Writing Adaptable Software: Mechanisms for Implementing Variabilities in Code and Models

OOPSLA 2007 Tutorial

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

This work is supported by

Copyright is held by the author/owner(s).
OOPSLA 2007, October 21–25, 2007,
Montréal, Québec, Canada.
ACM 07/0010.

völter SIEMENS ample

About me

völter.de
ingenieurbüro für softwaretechnologie

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

- Independent Consultant
- Based out of Göppingen, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Product Lines

völter SIEMENS ample

© 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

Software System Families

- Typically, MDD makes most sense in the context of **software system families** because developing modeling environments, generators, translators, etc. can be a lot of work and it pays only if reused.
- What is a **software system familiy?**

  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.**

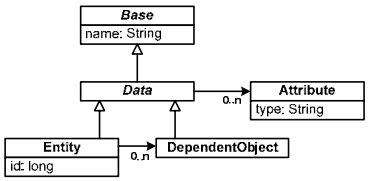
  Definition by Parnas

Variability Analysis

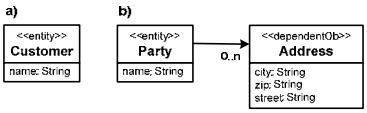
- **Variability analysis** discovers the variable and fixed parts of a product in a domain. Parts can be
  - Structural or behavioral
  - Functional or non-functional (technical)
  - Modularized or aspectual
- To define variable parts, we need to **have a commonality base:** a base platform, a common architecture
- There are **two kinds of variability:**
  - positive variability: add something (optional)
  - negative variability: removes something (essential)
- Another classification: **structural** vs. **non-structural** var.

Structural vs. Non-Structural Variability

- **Structural Variations**
  Example Metamodel



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

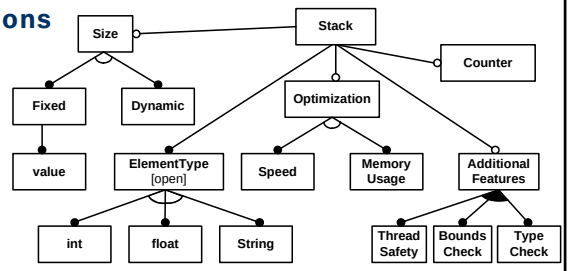


- **Non-Structural Variations**
  Example Feature Models

  Dynamic Size, ElementType: int, Counter, Threadsafe

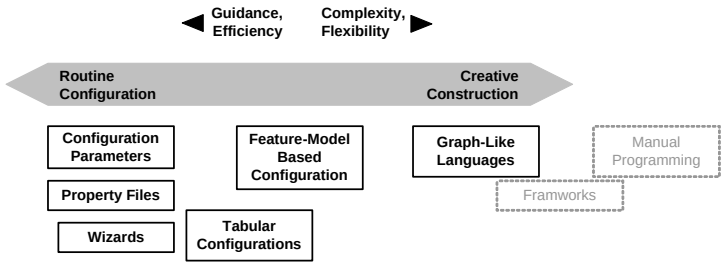
  Static Size (20), ElementType: String

  Dynamic Size, Speed-Optimized, Bounds Check



 © 2005-7 Markus Völter

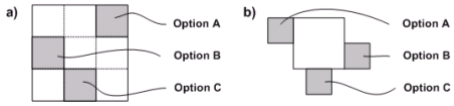
Routine Configuration vs. Creative Contruction



- 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 „form“ to the configuration side, the simpler it typically gets.

 © 2005-7 Markus Völter

Negative vs. Positive Variability



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

Typical Binding Times & Techniques

- For each of the variable features you need to define when you‘ll bind the feature
  - **source time:** manual programming, generators
  - **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)

Binding Time Tradeoffs

	flexibility	performance	code size	complexity
source time	-	+	+	-
compile time	+	+	+	-
link time	+	+	+	-
load time	++	+	+	+
run time	+++	-	-	+

CONTENTS

Manual Programming

- Handling variabilities by manually programming is the **simplest way** of handling variabilities.
- However, it is obviously **very inflexible**, since, whenever something changes you have to go back to the code and change it manually.
- Actually, there's **no variability in the code.**
- We won't elaborate this any further.

 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

Function / Method Overloading

- Assume the following piece of code:

```
public class Calculator {
   public int add( int x, int y ) {
     return x+y;
   }
   public double add( double x, double y ) {
     return x+y;
   }
}

// somewhere else..
Calculator c = new Calculator();
int res = c.add(3,5);    // calls the first one
double res2 = c.add(3.1415, 1.142); // calls the second one
```

- Here we use **compile-time overloading**, not polymorphism!

Preprocessors: C/C++ Examples

- The following statement **replaces the statement itself** with the **content of the file** specified as part of the statement:

```
#include „iostream.h“
```

- The next statement is a **conditional statement** that includes the part between the *#if* and the *#endif*.

```
#if defined (ACE_HAS_TLI)
   static ssize_t t_snd_n (ACE_HANDLE handle,
          const void *buf, size_t len, int flags,
          const ACE_Time_Value *timeout = 0,
          size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

- The *ACE_HAS_TLI* can **be considered a boolean variable** that can be *defined* or *undefined*.

```
#define ACE_HAS_TLI // defines ACE_HAS_TLI, sets it true
```

Simple Macros: C/C++ Examples (II)

- A typical case is to make sure **include files are only included once** per compilation unit.

```
#if !defined(ComponentHelloWorldIncluded)
#define ComponentHelloWorldIncluded

#include "components\HelloWorld\HelloWorldSI.h„
#include "components\HelloWorld\HelloWorldRI.h„
#include "container\Diagnosable.h"

class HelloWorld: public LifecycleInterface, public ComponentBase,
                  public HelloWorldSI,public HelloWorldRI,public Diagnosable {

private:

public:

    HelloWorld();
    ~HelloWorld();

    static int PARAMETER_NOT_DEFINED;
    static int DP_inputvoltage;
    static int DP_clientcount;

    int getDiagnosticParameter( int name );

};
#endif
```

 © 2005-7 Markus Völter

Simple Macros: C/C++ Examples (III)

- Macros can also be used **to define constants** (although specifically C++ provides better (and typesafe) means to achieve this):

```
#define MAX_ARRAY_SIZE 200
#define AUTHORNAME MarkusVoelter
```

- Processing is done via **strict text pattern matching**. Wherever the preprocessor finds the pattern, it replaces it. It has **no clue about language semantics**.
- Some **more complex expressions involving parameters** can also by preprocessed:

```
#define MAX(x,y) (x<y ? y : x)
#define square(x) x*x
```

 © 2005-7 Markus Völter

Simple Macros: Summary

- Macros are a useful means **to achieve simple text replacement**.
- In the **context of programming languages**, the problem is that macros are **not syntax- or semantic-aware** (and not type safe).
- As with almost anything, it **can be abused by being used too heavily** or by constructing formally legal, but nearly incomprehensible macro definitions.
- However, it is a **proven tool** and has been **used successfully** in many systems.

 © 2005-7 Markus Völter

Template Parameters (in C++)

- Unlike the generics implementation in Java, the templates in C++ are **completely static** – this is why this is a source time mechanism.
- For every instantiated template (i.e. Template parameter) a **completely new variant** of the respective generic class is created.
- Consequently, this approach is quite efficient – but potentially produces **large images**.

 © 2005-7 Markus Völter

Template Metaprogramming (in C++)

- Also called **compile-time metaprogramming**, because metaprograms „run“ while the program is compiled
- Uses the features of **C++ template instantiation**
- Programming style is **functional** and operates on types
- Note that some **awkward constructs** are required,
  - because C++ templates were not originally intended for this purpose
  - and many generally unknown and non-trivial features of the standard are used.
- Error reporting is usually clumsy

Static Aspect Weaving

- AOP can be used for **various reasons**
  - „fixing“ broken code
  - Separate cross-cutting (often technical) concerns
  - Handling **variants**
- Depending on the features we want in our system, we **add** additional pieces of (AspectJ) source code.
  - In our example, we can optionally add error handling
  - done by adapting/extending the **build path** of the respective project
- It happens **statically**, it‘s woven on byte code level
  - Can also be done at deployment time…
- Using advices, you can **attach additional behaviour** to existing code.

Static Aspect Weaving: Example

- We use a small **service framework** to illustrate the technique. Here is an introductory test case:

```
public void testSimpleAdding() {
    ServiceEngine engine = new ServiceEngine();
    engine.registerService( new CalculationService(),
       CalculationServiceContext.class );
   CalculationServiceContext ctx1 = new CalculationServiceContext(1,2);
    engine.addTask( ctx1 );
    CalculationServiceContext ctx2 = new CalculationServiceContext(3,4);
    engine.addTask( ctx2 );
    engine.run();
    assertEquals( 3, ctx1.getResult() );
    assertEquals( 7, ctx2.getResult() );
}
```

 © 2005-7 Markus Völter

Static Aspect Weaving: Example II

- One variant adds **error logging** to the service engine framework. Here is a test case; adding negative numbers is illegal, we expect an error in the log.

```
public void testSimpleAdding() {
    ServiceEngine engine = new ServiceEngine();
         engine.registerService( new CalculationService(),
    CalculationServiceContext.class );
    CalculationServiceContext ctx = new CalculationServiceContext(1,-2);
    engine.addTask( ctx );
    engine.run();
    assertEquals( -1, ctx.getResult() );
    assertNotNull( engine.logFor( ctx ) );
}
```

- The logFor(...) **operation is new**, as is the **functionality to create a log** in case the result is negative.

 © 2005-7 Markus Völter

Static Aspect Weaving: Example III

- Here's the aspect that implements that variant:

```
public aspect ErrorLogging {

  private Map<IServiceContext, Error> ServiceEngine.log = …

  public Error ServiceEngine.logFor(IServiceContext ctx) {
    return log.get(ctx);
  }

  public void ServiceEngine.log( IServiceContext ctx, Error err ) {
    log.put( ctx, err );
  }

  pointcut serviceExec( ServiceEngine e, IServiceContext c, IService s ) :
   execution(* ServiceEngine.executeService(IServiceContext, IService)) &&
     args(c,s) && target(e);

  Status around( ServiceEngine e, IServiceContext c, IService s ) :
    serviceExecution( ServiceEngine, IServiceContext, IService )
      && args(c,s) && target(e) {
    Status s = proceed(engine,ctx,srv);
    if ( s != Status.ok )
        engine.log( ctx, new Error(s, "no further information") );
    return s;
  }
}
```


CONTENTS

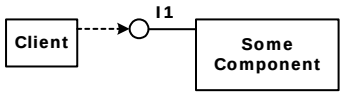
Component Deployment

- Consider a J2EE application server. When deplying EJBs you can
  - Pass in **configuration parameters**
  - „wire“ the **dependencies** to other components
  - Configure **security** and **transactions**,
  - … and generally **address QoS issues** by deploying on different hardware

 © 2005-7 Markus Völter

Component Deployment; Interceptors

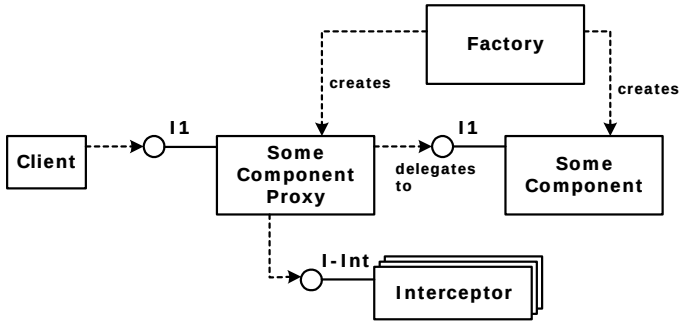
- In general, whenever you can **add interceptors** to a system, this allows you to add/configure certain cross-cutting concerns:
  - Typically, this consists of generating **proxies** [GoF] for application components
  - that can hook-in **interceptors** [POSA2].
  - Use a **factory** to instantiate the proxies if necessary.
- Consider you face the following situation:



 © 2005-7 Markus Völter

Component Deployment; Interceptors II

- You can replace this setup by the following:



- From a client's perspective, nothing has changed, the client still uses the interface I1. However, the client **actually talks to a proxy that handles CCC**, and then forwards to the real object.

 © 2005-7 Markus Völter

Component Deployment; Interceptors III

- Make sure that the join points are method calls; then the following **interceptor interface** can be used:

```
public interface Interceptor {
  public void beforeInvoke( Object target,
                            String methodName,
                            Object[] params );
  public void afterInvoke( Object target,
                           String methodName,
                           Object[] params,
                           Object retValue );
}
```

- The factory **determines which interceptors will be used** for a given object based on some kind of configuration (file).

 © 2005-7 Markus Völter

Component Deployment; Interceptors IV

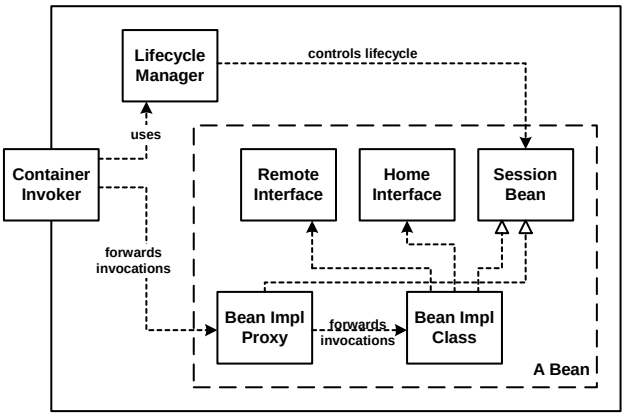
- The following is the basic structure of the proxy:

```
public class SomeComponentProxy implements I1 {
  private SomeComponent delegate;
  private Interceptor interceptor; // can also be a list
                                   // of interceptors
  public String someOperation( String p1, int p2 ) {
    Object target = delegate;
    String opName = "someOperation";
    Object[] params = {p1, p2};
    interceptor.beforeInvoke( target, opName, params );
    String res = delegate.someOperation( p1, p2 );
    interceptor.afterInvoke( target, opName, params, ret );
    return res;
  }
  // more operations of I1
}
```

 © 2005-7 Markus Völter

Component Deployment; Interceptors V

- **Example.** In the EJB scenario introduced above, the generated proxy would be the bean implementation class from the perspective of the application server, the real bean implementation would be an "implementation detail" of this class.



 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

DLL loading and Classloading

- In static languages such as C/C++, you can load **different DLLs** that define the same entry points.
- In Java you can use **class loading** ... Although the variability mechanism in fact will be a runtime solution using polymorphism

 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

Polymorhpism

- This is well known. The method that is invoked depends on the **runtime (dynamic) type** of the object on which you invoke the operation.
  - A factory is often used in conjuction
  - Related to the strategy & bridge patterns

 © 2005-7 Markus Völter

Polymorhpism II

- A simple test case:

```
public class PricingTest extends TestCase {
  private List<Product> products;
  protected void setUp() throws Exception {
    products = new ArrayList<Product>();
    products.add( new Product() ); products.add( new Product() );
    products.add( new Product() ); products.add( new Product() );
    products.add( new Product() );
  }
  public void testLinearPricing() {
    Customer normalCustomer = new Customer(false);
    int totalPrice = Factory.getPricingStrategy(normalCustomer).
                     calculatePrice( products );
    assertEquals( 500, totalPrice );
  }
  public void testRebatePricing() {
    Customer valuedCustomer = new Customer(true);
    int totalPrice = Factory.getPricingStrategy(valuedCustomer).
                     calculatePrice( products );
    assertEquals( 300, totalPrice );
  }
}
```

 © 2005-7 Markus Völter

Polymorhpism III

- Strategy Implementation and the Factory:

```
public abstract class PricingStrategy {
  public abstract int calculatePrice( List products );
}
public class LinearPricing extends PricingStrategy {
  public int calculatePrice(List products) {
    return products.size() * 100;
  }
}
public class RebatePricing extends PricingStrategy {
  public int calculatePrice(List products) {
    int count = products.size();
    if ( count > 3 ) count = 3;
    return count * 100;
  }
}
```

```
public class Factory {
  public static PricingStrategy getPricingStrategy( Customer c ) {
    if ( c.isValued() ) return new RebatePricing();
    else return new LinearPricing();
  }
}
```

 © 2005-7 Markus Völter

Metaprogramming

In as much as a computational process can be constructed to **reason about an external world** in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world,

so, too, a computational process could be made to **reason about itself** in virtue of comprising an ingredient process (interpreter) formally manipulating representations of ist own operations and structures.

Smith, The Reflection Hypothesis

Metaprogramming in OO Languages

- There are several terms in use:
  - **Introspection / Reflection**: read/modify the program
  - **Reification**: change the semantics of existing code
- Often, the term **Meta Object Protocol** is used
- **Example Languages:**
  - **CLOS**: Reification, Reflection, Introspection, MOP, Lisp in Lisp
  - **Smalltalk**: Reflection, Dictionary, (Smalltalk := nil)...
  - **Java**: Introspection, teilweise Reflection, java.lang.Class, java.reflect
  - **Self**: Reification, Reflection, Introspection
  - **Ruby:** Reflection, Introspection

Metaprogramming in OO Languages II

- I assume you all know Java Reflection … and it's also **not very interesting** (since it's not very powerful).
- Considering the current hype about **dynamic languages** such as Ruby, and the fact, that these languages
  - integrate with Java nicely (JRuby)
  - And that Java (at least, the VM) may even get native support for more dynamic languages (*invokedynamic* keyword)
- **… I will show an Example in Ruby**
  - It shows how to handle structural variability using metaprogramming

 © 2005-7 Markus Völter

Metaprogramming in OO Languages III

- Here is an entity class definition:

```
class Person < Entity
  properties :name, :firstname
  has_one :adr => Address
  has_many :addresses => Address
end
```

- And here is a test case:

```
class SimpleTests < Test::Unit::TestCase

  def test_people
    p = Person.new( :name => "Voelter", :firstname => "Markus")
    assert_equal p.name, "Voelter"
    assert_equal p.firstname, "Markus“
…
```

- Where do the **native Ruby properties** name and firstname come from, and how come they can be **intialized** via the => syntax?

 © 2005-7 Markus Völter

Metaprogramming in OO Languages IV

- Here is the class definition of Entity:

```
class Entity < WithProperties
end
```

- … and the WithProperties class ➔
- The properties key-word in really static method that is **executed during class definition.**
- It in turn **creates** the **initializer** and the **setters** and **getters**

```
class WithProperties
  def self.properties( *attrNames )
    define_method( :initialize ) do | values |
      attrNames.each do | attrName |
        instance_variable_set(
          ("@"+attrName.to_s).to_sym,
            values[attrName.to_sym] )
      end
    end
    attrNames.each do | attrName |
      getter = %Q{
          def #{attrName.to_s}
            @#{attrName.to_s}
          end
        }
      self.module_eval(getter)
      setter = %Q{
          def #{attrName.to_s}= (value)
            @#{attrName.to_s} = value
          end
        }
      self.module_eval(setter)
    end
  end
end
```

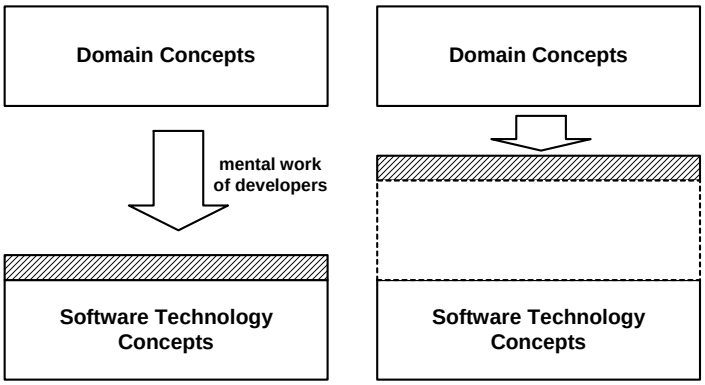
 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

What is MDSD?

- Domain Driven Development is about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.

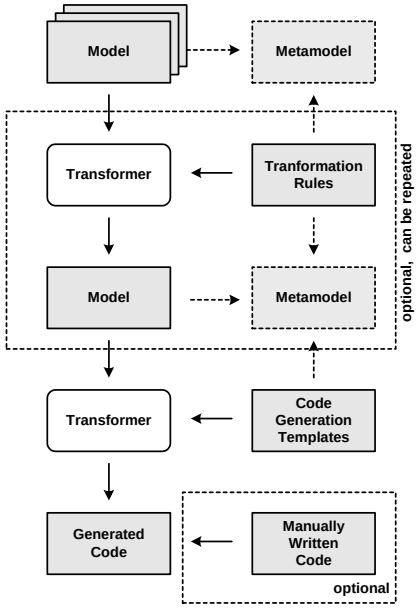


What is MDSD? II

- Model-Driven Software Development is about **making models first class development artefacts** as opposed to “just pictures”.
- Various aspects of a system are not programmed manually; rather they are **specified using a suitable modeling language**.
- The language for expressing these models is specific to the domain for which the models are relevant. The modeling languages used to describe such models are called **domain-specific languages** (DSL).
- Models have to be **translated into executable code** for a specific platform.
  - Such a translation is implemented using **model transformations.**
  - An approach based on **model interpretation** is also possible, but seldomly used – I will ignore this here!

How does MDSD work?

- Developer develops **model(s)** based on certain metamodel(s).
- Using **code generation templates**, the model is transformed to executable code.
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.



 © 2005-7 Markus Völter

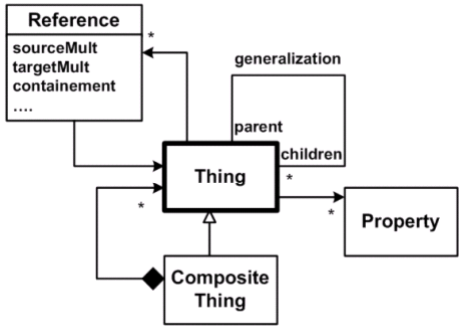
Models & Meta Models

- 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.

 © 2005-7 Markus Völter

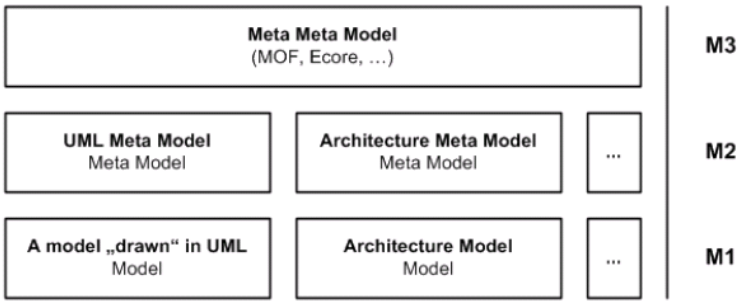
Meta Meta Models

- A **meta model also has a meta model**
  - after all, a meta model is a model that plays the role of the meta model for some other model.
- The **meta model's meta model** is called the meta meta model.
- A meta meta model typically looks more or less like that →



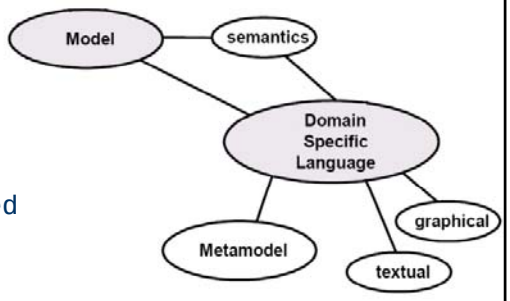
Meta Levels

- This diagram **illustrates** the various meta levels using UML as well as a custom meta model
- **Caveat:** Note that absolute meta levels (as shown here) can be a problem and lead to strange statements – better avoid them and consider this really only an example

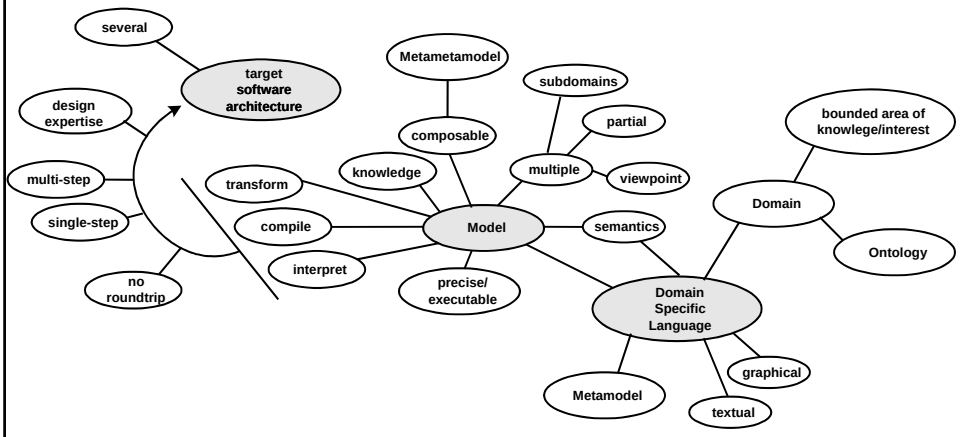


Domain Specific Language

- 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 MDSD? III



- Related Approaches (Specializations): MDA, SF, DSM, GP, ...

CONTENTS

 © 2005-7 Markus Völter

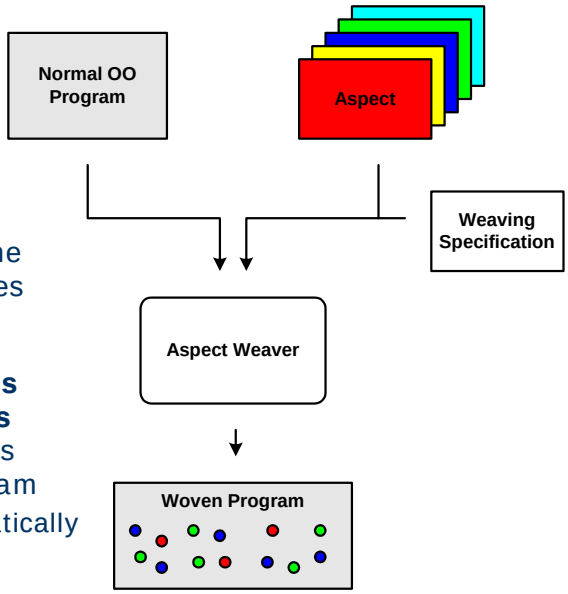
What is AOSD?

- AOSD is about **localizing cross-cutting concerns** into well-defined modules called aspects.
- Various approaches to AOSD are possible, including **language extension** (AspectJ) and **framework/infrastructure-based** approaches (such as Spring AOP, JBOSS AOP or AspectWerkz).
- A core characteristic of each AOSD tool is its **join point model**, i.e. the means by which the base code and the aspect code can be joined.
  - **Static** and **Dynamic** join points can be supported
  - The **granularity** of the join point model varies.
  - **Introductions/Inter-Type declarations** are often, but not always possible

 © 2005-7 Markus Völter

How does AOSD work?

- 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



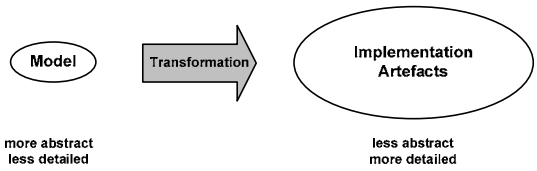
 © 2005-7 Markus Völter

CONTENTS

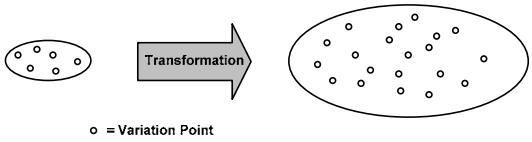
 © 2005-7 Markus Völter

What is MDD-AO-PLE

- As mentioned above, the core challenge of product line implementation, is the **implementation of the product variability**.
- Models are **more abstract** and hence **less detailed** than code



- Thus, the **variability is inherently less scattered**, making variability management on model level **simpler**!



 © 2005-7 Markus Völter

What is MDD-AO-PLE II

- AO is used in several ways:
  - On model level, we use it for **weaving models** and meta models
  - In the transformation, we **weave variants into transformations** and generators
  - And on code level, we use it to directly **implement fine-grained** implementation variants.
- We provide more details on all of these aspects ☺ later, as well as examples.
- **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.

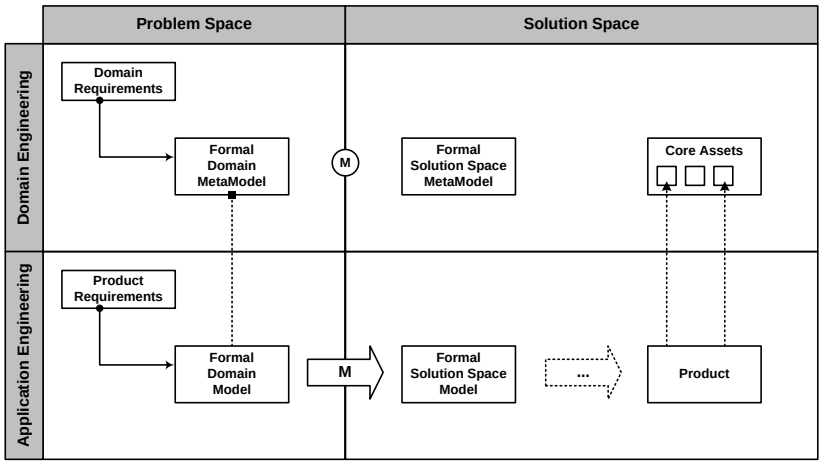
 © 2005-7 Markus Völter

What is MDD-AO-PLE III

- **Variability can be described more concisely** since in addition to the traditional mechanisms, variability is also described on model level.
- The **mapping from problem to solution domain** can be formally described automated using model-to-model transformations.
- Aspect-oriented techniques enable the explicit expression and **modularization of crosscutting variability** on model, code, and generator level.
- Fine grained traceability is supported since **tracing is done on model element level** rather than on the level of artifacts.

 © 2005-7 Markus Völter

What is MDD-AO-PLE IV

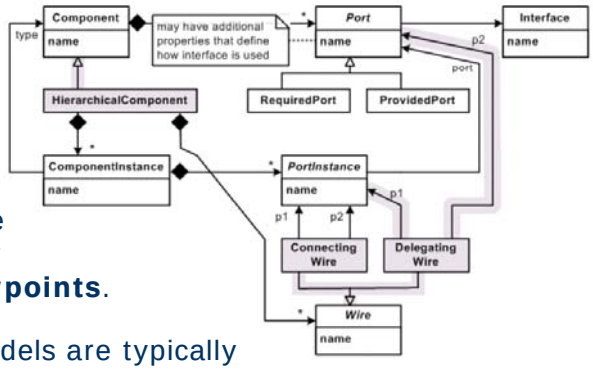


 © 2005-7 Markus Völter

CONTENTS

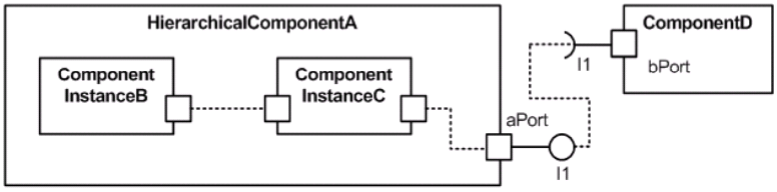
Architecture Meta Models

- An architectural meta model **defines formally** the concepts available for defining **software architectures**.
- This one defines the component definition viewpoint of a **component architecture**.
- Complete architecture meta models typically contain **several viewpoints**.
- Architecture meta models are typically defined **as part of domain engineering**



Architectural Models

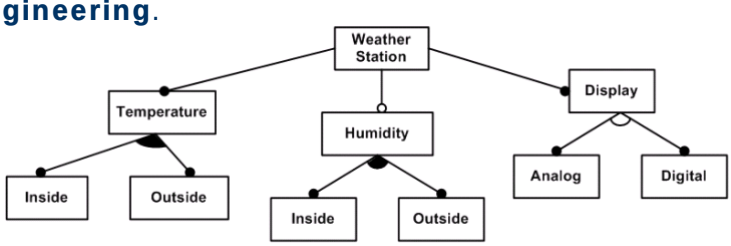
- An architecture model is an **instance** of an architecture meta model.



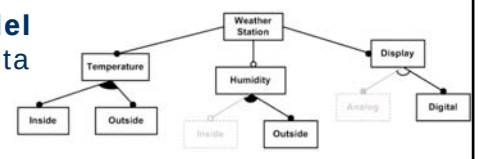
- They are defined during **application engineering**

Application Domain Meta Model and Model

- Application Domain Meta Models are **formalizations of domain requirements**. Often this meta model is a feature model. It is created as part of **domain engineering**.

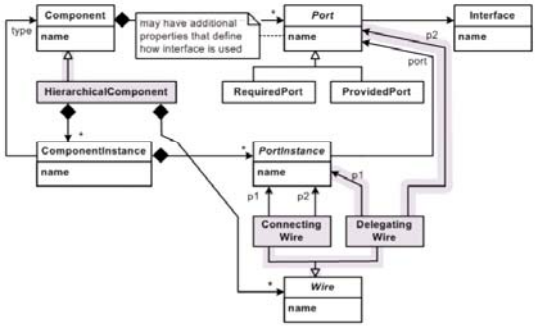


- An **application domain model** is then an instance of that meta model; it is created during **application engineering**.



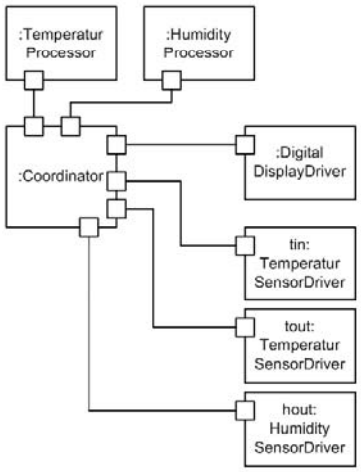
Software Domain Meta Model

- Software Domain Meta Models are **formalizations of the architecture** of the to-be-built product line (i.e. they are architecture meta models).
- Typically, this meta model is a NOT a feature model.
- It is created as part of **domain engineering**.



Software Domain Meta Model and Model

- An software domain model is then an **instance of that meta model**; it is created during application engineering.



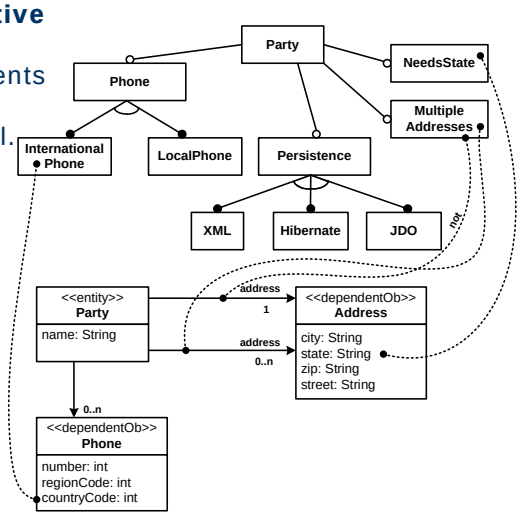
Variants and Models II

- It is especially useful to **combine structural and non-structural** variations
  - specifically, you may want to „configure“ structural models with the help of feature models,
  - we want to describe variants of structural models (and use these variants as generator input)
- Examples:
  - A party may have one or more addresses
  - A party may store telecontacts or not
  - In case of telephone numbers, you may want to store the country code
  - Addresses may have the *state* field (USA)

 © 2005-7 Markus Völter

Variants and Models III

- Implementation using **negative variability:**
  You can **assign** model elements of the structural model to features in the feature model.
  - The respecticve model elements are only there if the associated feature is selected,
  - And it‘s removed, if the feature is not there.
- Implementation using **positive variability**: model weaving (see later)



 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

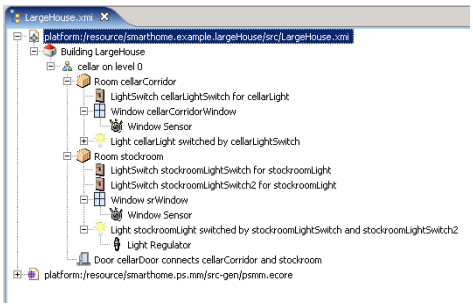
Intro to Case Study

- 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.

 © 2005-7 Markus Völter

Application Domain 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
  - It is based on Exeed, and it is basically an EMF tree view with customized icons and labels

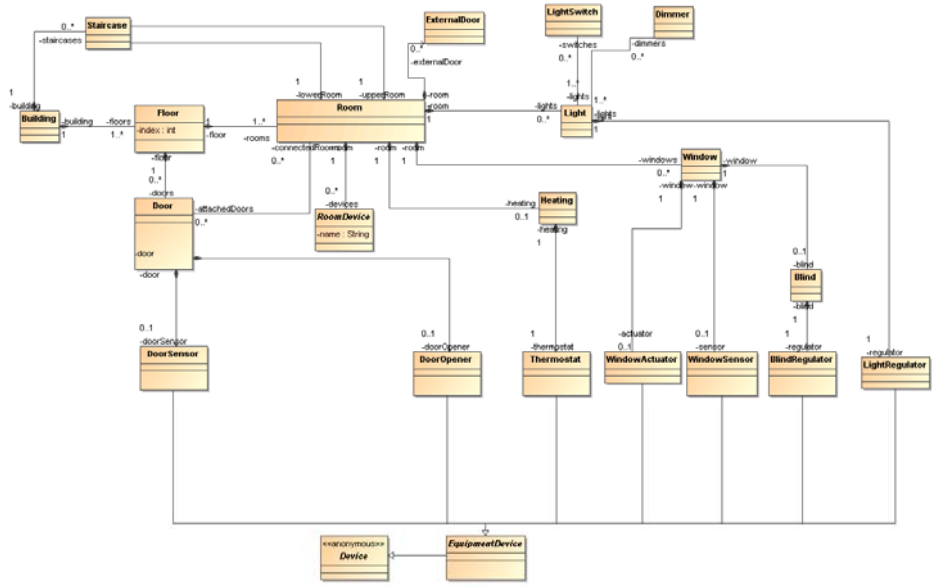


- Note that problem space modeling uses a **creative construction DSL** since describing a Smart Home is not just a matter of “ticking boxes”.
- A more convenient editor will be provided later.

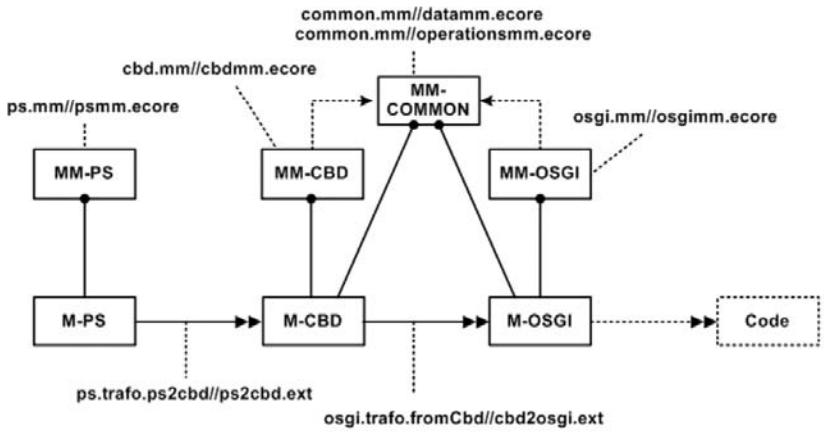
CONTENTS

- PLE Concepts
- Classical PLE Implementation
  - Source time
  - Compile time
  - Deployment/Configuration time
  - Link time
  - Run time
- MDD-AO-PLE
  - What is MDD
  - What is AO
  - What is MDD-AO-PLE
  - More Terms and Concepts
- **MDD-AO Implementation**
  - Intro to Case Study
  - **The Various (Meta-)Models**
  - Libraries
  - An Example House
  - Orthogonal Variability
  - Transformation and Template AO
  - AO Modeling
  - Code Level Aspects
  - Negative Variability
  - Testing
  - Enforcing Conventions
  - Product Line Evolution
- Summary

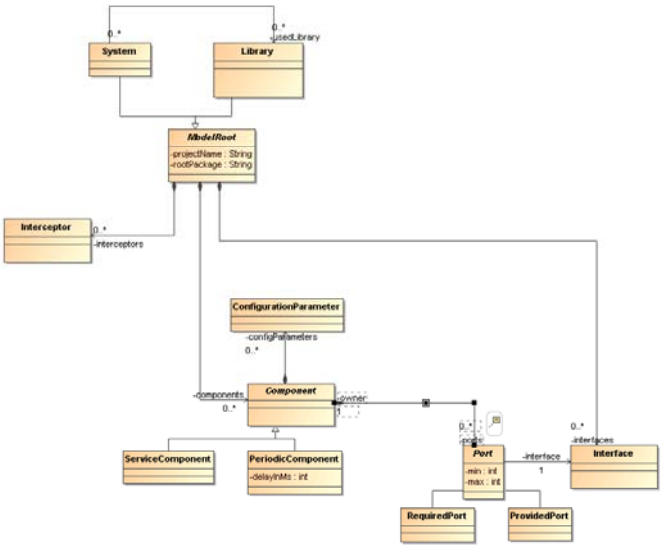
Application Domain Meta Model (PSMM)



Models and Transformations Overview

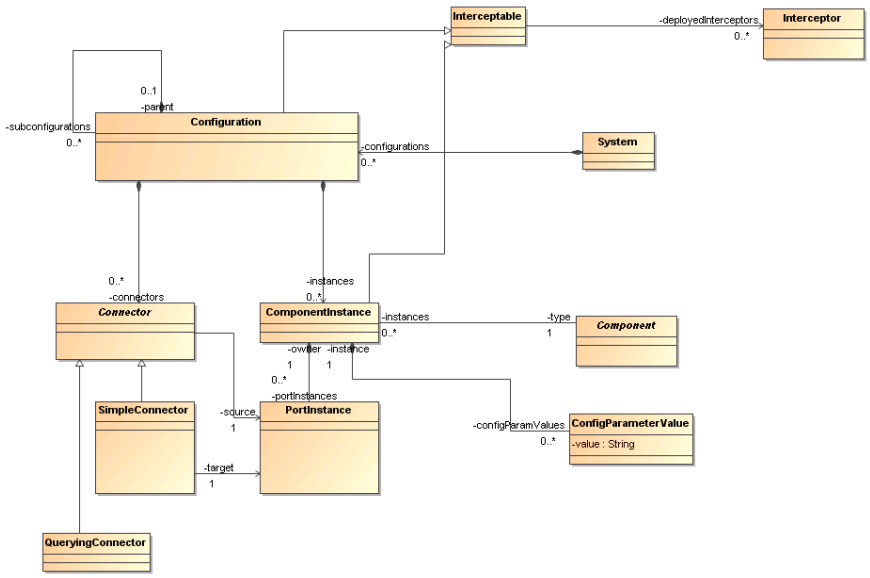


Software Domain Meta Model (CBDMM): Types Viewpoint



 © 2005-7 Markus Völter

Software Domain Meta Model (CBDMM): Configuration Vp.



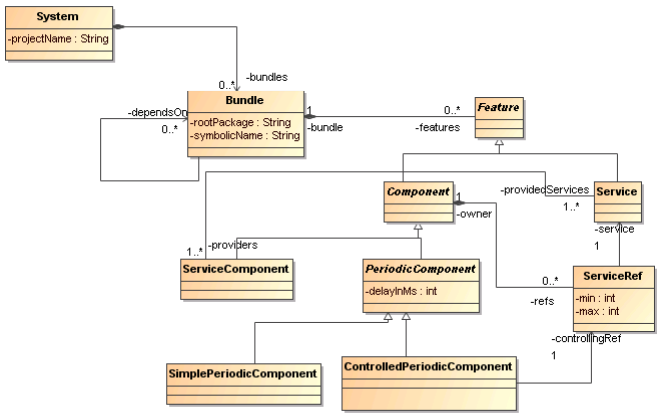
 © 2005-7 Markus Völter

Low Level Software Domain Modeling

- As part of our OSGi based implementation, we **use another M2M transformation** (and hence, another meta model).
- As far as the product line is concerned, that second model transformation is an **implementation detail** of our implementation technology.
- Other implementation technologies might choose to generate code directly from the CBD models.
- Again, **no concrete syntax** is available for this level of modeling.

 © 2005-7 Markus Völter

OSGI Meta Model (OSGIMM)



 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

Library Components

- 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.

 © 2005-7 Markus Völter

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.

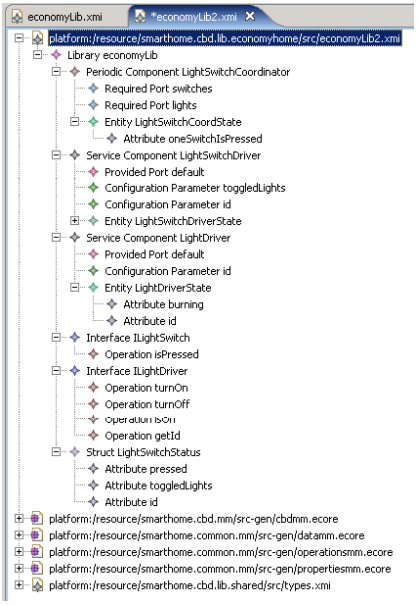
 © 2005-7 Markus Völter

The EconomyLib 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.

 © 2005-7 Markus Völter

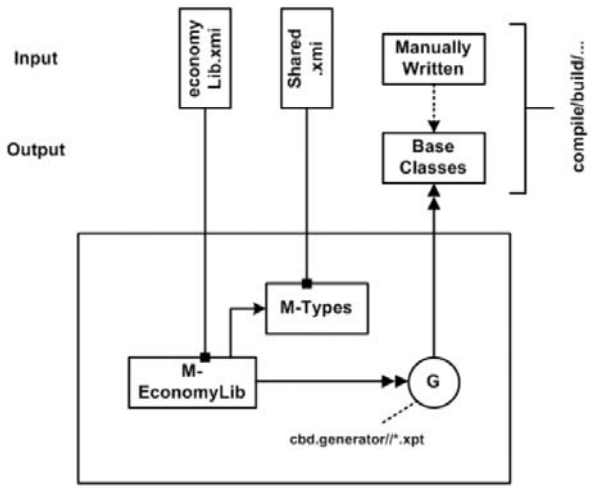
The EconomyLib Library: Part of the Model



- 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

 © 2005-7 Markus Völter

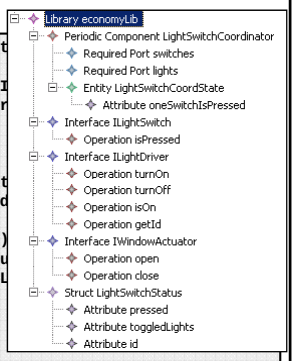
The EconomyLib Library: Generating CBD-Level code



 © 2005-7 Markus Völter

The EconomyLib Library: Manually written code I

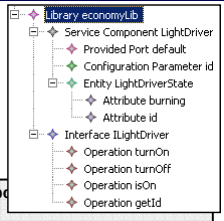
- This is the component that **switches lights** based on the status of the switches
- It is a **periodic component**, hence it has only an ***execute()*** operation.
- Note how it uses the ***switchesAll()*** operation to access all the switches it is connected to.




```
package smarthome.eco…witchCoordinat

public class LightSwitchCoordinatorI
      extends LightSwitchCoordinator

  @Override
  public void execute() {
    Collection<LightSwitchStatus> st
              switchesAll().isPressed
    for (LightSwitchStatus status :
      if (hasChanged( status.getId()
        String changedLights = statu
        parseLightsToSwitch(changedL
      }
    }
  }

  private boolean hasChanged(String id, boolean pressed) {
    // is the light switch in another position than
    // last time around?
  }

  private void parseLightsToSwitch(String lights) {
    // find out which lights this switch affects
    // and switch these lights
  }

}
```


The EconomyLib Library: Manually written code II

- This one represents a **light**.
- It is a **service component**, it **implements the operations** provided by the default port's interface
- You can also see how it accesses **configuration parameters** and its **internal state**




```
package smarthome.ecolib.compo

public class LightDriverImplementation
          extends LightDriverImplBase {

  public String getId() {
    return getConfigParamValueForId();
  }

  public boolean isOn() {
    return state().getBurning();
  }

  public void turnOff() {
   state().setBurning(false);
  }

  public void turnOn() {
    state().setBurning(true);
  }

}
```

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**.

 © 2005-7 Markus Völter

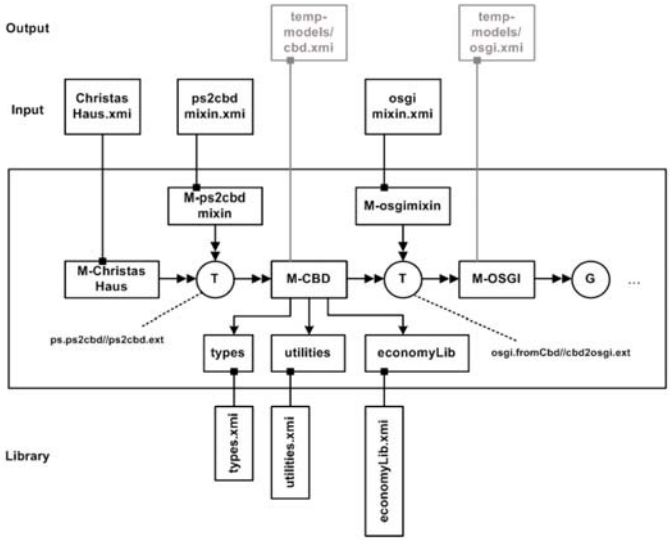
CONTENTS

 © 2005-7 Markus Völter

Example House: The App Domain Model

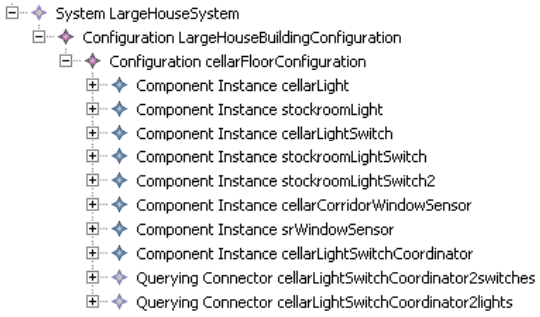
- A house with only **one level**, and **two rooms**, connected by **doors**.
- The rooms have **windows** as well as **lights** and **light switches**.

Example House: Models and Transformations



Example House: The Transformed CBD Model

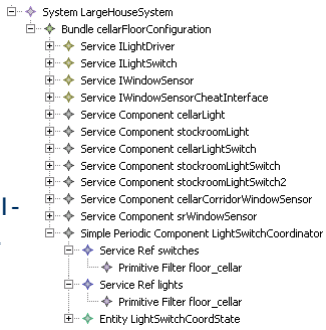
- For each of the lights and switches we have **instances** of driver **components** (the component types are taken from the library)
- We also have a **light switch coordinator** component instance for each floor that has light switches.

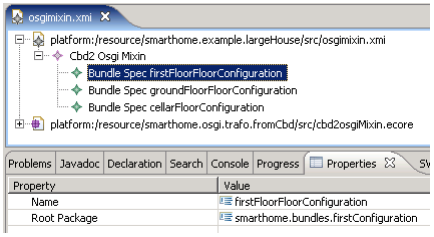


- We use **query based connectors** to connect the coordinator with the lights and the switches.
  - The query dynamically finds all lights and switches for a given floor, dynamically at runtime.
- We also have hierarchical configurations for the building and floors.

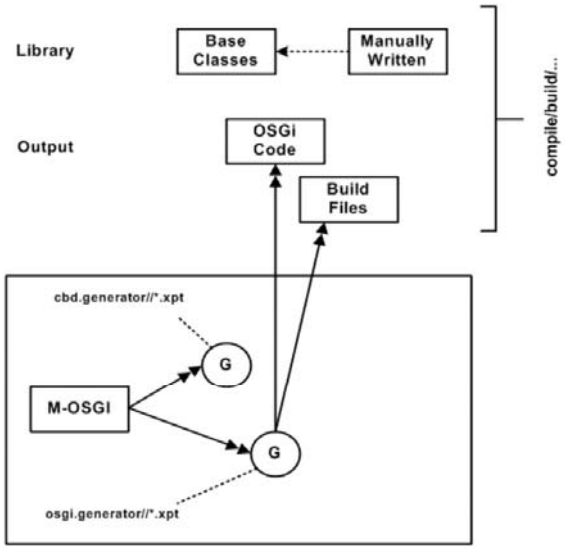
Example House: The Transformed OSGi Model

- 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



 © 2005-7 Markus Völter

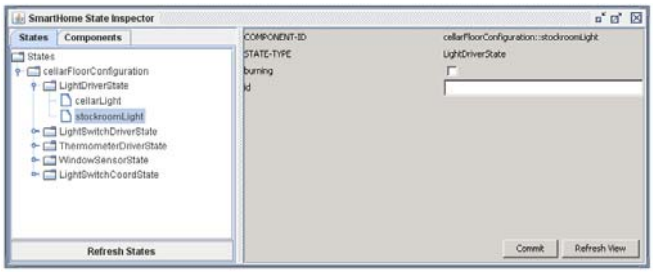
Example House: Generated Code

- 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)

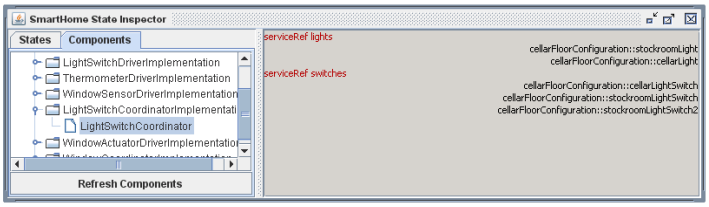
 © 2005-7 Markus Völter

Example House: Running the System (UI)

- A console allows the **inspection** and **change** of component states.



- It also shows the actual **connections of service refs**



 © 2005-7 Markus Völter

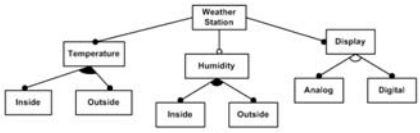
CONTENTS

- PLE Concepts
- Classical PLE Implementation
  - Source time
  - Compile time
  - Deployment/Configuration time
  - Link time
  - Run time
- MDD-AO-PLE
  - What is MDD
  - What is AO
  - What is MDD-AO-PLE
  - More Terms and Concepts
- **MDD-AO Implementation**
  - Intro to Case Study
  - The Various (Meta-)Models
  - Libraries
  - An Example House
  - **Orthogonal Variability**
  - Transformation and Template AO
  - AO Modeling
  - Code Level Aspects
  - Negative Variability
  - Testing
  - Enforcing Conventions
  - Product Line Evolution
- Summary

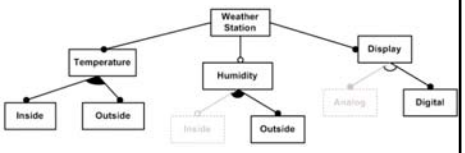
 © 2005-7 Markus Völter

Orthogonal Variability Management

- Configuration models are **"instances"** of Feature Models.
  - There is **no variability** left in them
  - They are basically a **list of selected features**
  - (there are also partial configurations and properties…)
- Feature Model



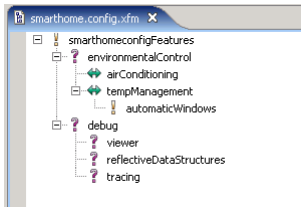
- Configuration Model



- From the perspective of a **backend** (i.e. generator, compiler, etc) **only the configuration model** is relevant!
  - … as long as we expect that the configuration is valid wrt. to the feature model and we have implemented the generator correctly.

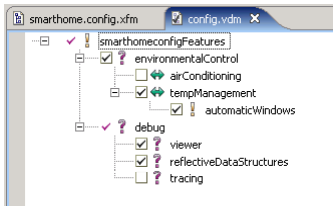
Orthogonal Variability Management II

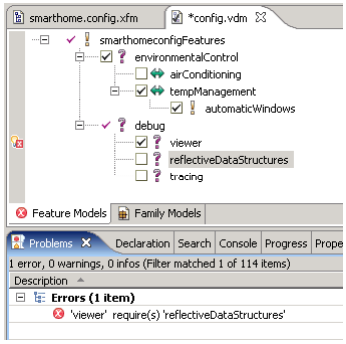
- oAW comes with a feature that allows domain architecture artifacts to **depend on whether certain features are selected**.
- An API is available that allows to plug in various **feature modeling tools**
  - In the simplest case, that API can be bound to a simple text file that contains a list of selected 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



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**.





 © 2005-7 Markus Völter

CONTENTS

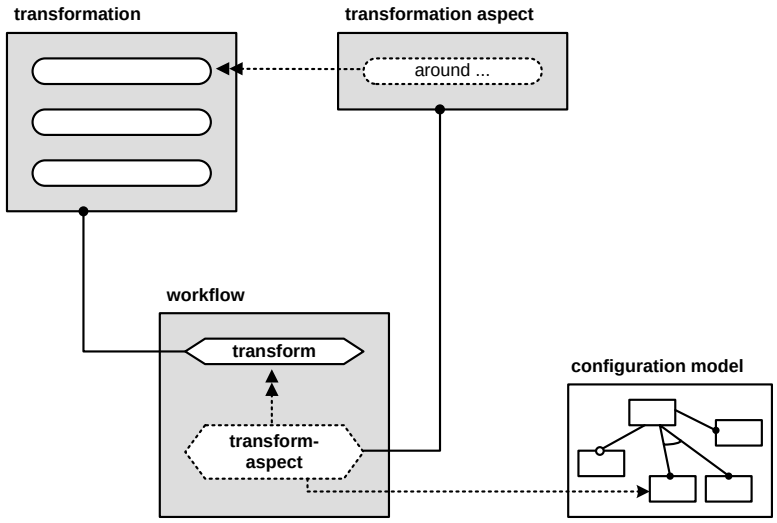
- PLE Concepts
- Classical PLE Implementation
  - Source time
  - Compile time
  - Deployment/Configuration time
  - Link time
  - Run time
- MDD-AO-PLE
  - What is MDD
  - What is AO
  - What is MDD-AO-PLE
  - More Terms and Concepts
- **MDD-AO Implementation**
  - Intro to Case Study
  - The Various (Meta-)Models
  - Libraries
  - An Example House
  - Orthogonal Variability
  - **Transformation and Template AO**
  - AO Modeling
  - Code Level Aspects
  - Negative Variability
  - Testing
  - Enforcing Conventions
  - Product Line Evolution
- Summary

 © 2005-7 Markus Völter

Optional Feature: Logging

- Tracing 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.tracing is selected, the transformation from PS to CBD level must make sure that the appropriate interceptor is configured for the components.

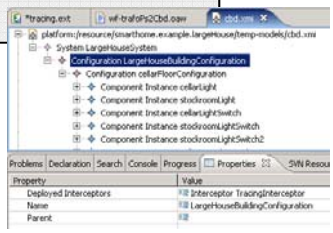
Optional Feature: Logging [Thumbnail]



Optional Feature: Logging, Implementation

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

```
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("TracingInterceptor") }
      ) ->
      s
    );
```



- We **advice** *ps2cbd::transformPs2Cbd*
- We then execute the **original definition** (*ctx.proceed()*)
- Then we **add**, to the top level config, the *Tracing Interceptor*

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:

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

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

- The stuff inside the <feature>…</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

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

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

- An Advice component basically takes the sub-elements and adds them to the component **refenced 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.**

 © 2005-7 Markus Völter

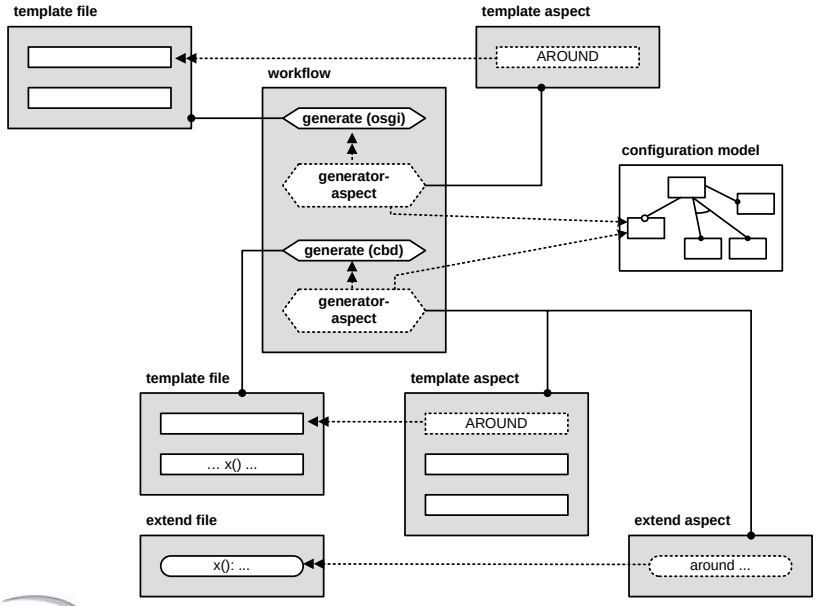
Optional Feature: Component State Viewer

- 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.

 © 2005-7 Markus Völter

Optional Feature: Component State Viewer [Thumbnail]



 © 2005-7 Markus Völter

Optional Feature: State Viewer, Implementation

- Here are the respective **feature-dependent aspects** in the workflow
- This one happens **on CBD level** because the data implementations are **independent of the runtime platform**

```
<feature exists="debug.viewer">
    <component adviceTarget="generator.cbdapi" class="oaw.xpand2.GeneratorAdvice">
        <advices value="data::api::reflect::reflectImpl"/>
        <extensionAdvices value="data::api::reflect::reflect"/>
    </component>
</feature>
```

- The second one influences the **generation of the OSGI activator**, since that one has to **publish the component states** once they are instantiated.

```
<feature exists="debug.viewer">
    <component adviceTarget="generator.osgistuff" class="oaw.xpand2.GeneratorAdvice">
        <extensionAdvices value="osgi::impl::viewer::viewerExtAdvices"/>
        <advices value="osgi::impl::viewer::viewerXptAdvices"/>
    </component>
</feature>
```

 © 2005-7 Markus Völter

Optional Feature: State Viewer, Implementation II

- **reflectImpl.xpt** adds around advice to a number of definitions in the code generation templates:
  - some are pure hooks, i.e. they are empty!

```
«AROUND data::api::data::body FOR ComplexType»
  «targetDef.proceed()»
  «EXPAND reflectionImplementation»
«ENDAROUND»

«AROUND data::api::data::imports FOR ComplexType»
  «targetDef.proceed()»
  import smarthome.common.platform.MemberMeta;
  import smarthome.common.platform.ComplexTypeMeta;
«ENDAROUND»

«DEFINE reflectionImplementation FOR Comple
  private transient ComplexTypeMeta __meta
  public ComplexTypeMeta __metaObject() {
    …
  }
  public void __metaSet( MemberMeta member,
    …
  }
  public Object __metaGet( MemberMeta membe
     …
  }
«ENDDEFINE»
```

```
«DEFINE typeClass FOR ComplexType»
  «FILE fileName()»
    package «implClassPackage()»;
    «EXPAND imports»
    public class … {
          «EXPAND body»
    }
  «ENDFILE»
«ENDDEFINE»

«DEFINE imports FOR ComplexType»«ENDDEFINE»

«DEFINE body FOR ComplexType»
  …
«ENDDEFINE»
```

 © 2005-7 Markus Völter

Optional Feature: State Viewer, Implementation III

- **reflect.ext** adds a newly implemented interface to an existing extension function

```
around data::api::dataapiutils::implementedInterfaces(ComplexType this):
  ((Collection)ctx.proceed()).add("smarthome.common.platform.ReflectiveComplexType");
```

- That original function is **called from a template** in order to find out which additional interfaces a data bean class needs to implement:

```
public class «implClassName()»
  «IF implementedInterfaces().size > 0»implements «ENDIF»
        «FOREACH implementedInterfaces() AS e SEPARATOR ", "»«e»«ENDFOREACH»  {
   «EXPAND body»
}
```

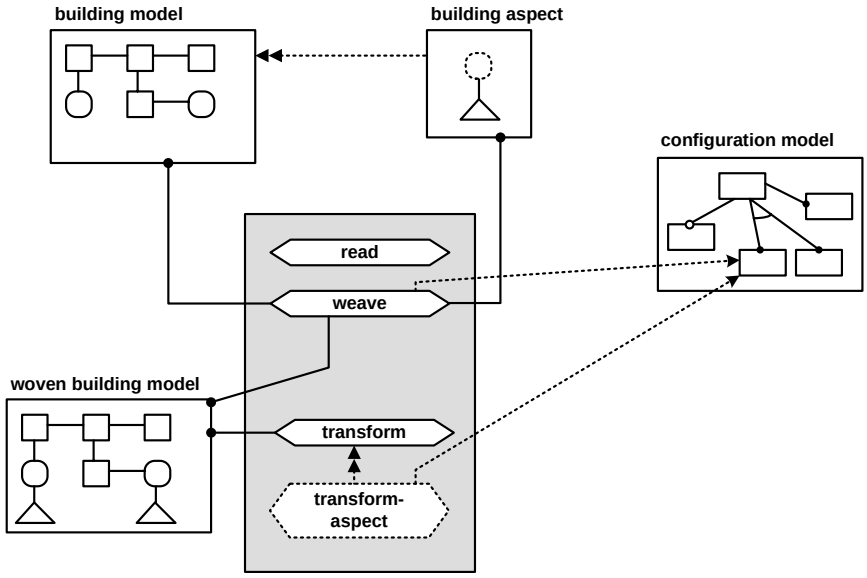
- The same mechanisms are used to “advice” the templates that generate the OSGi level code for the activator.

 © 2005-7 Markus Völter

CONTENTS

 © 2005-7 Markus Völter

Optional Feature: Automatic Windows [Thumbnail]



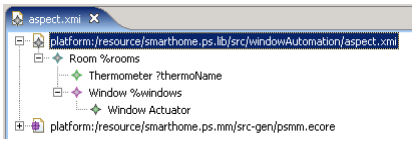
 © 2005-7 Markus Völter

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 PS model:



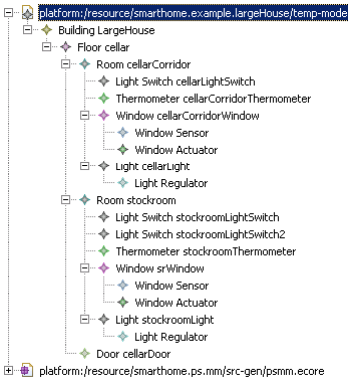
- Here are the **pointcut expressions** used in the aspect model:

```
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

Optional Feature: Automatic Windows, Implementation II

- 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 “adviced” into the original transformation.

 © 2005-7 Markus Völter

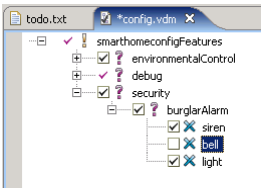
Optional Feature: Automatic Windows, Implementation III

- Here is the **workflow fragment** that configures all of this:

```
<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>
```

 © 2005-7 Markus Völter

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:

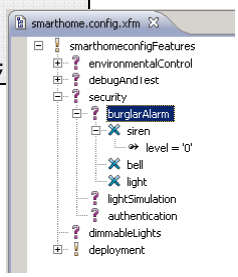
```
create System transformPs2Cbd( Building building ):
  …
  hasFeature("burglarAlarm") ? ( handleBurglarAlarm() -> this) : this;

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.

Optional Feature: Burglar Alarm III

- It is also possible to access **attributes of features** (if the feature modeling tool supports attributes).
- Here we set the **volume level** of the siren:

```
handleBurglarAlarm( System this ):
  …
  isFeatureSelected( "siren" ) ? (
     let siren = conf.addAlarmDevice("AlarmSiren"):
       siren.configParamValues.add( siren.createConfigParamForLevel() )
     ) : null ->
  …
);

private create ConfigParameterValue
     createConfigParamForLevel( ComponentInstance instance ):
  setName( "level" ) ->
  setValue((String)getFeatureAttributeValue( "siren", "level" ));
```



- The **feature model** needs to have the **level attribute**, of course.

 © 2005-7 Markus Völter

CONTENTS

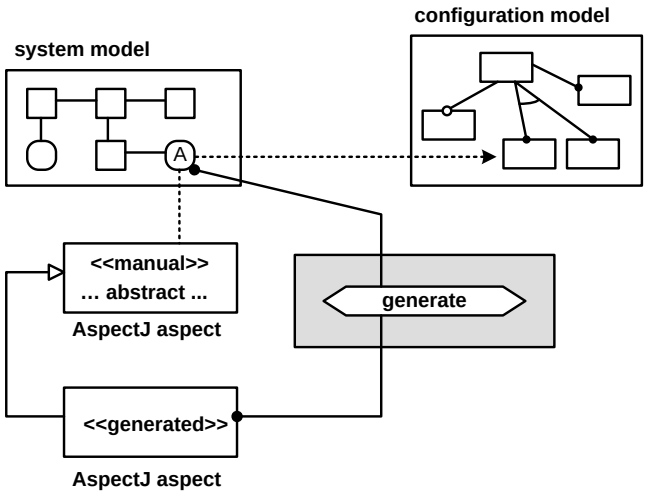
- PLE Concepts
- Classical PLE Implementation
  - Source time
  - Compile time
  - Deployment/Configuration time
  - Link time
  - Run time
- MDD-AO-PLE
  - What is MDD
  - What is AO
  - What is MDD-AO-PLE
  - More Terms and Concepts
- **MDD-AO Implementation**
  - Intro to Case Study
  - The Various (Meta-)Models
  - Libraries
  - An Example House
  - Orthogonal Variability
  - Transformation and Template AO
  - AO Modeling
  - **Code Level Aspects**
  - Negative Variability
  - Testing
  - Enforcing Conventions
  - Product Line Evolution
- Summary

 © 2005-7 Markus Völter

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

Complete Aspects [Thumbnail]



Complete Aspects

- Here is a **sample aspect** (trivialized authentication):

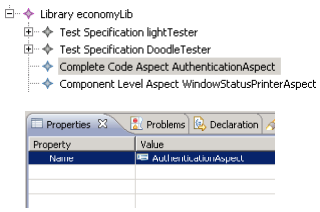
```
public abstract aspect AuthenticationAspect {

    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

```
public aspect AuthenticationAspectImpl extends AuthenticationAspect {

}
```

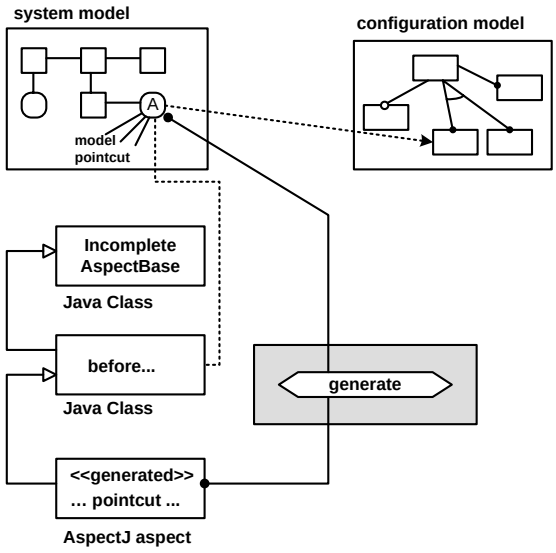

Complete Aspects II

- 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)



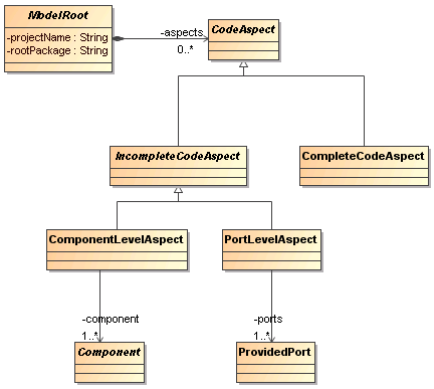
- Using a **naming convention** (enforced and checked by the recipe framework) the manually written code is associated with the model

Incomplete Aspects [Thumbnail]



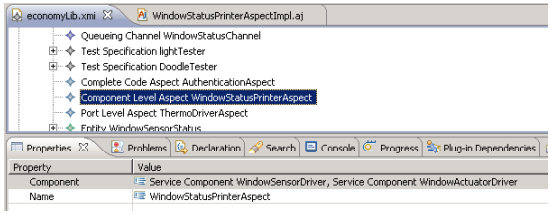
Incomplete Aspects

- Incomplete aspects need to **define their pointcut in the model**.
- Hence we need a joinpoint model for the CBD meta model:
  - Currently we support **operation executions** on service components as joinpoints (same location as the interceptors)
  - The granularity for selection is either a **complete component**, or **components' provided ports.**
- Here is the **meta model** of the joinpoint model:
- Note how easy it is to define a joinpoint model if you use your own domain-specific meta model



Incomplete Aspects II

- Here is the model part of a **component-level** incomplete aspect.
  - It specifies the components to whose methods it wants to advice



- The advice is **manually written**; the implementation class has to **extend a predefined base class**, and it needs to be abstract (conventions checked by recipes)
  - Developers implement a before or after method

```
public abstract class WindowStatusPrinterAspect extends IncompleteCodeAspectBase {
  protected void beforeMethodExecution(JoinPoint jp) {
    // do sensible stuff here
  }
}
```


Incomplete Aspects III

- Based on the pointcut information in the model, the **generator generates a sub-aspect** that contains a **suitable pointcut** – this aspect is then woven.

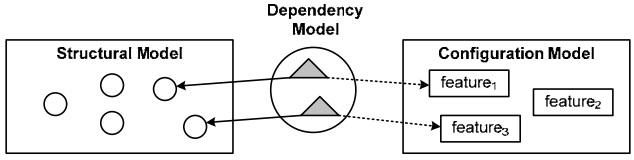
```
public aspect WindowStatusPrinterAspectImpl extends WindowStatusPrinterAspect {
  pointcut pcWindowSensorDriver( smarth….WindowSensorDriverImplementation tgt): (
      call (public * smartho….IWindowSensor.*(..)) ||
      call (public * sma….IWindowSensorCheatInterface.*(..)) ) && target(tgt);

  before( smarthome.ec….WindowSensorDriverImplementation tgt) :
       pcWindowSensorDriver(tgt) {
     this.beforeMethodExecution(thisJoinPoint);
  }

  after( sm….WindowSensorDriverImplementation tgt) :
       pcWindowSensorDriver(tgt) {
     this.afterMethodExecution(thisJoinPoint);
  }

  // more stuff..
}
```

- Note how the pointcut is **restricted** to **the operations of the interfaces** of the provided port, as **implemented by the respective component implementation** class.

CONTENTS

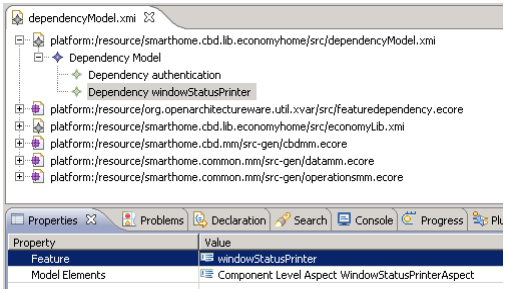
Negative Variability

- 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”



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



 © 2005-7 Markus Völter

Negative Variability for Aspects II

- A **cartridge call to the XVar tool** in the API-level code generator workflow configures the structural model.

```
<workflow abstract="true">

    <readConfig uri="${globalConfigurationModel}"/>

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

    <catridge 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>
```

 © 2005-7 Markus Völter

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**:

```
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.

```
<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.

 © 2005-7 Markus Völter

CONTENTS

- PLE Concepts
- Classical PLE Implementation
  - Source time
  - Compile time
  - Deployment/Configuration time
  - Link time
  - Run time
- MDD-AO-PLE
  - What is MDD
  - What is AO
  - What is MDD-AO-PLE
  - More Terms and Concepts
- **MDD-AO Implementation**
  - Intro to Case Study
  - The Various (Meta-)Models
  - Libraries
  - An Example House
  - Orthogonal Variability
  - Transformation and Template AO
  - AO Modeling
  - Code Level Aspects
  - Negative Variability
  - **Testing**
  - Enforcing Conventions
  - Product Line Evolution
- Summary

 © 2005-7 Markus Völter

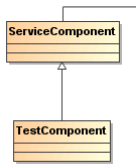
Testing in Product Lines

- In PLE, **testing is important**, just as in "normal" application development.
- We distinguish between
  - **testing the domain architecture** (an activity in domain engineering)
  - and **testing products** (an activity in application engineering)
- In testing products, there is an additional distinction:
  - Testing a specific product with **tests specific to that product**
  - Testing a set of features in a product based with **tests specific to features** of to combinations of features
- Ideally, the **domain architecture** should **support** testing
- **Tests** in that sense **are features** that depend on other ("real") features
  - they are included in a product's test suite if the features they test are included in the product

 © 2005-7 Markus Völter

Testing in SmartHome

- Testing is an activity **on the CBD level**
  - You cannot test on the level of the problem space
  - And OSGI-level would be too technology-specific
- Testcode is currently **written manually** – **wrapped as components**, to be deployable on the target environment.

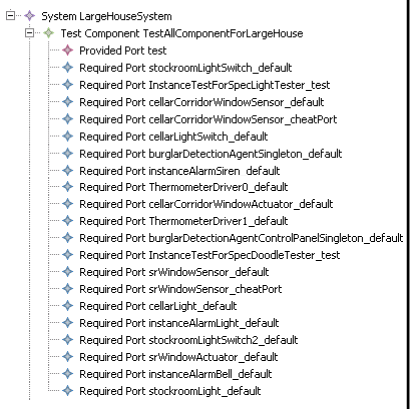


- Components that contain tests are actually **TestComponents** – the metamodel has been extended
- A TestComponent has to provide exactly one port that **provides the *ISystemTest*** interface (which has a *runTests()* method)
- A **test runner** is deployed into the system if tests should be executed (configuration model!)
  - The test runner finds all ports that provide *ISystemTest* and calls their *runTest()* method

 © 2005-7 Markus Völter

Product Specific Tests

- If a certain feature is selected in the configuration model, then a model-2-model transformation **automatically builds the following:**
  - A *TestComponent* that has a required port for each provided of each instance of each component in the system
  - A configuration and a singleton instance of this component.
- Based on the "rest" of the toolchain, the component is build, packaged, activated …



 © 2005-7 Markus Völter

Product Specific Tests II

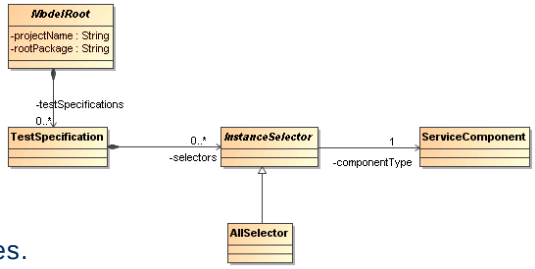
- The developer **implements the test inside that component** using the well-known CBD-level implementation idioms.

```
public class TestAllComponentForLargeHouseImplementation
           extends TestAllComponentForLargeHouseImplBase {

  @Override
  protected void runTestInternal() {
    if ( cellarLight_defaultPort() != null ) {
      cellarLight_defaultPort().turnOn();
      assertTrue( cellarLight_defaultPort().isOn(),
            "switching on the light did not work");
    } else {
      fail( "cellarLight_defaultPort not connected!" );
    }
  }

  @Override
  protected String getTestDescriptionInternal() {
    return "testing complete system (TestAllComponent)";
  }

}
```

 © 2005-7 Markus Völter

Feature-Dependent Tests

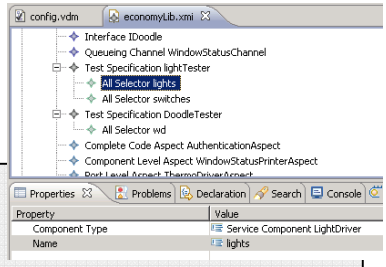
- A feature-dependent test is **only included in the system** if the features it depends on are instantiated in the system.
- Specifically, tests are included **if the set of components it tests are included** in the system
- ***TestSpecifications*** are used to describe such tests and their depedencies.



- The API level generator builds a ***TestComponent*** from the *TestSpecification* that has required ports to the provided ports of the respective components
  - The developer implements the test manually using the well-known CBD-Level idioms

Feature-Dependent Tests II

- The test specification shown here wants to test *LightDrivers* and LightSwitches.




```
public class TestForSpecLightTesterImplementation
             extends TestForSpecLightTesterImplBase {

  @Override
  protected String getTestDescriptionInternal() {
    return "testing functionality of lights and light switches";
  }

  @Override
  protected void runTestInternal() {
    for (int i = 0; i < rpForLightDriverDefaultPortCount(); i++) {
      ILightDriver d = rpForLightDriverDefaultPort(i);
      d.turnOn();
      assertTrue(d.isOn(), "light "+d.getInstanceInfo()+
           " is not turned on even after it has been turned on");
      d.turnOff();
      assertTrue(!d.isOn(), "light "+d.getInstanceInfo()+
           " is not turned off even after it has been turned off");
    }
  }
}
```

CONTENTS

 © 2005-7 Markus Völter

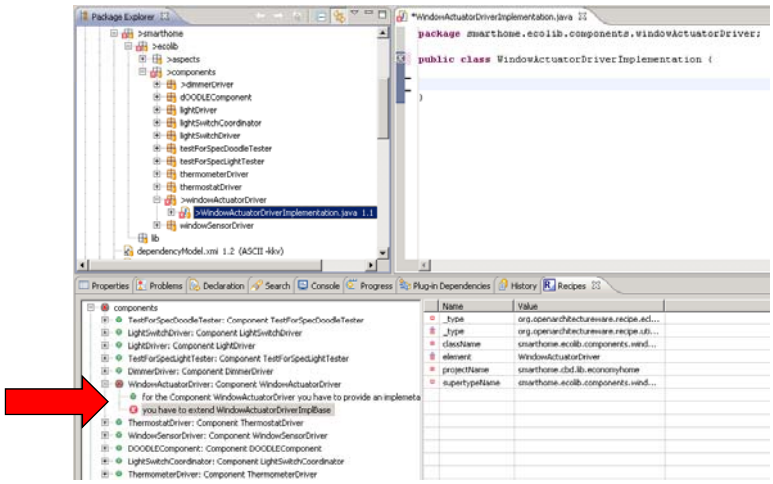
The use of Conventions

- Since we do not want to modify generated code, all kinds of **(naming) conventions** are used:
  - **Components:** Base class is generated, developers have to extend this base class
  - **Code Aspects:** Developers have to manually write Java classes with a certain name, inheriting from a given base class; and the class must be abstract
- We use the **oAW Recipe Framework** to notify developers of remaining manual coding steps:
  - The recipes check all the code in the IDE workspace – generated and manually written
  - They are created as part of the workflow
- As a consequence, **all the conventions are "toolified".**

 © 2005-7 Markus Völter

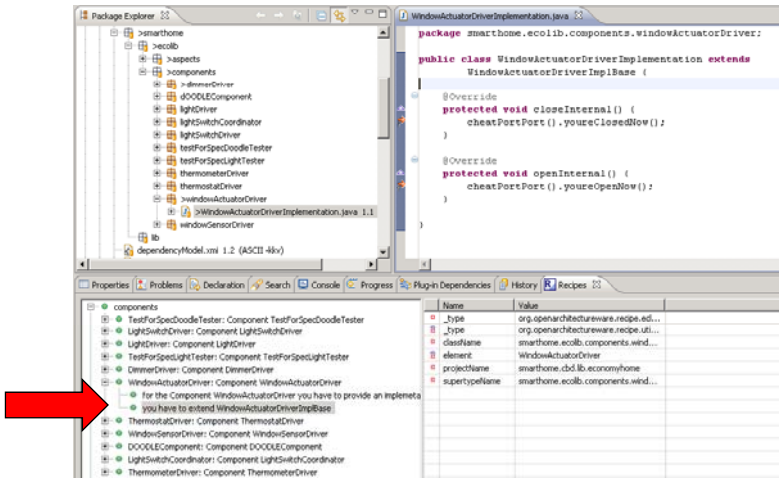
Recipe Framework I

- The manual class exists, but **does not extend** the generated base class



Recipe Framework II

- The extends has been added correctly.

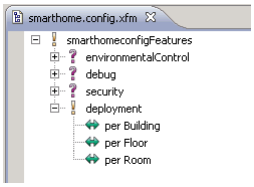


CONTENTS

 © 2005-7 Markus Völter

Unexpected: Deployment

- As a consequence of new application scenarios, it became necessary to change the **granularity of the deployment**:
  - Currently, we use one bundle per Floor
  - We now need one bundle per Room
- Instead of hardcoding that new requirement into the tool, we wil make the deployment a **configuration option**.
- The first step is to update the configuration model:



- We then have to change the **ps2cbd transformation**
  - In cbd2osgi, a *Configuration* is mapped to *Bundles*
  - So we have to make sure, we generate the appropriate set of *Configurations* from the problem space models.

 © 2005-7 Markus Völter

Unexpected: Deployment II

- Here is the original code:

```
create System transformPs2Cbd( Building building ):
    setName( building.name+"System" ) ->
    setConfigurations( { building.createBuildingConfiguration() } );

create Configuration createBuildingConfiguration( Building building ):
    setName( building.name+"BuildingConfiguration" ) ->
    setSubconfigurations( building.floors.createConfig() );

create Configuration createConfig( Floor f ):
    setName( f.name+"FloorConfiguration" ) ->
    instances.addAll(  f.rooms.lights.createInstance() ) ->
    instances.addAll(  f.rooms.devices.createInstance() ) ->
    instances.addAll(  f.rooms.windows.sensor.createInstance() ) ->
    instances.addAll(  f.rooms.select(r|r.heating !=
                              null).heating.thermostat.createInstance() ) ->
    f.rooms.devices.typeSelect(LightSwitch).size > 0 ?
                        handleLightCoordinator(f) : null;
```

- We create **one root configuration** for the building (which will not result in a bundle, since it is not a “root” configuration)
- Below that, there’s **one configuration per Floor**.

 © 2005-7 Markus Völter

Unexpected: Deployment III

- Here is the changed version:

```
create Configuration createBuildingConfiguration( Building building ):
    setName( building.name+"TopLevelConfiguration" ) ->
    setSubconfigurations( building.floors.rooms.getConfig() ) ->
    building.floors.rooms.populateConfig();

private populateConfig( Room r ):
    r.getConfig().instances.addAll(  r.lights.createInstance() ) ->
    ...;

private Configuration getConfig(Room r) :
    switch {
        case hasFeature("perFloor"): r.floor.createConfig()
        case hasFeature("perRoom"): r.createConfig()
        default : r.floor.building.createConfig()  //hasFeature(perBuilding)
    };

private Configuration getConfig(Floor f) :
    switch {
          case hasFeature("perFloor"): f.createConfig()
          case hasFeature("perRoom"): f.rooms.get(0).createConfig()
          default : f.building.createConfig()  //hasFeature(perBuilding)
     };

create Configuration createConfig( Floor f ): …

create Configuration createConfig( Room f ): …

create Configuration createConfig( Building f ): …
```

 © 2005-7 Markus Völter

Unexpected: Deployment IV: Summary

- The **only necessary change** was localized in the ps2cbd transformation (and in the feature model to select the alternative)
- All the bundle stuff, the generation of the ant files, build, deployment, etc. followed **without additional changes**.
- The effort was less than one hour.

CONTENTS

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!

© 2005-7 Markus Völter