Managing Variability in Product-Lines

Markus Voelter
voelter@acm.org
http://www.voelter.de

This work is supported by
About me

- Independent Consultant
- Based out of Göppingen, Germany
- Focus on
  - Model-Driven Software Development and DSLs
  - Software Architecture
  - Product Line Engineering

Markus Völter
voelter@acm.org
www.voelter.de
About AMPLE

- AMPLE == Aspect-Oriented, Model-Driven Product Line Engineering
  (buzzwords are important to get funded 😊)

- EU-funded research project

- I am working with SIEMENS, building practically useful tooling and verifying it with a case study

- Resulting tooling based on Eclipse/openArchitectureWare and is freely available from eclipse.org/gmt/oaw
  - Version 4.2 that includes all of them has been released Sept 17
About openArchitectureWare

- Well-known (and much used) toolkit for most aspects of **model-driven software development**

- Open Source at Eclipse GMT
  - **integrates** w/Eclipse Modeling projects (eg. EMF, GMF)
  - **Contributes** to various Eclipse Modeling projects (Workflow Engine, Model-to-Text, Textual Modeling Framework)

- Version **4.2** is current, has been released Sept. 2007

**Some Features:**
- Constraint Checking, Code Generation, Model-to-Model Transformation
- OCL-like expression language used throughout the tool
- Xtext Framework for building textual DSLs and Editors
- Support PLE in models, generators and transformations via AOP
- Editors and Debuggers for all those languages integrated in Eclipse
Managing Variability in Product Lines

CONTENTS

• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
Managing Variability in Product Lines

CONTENTS

• Introduction and Concepts
  • PLE and Variabilities
    • MDSD and AOSD
    • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
The idea of PLE is to not develop software products as single artifacts, but rather to develop a family of related products as efficiently as possible.

We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.

Definition by Parnas
Variability Analysis: A central building block for PLE

- **Variability analysis** discovers the variable and fixed parts of a product in a domain. Parts can be
  - Structural or behavioral
  - Functional or non-functional (technical)
  - Modularized or aspectual

- Central challenges wrt. to variabilities are:
  - **Identification**: where are the variabilities, what are the options?
  - **Kind of variability**: see above
  - **Description**: how do I describe the allowed alternatives
  - **Management**: what are the constraints between the various variation points
  - **Implementation**: how do I implement the respective variability in my software system?
Negative vs. Positive Variability

• Negative Variability (a) takes optional parts away from an „overall whole“
  • **Challenge:** the „overall whole“ can become really big and unmanageable

• Positive Variability (b) adds optional parts to a minimal core.
  • **Challenge:** How to specify where and how to join the optional parts to the minimal core

• **In Practice:** combine both
Managing Variability in Product Lines

Structural vs. Non-Structural Variability

- **Structural Variations**
  - Example Metamodel
    - Based on this sample metamodel, you can build a **wide variety of models**:

- **Non-Structural Variations**
  - Example Feature Models
    - Dynamic Size, ElementType: int, Counter, Threadsafe
    - Static Size (20), ElementType: String
    - Dynamic Size, Speed-Optimized, Bounds Check
Managing Variability in Product Lines

Routine Configuration vs. Creative Construction

• The **expressive power** of the language used to bind the variability (select an option) can vary widely

![Diagram showing Routine Configuration and Creative Construction]

- Guidance, Efficiency
- Complexity, Flexibility

- Configuration Parameters
- Property Files
- Wizards
- Feature-Model Based Configuration
- Tabular Configurations
- Graph-Like Languages
- Frameworks
- Manual Programming

• This slide (adopted from K. Czarnecki) is **important for the selection of DSLs** in the context of MDSD in general:

• The more you can move your DSL to the configuration side, the simpler it typically gets.
Typical Binding Times & Techniques

• For each of the variable features you need to define when you’ll bind the feature

  • **modeling time:** DSLs, transformations, generators

  • **source time:** manual programming

  • **Compile time:** function overloading, precompiler, template evaluation, static aspect weaving

  • **deployment/configuration time:** component deployment (impl. for an interface), environment variables

  • **link time:** DLLs, class loading

  • **run time:** virtual functions, inheritance & polymorphism, factory-based instance creation, delegation, meta programming, data driven (tables, interpreters)
Typical Binding Times & Techniques

• For each of the variable features you need to define when you’ll bind the feature

  • **modeling time:** DSLs, transformations, generators

  • **source time:** manual programming

  • **Compile time:** function overloading, precompiler, template evaluation, static aspect weaving

  • **deployment/configuration time:** component deployment (impl. for an interface), environment variables

  • **link time:** DLLs, class loading

  • **run time:** virtual functions, inheritance & polymorphism, factory-based instance creation, delegation, meta programming, data driven (tables, interpreters)
• Introduction and Concepts
  • PLE and Variabilities
  • **MDSD and AOSD**
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
What is MDSD?

- **DSL is defined** for a domain formalizing domain concepts into a custom meta model.
- Developer develops **model(s)** based on DSL
- Using **code generators**, the model is transformed to executable code (interpreters are also possible)
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.
Models & Meta Models

• A model is an abstraction of a real world system or concept.
  • It only contains the aspect of the real world artifact that is relevant to what should be achieved with the model.
  • A model is therefore less detailed than the real world artifact.

• MDD models are precise and processable.
  • Complete regarding the abstraction level or viewpoint.
  • The concepts used for building the model are actually formally defined.
  • The way to do this is to make every model conform to a meta model.

• The meta model defines the “terms” and the grammar we can use to build the model.
  • Models are instances of their respective meta models.
A Domain Specific Language (DSL) is a **formalism to build models**. It encompasses
- the **meta model** of the models to be built
- some textual or graphical (or other) **concrete syntax** that is used to represent ("draw") the models.

In the context of product line engineering **DSLs are used to bind variabilities**.
- Consequently, feature diagrams are a special kind of DSL, one that can be used to express **configurative variability**.
What is AOSD?

- Developer develops **program code**
- Developer develops (or reuses) **aspect code**
- Developers specifies the **weaving rules** (defines pointcuts)
- Aspect Weaver **weaves program and aspects together** and produces the „aspectized“ program
  - This may happen statically or dynamically
**Introduction and Concepts**
- PLE and Variabilities
- MDSD and AOSD
- **MD-AO-PLE**

**Implementation Techniques**
- Intro to Case Study
- Models, Code, Transformations
- Orthogonal Variability
- Transformation and Template AO
- AO Modeling
- Aspects on Code Level
- Negative Variability

**Summary**
The challenge of implementing and handling variabilities is more easily addressed on model level than on code level.

Models are more abstract and hence less detailed than code.

Thus, the variability is inherently less scattered, making variability management on model level simpler!
MD-AO-PLE building blocks

- **Variability can be described more concisely** since it is described on model level.

- The **mapping from problem to solution space** can be formally described using model-to-model transformations.

- AO enables the explicit expression and **modularization of crosscutting variability**: 
  - In models: **weaving models** and meta models
  - In transformation: **weave variant aspects into transformations** and generators
  - In code: **implement fine-grained** implementation variants.

- Additional benefit: **Fine grained traceability** is supported since **tracing is done on model element level** rather than on the level of artifacts.
**Definition:**
MDD-AO-PLE uses **models** to describe product lines. **Variants** are defined on model-level. **Transformations** generate running applications. **AO techniques** are used to help define the variants in the models as well as in the transformers and generators.
CONTENTS

• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
Intro to Case Study

- A **home automation system** called Smart Home.

- In homes you will find a wide range of electrical and electronic **devices**
  - lights
  - thermostats
  - electric blinds
  - fire and smoke detection sensors
  - white goods such as washing machines
  - as well as entertainment equipment.

- Smart Home **connects those devices** and enables inhabitants to monitor and control them from a **common UI**.

- The home network also allows the devices to **coordinate their behavior** in order to fulfill complex tasks without human intervention.
Problem Space Modeling

- The **domain expert** (i.e. a building architect) uses a suitable modeling language for building smart homes.

- Currently, we use a **simple tree editor** for that (based on Exceed, and it is basically an EMF tree view with customized icons and labels).

- Note that problem space modeling uses a **creative construction DSL** since describing a Smart Home is not just a matter of “ticking boxes”.

- A more convenient editor will be provided later.
• Solution Space is made up of a **component-based architecture** (CBD level) subsequently mapped to OSGi (OSGi-level)

• Various **meta models** describe all these different levels
Managing Variability in Product Lines

CONTENTS

• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
Application Domain to Software Domain Transformation

• We use an M2M transformation to map from the application domain to the software domain.

• Here are some examples of what that transformation has to do:
  • Lighting:
    - For each light in a room, instantiate a light driver component
    - For each light switch, instantiate a light switch component
    - For each room with lights, instantiate a light controller, that manages lights and the connected switches
  • Windows:
    - For each window, instantiate a window sensor component

• Note how the transformation only instantiates and connects software components. The components themselves are pre-built and are available in libraries.
• A house with only **one level**, and **two rooms**, connected by **doors**.

• The rooms have **windows** as well as **lights** and **light switches**.
Example House: The Transformed CBD Model

- For each of the lights and switches we have **instances** of driver **components** (the component types are taken from the library)

- We also have a **light switch coordinator** component instance for each floor that has light switches.

- We use **query based connectors** to connect the coordinator with the lights and the switches.
  - The query dynamically finds all lights and switches for a given floor, dynamically at runtime.

- We also have hierarchical configurations for the building and floors.
Example House: The Transformed OSGi Model

- **Leaf configurations** have been transformed into **bundles**.

- **Interfaces** (from the Lib!) are now **Services** in this model.

- **Component instances** have become **OSGI-level components** of the appropriate type.
  - Those use ServiceRefs with queries to find the respective provided services at runtime.

- Note how the **mixin model** specifies the root packages for the bundles to enable code generation.
Example House: Code Generation

- Library
  - Base Classes
  - Manually Written
- Output
  - OSGi Code
  - Build Files
  - Compile/build...

```
cbd.generator/*.xpt
M-OSGI
osgi.generator/*.xpt
```
Example House: Generated Code

- We generate the **OSGi bundle activators** which
  - Instantiate the components deployed in that bundle
  - Register the services of those components
  - Register generated service trackers for each of the component’s service refs ... using an LDAP expression to dynamically find the provided services

- We generate a **manifest file**
  - including the correct package exports and imports

- We generate an ant **build file** to assemble the bundle JARs
  - JAR will contain OSGI-level code as well as the CBD level code
  - The used libraries know their Eclipse project so we know from where we need to grab the implementation source code

- We generate a **batch file** that runs the OSGi runtime (Knopflerfish) with the correct configuration (xargs-file)
Component Libraries

• Library components are **predefined building blocks** to be used in products. There are three “flavors”:

• **Code-Only**: the aspect of the PL that is covered by the library component is not supported by generators,
  • The production process for the product will simply include/link/instantiate/deploy the component if it’s required as part of a product.
  • Example: an optional SNMP agent running on a system node

• **Model-Only**: PLA contains generators that can completely generate the component implementation from a model.
  • If the generator changes, the library component’s implementation is automatically adapted (since it’s regenerated).
  • Example: A reusable business process component specified as a component with an associated state machine

• **Model/Code Mix**: This is necessary if you can represent some aspects of a component via a model, but cannot represent others.
Model/Code Mix: The different levels of code

- There are **two kinds of source code** in the system.

- CBD-level code is **partly generated/partly hand-written**.
  - As the name implies, it does **not depend** on the concrete **implementation technology** (such as OSGi)
  - Base classes (and other skeleton artifacts) are **generated**, the manually written code is **integrated** in well-defined ways
  - This is the way, manually written **business logic** is integrated.

- Implementation-level code is **completely generated**
  - It is **specific** to the concrete **implementation technology**
  - It **wraps** or **uses** the CBD-level code and adapts it to the concrete implementation technology

- The **generation process is separated** into two phases, one for each kind of source code.
EconomyLib: An example library

- The EconomyLib library contains pre-built components, interfaces and data types that are needed for building Smart Homes of the Economy variety.

- Interfaces and data types are model-only, whereas components are model/code mixed, because they contain manually written code parts.

- Libraries such as the EconomyLib are CBD-level code. There is absolutely nothing in there that is specific to the concrete implementation technology.

- The library comes with a model file as well as a source code directory.

- Note that this library depends on another library that defines basic primitive types.
### EconomyLib: Part of the Model

- **The **LightSwitchCoordinator** orchestrates lights and switches**
- **The **LightSwitchDriver** proxies a light switch**
  - The state knows whether the switch is pressed or not
- **The **LightDriver** proxies an actual light**
  - Its state has an ID and it knows whether it is burning
- **ILightSwitch** is used to query a switch whether it is pressed
- **ILightDriver** can be used to turn a light on or off
EconomyLib: Generating CBD-Level code
This is the component that switches lights based on the status of the switches.

It is a periodic component, hence it has only an `execute()` operation.

Note how it uses the `switchesAll()` operation to access all the switches it is connected to.
Contents

• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
    • Transformation and Template AO
    • AO Modeling
    • Aspects on Code Level
    • Negative Variability

• Summary
Orthogonal Variability Management

- Orthogonal Variability: all the artifacts of a system **depend on a central configuration** model that describes the variability of those artifacts.

- Here: the “system” is the **MDSD tooling** for developing Smart Homes
• oAW comes with a feature that allows domain architecture artifacts to **depend on whether certain features are selected**.

• An API is available that allows to plug in various **feature modeling tools**
  • In the simplest case, that API can be bound to a simple text file that contains a list of selected features.
  • Another binding is available to Pure Systems’ pure::variants tool

• That configuration model **controls various aspects** of the model transformation and code generation process.
  • It is read at the beginning of the workflow and is available globally.

• Currently, we use it for the following **optional features**:
  • Tracing
  • Reflective Data Structures
  • Viewer (UI)
  • Automatic Windows
Managing Variability in Product Lines

Orthogonal Variability Management III

- The configuration is done via a pure::variants variant model (ps:vdm)

- pure::variants supports the interactive selection of features, while evaluating constraints and feature relationships to make sure only valid variants are defined.

- If a constraint is violated, the model is either automatically corrected, or an error is shown.
Managing Variability in Product Lines

CONTENTS

- Introduction and Concepts
  - PLE and Variabilities
  - MDSD and AOSD
  - MDSD-AO-PLE

- Implementation Techniques
  - Intro to Case Study
  - Models, Code, Transformations
  - Orthogonal Variability
  - Transformation and Template AO
    - AO Modeling
    - Aspects on Code Level
    - Negative Variability

- Summary
AO for generator artifacts

- Aspect Orientation is used to **encapsulate and “inject” transformation and generator code** that is only necessary for implementing a given feature.

- Transformation and generator aspects are **captured in separate files**.

- These files are only deployed **iff a certain feature is selected** in the configuration model.

- The **workflow** ties all these loose ends together.
**Optional Feature: Logging**

- Logging is simply about writing a stdout **log of the methods called** on Service Components as the system runs.
  - The runtime infrastructure (OSGI-level) supports the use of **interceptors** for any component.
  - Interceptors are available in **libraries** (just as the light switch components and their interface and the primitive types)
  - If the **model configures interceptors** for a given component, the generated activator actually instantiates them, **instantiates a proxy** for each component and **adds the interceptors to that proxy**.

- **In short**: if the feature debug.logging is selected, the transformation from PS to CBD level must make sure that the appropriate interceptor is configured for the components.
Optional Feature: Logging [Thumbnail]
Optional Feature: Logging, Implementation

- The implementation uses **AO for the model transformation language.** Here is the aspect:

```java
// logging. ext
import psmm;
import cbdmm;

extension ps2cbd;

extension org::openarchitectureware::util::stdlib::io;
extension org::openarchitectureware::util::stdlib::naming;

around ps2cbd::transformPs2Cbd(Building building):
  let s = ctx.proceed(): (  
    building.createBuildingConfiguration().
    deployedInterceptors.addAll(
      { utilitiesLib().interceptors.findByName("LoggingInterceptor") }  
    ) ->  
    s
  );
```

- We **advise** `ps2cbd::transformPs2Cbd`
- We then execute the **original definition** (`ctx.proceed()`)  
- Then we **add**, to the top level config, the `LoggingInterceptor`
Optional Feature: Logging, Implementation II

- Remember we only want to have these interceptors in the system **iff the feature debug.tracing is selected** in the global configuration model.

- That dependency is expressed in the workflow:

```xml
<component id="xtendComponent.ps2cbd" class="oaw.xtend.XtendComponent">
...
</component>

<feature exists="debug.logging">
  <component adviceTarget="xtendComponent.ps2cbd" class="oaw.xtend.XtendAdvice">
    <!-- references tracing.ext, file that contains aspect on prev. slide -->
    <extensionAdvices value="logging"/>
  </component>
</feature>
```

- The stuff inside the `<feature>...` tag is only executed **if the respective feature is selected** in the global configuration.

- The XtendAdvice component type is an aspect **component for the Xtend component** used for transforming models.
Optional Feature: Logging, Implementation III

- An Advice component basically takes the sub-elements and adds them to the component referenced by the `adviceTarget` attribute.

- In the case here, that target is the one that runs the PS to CBD M2M transformation.

- Using this mechanism, the configuration of aspect code (the `<extensionAdvices>` element) is **non-invasive**.
Optional Feature: Component State Viewer

• The viewer UI shown before is not generated. It is a **generic** piece of code that **reflects on the data structures** that it is supposed to render.

• To make this work, the following two additions have to be made to the generated system:

  • The component state data structures must feature a **generated reflection layer**
  
  • Whenever a component is instantiated in the activator, its state **has to be registered** with the viewer.

• These things are implemented using **generator aspects**, depending on the selection of the **debug.viewer** feature.
• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
    • Aspects on Code Level
    • Negative Variability

• Summary
• AO Modeling (aka Model Weaving) is about **applying AO techniques** for models and meta models.

• Aspect Models capture the parts of models that **represent elements** necessary for a “implementing” a given **feature**.

• **Pointcut expressions** are used to determine **where and how** those aspect models are “woven” into base models.

• A **model weaver** does the weaving.

• Note that AO modeling is NOT about drawing UML diagrams of AspectJ code… contrary to what some people suggest!
Optional Feature: Automatic Windows [Thumbnail]

- building model
- building aspect
- configuration model
- woven building model
  - transform
  - weave
  - read
Optional Feature: Automatic Windows

- Automatic windows are an optional feature on the PS level.
  - If we have at least one thermometer in a room,
  - We can automatically open the windows if the temperatures are above 25°C average, and close them if we are below 20°C.
  - We also need windows actuators for that.

- We want this feature, if the global configuration model has the environmentalControl.tempManagement.automaticWindows feature selected.

- To implement it,
  - We weave the necessary elements into the PS model
  - Advice the PS to CBD transformation to consider these additional elements
  - ... and then (for debugging purposes) write the modified model to an XMI file.
Optional Feature: Automatic Windows, Implementation

- Here is the **aspect** for the problem space model:

```
aspect.xml
```

- Here are the **pointcut expressions** used in the aspect model:

```
rooms( Building this ):       
    floors.rooms.select(e|e.windows.size > 0) ;
windows( Building this ):    
    rooms().windows;
thermoName( Thermometer this ):  
    ((Room)eContainer).name.toFirstLower()+"Thermometer";
```

- **rooms** returns all the rooms that have windows
- **windows** returns the windows in these rooms
- **thermoName** calculates a sensible name for the thermo device
• Here is the result of the example house **after weaving**.
  • The rooms now have a **thermometer** with a suitable name
  • The windows have an **actuator**

• The transformation must now be enhanced to transform those new devices into **instances of software components**.

• Also we need some kind of driver component that **periodically checks the temperature** of all thermometers, calculates the average, and then **opens or closes the windows**.

• This whole additional transformation is **located in a separate aspect transformation file** and is “advised” into the original transformation.
Here is the **workflow fragment** that configures all of this:

```xml
<feature exists="environmentalControl.tempManagement.automaticWindows">
    <!-- the stuff that enhances the M2M transformation -->
    <component adviceTarget="xtendComponent.ps2cbd"
                class="org.openarchitectureware.xtend.XtendAdvice">
        <extensionAdvice value="windowAutomation::extensionAdvices"/>
    </component>

    <!-- this launches the model weaver that adds the aspect to the PS model -->
    <cartridge file="org/openarchitectureware/util/xweave/wf-weave-expr"
               baseModelSlot="psmodel"
               aspectFile="platform:/resource/smarthome.ps.lib/src/windowAutomation/aspect.xmi"
               expressionFile="windowAutomation::expressions"/>

    <!-- and here we write the model for debugging purposes -->
    <component class="org.eclipse.mwe.emf.Writer">
        <useSingleGlobalResourceSet value="true"/>
        <uri value="${dumpFileUriPrefix}/psWithWindowAutomation.xmi" />
        <cloneSlotContents value="true"/>
        <modelSlot value="psmodel" />
    </component>
</feature>
```
Optional Feature: Burglar Alarm

- In the configuration feature model, you can select whether your house should **feature a burglar alarm system**; and if so, which kinds of **alarm devices** it should have.

- There is a library of **pre-built components** for these devices in the **securehome** library project.

- The **ps2cbd** transformation
  - Instantiates a **control panel** component (turn on/off)
  - Instantiates the burglar alarm **detection agent**
  - ... **connects** those two ...
  - And then instantiates an instance of each of the **alarm devices** selected in the feature model
  - ... and **connects** those to the agen.
Optional Feature: Burglar Alarm II

- Thumbnail:

- Here is (part of) the code:

```java
create System transformPs2Cbd( Building building ):
    ...
    hasFeature("burglarAlarm") ? ( handleBurglarAlarm() -> this) : this;

handleBurglarAlarm( System this ):
    let conf = createBurglarConfig():
        configurations.add( conf ) ->
        ...
        conf.connectors.add( connectSimToPanel( createSimulatorInstance(),
                                                 createControlPanelInstance() ) ) ->
        hasFeature( "siren" ) ? conf.addAlarmDevice("AlarmSiren") : null ->
        hasFeature( "bell" ) ? conf.addAlarmDevice("AlarmBell") : null ->
        hasFeature( "light" ) ? conf.addAlarmDevice("AlarmLight") : null
    );
```

- Note how we query the feature model from within the transformation instead of using aspects to contribute the additional behaviour to the transformation.
• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
Code Level Aspects

• Sometimes the simplest way to implement variability is to **aspects on code level (AOP)**

• Since we’re using Java as the implementation language, we’ll use **AspectJ** as the implementation language for code level aspects

• The following challenges must be addressed:
  • A certain aspect shall only be woven iff a certain **feature is selected** in the global configuration model
  • It might be necessary to define (in the models!) **to which joinpoints** an aspect should be woven

• We assume that aspect functionality is **hand-written**, they are available in **libraries**. We distinguish
  • **Complete aspects**: advice and pointcut handwritten, inclusion is optional based on feature configuration
  • **Incomplete aspects**: advice is handwritten, pointcut is generated based on information in the models
Here is a **sample aspect** (trivialized authentication):

```java
public abstract aspect AuthenticationAspect {
    pointcut pc(): call (public * smarthome.ecolib.components..*());
    before() : pc() {
        // do some fancy authentication here
    }
}
```

The aspect contains **all the relevant code** (hence the pointcut is extremely generic) and is completely handwritten.

The aspect is **abstract** to make sure it is **not woven** by default.

If it **should be woven** (see later for how this is determined) a concrete sub-aspect is automatically generated
- Which is then grabbed by the weaver and automatically woven

```java
public aspect AuthenticationAspectImpl extends AuthenticationAspect {
}
```
As with interceptors, components and other code-related architectural elements, aspects are represented in the library model.

- Provides awareness of the generated build file, etc.
- Allows the use of model-level negative variability (see below)

Using a naming convention (enforced and checked by the recipe framework) the manually written code is associated with the model.
• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
In negative variability, **elements of a structural model** are associated with **features in a configuration model**. If that feature is not selected, the respective elements of the structural models are **removed**.

- The oAW XVar tool does that

The dependencies between the structural model and the configuration model are **externalized into a dependency model**.

- This makes sure the meta model of the structural model need not be changed in order to make it “configurable”
Managing Variability in Product Lines

Negative Variability for Aspects

- We use negative variability to **remove the aspect definitions** (see previous topic) from the library model if a specific feature is not selected.

- Since the aspect model elements are removed from the model, no **aspect-subclasses are generated**, and hence, no aspect is woven.

- Here is the dependency model:
  - Structural Elements are referenced directly,
  - Features are referenced by name
A **cartridge call to the XVar tool** in the API-level code generator workflow configures the structural model.

```xml
<workflow abstract="true">
  <readConfig uri="${globalConfigurationModel}"/>

  <read
      uri="platform:/resource/smarthome.cbd.lib.economyhome/src/economyLib.xmi"
      modelSlot="ecomodel"/>

  <cartridge file="org/openarchitectureware/util/xvar/wf-xvar.oaw"
      dependencyFileUri="platform:/resource/smarthome.cbd.lib.economyhome/src/dependencyModel.xmi"
      baseModelSlot="ecomodel"/>

  <feature exists="dumpCBDAfterXVar">
    <cloneAndWrite uri="temp-models/cbdAfterXVar.xmi"
        modelSlot="ecomodel" />
  </feature>

  <!-- more... -->

</workflow>
```
Customizing Code

- Remember that our libraries contain a **mixture of models and code** – the implementation (“business logic”) is implemented manually in Java.

- Hence, if you want to define variants of library components, it is not enough to vary the models (and with it the generated code). You also need to **vary manually written code**.

- Consider making **the lights dimmable**:
  - The interface `ILightDriver` needs an operation `setLightLevel()`.
  - The state of the light driver component needs an additional attribute to keep track of the light level.
  - And the implementation code needs to change – it needs to implement the optional `setLightLevel()` operation.

- The variability in the models is handled as explained before.
Customizing Code II

- Variable code sections can be marked up using **special syntax**:

```java
public class LightDriverImplementation extends LightDriverImplBase {
    @Override
    protected String getIdInternal() {
        return getConfigParamValueForId();
    }

    ... 

    // # dimmableLights
    @Override
    protected int setLightLevelInternal(int level) {
        state().setEffectiveLightLevel(level);
        return level;
    }
    // ~# dimmableLights
}
```

- This piece of code is in a `.javav` file
  - Hence it is not compiled
  - It is **customized** into a `.java` file based on the configuration
Customizing Code III

- Here is the **workflow component** that handles the customization.

```xml
<workflow abstract="true">
  ...
  
  <catridge file="org/openarchitectureware/util/xvar/file/wf-xvarfile.oaw"
    sourcePath="platform:/resource/smarthome.cbd.lib.economyhome/src"
    sourceExt="javav"
    genPath="platform:/resource/smarthome.cbd.lib.economyhome/src-gen"
    genExt="java"
    useComments="false"/>

  </workflow>
```

- The component
  - looks for `sourceExt`-files in the `sourcePath` directory
  - customizes them,
  - And writes the result to `genExt`-files in the `genPath` directory.
CONTENTS

• Introduction and Concepts
  • PLE and Variabilities
  • MDSD and AOSD
  • MDSD-AO-PLE

• Implementation Techniques
  • Intro to Case Study
  • Models, Code, Transformations
  • Orthogonal Variability
  • Transformation and Template AO
  • AO Modeling
  • Aspects on Code Level
  • Negative Variability

• Summary
Summary

• It is essential to **explicitly describe** the variabilities wrt. to the various product in a product line.

• While you can directly map variabilities to implementation code, it is much better to use a model-driven approach and **map the variability to models**
  * because they are more coarse grained and there’s less to vary

• **Variant management tools** integrate well with the model-driven tool chain

• Generators, transformation languages and all the other **MDD tooling is mature** and can be used in practice.
  * Advanced tools have sufficient features to build variants of generators, transformations or models based on configuration data in feature models

THANKS!
• **Videos** of a the full presentation of these slides are at
  http://ample.holos.pt/pageview.aspx?pageid=50&langid=1

• **Papers** on the topic:
  • Product Line Implementation using Aspect-Oriented and Model-Driven Software Development (SPLC 2007)
    http://www.voelter.de/publications/index/detail-1395240160.html
  • Feature-Based Variability in Structural Models (MVSPL 2007)
    http://www.voelter.de/conferences/index/detail-1817579497.html
  • Handling Variability in Model Transf. and Generators (DSM 2007)
    http://www.voelter.de/conferences/index/detail-966723965.html

• **Tooling:** openArchitectureWare 4.2,
  eclipse.org/gmt/oaw, includes 3 hours of tutorial videos

• **AMPLE Project:** http://ample-project.net

• **Podcasts** on PLE and MDSD at
  Software Engineering Radio: http://se-radio.net