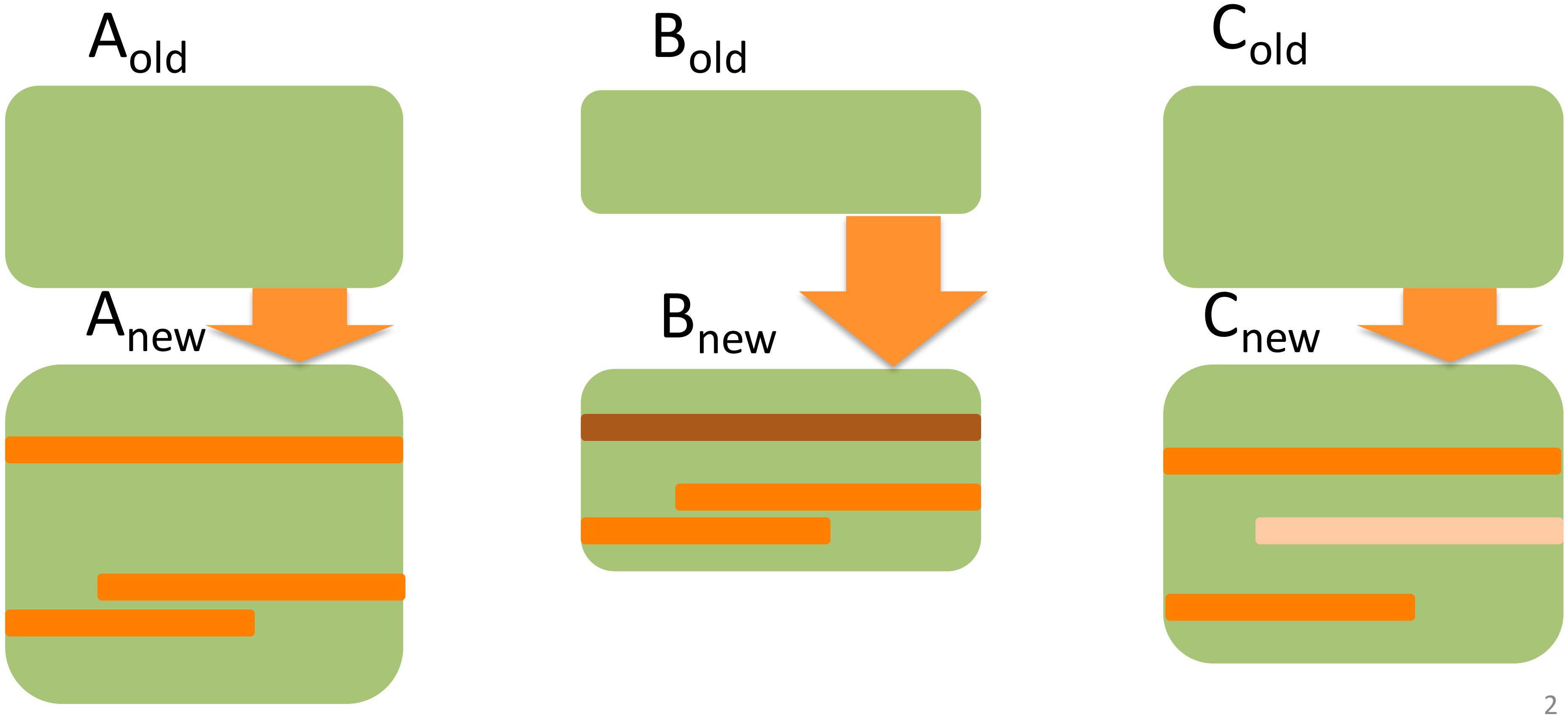
Does Automated Refactoring Obviate Systematic Editing?

Na Meng* Lisa Hua* Miryung Kim[+]
Kathryn S. McKinley[‡]

The University of Texas at Austin*
University of California—Los Angeles[+]
Microsoft Research[‡]

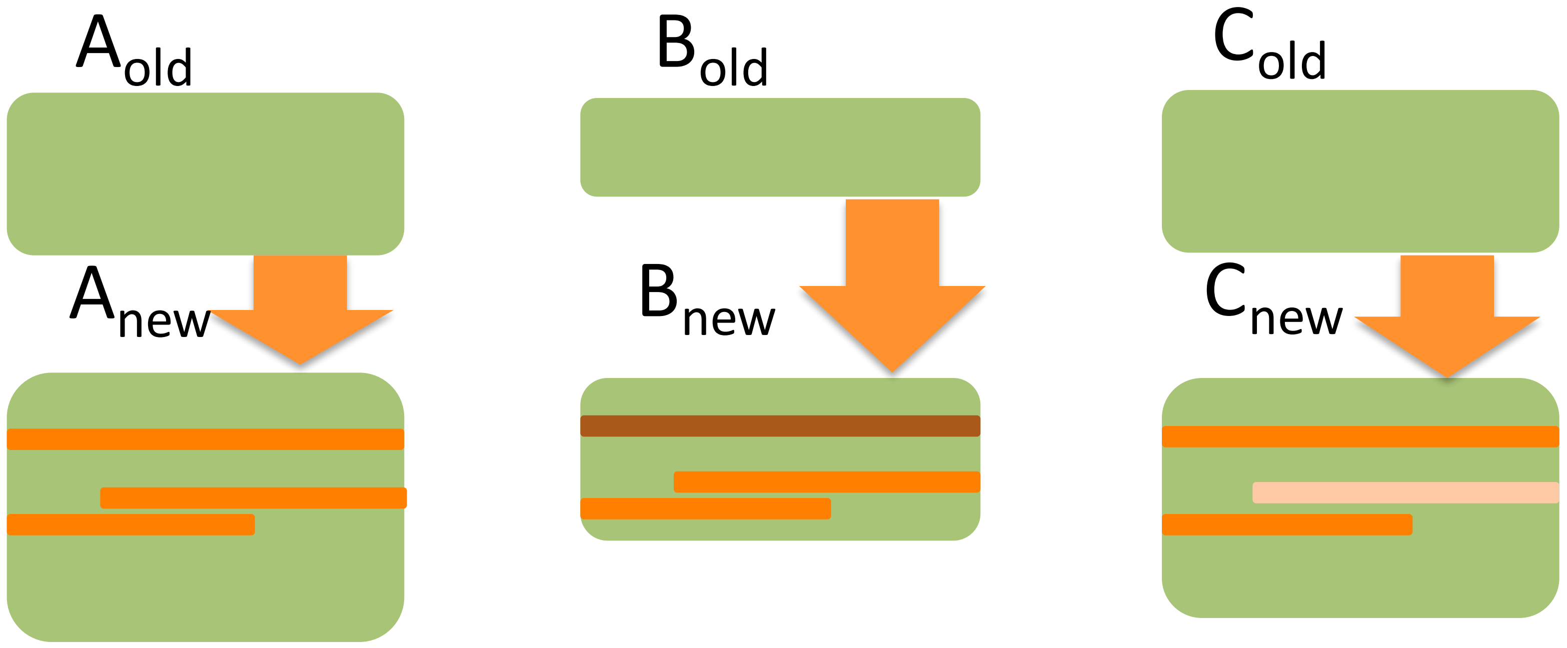
Motivating scenario

Pat needs to update database transaction code to prevent SQL injection attacks



Systematic editing tools

- Simultaneous text editing [2002], Linked Editing [2004], Clever [2009]
- Example-based program transformation [Meng et al.]

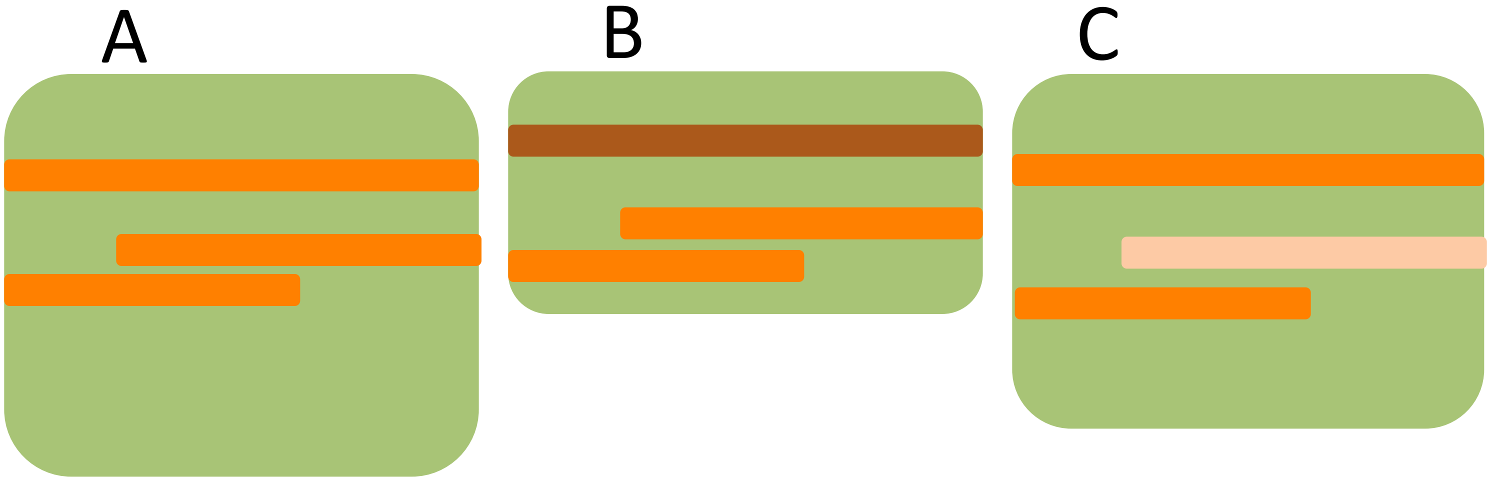


Systematic editing: Friend or foe?

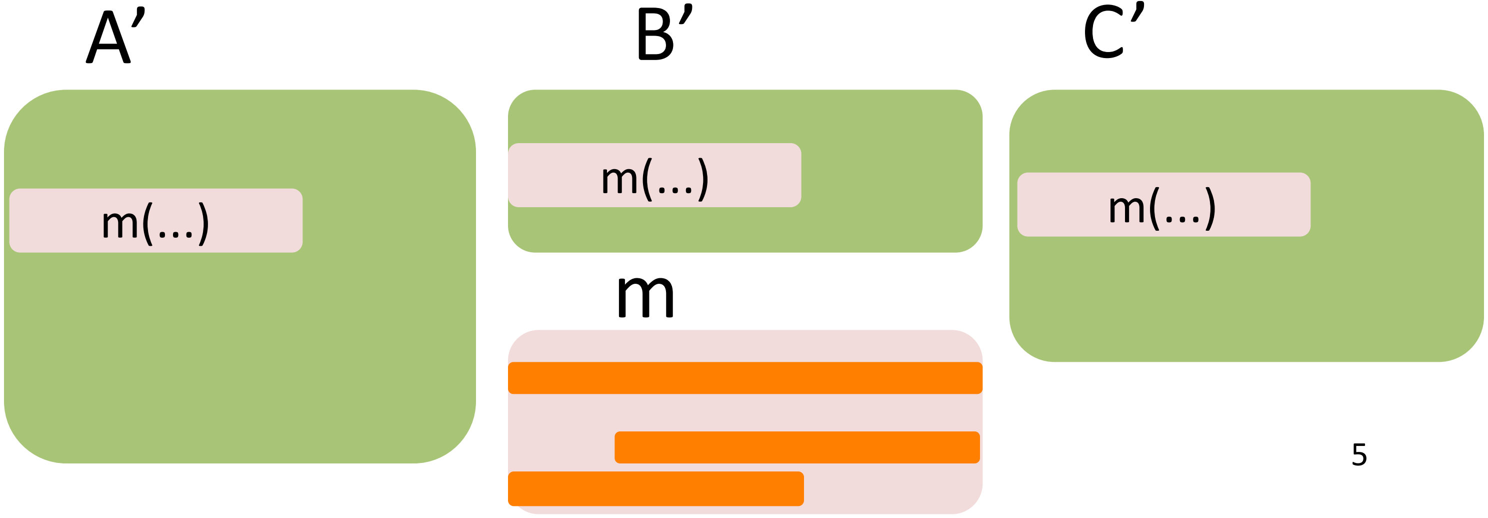
- **Friend**: Performs code change propagation
- **Foe**: Encourages code duplication

Code maintenance alternatives

Systematic editing



Clone removal refactoring

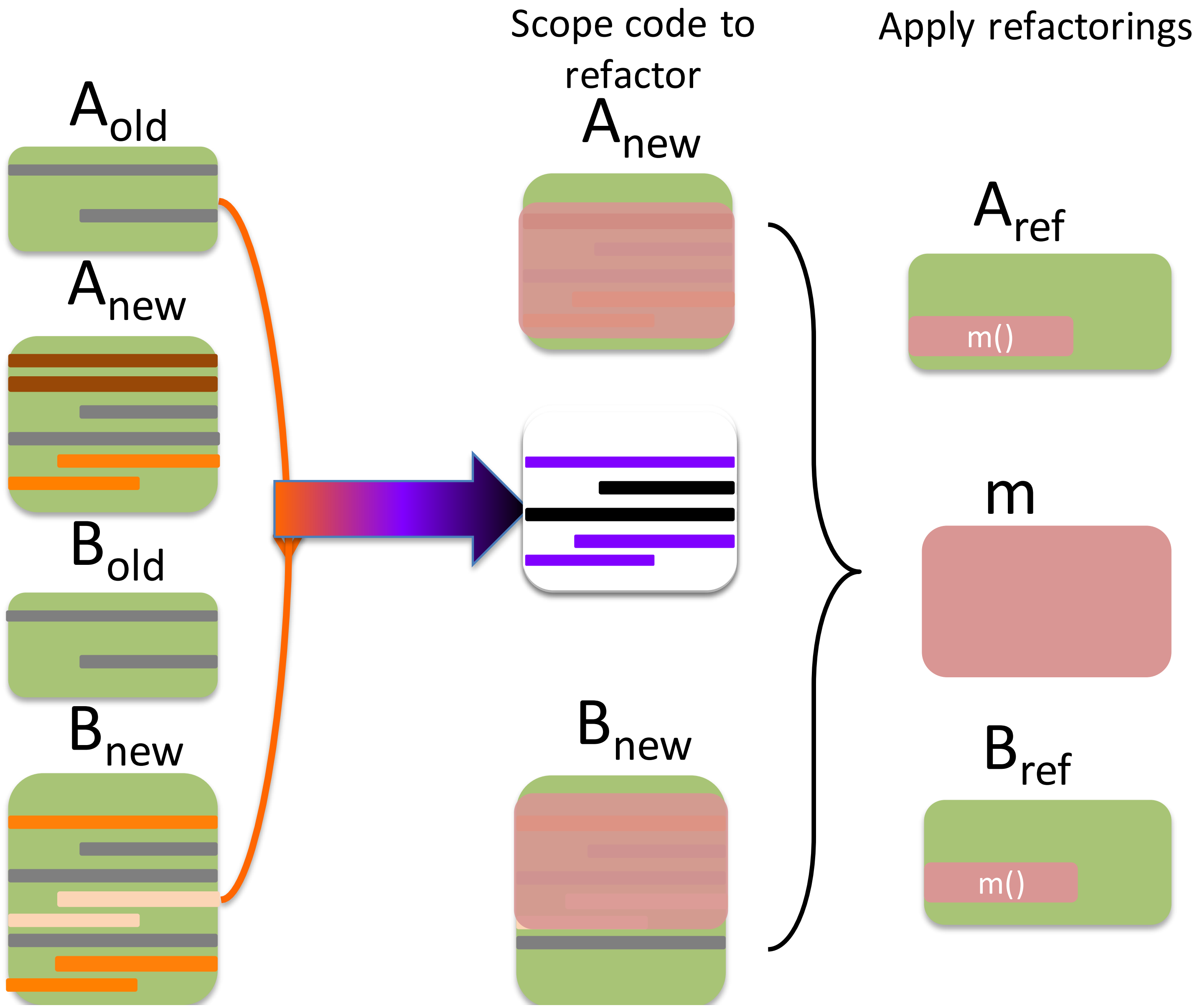


Does systematic editing encourage code duplication or should we remove code clones instead?

↓

We design a fully automated, clone removal refactoring technique

Rase: Exploiting systematic edits for clone removal refactoring

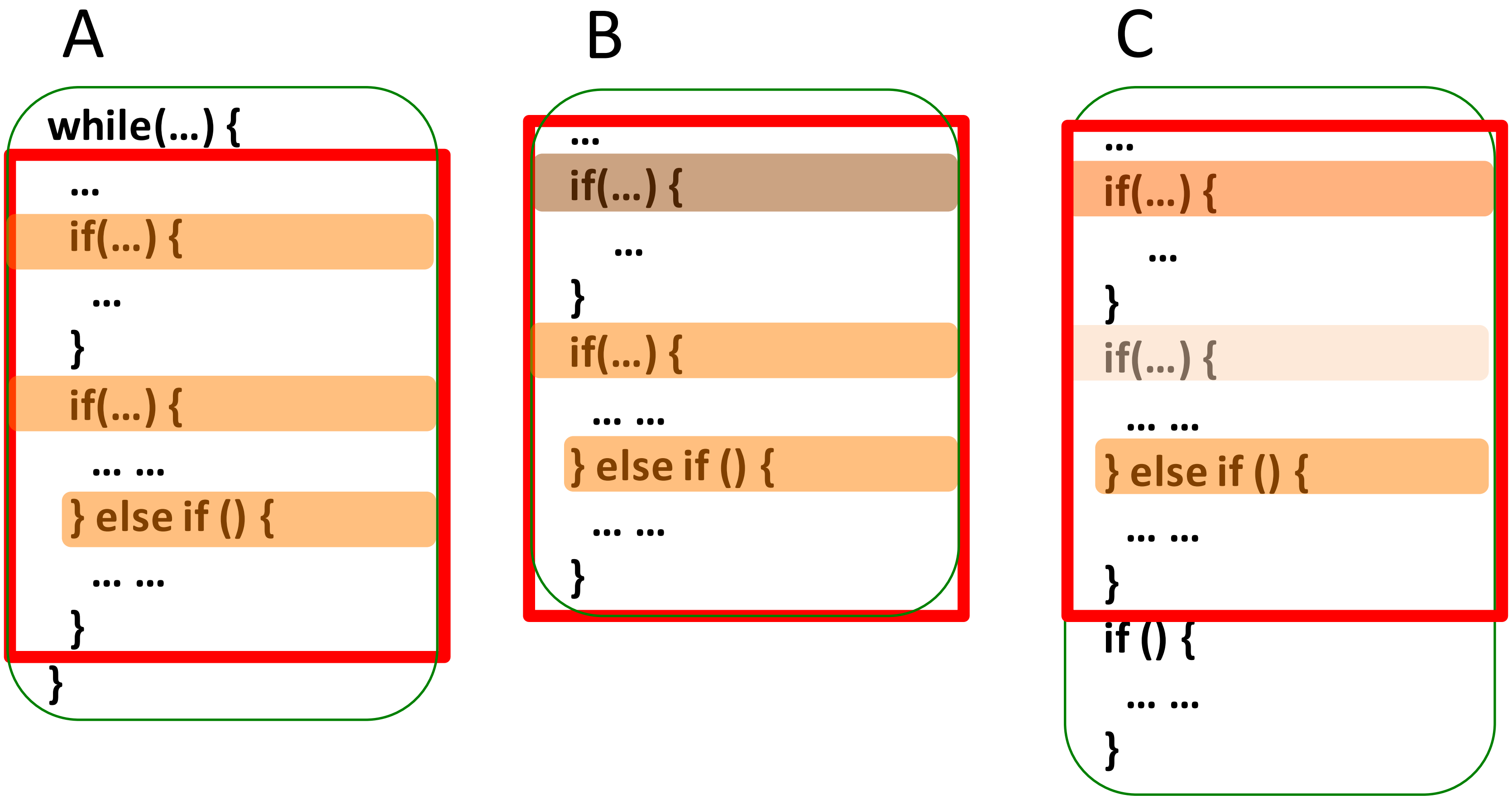


Rase Approach

- Input: Systematic edits
- Step 1: Scope refactoring region and analyze variations
- Step 2: Create and apply an executable refactoring plan
  - Extract method
  - Add parameter
  - Parameterize type
  - Form template method
  - Introduce return object
  - Introduce exit label

Step 1: Scope code to refactor

Refactor the maximum syntactically valid contiguous code clones enclosing edits



A

```
while(…) {
  …
  if(…) {
    …
  }
  if(…) {
    … …
  } else if () {
    … …
  }
}
```

B

```
…
if(…) {
   …
}
if(…) {
  … …
} else if () {
  … …
}
```

C

```
…
if(…) {
   …
}
if(…) {
  … …
} else if () {
  … …
}
if () {
  … …
}
```

Step 2: Create and apply an executable refactoring plan

Challenges to extract common code	**Refactorings**
Type variations	Parameterize type
Method variations	Form template method
Variable/Expression variations	Add parameter
Multiple variables to return	Introduce return object
Non-local jump statements	Introduce exit label

Type variations

```
public void A(IC c) {
  …
  Insert e = getEdit(c);
  …
}
```

Code to extract

```
public void B(RC c) {
  …
  Remove e = getEdit(c);
  …
}
```

Code to extract

Create generalized types

①Declare type parameters

```
class C<T0,T1>{
  public void extractMethod(T1  c){
    …
    T0 e = getEdit(c);
    …
  }
}
```

②Concretize the type usage

```
public void mA(IC c){
  new C<Insert,IC>().extractMethod(c);
}
public void mB(RC c){
  new C<Remove, RC>.extractMethod(c);
}
```

Method variations Form template methods

```
public void add() {
  …
  input.addCompareInput();
  …
}
```

Code to extract

```
public void remove() {
  …
  input.removeCompareInput();
  …
}
```

Code to extract

```
abstract class Template{
  public void extractMethod(…){
    …
    m(input);
    …
  }
  public abstract void m(Input input);
}
class Add extends Template {
  public void m(Input input){
    input.addCompareInput();
  }}
class Remove extends Template {
  public void m(Input input) {
    input.removeCompareInput();
  }
}
public void add(){
  new Add().extractMethod(…);
}
public void remove() {
  new Remove().extractMethod(…);
}
```

① Create a template method

② Dispatch function call

Multiple variables to output

Introduce return objects

Code to extract

```
public void foo() {
  …
  String str1 = …;
  …
  String str2 = …;

  System.out.println(
   str1 + str2);
}
```

→

```
class RetObj{
   public String str1;
   public String str2;
}
public RetObj extractMethod(…){
   …
   return new RetObj(str1, str2);
}

public void foo() {
   RetObj retObj = extractMethod(…);
   String str1 = retObj.str1;
   String str2 = retObj.str2;
   System.out.println(str1 + str2);
}
```

Non-local jump statements

Introduce exit labels

```
public void bar(){
  while(!stack.isEmpty()){
    …
    elem = stack.pop();
    if(elem == null)
      continue;
    if(elem.equals(known))
      break;
    push(elem.next());
  }
}
```

Code to extract

① Declare exit labels

② Modify non-local jumps

```
enum Label{CONTINUE, BREAK, FALLTHRU};
public Label extractMethod(…){
  …
  elem = stack.pop();
  if(elem == null)
    return Label.CONTINUE;
  if(elem.equals(known))
    return Label.BREAK;
  return Label.FALLTHRU;
}
```

③ Interpret labels

```
public void bar() {
  while(!stack.isEmpty()){
    Label l = extractMethod(…);
    if(l.equals(Label.CONTINUE))
      continue;
    else if(l.equals(Label.BREAK))
      break;
  }
}
```

Test Suite

- 56 similarly changed method pairs from
  - org.eclipse.compare
  - org.eclipse.core.runtime
  - org.debug.core
  - jdt.core
  - jEdit
- 30 similarly changed method groups from
  - Elasticsearch
  - jfreechart

Q1. Is clone removal refactoring feasible?

ID		edits	types	Δcode
Pair	2	15	E, A	-1
	9	77	E, R	-7
	22	285	E, F	-47
	29	56	E, L, R	4
Group	1	137	E, A, F, T	-7
	5	36	E, T	-6
	8	44	E, A, F	-4
	29	211	E	-149

Rase refactors

- **30 of 56 method pairs**
- **20 of 30 method groups**

E: extract method, R: introduce return object, L: introduce exit label, T: parameterize type, F: form template method, A: add parameter

Q2. Why does refactoring fail?

Reason	# method pairs	# method groups
Limited language support for generic types, e.g., v instanceof $T	7	2
Unmovable methods, e.g., super()	5	0
No edited statement found	8	2
No common code extracted	6	6

Q3. Is clone removal refactoring desirable?

- Average duration of version history: 1.3 years

		Feasible	Infeasible
Refactored		5	0
Unrefactored	Co-evolved	4	7
	Divergent	7	10
	Unchanged	34	19

“We don’t typically refactor unless we have to change the code for some bug fix or new feature. ”

Conclusion

- Rase leverages systematic edits to apply clone removal refactoring
- Automatic clone removal refactoring cannot obviate systematic editing
- Both clone removal refactoring and automated systematic editing are needed and they are complementary
- Determining refactoring desirability remains as further work