Transforming programs between APIs with Many-to-Many Mappings

Chenglong Wang[1], Jiajun Jiang[1], Jun Li[1], Yingfei Xiong[1], Xiangyu Luo[1], Lu Zhang[1] and Zhenjiang Hu[2]

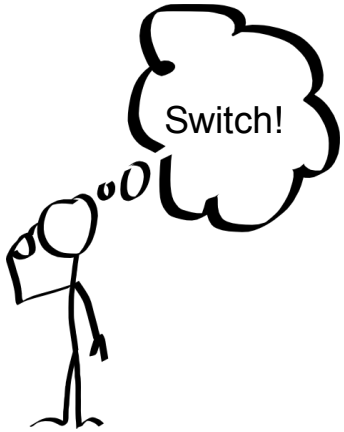
[1]Peking University, China
[2]National Institute of Informatics, Japan

APIs around us …

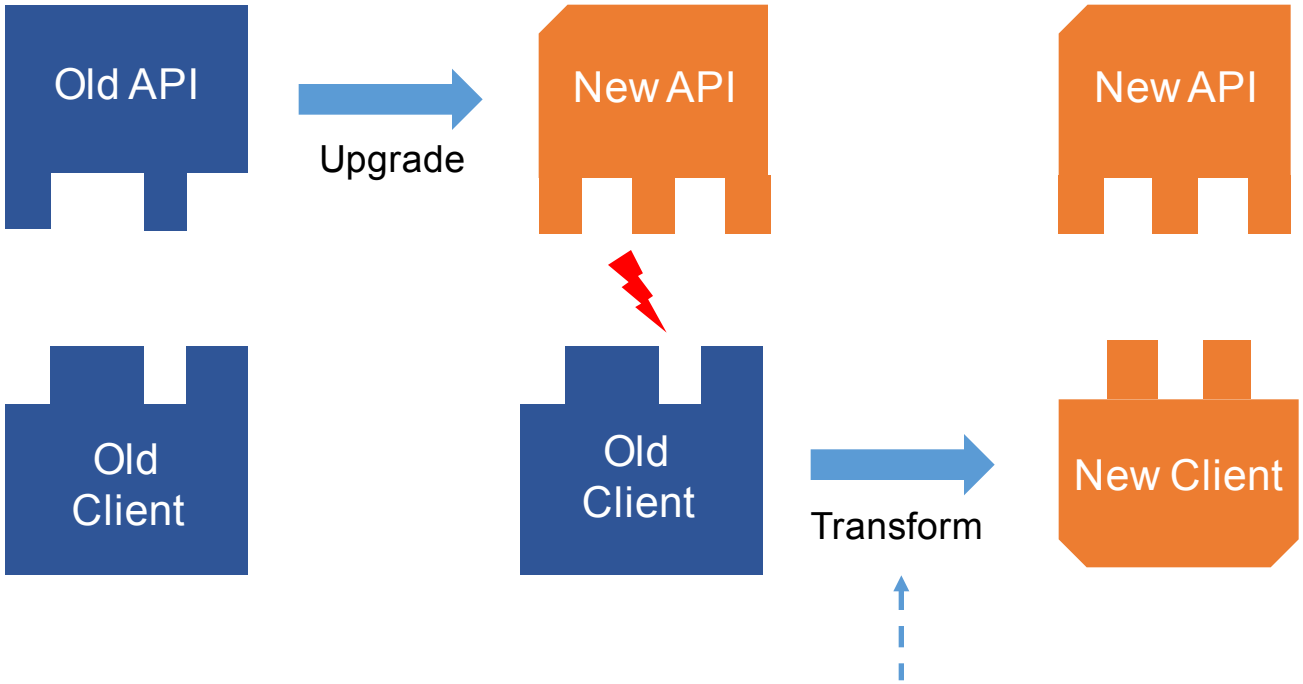
Python 2.x → Python 3.x

PhysX 2.x → PhysX 3.x

Google search API → Custom search API

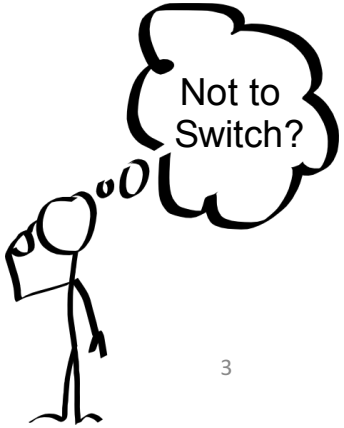


API changes can be incompatible …



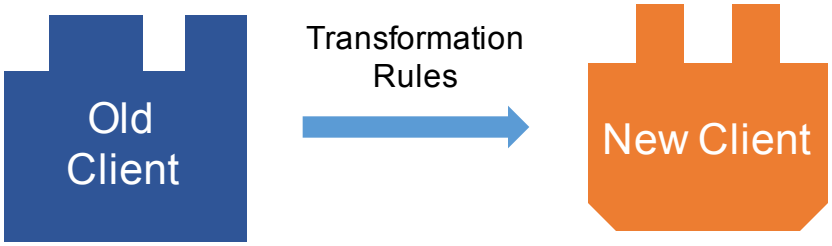
Manual Transformation:

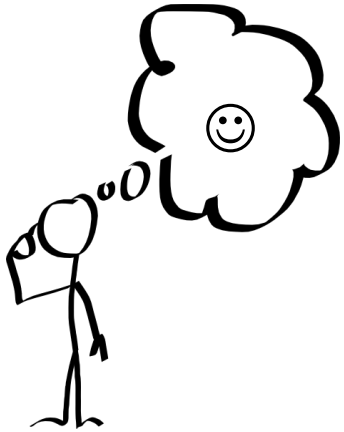
- Error-prone
- Time-consuming



Transformation Languages

- API providers write transformation rules to help transform all client programs automatically.





- General-purpose transformation languages (e.g. Strageto):
  - Difficult to program.
- Domain-specific languages for API transformation (e.g. Twinning):
  - Limited expressiveness: only capturing one-to-many mapping relations.

One-to-Many Mapping

- Definition: one invocation to an old API method is mapped to a sequence of method invocations to the new API.

```
button.setAlignmentX(align);
```

(Swing API in Java)

```
button.setAlignment(align);
```

(SWT API in Java)

Many-to-Many Mapping

- Definition: a sequence of invocations to the old API is mapped to a sequence of invocations to the new API.

```
1 b = new JButton();
2 parent.add(b);
```

(Swing API in Java)

→

```
1 b = new Button(parent, SWT.PUSH);
```

(SWT API in Java)

What's hard?

<u>Challenge</u>: The same sequence of API invocations may have different forms in different client programs.

- What we want:

```
b = new JButton();
parent.add(b);
```

→

```
b = new Button(parent, SWT.PUSH);
```

- Code Snippets:

```
// Case 1:
b = new JButton();
if(p != null){
    p.add(b);
}
```

↓

```
// Case 1'':
if(p != null){
    b = new Button(p, SWT.PUSH);
} else {
    b = new JButton();
}
```

```
// Case 2:
b = new JButton();
s = b.getUIClassID();
p = new JPanel();
p.add(b);
```

↓

```
// Case 2'':
p = new JPanel();
b = new Button(p, SWT.PUSH);
s = b.getUIClassID();
```

Insight: Normalization

Insight: Different forms of API invocation sequences can be semantics-equivalently normalized so that transformation is obvious.

- What we want:

```
b = new JButton();
parent.add(b);
```

→

```
b = new Button(parent, SWT.PUSH);
```

- Code Snippets:

```
// Case 1:
b = new JButton();
if(p != null){
    p.add(b);
}
```

```
// Case 2:
b = new JButton();
s = b.getUIClassID();
p = new JPanel();
p.add(b);
```

↓

```
// Case 1':
if(p != null){
    b = new JButton();
    p.add(b);
} else {
    b = new JButton();
}
```

↓

```
// Case 2':
p = new JPanel();
b = new JButton();
p.add(b);
s = b.getUIClassID();
```

Our solution: PatI Language

- Developers specify mapping relations between old and new APIs with only considering consecutive cases.

```
b = new JButton();
parent.add(b);
```

→

```
b = new Button(parent, SWT.PUSH);
```

- PatI automatically identify and transform different forms of the invocation sequence.

```
// Case 1:
b = new JButton();
if(p != null){
    p.add(b);
}
```

→

```
// Case 1':
if(p != null){
    b = new JButton();
    p.add(b);
} else {
    b = new JButton();
}
```

→

```
// Case 1'':
if(p != null){
    b = new Button(p, SWT.PUSH);
} else {
    b = new JButton();
}
```

PatI Syntax

- Looks like patch rules.

```
1 b = new JButton();
2 parent.add(b);
```

→

```
1 b = new Button(parent, SWT.PUSH);
```

```
//rule rButton
(b: JButton->Button,
 parent: JPanel->Composite) {
    - b = new JButton();
    - parent.add(b);
    + b = new Button(parent, SWT.PUSH);
}
```

Metavariable Declaration

Source Pattern

Target Pattern

Running Example

```
1 b = new JButton();
2 parent.add(b);
```

```
1 b = new Button(parent, SWT.PUSH);
```

```
1 btn = new JButton();
2 btn.setAlignmentX(alX);
3 System.out.print(alX);
4 if (b) {
5     component.getPanel().add(btn);
6 } else {
7     defaultBtn = btn;
8     defaultPnl.add(defaultBtn);
9 }
```

(A client program snippet using Swing Java API)

Transformation Pipeline

- Input:
  - An old client program using old APIs.
  - A set of transformation rules.

- Transformation Process

- Output:
  - A new client program using the new APIs.

Preprocessing

- Step 1: Transform the client program into 3-address form.

```
1 btn = new JButton();
2 btn.setAlignmentX(alX);
3 System.out.print(alX);
4 if (b) {
5     component.getPanel().add(btn);
6 } else {
7     defaultBtn = btn;
8     defaultPnl.add(defaultBtn);
9 }
```

```
1 btn = new JButton();
2 btn.setAlignmentX(alX);
3 System.out.print(alX);
4 if (b) {
5     gen1 = component.getPanel();
6     gen1.add(btn);
7 } else {
8     defaultBtn = btn;
9     defaultPnl.add(defaultBtn);
10 }
```

Matching

- Step 2: Match the client program with transformation rules.

```
1 b = new JButton();
2 parent.add(b);
```

→

```
1 b = new Button(parent, SWT.PUSH);
```

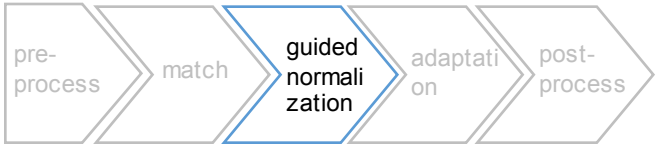
aliases

b ⟼ {btn, defaultBtn}
parent ⟼ defaultPnl

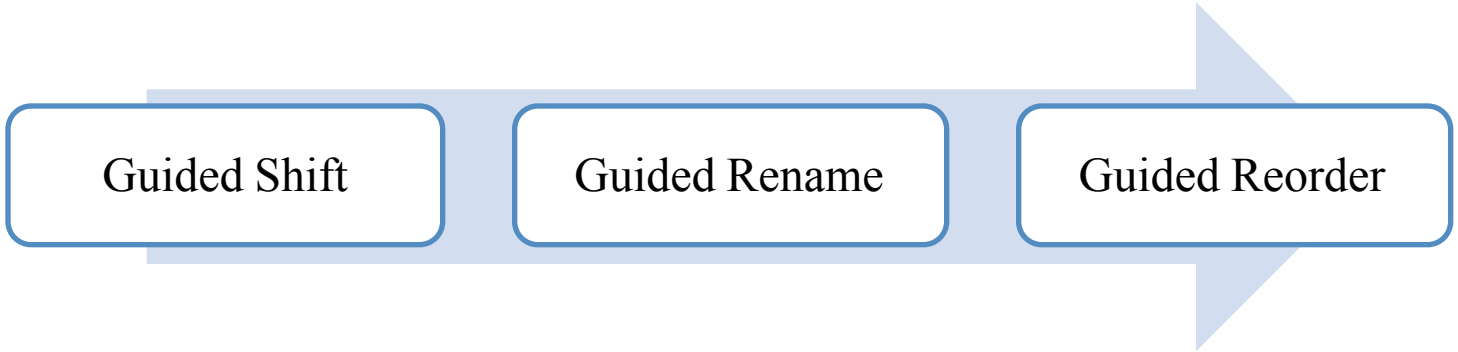
```
1  btn = new JButton();
2  btn.setAlignmentX(alX);
3  System.out.print(alX);
4  if (b) {
5      gen1 = component.getPanel();
6      gen1.add(btn);
7  } else {
8      defaultBtn = btn;
9      defaultPnl.add(defaultBtn);
10 }
```

b ⟼ btn
parent ⟼ gen1

Guided Normalization

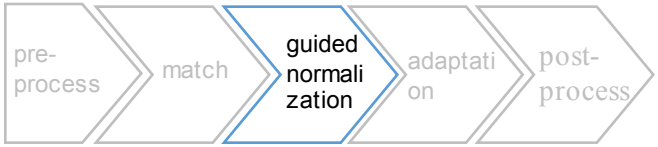


- Step 3: *Semantics equivalently* normalize client program so that all matched statements appear consecutively.



- Dependency analysis:
  To ensure normalization process is semantics-preserving.

Guided Normalization



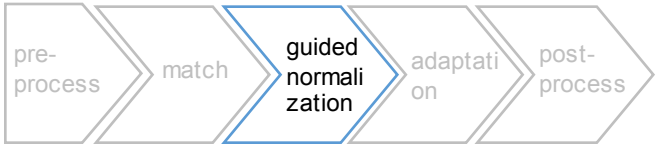
- Step 3.1: **Guided shift**
  - Shift statements into if-else branches to make matched API invocations appear in same block.

```
1  btn = new JButton();
2  btn.setAlignmentX(alX);
3  System.out.print(alX);
4  if (b) {
5      gen1 = component.getPanel();
6      gen1.add(btn);
7  } else {
8      defaultBtn = btn;
9      defaultPnl.add(defaultBtn);
10 }
```

```
1  System.out.print(alX);
2  if (b) {
3      btn = new JButton();
4      btn.setAlignmentX(alX);
5      gen1 = component.getPanel();
6      gen1.add(btn);
7  } else {
8      btn = new JButton();
9      btn.setAlignmentX(alX);
10     defaultBtn = btn;
11     defaultPnl.add(defaultBtn);
12 }
```

(Remark: Warnings will be generated when data dependency may be violated during normalization.)

Guided Normalization

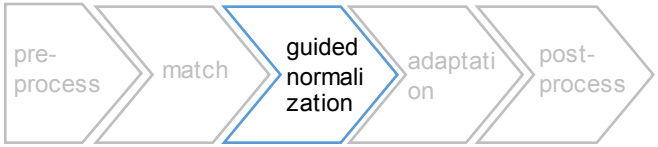


- Step 3.2: **Guided rename**
  - Rename aliases with a new name to make variables matched to the same meta-variable have same name.

```
System.out.print(alX);
if (b) {
    btn = new JButton();
    btn.setAlignmentX(alX);
    gen1 = component.getPanel();
    gen1.add(btn);
} else {
    btn = new JButton();
    btn.setAlignmentX(alX);
    defaultBtn = btn;
    defaultPnl.add(defaultBtn);
}
```

```
System.out.print(alX);
if (b) {
    btn = new JButton();
    btn.setAlignmentX(alX);
    gen1 = component.getPanel();
    gen1.add(btn);
} else {
    gen2 = new JButton();
    btn = gen2;
    btn.setAlignmentX(alX);
    defaultBtn = btn;
    defaultPnl.add(gen2);
}
```

Guided Normalization



- Step 3.3: **Guided reorder**
  - Reorder statements to make matched API invocations appear consecutively.

```
1  System.out.print(alX);
2  if (b) {
3      btn = new JButton();
4      btn.setAlignmentX(alX);
5      gen1 = component.getPanel();
6      gen1.add(btn);
7  } else {
8      gen2 = new JButton();
9      btn = gen2;
10     btn.setAlignmentX(alX);
11     defaultBtn = btn;
12     defaultPnl.add(gen2);
13 }
```

```
1  System.out.print(alX);
2  if (b) {
3      gen1 = component.getPanel();
4      btn = new JButton();
5      gen1.add(btn);
6      btn.setAlignmentX(alX);
7  } else {
8      gen2 = new JButton();
9      defaultPnl.add(gen2);
10     btn = gen2;
11     btn.setAlignmentX(alX);
12     defaultBtn = btn;
13 }
```

(Remark: Warnings will be generated when data dependency may be violated during normalization.)

Adaptation

- Step 4: Transform the program using the transformation rule.

```
b = new JButton();
parent.add(b);
```

→

```
b = new Button(parent, SWT.PUSH);
```

```
System.out.print(alX);
if (b) {
    gen1 = component.getPanel();
    btn = new JButton();
    gen1.add(btn);
    btn.setAlignmentX(alX);
} else {
    gen2 = new JButton();
    defaultPnl.add(gen2);
    btn = gen2;
    btn.setAlignmentX(alX);
    defaultBtn = btn;
}
```

→

```
System.out.print(alX);
if (b) {
    gen1 = component.getPanel();
    btn = new Button(gen1, SWT.PUSH);
    btn.setAlignmentX(alX);
} else {
    gen2 = new Button(defaultPnl, SWT.PUSH);
    btn = gen2;
    btn.setAlignmentX(alX);
    defaultBtn = btn;
}
```

Post-process

- Step 5: Recover program style by inlining temp variables.

```
1  System.out.print(alX);
2  if (b) {
3      gen1 = component.getPanel();
4      btn = new Button(gen1, SWT.PUSH);
5      btn.setAlignmentX(alX);
6  } else {
7      gen2 = new Button(defaultPnl, SWT.PUSH);
8      btn = gen2;
9      btn.setAlignmentX(alX);
10     defaultBtn = btn;
11 }
```

```
1  System.out.print(alX);
2  if (b) {
3      btn = new
           Button(component.getPanel(),
           SWT.PUSH);
4      btn.setAlignmentX(alX);
5  } else {
6      btn = new
           JButton(defaultPnl,
           SWT.PUSH);
7      btn.setAlignmentX(alX);
8      defaultBtn = btn;
9  }
```

Properties of Patl

- Guided-normalization transforms programs semantics-equivalently.
- Palt will match and transform all statements in one pass.

Evaluation

- Q1: Is our transformation language easy to use?
- Q2: How important is guided-normalization in transforming programs between APIs?
- Q3: How many warnings will be generated in real world cases?
- Q4: How many cases cannot be expressed by our approach?

Evaluation Setup

- Three real-world cases:
  - JDOM→ Dom4J
  - Google calendar v2 → v3
  - Swing → SWT

- Nine open source projects* using these APIs.

Client	KLOC	Classes	Methods	Case
husacct	195.6	1187	5977	Jdom/Dom4j
serenoa	12.2	52	523	Jdom/Dom4j
openfuxml	112.5	727	4098	Jdom/Dom4j
clinicaweb	3.9	74	213	Calendar
blasd	9.7	199	729	Calendar
goofs	8.6	78	643	Calendar
evochamber	12.8	132	868	Swing/SWT
swingheat	2.3	30	186	Swing/SWT
marble	1.6	10	56	Swing/SWT
Total	359.2	2489	13293	–

*: obtained from searchcode.com by searching the source API methods.

Evaluation

Q1: Is our transformation language easy to use?

- In total, 204 API changes are covered by 236 rules.
- 94.1% rules have a body no longer than 4 lines.

Q2: How important is guided-normalization in transforming programs between APIs?

- In average, 28% transformations are many-to-many transformations, and 65% of them need guided normalization.

Evaluation

Q3: How many warnings will be generated in real world cases?

- Totally, there are 30 warnings while transforming 4043 lines of code, in average 0.7% of codes report warnings for each project.

Q4: How many cases cannot be expressed using PatI?

- Totally, 84 (2%) lines of code cannot be expressed in PatI.
- Include class level transformation, loop unrolling etc.

Conclusion

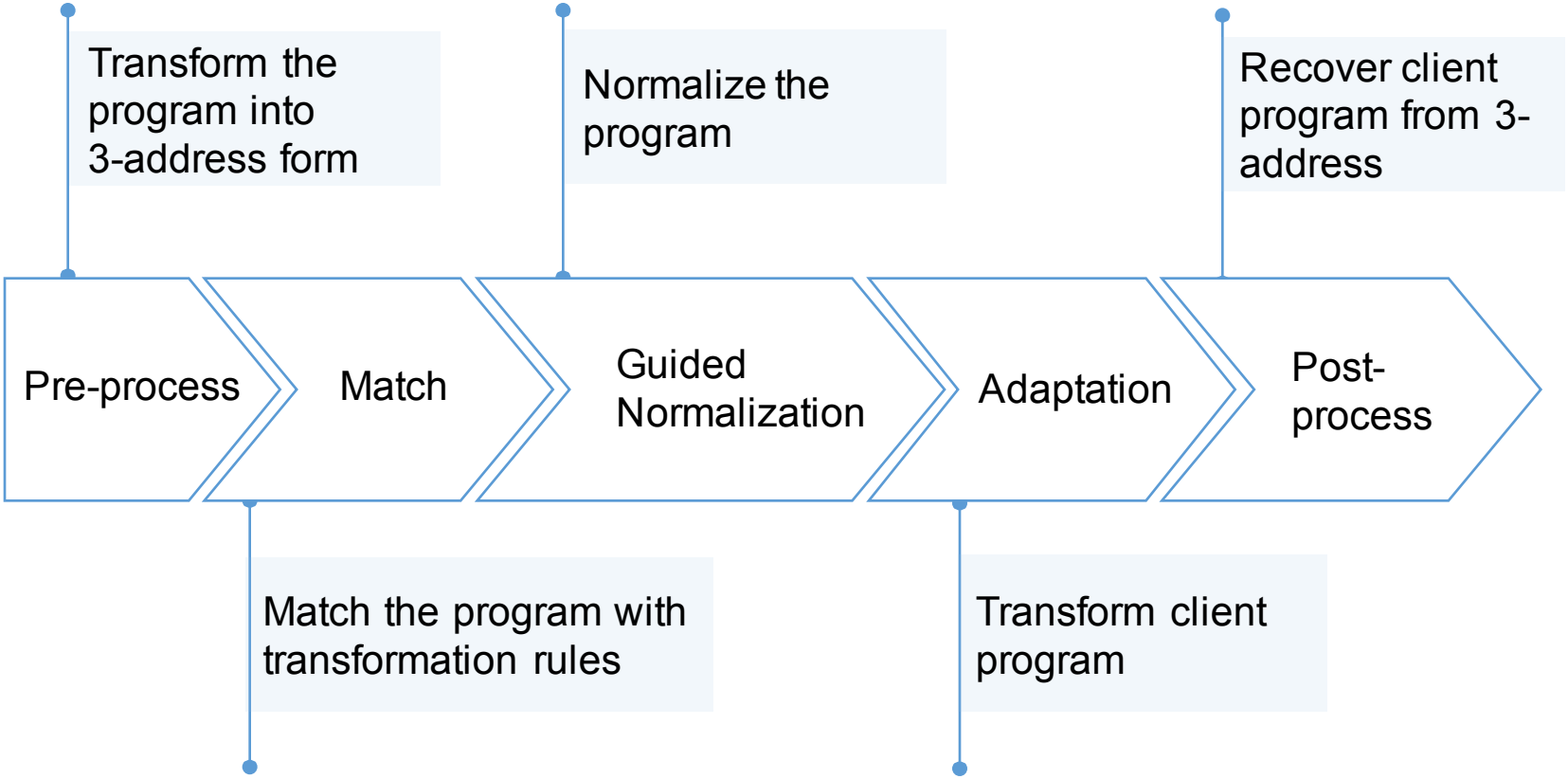
- Many-to-Many transformations are common in real programs and vary greatly.

- PatI is a practical language to solve the problem.
  - Developers only consider basic form transformation.
  - PatI match and transform all cases automatically.

Thank you!

Questions?

- Input: Old client program, Transformation rules
- Output: New client program



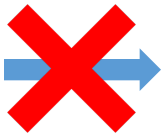
None-Expressible Cases

- A class may be split into two classes, each inherits part of the functionalities of the original class.
- To be transformed API invocations are across a ‘for’ statement.
- A class definition is transformed into a method invocation.

Warnings in Guided Normalization

- Warnings will be generated when data dependency may be violated during normalization.

```
1 b = new JButton();
2 if (b != null) {
3     p.add(b);
4 }
```



```
1 if (b != null) {
2     b = new JButton();
3     p.add(b);
4 } else {
5     b = new JButton();
6 }
```