicate an opportunity to remove redundancy. To investigate this question, we design and implement RASE, an automated refactoring tool that consists of six clone removal refactoring techniques: *extract method*, *parameterize type*, *form template method*, and *add parameter* to tackle variations in types, methods, variables, and expressions respectively; and *introduce exit label* and *introduce return object* to handle non-local jumps and multiple output variables.

By applying RASE to real-world systematic editing tasks, we observe that RASE improves refactoring feasibility by refactoring the region surrounding systematic edits as opposed to refactoring the entire method. This finding corroborates the community's understanding that the evolutionary characteristics of clones may be a better indicator for refactoring needs than the clones themselves. Despite this improvement, automated refactoring is feasible only in 58% of the cases in our test suite. We show it is very difficult to automate clone removal for the remaining 42%. Hand examination indicates that language, semantics, and lack of common code are main reasons when refactoring is infeasible, while refactoring is not always applied by developers even if refactoring is feasible. We conclude that developers need tool support for both systematic editing and automated refactoring.

While RASE automates clone removal based on systematic edits, the decision of whether to refactor or not depends on multiple complex factors such as readability, maintainability, and types of anticipated changes. Systematic edits serve only as one factor. Therefore, they are not sufficient to indicate clone removal is desirable. However, we believe that RASE's automated refactoring capability will support further research on refactoring cost/benefit analysis and recommendations.

VIII. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, CCF-1018271, CCF-0811524, CNS-1239498, and a Google Faculty Award.

REFERENCES

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *CSMR '07*, pages 81–90, 2007.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, page 292, 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*, page 326, 1999.

[4] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba. Supporting extract class refactoring in eclipse: The aries project. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[5] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *WCRE*, pages 85–94, 2009.

[6] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167, 2007.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[8] N. Göde. Clone removal: Fact or fiction? In *IWSC*, pages 33–40, 2010.

[9] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue. How to extract differences from similar programs? a cohesion metric approach. In *IWSC*, pages 23–29. IEEE, 2013.

[10] Y. Higo and S. Kusumoto. Identifying clone removal opportunities based on co-evolution analysis. In *IWPSE*, pages 63–67, 2013.

[11] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J. Softw. Maint. Evol.*, 20(6):435–461, 2008.

[12] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. *2011 15th European Conference on Software Maintenance and Reengineering*, 0:53–62, 2012.

[13] 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.

[14] N. Juillerat and B. Hirsbrunner. Toward an implementation of the "form template method" refactoring. *SCAM*, 0:81–90, 2007.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE*, pages 654–670, 2002.

[16] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196, 2005.

[18] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL*, pages 155–169, 2000.

[19] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *IWPC*, pages 33–, 2003.

[20] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, page 301, 2001.

[21] G. P. Krishnan and N. Tsantalis. Refactoring clones: An optimization problem. *ICSM*, 0:360–363, 2013.

[22] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth. Improving usability of software refactoring tools. In *Proceedings of the 2007 Australian Software Engineering Conference*, 2007.

[23] N. Meng, M. Kim, and K. McKinley. Lase: Locating and applying systematic edits. In *ICSE*, page 10, 2013.

[24] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[25] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *2002 USENIX Annual Technical Conference*, pages 161–174, 2001.

[26] M. Mortensen, S. Ghosh, and J. Bieman. Aspect-oriented refactoring of legacy applications: An evaluation. *IEEE Trans. Softw. Eng.*, 2012.

[27] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *ASE*, pages 123–134, 2009.

[28] D. Silva, R. Terra, and M. T. Valente. Recommending automated extract method refactorings. In *Proceedings of the 22Nd International Conference on Program Comprehension*, 2014.

[29] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, 2012.

[30] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC*, pages 173–180, 2004.

[31] N. Tsantalis. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, 2011.

[32] N. Tsantalis and A. Chatzigeorgiou. Ranking refactoring suggestions based on historical volatility. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, pages 25–34, Washington, DC, USA, 2011. IEEE Computer Society.

[33] S. A. Vidal and C. A. Marcos. Toward automated refactoring of crosscutting concerns into aspects. *J. Syst. Softw.*, 2013.

[34] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, 2011.