Delta-oriented Development of Software Product Lines

Ina Schaefer
Institut für Software Systems Engineering
Technische Universität Braunschweig
i.schaefer@tu-braunschweig.de

Gastvorlesung, Universität Magdeburg
13. Dezember 2010
Software Product Lines

- A product line is “a family of products designed to take advantage of their common aspects and predicted variabilities.” [Weiss; 1999]

- A product line is “a set of systems sharing a common set of features that satisfy the specific needs of a particular market segment.” [Clements, Northrop; 2001]
Product Lines in Industry

- Industrial-scale Approach for Reuse in Software Engineering
- Members of Product Line Hall of Fame: [http://www.splc.net/fame.html](http://www.splc.net/fame.html)
  - HP, Ericsson, Nokia, GM Powertrain, Boeing, Bosch, Lucent, Philips, Toshiba
- Commercially successful, e.g., HP Owen Firmware Cooperative:
  - 1/4 of the staff, 1/3 of the time, and 1/25 of the bugs
    (compared to previous single application engineering)
Product Line Development

Feature Model → Family Engineering → Product Line Artifacts Base → Application Engineering → Product
Product Line Development

Feature Model → Family Engineering → Product Line Artifacts Base

Feature Configuration → Automated Product Derivation → Product
Outline

• Delta Modelling
• Delta Modelling and Model-based Development
• Delta-oriented Programming in DeltaJava
• Pure Delta-oriented Programming
Delta Modelling of Product Lines

- Product for valid feature configuration.
- Developed with Standard Techniques

- Modifications of Core Product.
- Application conditions over product features.
- Partial ordering for conflict resolution.

Configuration

For a Feature Configuration:

• Determine product deltas with valid application condition.

• Determine linear ordering of product deltas compatible with partial ordering.

• Apply changes specified by product deltas to core product in the linear order.
Feature-based Variability

Example: Trading System Software Product Line
Feature-based Variability

Example: Trading System Software Product Line

Payment Features
Component Modelling

Component Core Model (for Cash Payment):

Component Delta Model (for Card Payment):
Configured Component Model (for Cash and Card Payment):
Delta Modelling

- Evolutionary Development by Adding Product Deltas
- Explicit Treatment of Combinations of Features by Complex Application Conditions
- Proactive, Reactive and Extractive SPL Development
- Usable with Different Modelling Formalisms and Implementation Techniques
- Model Refinements are Orthogonal to Variability Modelling.
Model-based Development

Feature Modelling

Feature Model $\xrightarrow{\text{create}}$ Feature Configuration

Modelling Level 1

Core Model$_1$ $\xrightarrow{\text{configure}}$ Delta Models$_1$, $\xrightarrow{\text{refine}}$ Product Models$_1$

Modelling Level 2

Core Model$_2$ $\xrightarrow{\text{configure}}$ Delta Models$_2$, $\xrightarrow{\text{refine}}$ Product Models$_2$

Implementation

Core Module $\xrightarrow{\text{configure}}$ Delta Modules $\xrightarrow{\text{refine}}$ Product Impl.


Delta-oriented Development of Software Product Lines
Model-based Development

Feature Modelling

Feature Model

create

configure

Feature Configuration

Modelling Level 1

Core Model₁

Delta Models₁

configure

Product Models₁

refine

Modelling Level 2

Core Model₂

Delta Models₂

configure

Product Models₂

refine

Implementation

Core Module

Delta Modules

configure

Product Impl.

refine

[...]


Delta-oriented Development of Software Product Lines
A Component Design can be refined to Class Diagrams:

- Each Component in Core Model is realized by a Class Core Diagram.
- Component Delta Models are realized by Class Delta Diagrams.

For each Component:

- 1 Class Core Diagram
- Set of Class Delta Diagrams
Core Class Diagram

Inventory

Cash Desk

Cash Desk
- Order
  + CashDesk()
  + void addProduct(int id)
  + void receiveMoney(float)
  + void cancelCurrentPayment()
  + void startPaymentProcess()
  + void endPaymentProcess()
  + void selectCashPayment()
  - void connectWithInventory()

Keyboard
  + Keyboard(CashDesk)
  + void terminate()
  + void setStatePrePaymentProcess()
  + void setStatePaymentProcess()
  + void setStatePaymentSelection()
  + void setStateCashPayment()
  + void enable()
  + void disable()

Display
  + CashDeskDisplay()
  + void displayProduct(Product, float)
  + void displayTotal(float)
  + void displayTotalAndReturn(float)
Core Class Diagram

Refinement of Core Cash Desk Component

Delta-oriented Development of Software Product Lines
Class Delta Diagram

- Bank
  - +void CreditCardScanned(int)
  - +void CreditCardPinEntered(int)
  - -connectWithBank()

- +void startPaymentProcess()

Cash Desk

Cash Desk

Card Reader

+void enable()
+void disable()
+void enterPinCode()
Class Delta Diagram

Refinement of Cash Desk Component Delta Model

Delta-oriented Development of Software Product Lines
Commutativity

If Model Refinement and Model Configuration are compatible, it holds that:

\[
\text{refine} \left( \text{configure}_i \left( \left( \text{Core}_i, \Delta s_i \right), fc \right) \right) = \text{configure}_{i+1} \left( \text{refine} \left( \text{Core}_i, \Delta s_i \right), fc \right)
\]
Compatibility

- Additions of Modelling Elements are refined to Additions only.
- Removals of Modelling Elements are refined to Removals only.
- Modifications of Modelling Elements are refined to Additions, Modifications and Removals, such that for each Modification operation \( \text{mod}(e) \) of a Modelling Element \( e \), it holds that

\[
\text{apply(refine}(e), \text{refine}(\text{mod } e)) = \text{refine}(e')
\]
Model-based Development

Feature Modelling

Feature Model \(\xrightarrow{\text{configure}}\) Feature Configuration

create

Modelling Level 1

Core Model\(_1\) Delta Models\(_1\) \(\xrightarrow{\text{configure}}\) Product Models\(_1\)

refine

Modelling Level 2

Core Model\(_2\) Delta Models\(_2\) \(\xrightarrow{\text{configure}}\) Product Models\(_2\)

refine

Implementation

Core Module Delta Modules \(\xrightarrow{\text{configure}}\) Product Impl.

refine

[...]
Model-based Development

Feature Modelling

Feature Model

configure

Feature Configuration

create

Modelling Level 1

Core Model\(_1\)  Delta Models\(_1\)

configure

Product Models\(_1\)

refine

Modelling Level 2

Core Model\(_2\)  Delta Models\(_2\)

configure

Product Models\(_2\)

refine

Implementation

Core Module  Delta Modules

configure

Product Impl.

refine

[...]

Delta-oriented Development of Software Product Lines
DeltaJava - A PL for SPL

- Extension of Java with Core and Delta Modules
- Core Product is implemented by Core Module.
- Product Deltas are implemented by Delta Modules.
- A Product Implementation is obtained by Application of Delta Modules to Core Module.
- Type System ensures Safety of Delta Application.

Example Product Line

Feature Model:

Expression Product Line

Data
- Lit
- Add
- Neg

Operations
- Print
- Eval
Core Module

A core module contains a set of Java classes.

core Print, Lit {
class Lit implements Exp{
    int value;
    Lit(int n) { value=n; }
    void print()
        { System.out.println(value); }
}
class Test {
    Exp a;
    void run()
        { a=new Lit(3); a.print(); }
}
}
Delta Modules

• Modifications on Class Level:
  • Addition, Removal and Modification of Classes

• Modifications of Class Structure:
  • Changing Super Class and Constructor
  • Adding/Removing Fields/Methods
  • Modifying Methods (wrapping with original call)

• Application Condition in when clause: Boolean Constraint on Features in Feature Model

• Partial Ordering of Delta Modules by after clauses
Delta Module for Add

delta DAdd when Add {
  adds class Add {
    Exp expr1;
    Exp expr2;
    Add(Exp a, Exp b) { expr1=a; expr2=b; }
  }
}
Delta for Add and Print

delta DAddPrint after DAdd when Add && Print{

  modifies class Add {
     adds void print()
     { expr1.print();
         System.out.print(" + ");
         expr2.print(); }
  }

  modifies class Test {
     adds Exp e;
     modifies void run()
     { original();
         e=new Add(a, a);
         e.print(); }
  }
}
Product Generation

For a Feature Configuration:

- Determine product deltas with valid application condition in `when` clauses.
- Determine linear ordering of product deltas compatible with partial ordering in `after` clauses.
- Apply changes specified by product deltas to core product in the linear order.
class Lit {
    int value;
    Lit(int n) { value=n; }
    void print() { System.out.print(value); }
}

class Add {
    Exp e1; Exp e2;
    Add(Exp a, Exp b) { e1=a; e2=b; }
    void print() { e1.print(); System.out.print(" + ");
        e2.print();}
}

class Test {
    Exp a; Exp e;
    void run() {  a=new Lit(3); a.print();
        e=new Add(a,a); e.print(); } 
}
Starting from a Simple Core (Proactive Development)

```java
features Lit, Add, Neg, Print, Eval
configurations Lit && Print && (Add | Neg | Eval )
core Lit, Print { [... ] }
delta DEval when Eval { [...] }
delta DAdd when Add { [...] }
delta DAddPrint after DAdd when Add && Print { [...] }
delta DAddEval after DAdd when Add && Eval { [...] }
delta DNeg when Neg { [...] }
delta DNegPrint after DNeg when Neg && Print { [...] }
delta DNegEval after DNeg when Neg && Eval { [...] }
```
Starting from a Complex Core (Extractive/Reactive Development)

```
features Lit, Add, Neg, Print, Eval
configurations Lit && Print && (Add | Neg | Eval )
core Lit, Print, Add, Eval  { [ ... ] }
delta DRemEval when !Eval && Add { [...] }
delta DRemAdd when !Add { [...] }
delta DRemAddEval when !Add && !Eval { [...] }
delta DNeg when Neg { [...] }
delta DNegPrint after DNeg when Neg && Print { [...] }
delta DNegEval after DNeg when Neg && Eval { [...] }
```
## Deltas vs. Feature Modules

[AHEAD, Batory et al. 2004]

<table>
<thead>
<tr>
<th>Feature-oriented Programming</th>
<th>Delta-oriented Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressiveness</strong></td>
<td>Design from Base Module</td>
</tr>
<tr>
<td><strong>Domain Features</strong></td>
<td>Bijection between Features and Feature Modules</td>
</tr>
<tr>
<td><strong>Optional Feature Problem</strong></td>
<td>Rearrange Code, Multiple Impl., Derivative Modules</td>
</tr>
<tr>
<td><strong>Safe Composition</strong></td>
<td>External Tools, Type Systems</td>
</tr>
<tr>
<td><strong>Evolution</strong></td>
<td>Refactoring</td>
</tr>
</tbody>
</table>

Table 3: Summary of Comparison

<table>
<thead>
<tr>
<th>Simple Core</th>
<th>Complex Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC feature modules</td>
<td>LOC delta modules</td>
</tr>
<tr>
<td>LOC delta modules</td>
<td>LOC delta modules</td>
</tr>
</tbody>
</table>

Table 4: Evaluation Results

- vLOC is the number of lines of code.
- The first implementation of the EPL follows the derivative module principle.
- In the second implementation of the EPL, the design of the delta modules has been chosen to mimic the first design.
- To evaluate the flexibility of delta-oriented programming, each example was implemented in Delta-oriented Programming starting from a simple core product and from a complex core product.
- For the EPL, the simple and the complex core products presented in Section 5 were used.
- The results are summarized in Table 4, showing the number of feature modules or delta modules and the corresponding lines of code required to implement the respective examples.
- The number of feature modules and delta modules does not differ significantly in the considered examples.
- For the first version of the EPL, the only reason that 34 feature modules are necessary is that also interfaces are implemented by separate modules.
- In the second version of the EPL, the number of delta modules plus the core module is actually the same as the number of feature modules, as ELT#V# encodes the same modular product line representation.
- In the Calculator and List examples, fewer delta modules are required because several feature modules could be combined into one delta module.
- In the GraphPL example, additional delta modules are used to factor out common code for combinations of features.
Delta-oriented Development of Software Product Lines

Simple Core Complex Core

t feature modules LOC t delta modules LOC t delta modules LOC

<table>
<thead>
<tr>
<th></th>
<th>Jak</th>
<th>DELTAJAVA Simple Core</th>
<th>DELTAJAVA Complex Core</th>
</tr>
</thead>
<tbody>
<tr>
<td># feature modules</td>
<td>12</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>LOC</td>
<td>98</td>
<td>123</td>
<td>144</td>
</tr>
<tr>
<td>EPL [1]</td>
<td>12</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>LOC</td>
<td>98</td>
<td>123</td>
<td>144</td>
</tr>
<tr>
<td>EPL [26]</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>LOC</td>
<td>98</td>
<td>117</td>
<td>124</td>
</tr>
<tr>
<td>Calculator [7]</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>LOC</td>
<td>75</td>
<td>76</td>
<td>78</td>
</tr>
<tr>
<td>List [7]</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>LOC</td>
<td>48</td>
<td>58</td>
<td>59</td>
</tr>
<tr>
<td>GraphPL [25]</td>
<td>19</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>LOC</td>
<td>2348</td>
<td>1407</td>
<td>1373</td>
</tr>
</tbody>
</table>

Table 2: Evaluation Results (LOC is the number of lines of code)
Applicability

A set of delta-modules is applicable to a class table if

- all removed or modified classes exists.
- for every delta modifies-clause,
  - all removed methods and fields exist.
  - all modified methods exists.
- if added classes, methods and fields do not exist.
Well-formedness

1. The core configuration is a valid feature configuration.

2. No delta module has a valid application condition for the core configuration.

3. The partial order ensures that for every feature configuration, the set of delta modules with valid application condition
   a. is applicable to the respective intermediate product
   b. and conflict-free (all conflicting modifications are ordered).
Constraint-based Type System

- Type check and generate constraints for core and delta modules in isolation.

- A product is type checked by
  - generating the product constraints (application of the delta module constraints to the core product constraints)
  - checking the derived product constraints against the product’s class signature table.

Implementation

- Based on Xtext Language Development Framework
- Using XPAND for Code Generation
- Eclipse plugin with syntax highlighting, background parsing with error marking, code completion
- Stand-alone compiler
Delta-oriented Development of Software Product Lines

Implementation

- Based on Xtext Language Development Framework
- Using XPAND for Code Generation
- Eclipse plugin with syntax highlighting, background parsing with error marking, code completion
- Stand-alone compiler
Implementation

Available at http://deltaj.sourceforge.net/
Core DOP

- **Strengths:**
  - Any product can be core product.
  - Feature combinations in application conditions.
  - Explicit ordering for conflict resolution.
  - Usable with different modelling and implementation languages.
Core DOP (2)

- **Weaknesses:**
  - Selection of core product (e.g., for mandatory exclusive features).
  - Only partial generalization of FOP.
  - Limited support for product line evolution (e.g., if core product is no longer valid).
Pure DOP

- Modifications of core product.
- Application conditions over product features in when clauses.
- Partial ordering for conflict resolution in after clauses.

- Product for valid feature configuration.

---

Pure DOP

- Product for valid feature configuration.

- Modifications of core product.

- Application conditions over product features in when clauses.

- Partial ordering for conflict resolution in after clauses.

---

Pure DOP

- Modifications of core product.
- Application conditions over product features in when clauses.
- Partial ordering for conflict resolution in after clauses.

Pure DOP

- Product for valid feature configuration.

- Modifications of core product.

Product line specification contains application conditions and ordering.

Pure DOP Product Line

Feature Model:

Expression Product Line

Data
- Lit
- Add
- Neg

Operations
- Print
- Eval
Delta Modules

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{delta} DLit & \textbf{delta} DLitPrint \\
\hline
\textbf{adds interface} Exp & \textbf{modifies interface} Exp \\
& \textbf{void} print(); \\
\textbf{adds class} Lit \textbf{implements} Exp & \textbf{modifies class} Lit \textbf{implements} Exp \\
& \textbf{adds void} print() \{ System.out.println(value); \} \\
\hline
\end{tabular}
\end{table}

...
Product Line Specification

features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg ]

  [ DPrint,
    DEval when Eval,
    DAddPrint when Add,
    DAddEval when (Add & Eval),
    DNegPrint when Neg,
    DNegEval when (Neg & Eval) ]
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg ]

  [ DPrint,
    DEval when Eval,
    DAddPrint when Add,
    DAddEval when (Add & Eval),
    DNegPrint when Neg,
    DNegEval when (Neg & Eval) ]
FOP and PureDOP

- **Lightweight Feature Java** [Delaware et al.; FOAL’09/FSE’09]

  FMD ::= feature φ {cd rcd}  
  rcd ::= refines class C extending C { fd; md rmd }  
  rmd ::= refines ms {s; Super(); s; return y; }  

  feature module  
  class refinement  
  method refinement
FOP and PureDOP

• Lightweight Feature Java [Delaware et al.; FOAL’09/FSE’09]

\[
\begin{align*}
\text{FMD} & ::= \text{feature } \varphi \{ \text{cd rcd} \} & \text{feature module} \\
\text{rcd} & ::= \text{refines class } C \text{ extending } C \{ \text{ fd; md rmd } \} & \text{class refinement} \\
\text{rmd} & ::= \text{refines ms } \{ \bar{s}; \text{ Super()}; \bar{s}; \text{return } y; \} & \text{method refinement}
\end{align*}
\]

• Embedding of LFJ into LPΔJ - Defines same set of products:

\[
\begin{align*}
\lbrack \text{feature } \varphi \{ \text{cd rcd} \} \rbrack &= \\
\delta \text{ta } \varphi \{ \text{ adds cd } \lbrack \text{rcd} \rbrack \} \\
\lbrack \text{refines class } C \text{ extending } C \{ \text{ fd; md rmd } \} \rbrack &= \\
\text{modifies } C \text{ extending } C \{ \text{ adds fd adds md } \lbrack \text{rmd} \rbrack \} \\
\lbrack \text{refines ms } \{ \bar{s}; \text{ Super()}; \bar{s}; \text{return } y; \} \rbrack &= \\
\text{modifies ms } \{ \bar{s}; \text{ original()} ; \bar{s}; \text{return } y; \}
\end{align*}
\]

Delta-oriented Development of Software Product Lines
Extractive PL Development

features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
[ DLitNegPrint when (!Add & Neg) ]

[ DLitAddPrint when (Add | !Neg) ]

[ DNeg when (Add & Neg),
DremAdd when (!Add & !Neg) ]

[ DNegPrint when (Add & Neg),
DLitEval when Eval,
DAddEval when (Add & Eval),
DNegEval when (Neg & Eval) ]
Extractive PL Development

Existing Products

- features: Lit, Add, Neg, Print, Eval
- configurations: Lit & Print
- deltas:
  - [DLitNegPrint when (!Add & Neg)]
  - [DLitAddPrint when (Add | !Neg)]
  - [DNeg when (Add & Neg), DremAdd when (!Add & !Neg)]
  - [DNegPrint when (Add & Neg), DLitEval when Eval, DAddEval when (Add & Eval), DNegEval when (Neg & Eval)]
Extractive PL Development

Existing Products

Feature Removal

| features | Lit, Add, Neg, Print, Eval |
| configurations | Lit & Print |
| deltas | [ DLitNegPrint \textbf{when} (!Add & Neg) ] |
| | [ DLitAddPrint \textbf{when} (Add | !Neg) ] |
| | [ DNeg \textbf{when} (Add & Neg), DremAdd \textbf{when} (!Add & !Neg) ] |
| | [ DNegPrint \textbf{when} (Add & Neg), DLitEval \textbf{when} Eval, DAddEval \textbf{when} (Add & Eval), DNegEval \textbf{when} (Neg & Eval) ] |
**Product Line Evolution**

Expression Product Line

- **Data**
  - Lit
  - Add
  - Neg
  - Sub

- **Operations**
  - Print
  - Eval

---

Delta-oriented Development of Software Product Lines
features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas

[ DLit,
  DAdd when Add,
  DNeg when Neg,
  DSub when Sub ]

[ DLitPrint when Print,
  DLitEval,
  DAddPrint when (Add & Print),
  DAddEval when Add,
  DNegPrint when (Neg & Print),
  DNegEval when Neg,
  DSubPrint when (Sub & Print),
  DSubEval when Sub ]
**Product Line Evolution**

features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas
[ DLit,
  DAdd when Add,
  DNeg when Neg,
  DSub when Sub ]

[ DLitPrint when Print,
  DLitEval,
  DAddPrint when (Add & Print),
  DAddEval when Add,
  DNegPrint when (Neg & Print),
  DNegEval when Neg,
  DSubPrint when (Sub & Print),
  DSubEval when Sub ]
### Product Line Evolution

#### Change in Feature Model

**Features**: `Lit, Add, Neg, Sub, Print, Eval`  
**Configurations**: `Lit & Eval & choose1(Neg,Sub)`  
**Deltas**:

- `[ DLit,  
  DAdd `when` Add,  
  DNeg `when` Neg,  
  DSub `when` Sub ]`

- `[ DLitPrint `when` Print,  
  DLitEval,  
  DAddPrint `when` (Add & Print),  
  DAddEval `when` Add,  
  DNegPrint `when` (Neg & Print),  
  DNegEval `when` Neg,  
  DSubPrint `when` (Sub & Print),  
  DSubEval `when` Sub ]`

---

#### New and Reconfigured Delta Modules
Pure DOP vs. Core DOP

- Core DOP PL can be transformed to Pure DOP PL by introducing core product in first delta module.
- Pure DOP PL can be transformed to Core DOP PL by adding empty product to Core DOP PL.
Conclusion

- Delta Modelling of Software Product Lines
- Model-driven Software Product Line Engineering
- Delta-oriented Programming with Delta Java
- Pure Delta-oriented Programming
Ongoing Student Projects

- Teaching Concept for Core DOP
- Relation between DOP and Refactoring
- Selection of Core Product in Core DOP
Open Student Projects

- Implementation of Pure DOP Compiler
- Integration of Pure/Core DOP into Feature IDE
- Visualization of Core/Pure DOP Product Lines
- Transformation of Core DOP Product Lines
- Patching Pure/Core DOP Product Lines
- Delta-oriented Model-based Development of SPL