Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

TECHNISCHE
UNIVERSITÄT
DARMSTADT

2015-06-23

Delta-Oriented Programming

FOP Issues and Possible Improvements

Feature-Oriented Programming (FOP) Issues

- 1:1 mapping of features to feature modules too rigid: resolving feature interaction breaks 1:1 mapping
- Implicit & inflexible feature module application order
- Feature modules cannot remove code

Improvements (Goals)

- Flexible mapping between features and modules
- Easily configurable application order
- Support **extractive**, **reactive** and **proactive** SPL development

Extractive, Reactive and Proactive SPL Development[1]

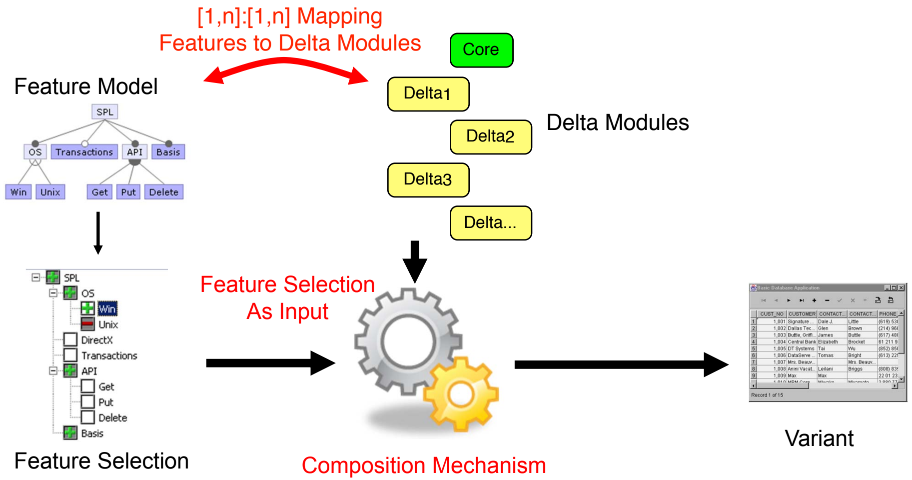
Extractive: develop SPL by reusing existing (legacy) systems: define products by extracting the required features (this requires ability to remove code).

Reactive: develop minimum number of features and products, reacting to immediate user requirements – when requirements are sketchy and likely to change. As requirements consolidate, incrementally expand the SPL.

Proactive: develop the complete SPL upfront – when requirements are well-defined and stable.

[1] [Krueger 2002]

Overview: Delta-Oriented Programming



Delta Modules (Deltas)

Basic idea: describe differences between variants in a modular way

Delta modules can…

- Add, remove or modify classes and interfaces.
- Permitted class modifications are:
  - addition and removal of fields
  - addition, removal and modification of methods
  - extending the list of implemented interfaces
  - (others, depending on the language)

Feature Modules vs. Delta Modules[2]

Approach	One	Multiple	Fragments	Extend	Replace	Remove
	Features			**Actions**		
Feature Module	X	-	-	X	X	-
Delta Module	X	X	X	X	X	X

[2][Schulze, Richers, and Schaefer 2013]

Delta-Oriented Programming Languages

DeltaJ[3]

- Extension of Java 1.5

Abstract Behavioural Specification (ABS)[4]

- Multi-paradigm (functional, object-oriented)
- Actor-based concurrency model
- Designed for abstract, yet precise modelling & specification

[3] https://www.tu-braunschweig.de/isf/research/deltas/
[4] http://www.abs-models.org/

SPL Engineering with the ABS Language

The ABS language provides four concepts to support software product line engineering:

1. Feature models
2. Product declarations
3. Deltas
4. SPL configuration

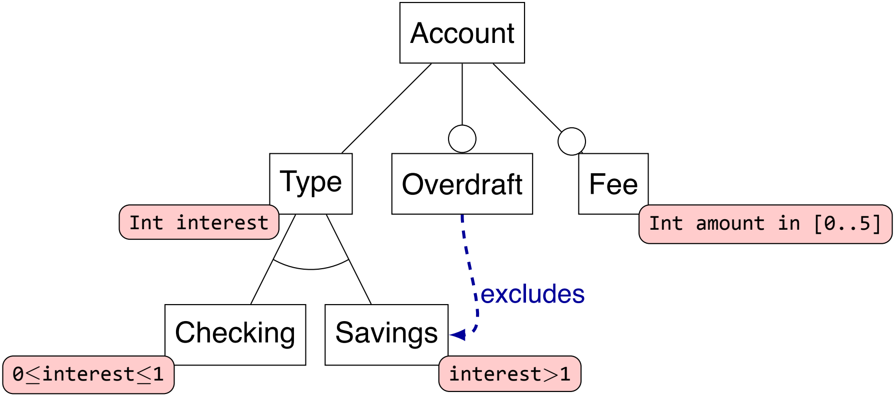
Feature Modelling in ABS

Based on "standard" feature modelling approach, with few extensions:

- Support for **feature attributes** (Integers and Booleans)
- Support for defining **additional (cross-tree) constraints**
- Support for **cardinalities** (specify precisely how many features out of a group of features)
- Feature models are expressed using **textual language** (as opposed to feature diagrams).

Feature Model Example[5]

Feature Diagram for an SPL of Bank Accounts



[5] [Hähnle 2013]

Feature Model Example (cont.)

ABS implementation of bank accounts feature model:

```
root Account {
  group allof {
    Type {
      group oneof {
        Checking {ifin: Type.interest == 0 || Type.interest == 1;},
        Savings {ifin: Type.interest > 1; exclude: Overdraft;}
      }
      Int interest; // interest rate of account
    },
    opt Fee {Int amount in [0..5];},
    opt Overdraft
  }
}
```

Feature Model Semantics

Straightforward translation of feature model to propositional logic formula over booleans and integers.

ABS compiler includes constraint solver that can:

- find all solutions (=valid feature selections) for a feature model,
- check whether given feature selection satisfies the feature model.

Products

Products are named feature selections, where feature attributes are assigned concrete values. Examples:

```
// Checking Accounts
product Basic (Type{interest=0}, Checking, Overdraft, Fee{amount=1});

product Earner (Type{interest=1}, Checking, Overdraft);

product Student (Type{interest=0}, Checking);

// Savings Accounts
product BasicSaver (Type{interest=2}, Savings);

product BonusSaver (Type{interest=6}, Savings);

// an invalid feature selection (does not satisfy feature model)
product SavingsWithOverdraft (Type{interest=1}, Savings, Overdraft);
```

Delta Modules (Deltas)

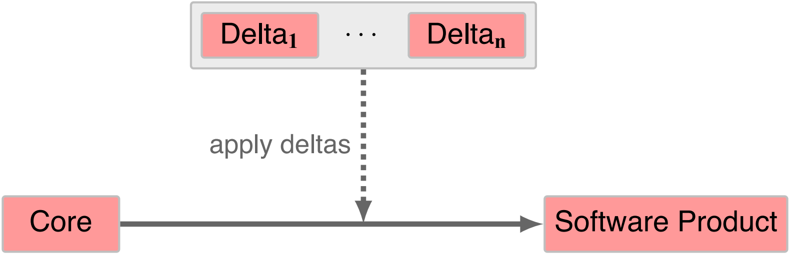
ABS has

- No subclassing, only subtyping (no **extends**, only **implements**)
- No traits, mixins, aspects, context layers, …

Instead: **Deltas**

- Base product (the **core**) with minimal (common) functionality
- Variants (products) obtained by applying **deltas** to base product
- Deltas allow **code reuse** across software products but not within individual products.

Application of Deltas



- ABS deltas add, remove or modify classes and interfaces
- Class modifications:
  add & remove fields; add, remove & modify methods, change list of implemented interfaces, …
- Given a product, compiler **determines the applicable deltas** and generates the corresponding software product.

Example: Core of Accounts SPL

```
module Account;

interface Account {
    Int deposit(Int x);
}

class AccountImpl(Int aid, Int balance) implements Account {
    Int deposit(Int x) {
        balance = balance + x;
        return balance;
    }
}
```

Example: Deltas of Accounts SPL

```
delta DFee (Int fee); // Implements feature Fee
uses Account;
modifies class AccountImpl {
    modifies Int deposit(Int x) {
        Int result = x;
        if (x >= fee)
            result = original(x-fee);
        else
            result = original(x);
        return result;
    }
}
```

```
delta DSave (Int i); // Implements feature Savings
uses Account;
modifies class AccountImpl {
    removes Int interest; // field removed & added with new initial value
    adds Int interest = i;
}
```

Accessing Previous Behaviour

Similarly to FOP…

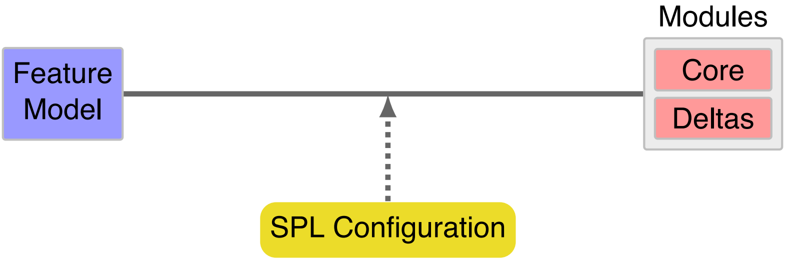
- Methods can be modified in several steps by applying several deltas in sequence.
- The “previous” method behaviour can be accessed by calling **original**.

As in previous example:

```
delta DFee (Int fee); // Implements feature Fee
uses Account;
modifies class AccountImpl {
    modifies Int deposit(Int x) {
        Int result = x;
        if (x >= fee)
            result = original(x-fee);
        else
            result = original(x);
        return result;
    }
}
```

Mapping Features to Deltas

Problem space: **Feature model** Solution space: **Core + Deltas**



How are they connected? **SPL Configuration**:

- **application conditions** to associate features and delta modules
- **temporal** delta ordering (partial)
- feature **attribute value** passing to delta modules

Mapping Features to Deltas

SPL Configuration Example

```
productline Accounts;
features Type, Fee, Overdraft, Check, Save;

delta DType (Type.interest) when Type;

delta DFee (Fee.amount) when Fee;

delta DOverdraft after DCheck when Overdraft;

delta DSave (Type.interest) after DType when Save;

delta DCheck after DType when Check;
```

- application condition (ensure suitable feature implementation)
- order of delta application (conflict resolution)
- feature attribute value passing

Mapping Features to Deltas (cont.)

Application conditions...

- Associate deltas with features.
- Are **boolean formulas** over features and feature attributes:
- **when** formula yields `true`, given delta is applied.
- Enable flexible mapping of feature to deltas: 1:1, 1:n, n:1, n:m

Examples:

```
productline ExampleSPL;
features A, B, C, F;

delta D1 when A; // Mapping feature A to delta D1 (1:1)
delta D2 when !A && (B || (C && F)); // features A, B, C, F to delta D2 (4:1)
delta D3 when C; // Mapping feature C to deltas D3...
delta D4 when C; // ...and D4 (1:2)
delta D5 when F.a >= 0; // Mapping feature attributes to deltas...
delta D6 when F.a < 0;
```

Delta Application Order

The order in which deltas are applied is important; deltas applied later in the sequence may depend on elements that have been added (or removed) earlier. Example:

```
delta D1;
modifies class C {
    removes Unit foo(); // remove method foo from class C
}
delta D2;
modifies class C {
    adds Int x=0;
    adds Int foo() { ... return x; } // re−add method foo in class C
}
```

Delta ordering ensures that delta `D2` is always applied **after** `D1`:

```
delta D1 when ...;
delta D2 after D1 when ...;
```

Delta Composition Mechanism

- How are the modifications described by a sequence of deltas applied to a core program?

Delta Composition Mechanism: Example

Core:

```
module Graph;
class Edge {
    Unit print() {
        System.out.print(" Edge between " + node1 + " and " + node2);
    }
}
```

Deltas:

```
delta Directed;
modifies class Graph.Edge {
    adds Node start;
    modifies Unit print() {
        original(); System.out.print(" directed from " + start);
    }
}
delta Weighed;
modifies class Graph.Edge {
    adds Node weight;
    modifies Unit print() {
        original(); System.out.print(" weighed with " + weight);
    }
}
```

Delta Composition Mechanism: Example (cont.)

- Method modifications are added to the class under different names. The method modification added by the last applied delta retains the actual method name.
- **original** calls are renamed to match the renamed methods.

```
module Graph;
class Edge {
   Unit core$print() {
      System.out.print(" Edge between " + node1 + " and " + node2);
   }
   Unit Directed$print() {
      Core$print(); System.out.print(" directed from " + start);
   }
   Unit print() {
      Directed$print(); System.out.print(" weighed with " + weight);
   }
}
```

Feature Interactions

Example: Doubly linked list (cf. FOP lecture)

forward link feature:

```
delta DForwardLink;
modifies class MyList {
    adds Node first;
    adds Unit insertAtEnd(Node n) {
        n.next = first; first = n;
    }
}
modifies class Node {
    adds Node next;
}
```

backward link feature:

```
delta DBackLink;
modifies class MyList {
    adds Node last;
    adds Unit insertAtBeginning(Node n) {
        n.prev = last; last = n;
    }
}
modifies class Node {
    adds Node prev;
}
```

SPL configuration:

```
productline ListPL;
features ForwardLink, BackLink;
delta DForwardLink when ForwardLink;
delta DBackLink when BackLink;
delta DDoubleLink after DForwardLink, DBackLink when ForwardLink && BackLink;
```

Conflict-Resolving Deltas

- Deltas can be used to resolve interactions (or any type of conflicts) among other deltas.
- Example: define a third delta that fixes our doubly linked list: Delta `DDoubleLink` is only applied when both `ForwardLink` and `BackwardLink` features are selected (as specified in the SPL configuration).

```
delta DDoubleLink;
modifies class MyList {
    modifies Unit insertAtEnd(Node n) {
        if (first == null) last = n; else first.prev = n;
        original(n);
    }
    modifies Unit insertAtBeginning(Node n) {
        if (last == null) first = n; else last.next = n;
        original(n);
    }
}
```

Conflict-Resolving Deltas (cont.)

After applying deltas `DForwardLink`, `DBackLink` and `DDoubleLink` to a (bare-bones) core program:

```
class MyList {
    Node first; Node last;

    Unit DForwardLink$insertAtEnd(Node n) {
        n.next = first; first = n;
    }
    Unit insertAtEnd(Node n) {
        if (first == null) last = n; else first.prev = n;
        DForwardLink$insertAtEnd(n);
    }
    Unit DBackLink$insertAtBeginning(Node n) {
        n.prev = last; last = n;
    }
    Unit insertAtBeginning(Node n) {
        if (last == null) first = n; else last.next = n;
        DBackLink$insertAtBeginning(n);
    }
}
class Node {
    Node prev; Node next;
}
```

Summary Delta-Oriented Programming

- **Feature traceability:** Similarly to FOP, features are traceable between problem and solution space.
- **Mapping features to code modules:** More flexible than FOP thanks to application conditions: a delta may implement more than one feature, but may also implement only fragments of one or more features.
- **Composition:** Delta application order can be controlled explicitly from within the language.
- **Conflict resolution:** Conflicts, such as feature interactions, can be easily resolved with conflict-resolving deltas.
- **SPL development:** Deltas are more general than feature modules as they can also remove code. Hence they support extractive, reactive and proactive SPL development.

This Week's Reading Assignment

- Wong, P. Y., Albert, E., Muschevici, R., Proença, J., Schäfer, J., and Schlatte, R. **The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems.** Journal on Software Tools for Technology Transfer 14 (2012), 567–588.
- Only necessary to read the relevant Sections 1 and 4.

- Download: `http://link.springer.com/article/10.1007/s10009-012-0250-1`
- Freely accessible from within the TUD campus network

References I

Hähnle, Reiner (2013). "The Abstract Behavioral Specification Language: A Tutorial Introduction". In: **Formal Methods for Components and Objects**. Vol. 7866. LNCS. Springer, pp. 1–37.

Krueger, Charles W. (2002). "Easing the Transition to Software Mass Customization". In: **Software Product-Family Engineering**. Vol. 2290. LNCS. Springer, pp. 282–293.

Schulze, Sandro, Oliver Richers, and Ina Schaefer (2013). "Refactoring delta-oriented software product lines". In: **Aspect-Oriented Software Development Conference**. AOSD '13. ACM Press, pp. 73–84.