Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

TECHNISCHE
UNIVERSITÄT
DARMSTADT

2015-06-16

Feature-Oriented Programming

Introduction

- There are many ways to implement Software Product Line variability.
- A widely use mechanism is **conditional compilation** using C preprocessor directives (`#ifdef...`)
- Examples: Linux kernel, many other open source projects

Variability by Conditional Compilation

```
static int _rep_queue_filedone(...)
    DB_ENV *dbenv;
    REP *rep;
    __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
    COMPQUIET(rep, NULL);
    COMPQUIET(rfp, NULL);
    return (__db_no_queue_am(dbenv));
#else
    db_pgno_t first, last;
    u_int32_t flags;
    int empty, ret, t_ret;
#ifdef DIAGNOSTIC
    DB_MSGBUF mb;
#endif
                              ...
```

Excerpt from Oracle's Berkeley DB

Conditional Compilation (cont.)

- Designed in the 1970s and hardly evolved since
- Not designed for large-scale variability
- Difficult to determine when which code is actually compiled into the system
- Programming errors are easy to make and difficult to detect
- Source code rapidly becomes a maze
- Preprocessor diagnostics are poor

Conditional Compilation Example

```
#include <stdio.h>
#ifdef WORLD
char * msg = "Hello World\n";
#endif
#ifdef BYE
char * msg = "Bye Bye World\n";
#endif
main() {
    printf(msg);
}
```

- Two features: `WORLD` and `BYE`
- Four possible products: $\emptyset$, $\{\texttt{WORLD}\}$, $\{\texttt{BYE}\}$, $\{\texttt{WORLD}, \texttt{BYE}\}$
- But: only two variants will actually compile
- How to define which products are valid?
- How to ensure products are type-safe (i.e. can be compiled) without generating each product?

Feature-Oriented Software Development (FOSD)

FOSD is a form of **Software Product Line Engineering**:

1. Domain engineering: develop set of reusable, variable assets.
2. Application engineering: construct software products (variants) by composing assets according to user requirements.

Method:

- Decompose a software system in terms of the **features** it provides.
- Specify dependencies between features using a **feature model**.
- Construct system variants based on **feature selection**.
- Features are **implemented** using a **feature-oriented programming** language.

Feature Traceability

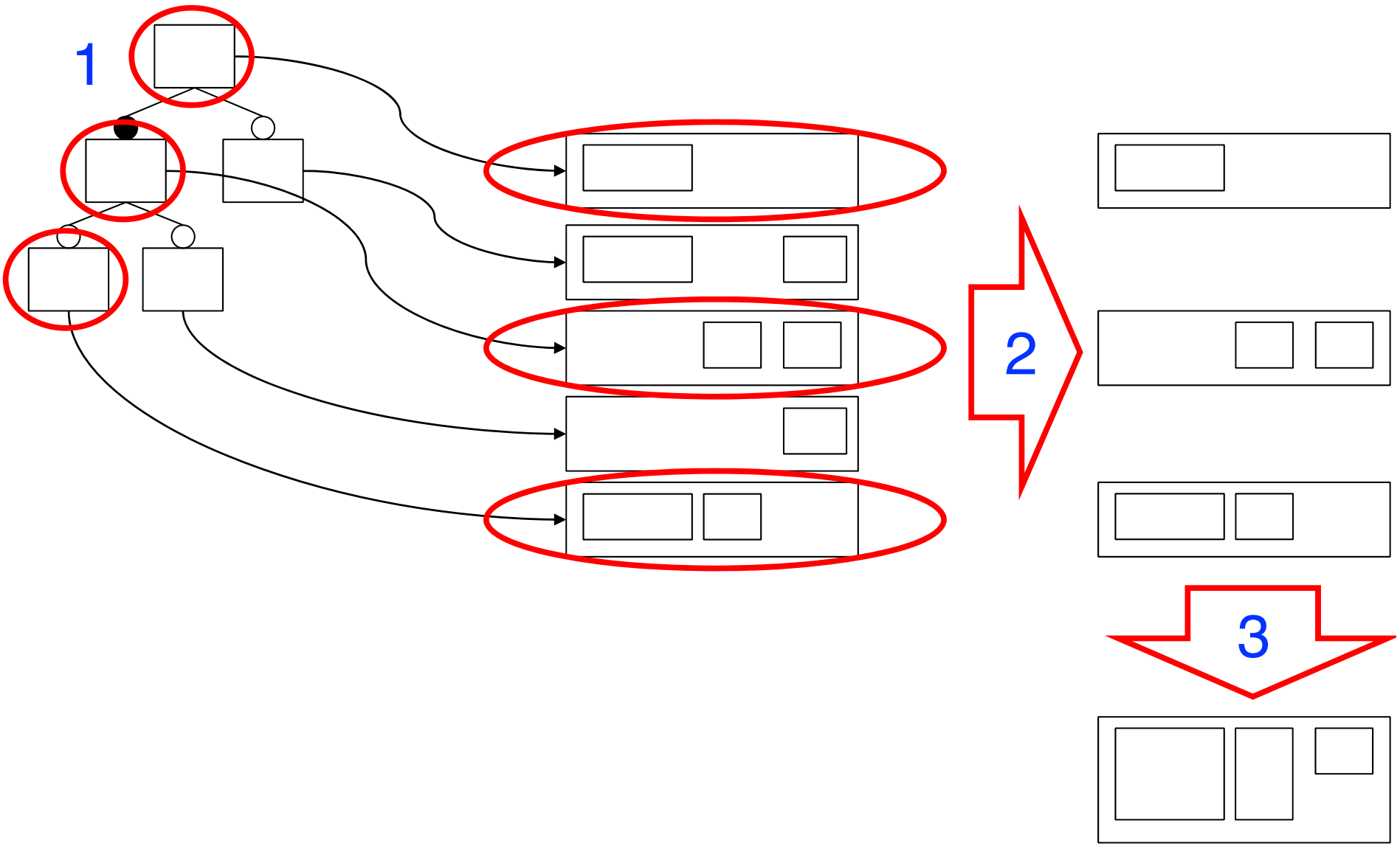
FOSD aims at the systematic application of the feature concept in all phases of the software life cycle. In other words, it establishes **feature traceability** between problem and solution space.

- **Transparency**: easy to see how a feature is implemented
- **Consistency**: if feature model changes, it is evident which parts of the implementation need to be adapted (and vice-versa).
- **Automation**: given a set of requirements, the appropriate system variant can be generated automatically.

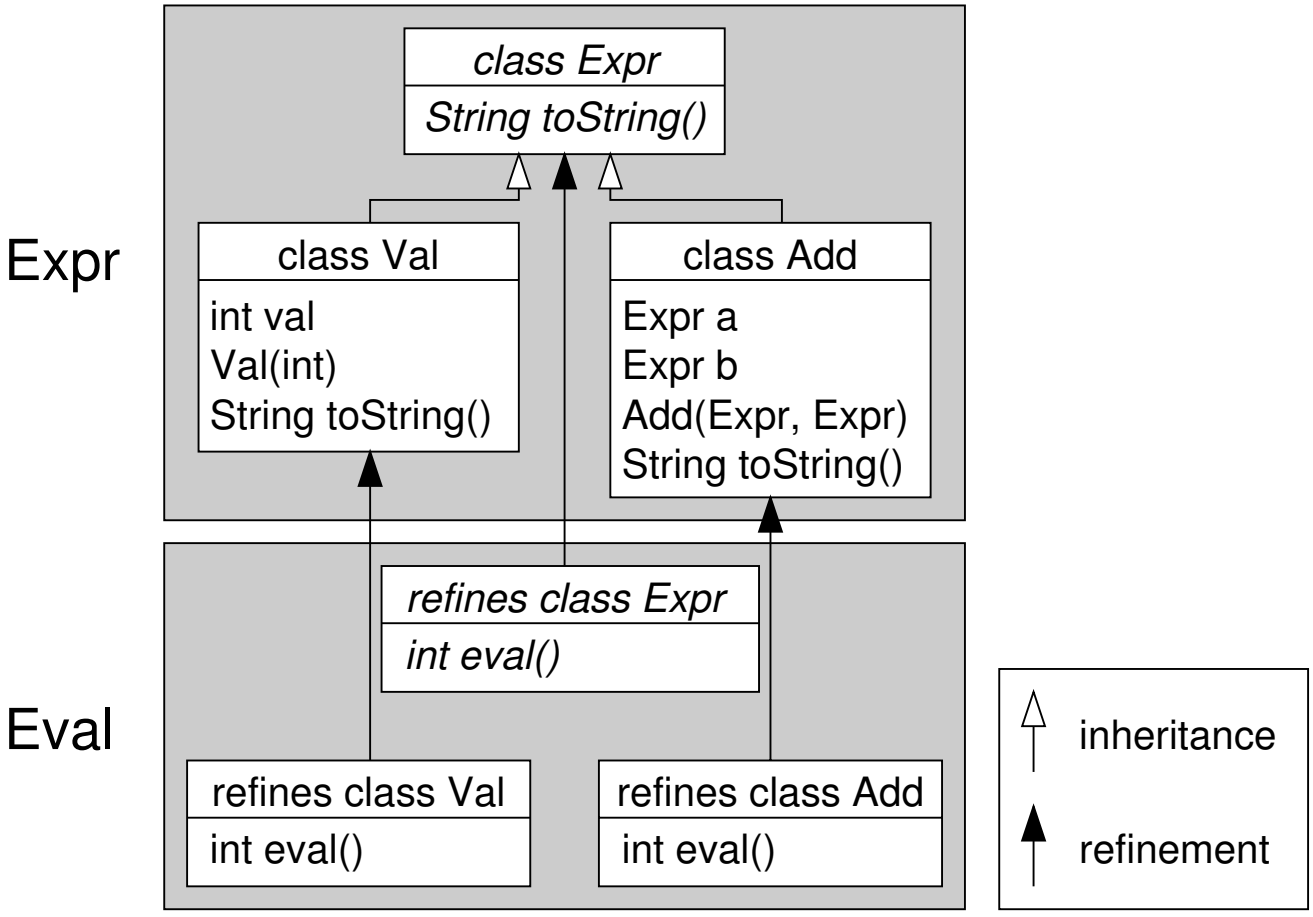
Feature-Oriented Programming (FOP)

- Language-based FOSD solution (extensions for Java, C, C++, C#, Haskell, etc.)
- Features are (often) cross-cutting concerns $\Rightarrow$ additional modularisation mechanism is needed (apart from classes).
- Each **feature** is implemented as a **feature module**.
- Variant generation by composition of feature modules
  1. Select features from feature model
  2. Select feature modules based on features
  3. Compose feature modules to obtain software product

Variant Generation by Feature Module Composition



Feature Modules: Example



A system that processes language expressions: `Expr` is the base program. The feature `Eval` adds the capability to evaluate expressions.

Example: `Eval` Feature Implementation with Jak[1]

Base program:

```
abstract class Expr { abstract String toString(); }
class Val extends Expr {
    int val;
    Val(int n) { val = n; }
    String toString() { return String.valueOf(val); }
}
class Add extends Expr {
    Expr a; Expr b;
    Add(Expr e1, Expr e2) { a = e1; b = e2; }
    String toString() { return a.toString() + "+" + b.toString(); }
}
```

`Eval` feature module:

```
refines class Expr { abstract int eval(); }
refines class Val {
    int eval() { return val; } }
refines class Add {
    int eval() { return a.eval() + b.eval(); }
}
```

[1] FOP extension of Java [Kästner and Apel 2013]

Feature Modules

- A feature module implements a feature (1:1 mapping).
- Features **crosscut** multiple classes $\Rightarrow$ a feature module refines multiple classes.
- A feature module **refines** the content of an object-oriented base program by
  - adding new classes,
  - adding new fields,
  - adding new methods or modifying existing methods.
- Feature modules are applied sequentially to a base program.
- The order in which feature modules are applied is important; earlier feature modules in the sequence may add elements that are refined by later features.

FOP Languages

- Many languages and variants with (slightly) different syntax
- For Java alone, several variants exist[2]: Jak, FeatureHouse, AHEAD, …
- Part of complete tool suite (e.g. Eclipse IDE) including tools for feature model specification, feature selection and variant generation.

[2] [Kästner and Apel 2013]

Accessing Previous Refinement

- Methods can be refined in several steps (by applying several feature modules in sequence).
- The “previous” method behaviour can be accessed by calling **original** or **Super** (depending on FOP dialect).

Accessing Previous Refinement: Example

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

AHEAD:

```
refines class Edge {
    private Node start;
    void print() {
        Super().print();
        System.out.print(" directed from " + start);
    }
}
```

FeatureHouse:

```
class Edge {
    private Node start;
    void print() {
        original();
        System.out.print(" directed from " + start);
    }
}
```

Feature Modules: Composition Mechanisms

Two approaches:

1. Flattening
2. Mapping refinements to class inheritance hierarchy

Feature Modules Composition: Flattening

- Refinements directly modify the target class.
- **Super/original** calls are inlined.
- Result is one single class.

Example of `Edge` class after refinement with two feature modules:

```
class Edge {
    private Node start;
    private int weight;
    void print() {
        System.out.print(" Edge between " + node1 + " and " + node2);
        System.out.print(" directed from " + start);
        System.out.print(" weighted with " + weigth);
    }
}
```

Feature Modules Composition: Inheritance Hierarchy

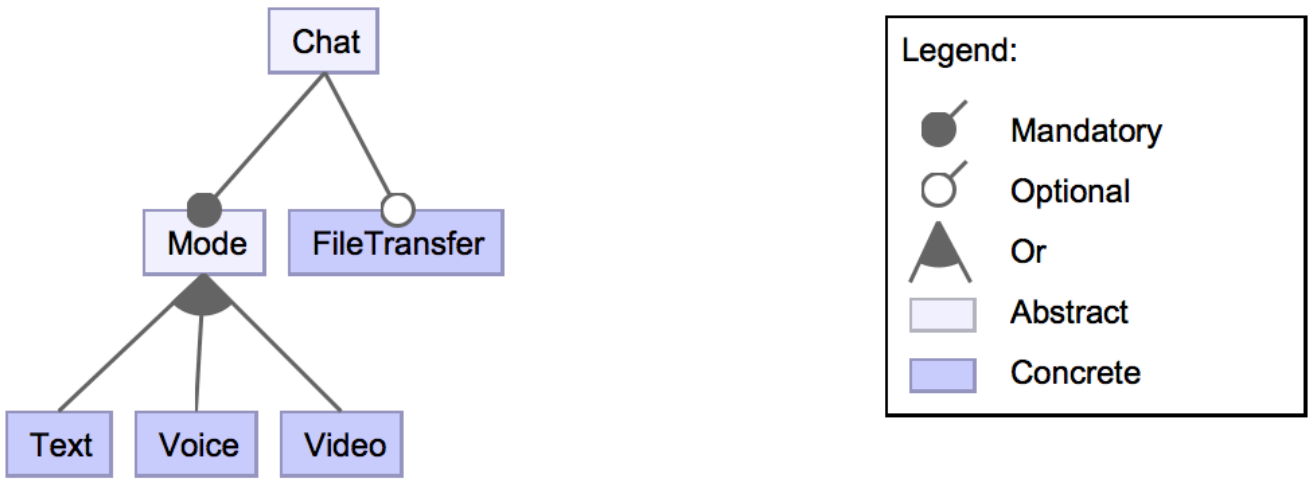
- A new subclass for each class refinement.
- Class renaming, such that subclass corresponding to final refinement has the actual class name.
- **Super** calls substituted with **super** calls.

```
class Edge$$Base {
    void print() { ... }
}
class Edge$$Directed extends Edge$$Base {
private Node start;
    void print() {
        super.print(); System.out.print(" directed from " + start);
    }
}
class Edge extends Edge$$Directed {
    private int weight;
    void print() {
        super.print(); System.out.print(" weighted with " + weight);
    }
}
```

Feature Modules: Application Order

- Feature modules are applied in fixed, linear order.
- Order is determined implicitly:
  by pre-order traversal of the feature diagram (tree).
- User-defined order possible.
- Same order applies to all configurations.

Example:



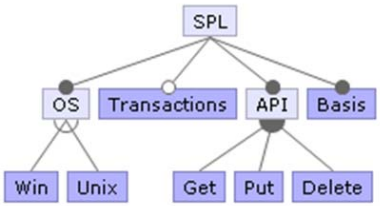
Application order: `Text`, `Voice`, `Video`, `FileTransfer`

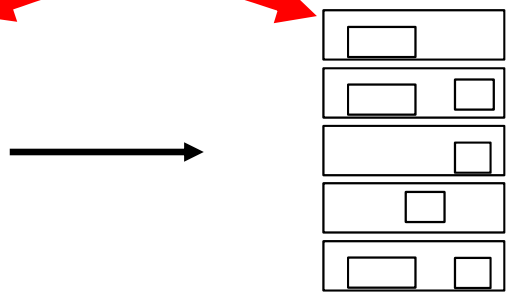
Overview: Feature-Oriented Programming

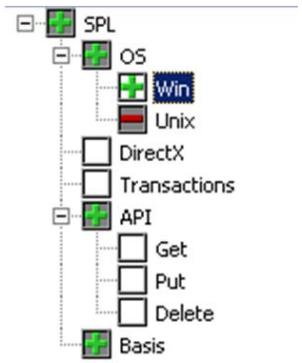
1:1 Mapping
Features to Feature Modules

Feature Modules

Feature Model



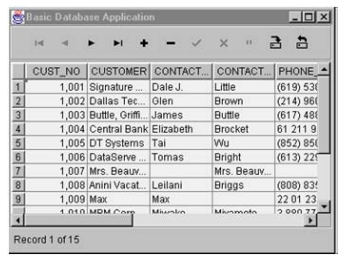




Feature Selection

Feature Selection
As Input

Composition Mechanism



Variant

Feature Interactions

A feature interaction is a situation in which two or more features exhibit unexpected behavior that does not occur when the features are used in isolation.

Example: Doubly Linked List[3]

forward link feature:

```
refines class List {
    Node first;
    void insertAtEnd(Node n) {
        n.next = first; first = n;
    }
}
refines class Node {
    Node next;
}
```

backward link feature:

```
refines class List {
    Node last;
    void insertAtBeginning(Node n) {
        n.prev = last; last = n;
    }
}
refines class Node {
    Node prev;
}
```

[3] [Apel and Kästner 2009]

Feature Interaction Example (cont.)

After applying both features to a (bare-bones) base program:

```
class List {
    Node first; Node last;
    void insertAtEnd(Node n) {
        n.next = first; first = n;
    }
    void insertAtBeginning(Node n) {
        n.prev = last; last = n;
    }
}
class Node {
    Node prev; Node next;
}
```

- When adding an element with method `insertAtEnd`, the backward reference `prev` is not adjusted accordingly and the reference to `last` is not set properly in the case the list is empty.
- List does not behave as intended: the two features **interact**.

Feature Interaction Example (cont.)

Corrected behaviour:

```
class List {
    Node first; Node last;
    void insertAtEnd(Node n) {
        if (first == null) last = n; else first.prev = n;
        n.next = first; first = n;
    }
    void insertAtBeginning(Node n) {
        if (last == null) first = n; else last.next = n;
        n.prev = last; last = n;
    }
}
class Node {
    Node prev; Node next;
}
```

- Highlighted code needs to be added to fix the problem.
- How/where to specify it?

Feature Interactions (cont.)

- Feature interactions require additional code to fix the unexpected behaviour.
- The additional code in part of neither feature.
- How/where to specify the additional fixing code?
- Possible solution: multiple feature modules per feature, some specifically for resolving interactions; apply “interaction fixing” modules only when interactions occur.
- Solution contradicts FOP idea of “one feature module for each feature” (violates the separation of concerns principle).

FOP Summary

Feature modules

- A system decomposition mechanism for modularising features
- Refine a base program
- Refinements add/modify classes: add fields, add or modify methods
- Composed iteratively (step-wise refinement)
- Composition is static (at compile time).
- Result of composition is a variant of a Software Product Line.

FOSD & FOP: Evaluation

- Features are easily traceable between problem and solution space due to 1:1 mapping of features to feature modules.
- But: 1:1 mapping between features and feature modules sometimes too rigid: not clear how to resolve feature interactions.
- Feature modules do not support removal of program elements.
- Rigid application order of feature modules.
- How to type-check/test/verify/analyse all SPL variants without generating each one? – Ongoing research problem.

This Week's Reading Assignment

- Kästner, C., and Apel, S. **Feature-oriented software development: A Short Tutorial on Feature-Oriented Programming, Virtual Separation of Concerns, and Variability-Aware Analysis**. In Generative and Transformational Techniques in Software Engineering IV, R. Lämmel, J. Saraiva, and J. Visser, Eds., vol. 7680 of LNCS. Springer, 2013, pp. 346–382.

- Download: `http://link.springer.com/content/pdf/10.1007/978-3-642-35992-7_10.pdf`
- Freely accessible from within the TUD campus network

References I

Apel, Sven and Christian Kästner (2009). “An Overview of Feature-Oriented Software Development”. In: **Journal of Object Technology** 8.5, pp. 49–84.

Kästner, Christian and Sven Apel (2013). “Feature-Oriented Software Development”. In: **Generative and Transformational Techniques in Software Engineering IV**. Vol. 7680. LNCS. Springer, pp. 346–382.

Slides partly based on the lecture “Software Product Lines” by Sven Apel et al., Passau University.