Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line

FOSD meeting, March 13-17, 2017
Wolfram Fenske,*, Jens Meinicke,*,† Sandro Schulze,*, Steffen Schulze,*, Gunter Saake*

* University of Magdeburg, Germany
† Carnegie Mellon University, USA

@sanschul sanschul@ovgu.de
Motivation: Creating Variants via Clone & Own

- Clone & own (C&O): Copy an existing product and adapt it to new requirements
Motivation: Creating Variants via Clone & Own

- Clone & own (C&O): Copy an existing product and adapt it to new requirements
Motivation: Creating Variants via Clone & Own

- Clone & own (C&O): Copy an existing product and adapt it to new requirements
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

\[ p_1 \ldots \ldots \ldots \]
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

\[ \begin{align*}
  p_1 & \quad p_2 \\
  \ldots & \quad \ldots \\
  \ldots & \quad \ldots 
\end{align*} \]
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

\[ p_1 \]
\[ p_2 \]
\[ \vdots \]
\[ \vdots \]
\[ \vdots \]
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

\[ p_1 \]
\[ p_2 \]
\[ \ldots \]
\[ p_n \]
\[ \ldots \]
\[ \ldots \]
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

+ Initially cheap and easy
- Lack of traceability
- High synchronization effort
- High maintenance & evolution costs
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

+ Initially cheap and easy
- Lack of traceability
- High synchronization effort
- High maintenance & evolution costs

Software Product Line (SPL) Development
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

- Initially cheap and easy
- Lack of traceability
- High synchronization effort
- High maintenance & evolution costs

Software Product Line (SPL) Development

- High initial costs
- Improved traceability
- Minimal synchronization costs
- Low maintenance & evolution costs
Unsystematic vs. Systematic Reuse

“Clone & Own” Variant Development

- Initially cheap and easy
- Lack of traceability
- High synchronization effort
- High maintenance & evolution costs

Software Product Line (SPL) Development

- High initial costs
+ Improved traceability
+ Minimal synchronization costs
+ Low maintenance & evolution costs
Proposed Step-Wise Migration Process

Original, Cloned Products

Initial SPL

Clone Detection & Variant-Preserving Refactoring

Final SPL

Configurations:
C1 = \{p_1\}
C2 = \{p_2\}
...
Cn = \{p_n\}

Extractive & Preparatory Refactoring

Configurations:
C1 = \{f_1, p_1\}
C2 = \{f_1, f_2, p_2\}
...
Cn = \{f_2, p_n\}

p_1 \lor p_2 \rightarrow f_1
p_2 \lor p_3 \lor ... \lor p_n \rightarrow f_2
...

Variant-Preserving Refactorings for Migrating Cloned Products | Wolfram Fenske | SANER Feb. 21–24 2017
Key Points of Migration Process

• Step-wise: help with the time-consuming, error-prone tasks but leave big design decisions to developers

• Variant-preserving refactoring [Schulze et al., VaMoS ’12] for clone consolidation

• Preparatory refactoring to align divergent clones

• Feature-oriented programming (FOP) as the variability mechanism

• Integrated tool support
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

Pull Up (OOP)

```
class Super

class A
common: int
a: int

class B
common: int
b: int
```

Move code within the **inheritance** hierarchy
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

Move code within the *inheritance* hierarchy

---

```
class Super

class A
common: int
a: int

class B
common: int
b: int
```
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

Pull Up (OOP)

class Super
common: int

class A
common: int
a: int

class B
common: int
b: int

Move code within the inheritance hierarchy
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

Pull Up (OOP)

```
class Super
    common: int

class A
    common: int
        a: int

class B
    common: int
        b: int
```

Move code within the **inheritance** hierarchy

Pull Up To Common Feature (FOP)

```
feature Common

class C
    common: int
        a: int

feature A

class C
    common: int
        a: int

feature B

class C
    common: int
        b: int
```

Move code within the **refinement** hierarchy
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

Pull Up (OOP)

**class Super**
common: int

**class A**
common: int
a: int

**class B**
common: int
b: int

Pull Up To Common Feature (FOP)

**feature Common**

**feature A**

**class C**
common: int
a: int

**feature B**

**class C**
common: int
b: int

Move code within the *inheritance* hierarchy

Move code within the *refinement* hierarchy
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

Pull Up (OOP)

```
class Super
  common: int

class A
  common: int
  a: int

class B
  common: int
  b: int
```

Move code within the *inheritance* hierarchy

Pull Up To Common Feature (FOP)

```
feature Common

class C
  common: int

feature A

class C
  common: int
  a: int

feature B

class C
  common: int
  b: int
```

Move code within the *refinement* hierarchy
Refactorings — Clone Consolidation via “Pull Up To Common Feature”

### Pull Up (OOP)

- **class Super**
  - common: int

- **class A**
  - a: int

- **class B**
  - b: int

Move code within the *inheritance* hierarchy

### Pull Up To Common Feature (FOP)

- **Feature Common**

- **class C**
  - a: int
  - b: int

(See paper for fancy details)

Move code within the *refinement* hierarchy

More complex in general.
Refactorings — Aligning Divergencies via “Rename”

```
feature A
  class C
  common: int
  a: int

feature B
  class MyC
  common: int
  b: int

feature C
  class MyC
  common: int
  c: int
```
Refactorings — Aligning Divergencies via “Rename”

feature A

class C
common: int
a: int

feature B

class MyC
common: int
b: int

dfeature C

class MyC
common: int
c: int
Refactorings — Aligning Divergencies via “Rename”

```
feature A
class C
  common: int
    a: int

feature B
class MyC
  common: int
    b: int

feature C
class MyC
  common: int
    c: int
```
Refactorings — Aligning Divergencies via “Rename”
Refactorings — Aligning Divergencies via “Rename”

Feature A

```
class C
| common: int |
| a: int      |
```

Feature B

```
class C
| common: int |
| b: int      |
```

Feature C

```
class C
| common: int |
| c: int      |
```
Refactorings — Aligning Divergencies via “Rename”

(See paper for fancy details)
Evaluation

Original Products

- ApoClock
- ApoDice
- ApoMono
- ApoSnake
- myTreasure
Evaluation

Original Products | Initial SPL

ApoClock | ApoDice | ApoMono | ApoSacoine | myTreasure

ApoClock | base

ApoDice | myTreasure

...
Evaluation

Original Products
- ApoClock
- ApoDice
- ApoMono
- ApoSnake
- myTreasure

Initial SPL
- base

LOC
- ApoClock
- ApoDice
- ApoMono
- ApoSnake
- myTreasure
- Common*

Initial LOC
- LOC without rename
- LOC with rename

Initial SPL
Evaluation

Original Products
- ApoClock
- ApoDice
- ApoMono
- ApoSna
ke
- myTreasure

Initial SPL
- base
- ApoClock
- ApoDice
- ... myTreasure

"Pull Up" only
- base
- ... Common1
- ApoClock
- ApoDice
- ... myTreasure

LOC
- ApoClock
- ApoDice
- ApoMono
- ApoSna
ke
- myTreasure
- Common*

Initial LOC
- LOC without rename
- LOC with rename

Initial SPL
Evaluation

Original Products

ApoClock  ApoDice  ApoMono  ApoSnake  myTreasure

Initial SPL

```
base

ApoClock  ApoDice  ... myTreasure
```

"Pull Up" only

```
base

ApoClock  ApoDice  ... myTreasure
```

### LOC

<table>
<thead>
<tr>
<th>Product</th>
<th>Initial LOC</th>
<th>LOC without rename</th>
<th>LOC with rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApoClock</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>ApoDice</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>ApoMono</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>ApoSnake</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>myTreasure</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>Common</td>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
</tbody>
</table>
Evaluation

Original Products

ApoClock
ApoDice
ApoMono
ApoSnake
myTreasure

Initial SPL

ApoClock  ApoDice  …  myTreasure

“Pull Up” only

ApoClock  ApoDice  …  myTreasure

“Rename” & “Pull Up”

ApoClock  ApoDice  …  myTreasure

LOC

Initial SPL

“A Pull Up” only

ApoDice $\lor$ myTreasure $\rightarrow$ Common2

…
Evaluation

Original Products

- ApoClock
- ApoDice
- ApoMono
- ApoSnake
- myTreasure

Initial SPL

```
ApoClock
ApoDice
myTreasure
```

“Pull Up” only

```
ApoClock
ApoDice
myTreasure
```

“Rename” & “Pull Up”

```
ApoClock
ApoDice
myTreasure
```

ApoDice ∨ myTreasure → Common2

```
ApoClock
ApoDice
myTreasure
```

Initial LOC

<table>
<thead>
<tr>
<th>Product</th>
<th>Initial LOC</th>
<th>LOC without rename</th>
<th>LOC with rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApoClock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoDice</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoMono</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoSnake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myTreasure</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
```

LOC

<table>
<thead>
<tr>
<th>Product</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApoClock</td>
<td></td>
</tr>
<tr>
<td>ApoDice</td>
<td></td>
</tr>
<tr>
<td>ApoMono</td>
<td></td>
</tr>
<tr>
<td>ApoSnake</td>
<td></td>
</tr>
<tr>
<td>myTreasure</td>
<td></td>
</tr>
<tr>
<td>Common*</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation — Discussion

• Naming in case study exaggerates efficacy of “Rename” (e.g. class ApoClockMenu in ApoClock vs. class ApoDiceMenu in ApoDice)

• Why do clones remain?
  • Long similar, but not identical methods (Type-3 clones) — more preparatory refactorings needed
  • Differing releases of 3rd-party libraries w/ conflicting APIs
Conclusion & Future Work

• Step-wise process to migrate from clone & own to SPL
• Variant-preserving refactorings (Pull Up and Rename)
• Case study shows feasibility

Future work:
• Further case studies
• More (preparatory) refactorings (e.g., “Extract {Method, Field, Constant …}”)
• Make code similarities more understandable
• Support for other languages (e.g., C)
Limitations

- We force developers
  - to choose a variability mechanism of our choice —> no further mechanism is supported
  - to migrate the whole project —> **Risky!**
- We take only text-based (syntactical) information into account
- We omit possibilities of alternative features
extract information by means of flexible and customizable reverse engineering
....extract information by means of flexible and customizable reverse engineering

....keep this information in a language-independent format
extract information by means of flexible and customizable reverse engineering

keep this information in a language-independent format

migration-on-demand by providing reengineering techniques for several variability mechanisms
Source Code

Intermediate Format

Variability + Mapping

Virtual Merge

Match

Compare

CCProgram

name : EString

CCNamespace

name : EString

CCFile

relativePath : EString

CCClassifier

name : EString

type : CCClassifierType = NOT_DEFINED

CCMethod

belongsToRelation : CCBelongsToRelation = NOT_SET

constructor : EBoolean = false

visibility : CCVisibility = PUBLIC

name : EString

CCNestedDataType

CCNestedGroupType

CCSimpleParameterType

[0..*] namespaces

[0..1] parentProgram

[0..*] files

[0..1] parentNamespace

[0..*] classifiers

[0..*] methods

[0..1] parentClassifier

[0..*] returnType

[0..*] nestedDataType

[0..*] subTypes
Intermediate Format

Comparison

Source Code

Domain knowledge

Variability + Mapping

Historical data
Intermediate Format

Source Code

Encode & Combine

Compare

Match

Virtual Merge

Variability + Mapping
Source Code

Intermediate Format

Encode & Combine

Variability + Mapping

Virtual Merge

Match

Compare

Virtual Merge
Source Code

Intermediate Format

Encode & Combine

Compare

Match

Variability + Mapping

Migration
KEEP
CALM
AND
REFACTOR
YOUR CODE