Multi-Paradigm Design with Feature Modeling in Aspect-Oriented Software Development

Erasmus Mobility at Lancaster University

Lecture 3

Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, Slovakia
vranic@fiit.stuba.sk
http://fiit.stuba.sk/~vranic/

September 16–19, 2008
Overview

1. Introduction
2. Feature Modeling
3. Aspect-Orientation and Software Product Lines
4. Multi-Paradigm Design with Feature Modeling for AspectJ
5. Summary
Introduction

- Notion of paradigm\(^1\)
  - large-scale view—traditional (object-oriented programming, functional programming, etc.); imprecise
  - small-scale view—paradigms as configurations of commonality and variability (map to directly to language mechanisms)

- Programming languages are often categorized according to (large-scale) paradigms they support
- Multi-paradigm languages: are there any other?
- Multi-paradigm design: how to select a paradigm appropriate for the problem being solved
- Multi-paradigm design with feature modeling for AspectJ

---

Multi-Paradigm Design with Feature Modeling

- Application Domain Feature Modeling
  - application domain related information
  - application domain feature model
- Transformational Analysis
  - application to solution domain mapping (paradigm instances)
- Solution Domain Feature Modeling
  - solution domain related information
  - solution domain feature model (paradigm model)
- Code Skeleton Design
  - code skeleton
Feature Modeling

- Captures connections among features and variability
- Feature model: a set of feature diagrams plus additional information
- Based on the notions of *domain*, *concept*, and *feature*
  - Features: common and variable
  - Concept instances: concept specializations
- Various notations exist, e.g. FODA, ODM, or Czarnecki-Eisenecker
- Notation used here is based on Czarnecki-Eisenecker feature modeling adapted to multi-paradigm design\(^2\)

---

Feature Variability (1)

- **Mandatory** features (edges ended by filled circles)
- **Optional** features (edges ended by empty circles)
Feature Variability (2)

- *Alternative* features (empty arc)
- *Or*-features (filled arc)
Feature Variability (3)

- Arcs modify the meaning of edges
  - Mandatory/Optional alternative features
  - Mandatory/Optional or-features
Feature Variability (4)

- Open features
  - Further variable subfeatures are expected
  - Denoted by square brackets; ellipsis is sometimes added
A feature can be included into a concept instance only if its parent has been included.

Features of all types may appear at any level.
Concept References

- Denoted by \( \mathbb{R} \) ((R) in diagrams in this presentation)
- May be expanded as needed
Binding time/mode

- When/how a feature will be bound
- Common binding times are source code, compile, link, load, and runtime
- Binding mode: static or dynamic
Additional Information in Feature Models

- Information associated with concepts and features
  - Textual information: description, presence rationale, inclusion rationale, note
  - Binding time/mode
- Constraints and default dependency rules
  - An example constraint:

\[ f1 \Rightarrow f6 \]
Feature Modeling

Concept instantiation

- C1
- f3
- f5
- f6
- f7

- compile time
- source time
- run time

- f2
- f4

- compile time
- source time
- run time
Parameterization in Feature Models

- Parameterized feature and concept names

  \[ \forall <i> \in N \ p<i> \cdot h \lor g \]

- Parameterized concepts

Aspects and Variability (1)

- In traditional approaches to implementation, the given feature code may be scattered across several components.
- This is especially important for variable features, because they are being bound and unbound according to the chosen configuration.
- Lee et al.\(^3\):
  - Common features implemented as usual
  - Variable features implemented with aspects

In reality, a more thorough analysis is needed for each feature in order to determine how it should be implemented.

General rules of aspect-oriented approach apply: features that crosscut other features should be implemented in the aspect-oriented way regardless of being variable or not.

Specific issues related to product lines should be considered (with respect to feature interdependencies).

Binding time should be considered, too.
Feature Interaction Problem

- Some features depend on other features
- A feature may require the presence or absence of another feature
- This relationship may be uni- or bidirectional
- Abstract aspects—the way how to separate dependencies
  - Dependencies are implemented as concrete pointcuts in concrete aspects
  - The functionality itself is in an abstract aspect
Multi-Paradigm Design with Feature Modeling

- Multi paradigm design with feature modeling (MPDfm)\(^4\) — a method for paradigm selection
- Software development process can be viewed as a mapping of the application (problem) domain to the solution domain
- Software development paradigm then determines how to express application domain concepts in terms of solution domain concepts
- Concepts of the solution domain correspond to programming language mechanisms
- Individual concepts of the solution domain (e.g., class in Java) may be considered as paradigms

The approach is based on Coplien's multi-paradigm design\(^5\)

---


Cplien’s Multi-Paradigm Design

Variability tables (from application domain SCVR analysis)

<table>
<thead>
<tr>
<th>Parameters of variation</th>
<th>Meaning</th>
<th>Domain</th>
<th>Binding</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output medium</td>
<td>...</td>
<td>Database, RCS file, TTY, UNIX file</td>
<td>Run time</td>
<td>UNIX file</td>
</tr>
<tr>
<td>Structure, Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Family table (from solution domain SCVR analysis)

<table>
<thead>
<tr>
<th>Commonality</th>
<th>Variability</th>
<th>Binding</th>
<th>Instantiation</th>
<th>Language Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related operations and some structure (positive variability)</td>
<td>Algorithm (especially multiple), as well as (optional) data structure and state</td>
<td>Compile time</td>
<td>Optional</td>
<td>Inheritance</td>
</tr>
<tr>
<td></td>
<td>Algorithm, as well as (optional) data structure and state</td>
<td>Run time</td>
<td>Optional</td>
<td>Virtual functions</td>
</tr>
</tbody>
</table>
MPD_{FM} Activities (repeated)

- Application Domain Feature Modeling
- Solution Domain Feature Modeling
- Transformational Analysis
- Code Skeleton Design

application domain related information → Application Domain Feature Modeling → application to solution domain mapping (paradigm instances) → code skeleton

solution domain related information → Solution Domain Feature Modeling → code skeleton

application domain feature model → Transformational Analysis → solution domain feature model (paradigm model)
Modeling Paradigms in MPD_{FM}

- Paradigm identification
  - Directly and indirectly usable paradigms
  - Paradigm hierarchy
- Binding time identification
  - Determining the sequence of binding times available in the solution domain
  - E.g., in AspectJ method body has the runtime binding
- First-level paradigm model
- Modeling individual paradigms
First-Level Paradigm Model

- First-level paradigm model consists of directly usable paradigms
  - Features of the solution concept
  - Introduced as concept references (usually in plural)
  - Their variability and binding time has to be determined

- Example: AspectJ first-level paradigm model

```
 AspectJ Program
  +---+    +---+    +---+    +---+    +---+    +---+    +---+  
  |   |    |   |    |   |    |   |    |   |    |   |    |   |    |   |    |
  V   |    V   |    V   |   V    V   |    V   |   V    V   |   V    V   |
```

- Classes(R)
- Interfaces(R)
- Inheritances(R)
- Aspects(R)
Modeling Individual Paradigms

- Each paradigm is introduced in a separate feature diagram
  - Solution domain concepts
  - May refer one to another

- Auxiliary concepts
  - Concepts that paradigms refer to
  - But they are not considered to be paradigms themselves

- Binding time (variable features)
- Instantiation (e.g., class–objects) is modeled with features
Structures and Relationships

- Structural paradigms correspond to the main constructs (structures) of the programming language.
- Relationship paradigms are about relationships among the programming language structures.
- In transformational analysis a node in the application domain feature model:
  - May correspond to the root of a structural paradigm.
  - But can’t correspond to the root of a relationship paradigm.
AspectJ Aspect-Oriented Paradigms

- Aspect paradigm—structural paradigm (modularization)
- A container for further aspect-oriented paradigms: advice, pointcut, and inter-type declaration
- These paradigms are structural paradigms (corresponds to their task—to capture crosscutting concerns)
Constraint: abstract ∨ final
Advice and Pointcut

Constraints:
1. \( \text{abstract} \lor \text{Body} \)
2. \( \text{Access} \Rightarrow \text{Name} \)
Transformational Analysis in MPD_{FM}

- Bottom-up instantiation of paradigms over application domain concepts at source time
- Application domain concepts are considered one by one
  1. The corresponding structural paradigm is determined
  2. The corresponding relationship paradigms for each relationship in it are determined
Paradigm Instantiation in MPD_{FM}

- Paradigm instantiation in MPD_{FM} is actually concept instantiation
  - Understood as concept specialization
  - Concept instances are represented by feature diagrams
  - Binding time is being taken into account

- Bottom-up instantiation

- Inclusion of paradigm nodes is determined by the mapping of the nodes of application domain concepts
  - Conceptual correspondence
  - Binding time correspondence
Transformational Analysis Example (1)
Transformational Analysis Example (2)
Transformational Analysis Example (3)
Transformational Analysis Example (4)
Multi-Paradigm Design with Feature Modeling in Aspect-Oriented Software Development

Multi-Paradigm Design with Feature Modeling for AspectJ

Transformational Analysis Example (5)

Debugging Code
Memory Management
inserting line
removing line
static Debugging Code
reading writing
Advice
around
after
throwing
returning
before
Body
Pointcut(R)
context
Return value type
Type(R)
Type(R)
Type(R)
Debugging Code.File
Pointcut(R)
Transformational Analysis Example (6)
Transformational Analysis Example (7)
Debugging Code
Memory Management
Inserting line
Removing line
Advice
Before
After
Returning
Throwing
Return value type
Debugging Code.File (R)

Pointcut
Context
Body
Final
Static join points
Dynamic join points
Join points
Compile time
Run time
Type (R)
Type (R)
Type (R)

Multi-Paradigm Design with Feature Modeling in Aspect-Oriented Software Development
Multi-Paradigm Design with Feature Modeling for AspectJ
Transformational Analysis Example (9)
Transformational Analysis Example (10)
Transformational Analysis Example (11)
Transformational Analysis Example (12)
Multi-Paradigm Design with Feature Modeling in Aspect-Oriented Software Development

Multi-Paradigm Design with Feature Modeling for AspectJ

Transformational Analysis Example (13)
Code Skeleton Design

- Code is designed by traversing the trees of paradigm instances
- Structural paradigm instances are considered first
- Example: the aspect of the file debugging code

```java
aspect FileDC {
    before(File f): target(f) && call(* File.read(..)) {
        ...
    }

    after(File f): target(f) && call(* File.write(..)) {
        ...
    }
}
```
Summary

- **MPDFM**: a method of paradigm selection based on feature modeling
- Paradigms are viewed as solution domain concepts
- The key activity: transformational analysis performed as a bottom-up paradigm instantiations over application domain concepts
- Transformational analysis can be applied to all application domain concepts, but can also be restricted to critical ones
- The AspectJ paradigm model
- Further research:
  - Use of MPDFM for early aspect identification
  - Use of feature modeling adapted to MPDFM to deal with the interaction of aspect-oriented change realizations