ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

MODEL TO CODE TRANSFORMATIONS FOR SOFTWARE PRODUCT LINES

Author:

Carlos Nebrera Cuevas

Supervised by:

Lidia Fuentes Fernández

Pablo Sánchez Barreiro

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, June 2009

INGENIERÍA INFORMÁTICA

TRANSFORMACIONES DE MODELO A CÓDIGO PARA LINEAS DE PRODUCTOS SOFTWARE

Realizado por

Carlos Nebrera Cuevas

Dirigido por

Lidia Fuentes Fernández

Pablo Sánchez Barreiro

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Junio de 2009

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFOMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente Dº/Dª.____

Secretario Dº/Dª. ____

Vocal Dº/Dª. ____

para juzgar el proyecto Fin de Carrera titulado:

TRANSFORMACIONES DE MODELO A CÓDIGO PARA LINEAS DE PRODUCTOS SOFTWARE

del alumno Dº/Dª. Carlos Nebrera Cuevas

dirigido por Dº/Dª. Lidia Fuentes Fernández

Pablo Sánchez Barreiro

ACORDÓ POR______________________ OTORGAR LA CALIFICACIÓN DE ________________________

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

El Presidente El vocal El Secretario

Fdo: Fdo: Fdo:

Málaga, a de del 2009

Table of Contents

List of Figures **5**

List of Tables **6**

Acronyms **7**

CHAPTER 1: Introduction **9**

1.1 Feature-Oriented Model-Driven Software Product Lines 9

1.2 The AMPLE project 12

1.3 Motivation and Scope of this work 13

1.4 Structure of the thesis 14

CHAPTER 2: Background **15**

2.1 Software Product Line 15

2.2 Smart Home Case Study 18

2.3 Model-Driven Development 21

2.4 Feature-Oriented Programming 22

2.5 The CaesarJ Language 24

2.6 The openArchitectureWare Suite 28

CHAPTER 3: Variability Management with AOP and MDD Technologies **31**

3.1 A Taxonomy of Variation in SPL 31

3.2 AOP/MDD Mechanism for Variability Management 35

3.3 Results of the evaluation 38

3.4 Conclusions 41

CHAPTER 4: Model-Driven Feature Oriented Software Product Lines: The TENTE Approach **43**

4.1 TENTE OVERVIEW 43

4.2 Domain Engineering 47

4.2.1 Architectural Design 47

4.2.2 Code Generation 53

4.2.2 Component Implementation 70

4.3 Application Engineering 71

4.3.1 Configuration of a Specific Architecture 71

4.3.2 Code Generation 74

4.4 Traceability information gathering 80

CHAPTER 5: Related Work **83**
CHAPTER 6: Conclusions and Future Work **85**
6.1 Discussion 85
6.2 Evaluation 90
6.3 Future work 92
References.. **95**
APENDIX A. TENTE Plug-in User Manual.. **103**
A.1 Install and Uninstall the TENTE Eclipse plug-in.. 103
A.2 Uninstallation 105
A.3 Updating.. 106
A.4 Generation of Code Skeletons.. 107
A.5 Code Generation of Specific Products 111
APPENDIX B. Smart Home User Manual.. **115**
B.1 Generating and Executing 115
B.2 UI Description 117
B.3 Functionality.. 119

List of Figures

Figure 2-1 Software Product Line Engineering process 16
Figure 2-2 Object-Oriented version of the Smart Home case study........ 24
Figure 2-3 CaesarJ example for the Smart Home case study 26
Figure 2-4 oAW general architecture diagram 29
Figure 3-1 Code generation general schema 37
Figure 4-1 General overview of TENTE........ 44
Figure 4-2 Simplified SmartHome Cardinality based Feature Model........ 48
Figure 4-3 Simplified SmartHome Component View 48
Figure 4-4 Simplified SmartHome Composite Structure View........ 49
Figure 4-5 VML specification of the Smart Home case study 51
Figure 4-6 Two-level family classes schema 53
Figure 4-7 Package mapping, model example 57
Figure 4-8 `HouseGateway` architectural component........ 59
Figure 4-9 Code generated when transforming the `HouseGateway` component........ 60
Figure 4-10 Separation of implementation files for components 60
Figure 4-11 Floor inner class........ 61
Figure 4-12 Code generated when transforming the `Floor` inner class 62
Figure 4-13 `INotify` interface........ 63
Figure 4-14 Code generated when transforming the `INotify` interface 63
Figure 4-15 `INotify` interface declared in two different packages related by a merge. 64
Figure 4-16 Code generated when transforming the `INotify` interface of the `HeaterManagement` package........ 64
Figure 4-17 `HouseGateway` component with two ports........ 66
Figure 4-18 Code generated when transforming the `HouseGateway` component with two ports 67
Figure 4-19 Provides relationship between the `services` port and the `INotify` interface 68
Figure 4-20 Code generated when transforming the provides relationship between the `services` port and the `INotify` interface 68
Figure 4-21 Requires relationship between the `request` port and the `INotify` interface 69
Figure 4-22 Code generated when transforming the requires relationship between the `request` port and the `INotify` interface 70
Figure 4-23 Final product configuration 72
Figure 4-24 Package structure of the architectural model of a specific product. 73
Figure 4-25 Application model composite structure diagram........ 74
Figure 4-26 Code generated when transforming the leaf package of Figure 4-24........ 76
Figure 4-27 Generated initialization code for a specific product........ 77
Figure 4-28 Code generated when transforming the inner classes corresponding to the configuration of Figure 4-25........ 78
Figure 4-29 Code generated when transforming the attribute initialization corresponding to the configuration of Figure 4-24........ 78
Figure 4-30 Code generated when transforming the port connection corresponding to the model of Figure 4-24 79
Figure 4-31 Gathering traceability information with aspectual templates 81

List of Tables

Table 3-1 Variation in structure results 38
Table 3-2 Variation in data results 38
Table 3-3 Variation in behavior results 39
Table 3-4 Variation in quality results 39
Table 3-5 Variation in environment results 40
Table 3-6 Variation in technology results 40
Table 4-1 Correspondence between architectural elements in UML 2.0 and implementation artifacts in CaesarJ (1) 55
Table 4-2 Correspondence between architectural elements in UML 2.0 and implementation artifacts in CaesarJ (2) 56
Table 4-3 Code generated for the architectural model depicted in Figure 4-7 58
Table 4-4 Correspondence between architectural elements and implementation artifacts at the application engineering level. 75

Acronyms

AHEAD	Algebraic Hierarchical Equations for Application Design
AMPLE	Aspect-Oriented Model-Driven Product Line Engineering
AO	Aspect-Oriented
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
ATF	AMPLE Traceability Repository
CRM	Customer Relationship Management
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
FMP	Feature Modeling Plug-ins
FOP	Feature Oriented Programming
GUI	Graphical User Interface
JET	Java Emitter Templates
J2EE	Java 2 Enterprise Edition
MDD	Model Driven Development
oAW	openArchitectureWare
OO	Object-Oriented
OOP	Object-Oriented Programming
SQL	Structured Query Language
SPL	Software Product Line
UML	Unified Modeling Language
VML	Variability Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

CHAPTER 1: Introduction

This Master Thesis presents TENTE, a Feature-Oriented Model-Driven process for Software Product Lines, which has been developed in the context of the AMPLE project. This chapter introduces the terms Feature-Oriented, Aspect-Oriented and Model-Driven Software development as well as the concept of Software Product Line. Then, we situate this work in the context of the AMPLE project and we present the motivation and goals of this work.

1.1 Feature-Oriented Model-Driven Software Product Lines

This Master Thesis combines technologies coming from different software development paradigms. The pivotal element is variability management in Software Product Lines. By means of applying Model-Driven and Feature-Oriented[1] techniques, we aim to improve the state-of-art of current Software Product Line practices. We introduce each one of these terms in the following.

Software Product Lines

A Software Product Line (SPL) aims to create the infrastructure for the rapid production of software systems for a specific market segment, where these software systems are similar, and therefore they share a subset of common features, but they also present some variations between them (Pohl et al, 2005, Clements and Northrop, 2002, Kaköla and Dueñas, 2006, Laguna et al. 2007, Hallsteinsen et al, 2006).

The main goal in Software Product Line is to, as automatically as possible, to construct specific products where a set of choices and decisions has been adopted on a common model, known as family or reference model, which represents the whole family of products that the Software Product Line covers. Software Product Line

[1] Feature-orientation is considered by several authors as a special form of aspect-orientation (Herrman, 2002; Mezini and Ostermman, 2004, Aracic et al, 2006; Gasiunas and Aracic, 2007). We will use feature-orientation instead of aspect-orientation throughout this work.

Development is often comprised of two different, but related software development processes, known as *domain engineering* and *application engineering*.

At the domain engineering level, we start from requirements documents that describe a family of similar products for a specific market segment. Then, we design a reference architecture and implementation for this family of products. This reference architecture contains the elements that are common to all the products in the family, but it must also contain mechanism for allowing the different variations introduced by the different products belonging to the family.

At the application engineering level, we start from a requirement document for a specific inside this family. This requirement document establishes with specific variations must be included in this specific product. With this information, we introduce these variations in the reference architecture and implementation, obtaining as a result a single software product. Therefore, the main benefit of adopting a Software Product Line approach is the reduction of time and development effort for developing specific products belonging to a same family.

Software Product Line Engineering introduces new issues as compared to engineering of single software-based systems: *variability design and management*, and *product derivation.*

Variability design is concerned with the incorporation of variation mechanisms (e.g. plug-in components, Aspect-Oriented enhancements) into the core assets that enable the construction of a set of reusable software assets that represents the complete range or family of products, including both their commonalities and their variations (Bayer et al, 2006).

Product Derivation is the process of constructing specific software products, after a specific configuration, i.e. a valid set of variants, has been selected, following the directives for composing common and variable software assets (Deelstra et al, 2005).

Model-Driven Development

Model-Driven software development (MDD) (Beydeda et al. 2005, Pastor and Molina, 2007) is a new technology for software development where models are first-class citizens of the software development process, instead of simple mediums for documentation purpose or inter-team artifacts. Using a Model-Driven approach, a software product is obtained by successive refinement of models defined at different

abstraction levels. These models can be automatically processed by tools, enabling part of a model at a specific abstraction level to be automatically generated from the models defined at higher abstraction levels. Thus, each property of a software system (e.g. distributed communication) can be specified at the most suitable abstraction level for that property and successively refined by means of automatic model transformation until the implementation code is obtained (Beydeda et al. 2005, Pastor and Molina, 2007). The main benefit of Model-Driven techniques is the automation of repetitive tasks, which leads to a reduction in the development effort and helps to increase quality.

In a Software Product Line, the composition of a product from a configurable set of reusable software assets is a time consuming and cumbersome process. For instance, at the code level, the instantiation of a specific product inside an SPL often implies writing large configuration files and compilation scripts, with intricate dependencies between them. In order to overcome this shortcoming, different attempts of applying model-driven techniques to Software Product Line Engineering have emerged during recent years (Haugen et al, 2005). They aim to automate repetitive, error-prone and time consuming tasks of the SPL development process, such as the composition of reusable software assets.

Feature-Oriented Software Development

A Software Product Lines is often decomposed into a set of interconnected features, where a feature could be defined as "a unit of variation". A feature represents a certain aspect of a family of product that may be included, or not, in a certain product. Ideally, features should be well-modularized in single software modules, facilitating their composition, maintenance, independent development and evolution. Traditional software development techniques, such as object-orientation (Meyer, 2001), often contribute to achieve this goal. For instance, by using a strategy pattern (Gamma et al, 1995), we can encapsulate different features represented by different strategies in separate subclasses, improving modularization and, as a consequence, maintenance and evolution. Nevertheless, there are some features that cannot be well-encapsulated using this kind of traditional techniques.

Feature-Oriented software development aims to improve separation of concerns by means of providing new encapsulation units, such as family classes, and new composition mechanisms, such as mixin composition, that allows the encapsulation of

features of a Software Product Lines in single software modules, easing variability management and feature modularization.

As already commented, TENTE is part of a bigger picture, which is the AMPLE project. Motivation and challenges of this project are described in the next section.

1.2 The AMPLE project

The AMPLE project is a research project funded by the European Commission inside the 6th Framework Programme. It is integrated by eight partners, which are divided in five universities (University of Lancaster, Universidade Nova de Lisboa, Technische Universităt Darmstad, Ecole des Mines de Nantes, University of Twente and Universidad de Málaga) plus 3 industrial partners (Siemens AG, SAP AG and Holos) .

Current industrial practice in SPL engineering are based on manual processes, which often rely in programming tricks such as conditional compilation and preprocessors, which are inadequate substitute for proper programming language support for variability. Similarly, there is no a systematic management of traceability information for relating variable software artifacts across a SPL engineering lifecycle.

The aim of AMPLE is to provide a Software Product Line (SPL) development methodology, from requirements until implementation, that offers improved modularization of variations, their holistic treatment across the software lifecycle and maintenance of their (forward and backward) traceability during SPL evolution. For achieving this goal, novel Aspect-Oriented and Model-Driven techniques are applied from early requirements engineering until implementation.

Aspect-Oriented Software Development (AOSD) can improve the way in which software is modularized, localizing its variability in independent aspects as well as improving the definition of complex configuration logic to customize SPLs. Model-Driven Development (MDD) can help to express concerns as a set of models without technical details and support traceability of the high-level requirements and variations through model transformations.

Next section describes motivation and scope of TENTE, which has become the backbone of the AMPLE project.

1.3 Motivation and Scope of this work

This master thesis presents TENTE (AMPLE D2.4, 2008), a Feature-Oriented Model Driven process for Software Product Lines. TENTE covers the architecture design and implementation software development stages, both at the domain and the application engineering levels.

TENTE uses advanced techniques for the separation of concerns, such as *family-classes plus mixin composition* (Herrman, 2002; Mezini and Ostermman, 2004, Aracic et al, 2006; Gasiunas and Aracic, 2007), both at the architectural design and the implementation level. This contributes to improve feature modularization, which eases variability management as well as feature maintenance and evolution. Separation of variants is maintained both at the architecture and implementation levels, so benefits of such a separation are maintained through this part of the software development lifecycle.

TENTE uses Model-Driven techniques, such as code generation, to automate repetitive tasks of Software Product Line engineering, especially for the derivation of specific products. Complete implementations of specific products can be automatically obtained by simply providing a selection of features to be included in a product and running a code generator.

TENTE uses UML 2.0 (UML 2.0) as language for modeling software architectures and CaesarJ (Aracic et al, 2006), a Feature-Oriented programming language as implementation language. Code generators have been implemented in xPand, the model-to-text transformation language of the openArchitectureWare Model-Driven suite.

TENTE has been integrated with Aspect-Oriented requirements engineering techniques for Software Product Lines developed in the context of the AMPLE project, such as Arborcraft (Noopen et al, 2009) or VML4RE (Alférez et al, 2008). TENTE also uses tools developed in the context of the AMPLE projects, such as VML4Arch (Loughran et al, 2008, Sánchez et al, 2008), a language for variability management in architectural models or ATF, a traceability framework (Anquetil et al, 2009). Nevertheless, the description of these techniques and tools is beyond the scope of this work.

In order to evaluate TENTE and its associated code generators, it has been applied to two industrial case studies used in the context of the AMPLE project. The first case study is Smart Home Software Product Line, provided by Siemens AG. This case study has been completely redeveloped following the TENTE approach, extracting positive results. Three different products, i.e. three different automatic houses, were automatically generated from the domain engineering infrastructure and successfully tested. The second case study is a kind of Customer Relationship Management (CRM) application for assisting sales processes provided by SAP AG. For this case study, a reference architecture have been created as well as different specific products. The goal of applying TENTE to this case study was to evaluate the expressiveness of the approach, and to demonstrate that TENTE can be applied to Software Product Lines different from the Smart Home. Throughout this thesis, the SmartHome case study would be used as an example to illustrate the different concepts and ideas.

1.4 Structure of the thesis

After this introduction, this thesis is structured as follows: Chapter 2 comments on concepts and tools that have been used during the development of this thesis. It also gives a brief description of the case study used to evaluate TENTE. Chapter 3 is an evaluation of the main AOP and MDD variability management for Software Product Line engineering. Chapter 4 explains the TENTE approach, a Feature-Oriented Model-Driven process for variability management in Software Product Line engineering. Chapter 5 comments on related work. Finally, Chapter 6 contains some conclusions and feature work.

The first appendix explains how to install the TENTE tool and a practical example about how to use it. The second appendix describes the Smart Home case study implementation, which was developed for evaluating TENTE.

CHAPTER 2: Background

This chapter provides some background on the techniques, technologies and tools used by the TENTE approach. The chapter starts introducing Software Product Line Development. Then, we describe the Smart Home case study, a traditional exemplar of Software Product Line Engineering. Next, we explain the novels Model-Driven Development and Feature-Oriented Programming. Then, we provide an overview of the CaesarJ language, a specific language supporting Feature-Oriented programming through family polymorphism plus mixin composition.

CaesarJ is the implementation language selected for TENTE, all the model to code transformations has CaesarJ as target language. CaesarJ is a Feature-Oriented Language based in Java that allows us to simplify the transformation process by transforming model features directly in family classes. Finally sections 2.5 and 2.5 describe the Smart Home and Sales Scenario use cases that has been developed to test the SPL development process.

2.1 Software Product Line

A Software Product Line (SPL) aims to create the infrastructure for the rapid production of software systems for a specific market segment, where these software systems are similar, and therefore they share a subset of common features, but they also present some variations between them (Pohl et al, 2005, Clements and Northrop, 2002, Kaköla and Dueñas, 2006, Laguna et al. 2007, Hallsteinsen et al, 2006). The main goal of a Software Product Line is to decrease development time and cost and increase quality of the products derived from the product line asset base. A product is one concrete variant of all possible base asset configurations.

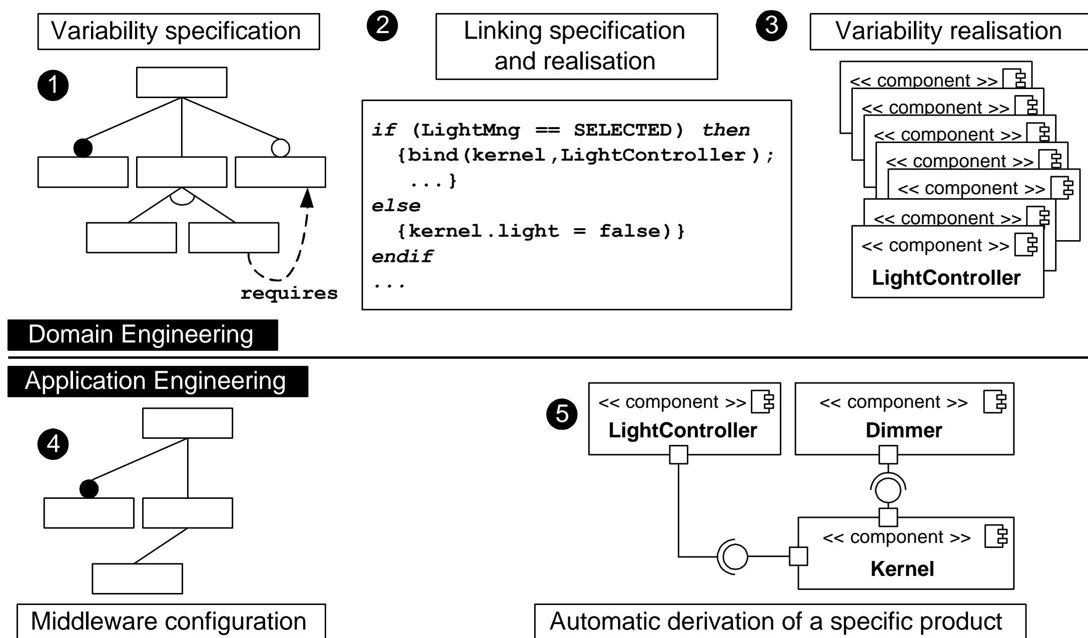


Figure 2-1 Software Product Line Engineering process

Software Product Line Engineering is comprised of two phases: *Domain Engineering* and *Application Engineering* (see Figure 2-1). *Domain Engineering* deals with the creation of the infrastructure or Product-Line Architecture, which will enable the rapid, or even automatic, construction of specific software systems within the family of products a SPL covers. *Application Engineering* is concerned with the engineering of specific products or single software systems using the infrastructure previously created at the Domain Engineering level (Fuentes et al, 2009). The different elements of this process, labeled in the figure, are described below:

1. First of all, variability of the family of products to be developed is analyzed and specified using a feature model (Czarnecki et al, 2005). The feature model specifies the different kinds of variations that can exist between different specific products that can be derived from the SPL. The feature model for SPL plus the constraints between features (i.e. dependencies and interactions between features) represent the 'variability specification', or, using SPL terminology, the *problem space*.
2. Once variability of a family of products has been identified, engineers must develop a system that supports this variability, i.e. software engineers and architects must design a flexible architecture that enables its customization,

including and excluding functionality and components as required. Different mechanisms are available for this purpose, from low-level mechanisms such as conditional compilation, parameterization or generics; to more high-level mechanisms, such as model transformations. As a result of this process, a reference architecture and a reference implementation are obtained. This step represents 'variability realization', or, using SPL terminology, the *solution space*. This reference implementation contains all the components that are required for implementing any product within the SPL family. The only remaining task for obtaining a specific product from this reference architecture and implementation is to appropriately instantiate and connect these components according to the features that must be included in the specific product.

3. The connection between a feature model and a reference architecture is rarely a trivial one-to-one mapping. For instance, the inclusion of the feature *Smart Heating Management,* of the SmartHome case study (see section 2.2), will influence the *HouseGateway, Heater* and *Windows* components, since several operations need to be overridden. Moreover, GUIs are also affected, since we need to add new panels and buttons. Different SPL tools and languages, such as pure::variants (Beuche, 2003), Gears (Krueger, 2007), VML (Loughran et al, 2008) or fmp2rsm (Czarnecki et al, 2005b), support the definition of mappings between a feature model and a reference architecture/implementation, often called *family model*. This mapping usually specifies which actions must be performed when a certain feature is selected. These actions range from generating a certain part of the code of a component to the setting of certain parameters or inclusion/exclusion of certain components from a compilation unit. For instance, in the SmartHome case study (section 2-2), if the *Light Management* feature is selected, the *HouseGateway* controller needs to be redefined, adding the ports and interfaces required for connecting the components related to light management, such as *LightControllers* or *Dimmers*. This kind of rule represents the mapping between *problem space* and *solution space*, and is the basis for the automatic derivation of specific products at the application engineering level, as is explained in the next two points. Once the domain engineering infrastructure or SPL infrastructure has been created, specific products, i.e. a Smart Home with a specific number of rooms and floors and a specific selection of facilities, can be automatically derived. The first step

in this process is the creation of a well-formed configuration, i.e. a configuration that satisfies the constraints specified by the feature model. For instance, in the SmartHome case study, a constraint is that if Smart Energy Management is selected, we must also select the Window Management and Heater Management features, such as both are required by the Smart Energy Management. This configuration specifies which features must be included in the specific product being engineered. Different techniques can be used for creating configurations, such as using a simple feature modeling tool, e.g. fmp (Czarnecki et al, 2005b), dedicated wizards or even Domain-Specific Languages (DSLs) (Santos et al, 2008).

4. Finally, using the configuration created in the previous step, and the mapping created in Step 3, SPL tools, such as *pure::variants* or VML, are able to automatically generate the specific product that corresponds to the desired configuration. This is achieved by interpreting and executing the rules that specify the mapping between the feature model and the reference architecture. As a result of executing these rules, the components that will comprise the specific product, plus their appropriate instantiation, initialization, configuration and compilation files are automatically obtained.

2.2 Smart Home Case Study

The reason for dealing with home automation systems was to get insight into a domain in which the application of Software Product Line Engineering might bring important benefits.

Most everyday-life technical devices can be controlled by microprocessors. Home automation integrates such devices into a network. The network allows the coordination of the functions provided by different subsystems in order to fulfill complex tasks without human intervention.

The home automation domain tackles as major goals: comfort, security and cost saving. Comfort is increased by automating tedious tasks and by an intuitive and well-designed user interface. Security is addressed by identification mechanisms and surveillance equipment like outdoor cameras. Notification mechanisms additionally

contribute to security by allowing for immediate reaction. A similar reasoning holds for life safety. Low costs helps to reduce running costs by smart energy management.

The user must be able to access all devices via a common user interface such as a touch screen. In addition, the residents can use Internet applications and mobile computers to control their home from any place.

Building Blocks of a Home Automation System

Sensors and actuators are mechanical or electronic components that measure and respectively influence physical values of their environment. Smart control devices read data from sensors, process this data, and activate actuators, if necessary. For many control and automation tasks a smart control device can act autonomously.

The home gateway is the central server of a smart home. It offers the processing and data storage capabilities required for complex applications. Users such as residents or technicians can access the services offered by the home gateway via different front-ends that interact with the home gateway and provide a user interface.

User management is a necessary component of the home gateway software. Each individual user has different access rights and different preferences with regard to the system functions. This kind of information is stored in the database of the home gateway and can be accessed by other devices such as electronic door locks.

To avoid additional cabling, power-line communication or wireless communication can be used. A realistic home automation system is inclined to employ a heterogeneous network made up of various network standards and various communication media.

The devices connected to the home network can also differ greatly with respect to their functionality and their software and hardware. As a consequence, the software architecture of a smart home must be able to cope with all kinds of networks and technical devices.

A specific Smart Home Product Line

In this document, we will focus on the development of a Smart Home Software Product Line, with a variable number of floors and rooms (note: the number of rooms per floor is also variable) that offers the following services, categorized in basic and complex facilities:

Basic Facilities

(1) Light management: Inhabitants must be able to switch on, switch off and adjust the intensity of the different lights placed in a room. The number of lights per room is variable. The adjustment should be performed specifying an intensity value. Lights can be controlled individually, at room level, floor level or at full house level.

(2) **Window management:** Inhabitants have to able to have windows managed, specifying the percentage aperture for each window. In addition, if the window would have blinds, these should be rolled up and down automatically. Like for light management, windows and blinds can be controlled individually, at room, floor and full house level.

(3) **Heater Management:** Inhabitants must be able to adjust the heaters of the house to their preferred value. Heater power is automatically adjusted according with the selected temperature. It is possible to select temperatures for the individual heaters, rooms, all rooms in a floor or the whole house. Each room can contain several heaters and their own thermometer so we can have different climate zones in the same room.

Complex facilities

(1) **Smart Heating Management:** The heating control will adjust itself automatically in order to save energy. Once the internal temperature is selected, the house will check the external temperature in the room. If the desired temperature can be acquired by opening the windows, they will be opened and heaters will be switched off. The `Smart Heating Manager` will control the temperature changes to adjust windows aperture and heaters power in order to save energy. In case there is more than one heater, the average selected temperature will be use as reference for the `Smart Heating Manager`.

The complex facilities are optional. Each customer can select the number of facilities he or she desires, although she or he must place devices and facilities at least in three rooms. Otherwise the setting up of the Smart Home will not be cost-effective.

A Specific Smart Home

In order to show how the configuration process works, a specific Smart Home will be derived. It will have two floors (first and second floor) and two equipped rooms per floor All the rooms have Window Management, Light Management and Heating Management. These features can be controlled individually, at room, floor or full house level. Smart Energy Management has been also selected by the user in order to save energy. It could be activated individually for each room or for the full house. All this functions will be controlled through a GUI installed in the House Gateway of the SmartHome.

2.3 Model-Driven Development

Model-Driven Engineering (MDD) (Beydeda et al, 2005, Pastor and Molina, 2007) is a new technology for Software Development where models are no longer simple mediums for describing software systems or facilitating inter team communication. Models are now first citizens of the software development process, and even the code is managed as a model. Using Model-Driven Development, a software system is obtained through the definition of different models at different abstraction layers. Models of a certain abstraction layer are derived from models of the upper abstraction layer, by means of model transformations.

A model transformation specifies how an output model is constructed based on the elements of an input model. Model transformations languages aim to automate the process of deriving one model from another one. Thus, when the mapping between two different kinds of models is known, e.g. the mapping between an entity-relationship database model and a relational database model, model transformations can provide the following benefits:

- Repetitive, laborious and error-prone tasks, required to create a model from another model are avoided, as transformations are executed by a computer.
- Best practices can be encapsulated in model transformations, ensuring target model quality.
- Knowledge encapsulated in a transformation can be easily reused, as software developers applying the model transformations do not need to know the details about how the mapping is performed. For example, a database architect can construct an

entity-relationship model and then apply a transformation in order to produce a relational model without knowing how the transformation is exactly performed.

- Changes can be easily managed, as they can be done at the corresponding abstraction layer and propagated quickly to lower abstraction levels by model transformation. For example, adopting a replication and load balance strategy in a system can be considered an architectural change. The architectural model in the Model-Driven process would be updated and then the change propagated to design, implementation and deployment models.
- When several transformations, from a source model to different kinds of target models are available, the same source model is reused to generate different systems. For instance, if transformations from relational models to SQL and XML models are available, the same relational model can reused to generate different implementation of the same database.

2.4 Feature-Oriented Programming

Feature-Oriented Programming (FOP) (Prehofer, 2001) is a new software methodology which holds that the best way to remove redundancy and improving efficiency, is to create a large number of minor features, which are then linked together in the functions/methods/procedures (from this point on referred to as functions) which takes care of the core functionality. In terms of abstraction, Feature-Oriented Programming primarily focuses on the features of a system, instead of the objects that comprise it (as one would do in an Object-Oriented language).

Feature–Oriented programming can be seen as a modular methodology, which promotes small, tightly focused, but still general purpose, functions, at the expense of large, specific, functions. Most functions should be seen as a single specific tool to complete a single general task (such as a tool in a Swish army knife), while the program itself should have a clear specific purpose. One task that could easily be envisioned would be a sorting function, which relies on a predicate/comparator to know how to sort, since it is so universally useful. Along the same lines one would find that most mathematical functions today are integrated directly into many programming languages' library's (because they are used so often).

The proponents of Feature-Oriented Programming hold that it would create greater consistency in programming, since the programs would only depend on code being written once, and then referred to (though the code might contain features to change it behavior under certain conditions). Another benefit is that most functions would be placed centrally, which would mean, unlike Object-Oriented Programming which spread their features across several objects – even though the chain of program calls would always be the same, the functions would be readily available to most of the code.

A *feature* is a prominent or salient part of an object or thing. Everyday objects like cars, houses, or dogs are distinguished among similar objects by the set of features they exhibit such as color, size, or breed. A similar scenario can be applied to programs where features correspond to the functionality that programs provide. For instance, consider a word processor program with typical features of file loading and saving, editing options, spelling checker, font formatting, printing options, and so on.

Abstracting programs in terms of features facilitates their understanding. More importantly, it opens the possibility to think about constructing programs that provide different combinations of features. The set of such programs is called a *Software Product Line*, and the members are thus distinguished by the combinations of features they support.

To carry along with the implementation of product lines, the next step is modularizing features so that we could use them to assemble particular products. Unfortunately, conventional modularization approaches like functions, classes or packages are not appropriate for feature modules. A typical feature implementation spreads over several module (class, package) boundaries. Furthermore, a single module (class, package) may contain intertwined fragments from multiple features.

Programmers must lower their abstractions from features to those provided by the underlying programming languages, a process that is far from simple let alone amenable to significant automation. Hence the gap between feature abstractions and their modularization severely hinders product line development. The problem is that feature modularity is not well understood and is not well supported in conventional programming languages.

Next section describes CaesarJ, a language that supports Feature-Oriented programming through family polymorphism plus mixin composition and which has been used as target language of the TENTE approach.

2.5 The CaesarJ Language

CaesarJ (Aracic et al, 2006) is a language developed by the Technical University of Darmstadt, which unifies aspects, classes and packages in a single powerful construct, called *family class*, which helps to solve a set of different problems of both Aspect-Oriented and Component-Oriented programming. CaesarJ integrates Aspect-Oriented constructs for join-point interception with advanced modularization techniques, like virtual classes and propagating mixin composition, enabling the development of large-scale aspectual components. Moreover, CaesarJ supports Feature-Oriented Programming (Prehofer, 2001), by means of virtual classes and mixin composition (Gasiunas and Aracic, 2007).

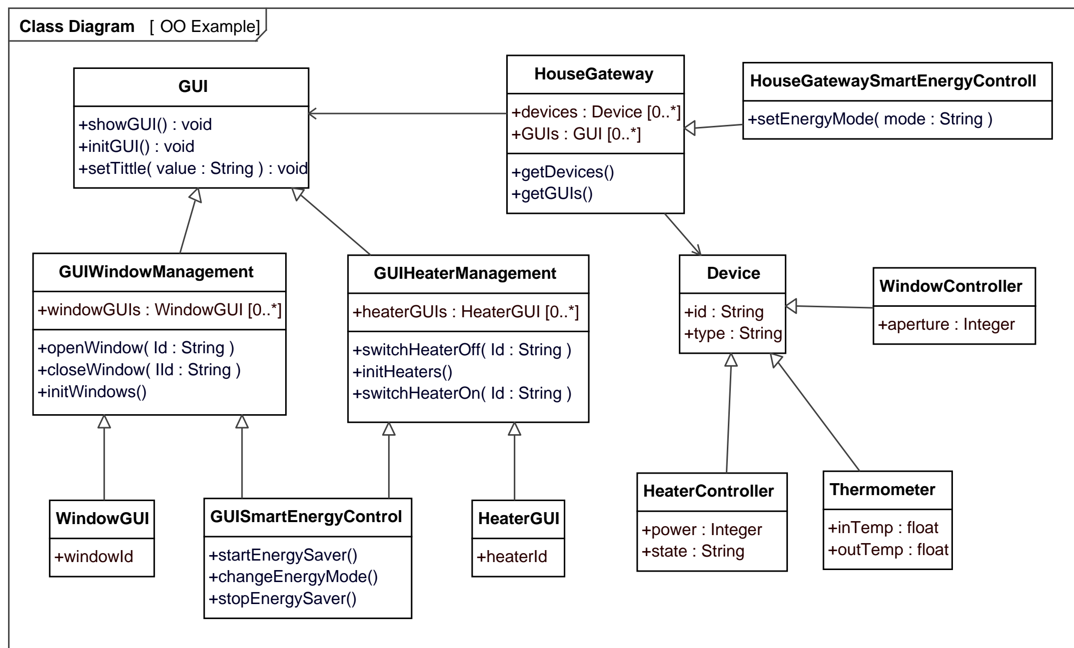


Figure 2-2 Object-Oriented version of the Smart Home case study.

In Object-Oriented Programming, dependencies between classes are not isolated in bigger entities as family classes. If new functionality is added to an existing set of classes, it is usually done by means of inheriting and extending the existing classes. This is illustrated in Figure 2-2 for the Smart Home case study. In this version, for instance, a new child class named `HouseGatewaySmartEnergyControl` extends the `HouseGateway` class in order to add the methods required by the `SmartEnergyControl`

feature. A new `GUISmartEnergyControl` class extends the classes that represent the GUIs for `Window` and `Heater Management`. The main problem of this technique is that inheritance is used at class level. Therefore, an increment in functionality represented by a set of new child classes that extend a set of parent classes cannot be managed consistently as an encapsulated unit.

The first problem of this lack of encapsulation is an increase in the complexity for managing the selection of features. Even if the classes belonging to a same feature are separated in packages, it is necessary to select the concrete classes that are going to be used in a specific product. For instance in order to have *window management* in the specific product, we have to select the `GUIWindowManagement` and `WindowController` concrete classes. Since normal inheritance is used, each class has a different name and therefore the references to those concrete classes must also to be updated. The bigger the number of features of a family of products is, the more complex the relationships between concrete classes become, resulting unmanageable for medium-size Software Product Lines. If multiple inheritance is not allowed, which is the case of popular languages such as Java, the problem becomes even more complex, since it is not trivial to extend several class at the same time. For instance, a class like `GUISmartEnergyControl` would not be allowed. We would need to create two concrete classes, one inheriting from `GUIWindowManagement` and other from `GUIHeaterManagement`, and combine both classes in another one that contains them. This increases the complexity of the relationships and references between classes.

In order to overcome these problems, the CaesarJ improves the OOP type system, by encapsulating dependencies between classes in *family classes*, transforming the OO classes in *virtual classes*, and introducing the mixin composition mechanism. Figure 2-3 shows an implementation of the SmartHome case study using CaesarJ. We will use this figure to illustrate the examples.

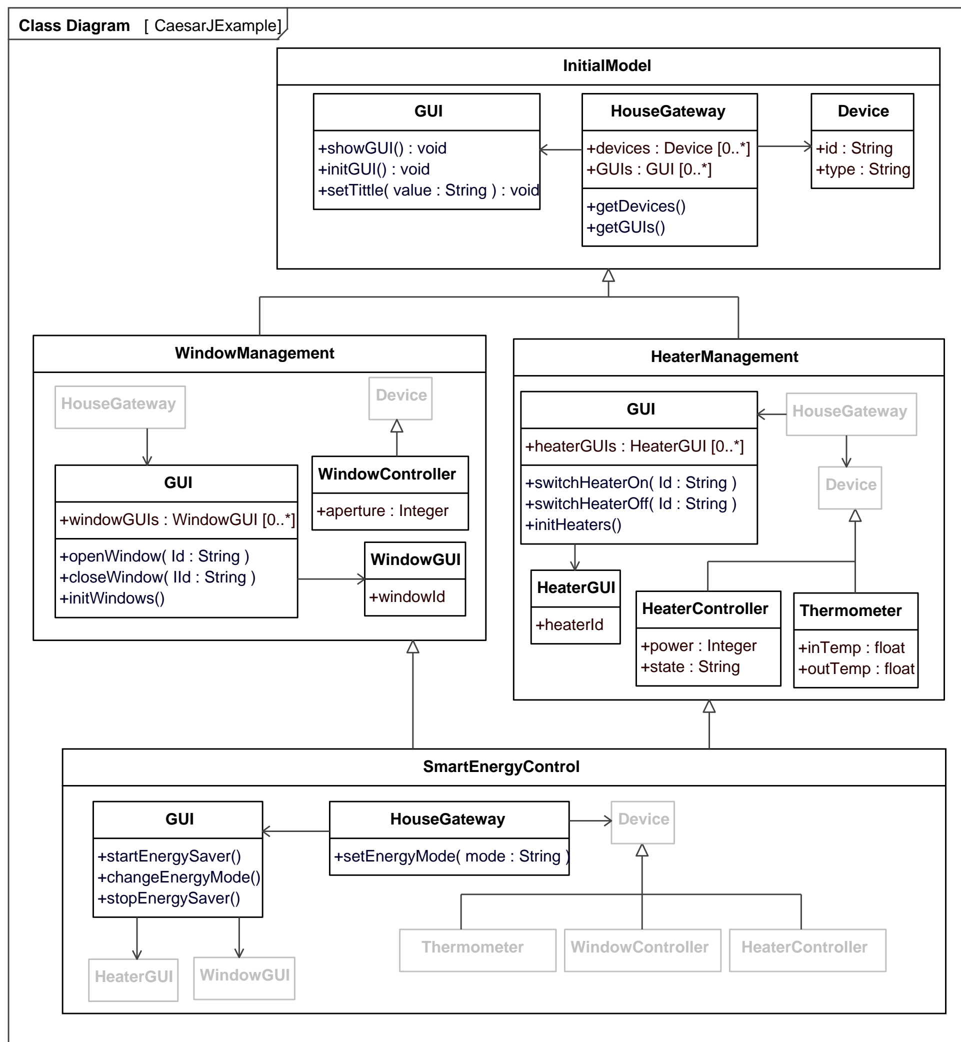


Figure 2-3 CaesarJ example for the Smart Home case study

Virtual classes are inner classes that can be refined in the subclasses of the enclosing class. We will call an enclosing class *family class,* because they define families of objects that are instances of the virtual classes a family class contains. According to the FOP approach, features are modeled as family classes; therefore we can consider that a family class is most likely a feature. For instance, in Figure 2-3, `HouseGateway` is an inner virtual class of the `InitialModel` family class. This inner virtual class is then refined in the `WindowManagement`, `HeaterManagement` and `SmartEnergyControl` virtual classes. A refinement of a virtual class, also known as a further binding, implicitly inherits from the class it refines. For instance, in Figure 2-3, the `HouseGateway` class in the `WindowManagement` family class implicitly inherits from the `HouseGateway` class of the `InitialModel` family class. In a refinement, we can add new methods, fields and inheritance relationships as well as override the inherited

methods. In each family class all references to a virtual class are always bound to its most specific refinement. For instance, in Figure 2-3, all references in the `SmartEnergyControl` family class to a `HouseGateway` class refer to the `HouseGateway` class refinement of that `SmartEnergyControl` family class. Since family classes can contain a set of classes, they can be used instead of packages. In this way we can enjoy the benefits of inheritance, interfaces and polymorphism at the scale of sets of classes. These features are useful for implementation of large scale extensible components.

Mixin composition is a form of multiple inheritance, which is based on linearization of the inheritance graph. Inheritance linearization (Ernst, 1999) is a common mechanism to reduce a multiple inheritance graph to an ordered list, so that the order of the elements in the list determines the behavior in a case of ambiguity. The linearization defines the overriding order of inherited methods.

CaesarJ implements a propagating mixin composition, which means that the composition propagates into virtual classes: all inherited declarations of virtual classes with the same name are composed by mixin composition. Since virtual classes may also have super classes, these are composed with mixin composition as well. For instance, in Figure 2-3, the `SmartEnergyControl` family class inherits from `HeaterManagement` and `WindowsManagement` at the same time. In order to avoid conflicts due to multiple inheritance, mixin composition is applied to these inheritance relationships. The propagating mixin composition provides a large-scale multiple inheritance that allows to compose independent extensions of large scale components.

The main benefit of the *family class* encapsulation appears when we want to evolve an existing set of classes, adding new functionality over them. Each family class, such as `WindowManagement`, contains a set of classes that implements a new feature by means of extending a set of existing classes. These extensions do not require modifications of the classes being extended and names are preserved. For instance, the `WindowManagement` family class adds new functionality to the classes of the `InitialModel` family class. More specifically, a new class `WindowGUI` is created and the `GUI` class is extended and associated with `WindowsGUI`. It should be noticed the `GUI` class maintains the same name.

Since classes are grouped by features, it is possible to instantiate a product just instantiating family classes. Moreover, a main benefit of CaesarJ is that since class names are not modified in each refinement performed by a family class, all references to

a class are automatically re-bound to the refined class. For instance, the `HouseGateway` class in the `WindowManagement` class will refer to the extended version of the `GUI` class, containing the new methods (e.g. `openwindows(id:String)`) added in the `WindowManagement` refinement, instead of the original `GUI` class defined in the `InitialModel` family class. This reference updating is automatically ensured by the CaesarJ type system and eliminates the burden of having to select between different concrete versions of a class, according to the features selected. Finally, multiple inheritance is allowed at family class level. Therefore, new features that made use of more than one of the previous features, like `SmartEnergyControl` can redefine common classes like `GUI` without conflicts.

Hence, CaesarJ seems to be a suitable language to implement Software Product Lines, since CaesarJ, as compared to other programming languages facilitates that:

- Coarse-grained reusable assets, or architectural increments, can be encapsulated into family classes, enabling the separation of these reusable assets at the code level. For instance, coarse-grained features such as `LightManagement` can be well-encapsulated into a single family class.
- Propagating mixin composition allows a feature can extend several features, i.e. a feature can refine several features at the same time.

Virtual classes were originally introduced in BETA (Madsen and Mollen, 1989) and were further developed in gbeta (Ernest, 1999), which supplemented them with mixin composition and family polymorphism. CaesarJ provides a solid implementation of these concepts on the Java Virtual Machine and combines them with language features for crosscutting composition, such as pointcuts and advices.

2.6 The openArchitectureWare Suite

openArchitectureWare (oAW) is a suite of tools and languages for Model-Driven development. This section will briefly explain the different parts of oAW. Since the oAW offers a wide range of languages and features, we focus on this section on those parts of the oAW suite that have been explicitly used for the development of TENTE. Figure 2-4 shows the components of by the TENTE approach and their interrelationships. This Figure is explained below.

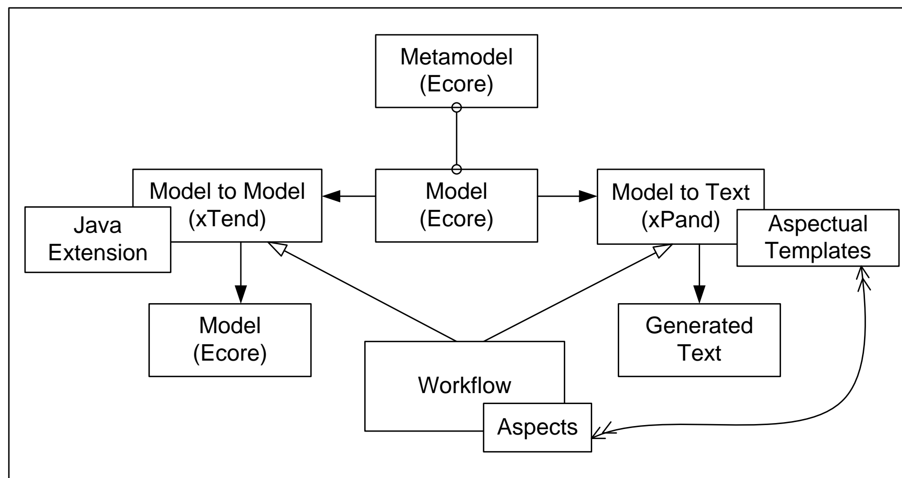


Figure 2-4 oAW general architecture diagram

The first step of a MDD process using oAW is to define a *metamodel* for the *models* we are going to deal with. The metamodel has to be specified in *Ecore*, the metamodelling language of the Eclipse Modeling Framework (EMF) (Budinsky et al, 2003). Once the metamodel has been specified defined, we can construct models that conform to this metamodel. EMF provides certain mechanism for defining models conforming to an *Ecore* metamodel, although they are really basic mechanism, which are not really appropriated for constructing large-scale models.

A better option is to reuse a previously existing metamodel, such as UML (UML, 2005). This has been the one followed in this thesis. This allows us to use third-party tools for model construction, and avoids the need of defining a new metamodel from scratch. oAW provides an adaptor for the UML metamodel, which also supports the use of UML Profiles.

Once we have defined a model, we can use the different languages provided by the oAW suite for manipulating this model in several ways. More specifically, we can use model-to model (M2M) and model-to-text (M2T) transformations.

Model-to-model transformations (Figure 2-4, left side) accept a model as output and generate a target model as output. The input model is transformed according to a set of rules defined in a model transformation language. The model-to-model transformation language of the oAw suite is called *xTend*. This language provides a set of predefined functions for manipulating model. These functions can be extended with new functions, which can be expressed as in the own xTend language and also in Java, in case this were

required (this option is useful for manipulating complex data structures that requires some optimization or for interacting with third-party tools).

Model-to-text transformations (Figure 2-4, right side) accept a model as output and generate text as output. This text is normally code for a specific programming language, or any other implementation-related artifact, such as configuration or deployment files. A model-to-text transformation generates one or several text files from a source model, following the transformation rules specified often in a template-based languages, which specifies how text must be produced according to the elements and the values of these elements in the source model. The model-to-text transformation language of the oAW suite is the *xPand* language.

The xPand language supports the definition of *aspectual model-to-text transformations*. An aspectual model-to-text transformations is a special kind of template which, following the *pointcut plus advice* mechanism of aspect-orientation (Kiczales et al, 1997). The aspectual template captures the execution of an *xPand* rule in a Model-to-Text transformation and generates the desired code. The aspectual template can generates code in the same file as the original template, or in a different one. This allows modifying the code generation without modifying the original templates. The aspectual templates to be applied in a transformation are defined in the workflow of the transformation.

In oAW, we can construct chains of model transformations, where the output of a model transformation is used as input for the following. These chains of model transformations are defined through a special kind of scripts, which are called in oAW *workflows*. A workflow, which is specified in a XML-based language, specifies which files contain the source models for the transformations, the metamodels for these models, what transformations should be invoked, the ordering between these transformations, what outputs are produced and where these outputs must be stored.

In order to apply aspectual templates to a transformation a new workflow is created. This workflow calls the original workflow and specifies which aspectual templates are applied over the original transformation. Using this technique is possible to use several aspectual templates over a transformation by changing the workflow that applies the template.

CHAPTER 3: Variability Management with AOP and MDD Technologies

Aspect-Oriented Programming (AOP) (Kiczales et al, 1997) and Model-Driven Software Development (Beydeda et al. 2005) have appeared in the recent few years as new technologies that improve the development of software systems. Both technologies have revealed initially to have important benefits regarding placement and configuration of variations in the context of a Software Product Line.

This chapter evaluates currently existing Aspect-Oriented Programming (AOP) and Model-Driven Development (MDD) tools and technologies regarding variability management, in Software Product Lines (SPL). They are compared to traditional tools and techniques in this context. The goal of this evaluation is to identify what are the novel and positive contributions of AOP and MDD related to variability management, at the implementation level, in SPL. Then, strengths and weaknesses of AOP and MDD will be analyzed, how each one can complement each other will be finally discussed. We have carried out this task before defining TENTE, in order to get better insights about how MDD/AOP can help to variability management in SPL. This chapter is a survey of a longer technical report (AMPLE D2.2, 2007). Interested reader can find further details about this evaluation in such a document.

3.1 A Taxonomy of Variation in SPL

This section gives a brief explanation of the different kinds of variability that might appear in a SPL. This taxonomy will serve to evaluate each AOP/MDD mechanism of variability management against each kind of variability described.

(1) **Variation in structure**

A Software Product Line is said to have variation in structure when is possible to derive two different products with a different structure, although they could offer the same functionality. For instance, in the case of the Smart Home, it is possible two

derive two specific homes with the same facilities, but a different number of rooms, floors or lights per rooms. These two products differ in number of feature instances. Structural variation is mainly concerned with the creation of a different number of instances when the product is started. We called to this problem, "*the problem of the variable constructor*". For instance, in the SmartHome case study, if `Light Management` is selected, the `HouseGateway` component will have to keep references to all the light devices deployed in the House. These corresponding light objects need also to be created. As the creation of the light objects as the setting of the references to these lights are variable pieces of code.

Additionally, a second problem could appear when the presence or not of one object motivates a change in the behavior of other objects. We named this second problem, "*the problem of the structurally dependent behavior*". For instance, in the SmartHome case study, if `Smart Heating Management` is selected, the heaters have to notify to the `HouseGateway` about temperature changes, so the order to open or close the windows can be given. Nevertheless, this behavior is only necessary if there are windows to be managed in the same room of the heater, therefore it depends of the structure of the specific product.

(2) **Variation in data**

Variation in data refers to the possibility of deriving two specific products from the same SPL which operate with different input, output or intermediate data. We distinguish basically two different kinds of variation in data:

(1) The data types are not the same. For instance, in the SmartHome case of study, we could have had a heater that works with temperatures expressed as integers, but the thermometer it is connected to sends temperatures measured as float numbers.

(2) The data types are the same, but they are semantically different. For instance, heaters in the SmartHome SPL accept float numbers as temperature values, but these float numbers could have expressed temperatures either in Celsius or Fahrenheit.

(3) **Variation in behavior**

Variation in behavior refers to the possibility of deriving two specific products from the same SPL which present different behavior.

We distinguish three different kinds of variation in behavior (which are not mutually exclusive):

- Variation that implies the addition of new software modules which provides new functionalities. (e.g. a new feature like light management is added).
- Variation in the implementation of a service (e.g. different LightController classes can implement a same ILightManagement interface).
- Variation in how the different services are coordinated (e.g. smart heater management controlling the window and heater management).

(4) **Variation in quality**

Variation in quality refers to the possibility of deriving two specific products from the same family which present different quality attributes.

These variations in quality can be due to different reasons: (1) a variation in the internal implementation of one or more methods of one or more classes (e.g. implementation of a method with different performance); (2) a variation in the quality attributes (e.g. different schemas) that are applied over a software module.

For instance, in the Smart Home case of study, the HouseGateway might use different data structures for storing the identifier for the different devices. Each data structure will have a different performance and memory consumption. Similarly, different fault-tolerance schemas could be applied.

(5) **Variation in environment**

Variation in environment refers to the possibility of deriving two specific products from the same family that are deployed in environments with different characteristics.

We identify three different kinds of problems we need to address:

- The software modules that comprise a specific product are distributed into different kind of nodes (e.g. a SmartHome GUI component could be deployed in lightweight devices, such as PDA, or on a common PC).

- The software modules that are part of a specific product could be distributed on nodes differently. For instance a typical web information system is comprised of web interfaces, a business layer and a database. This information system can be deployed using a two-tier schema (the business layer and the database layer are deployed in the same server) or a three-tier schema (the business layer and the database layer are deployed in different servers).
- A specific product deals with different external services (e.g. a SPL could require a payment platform, such as CreditCard service. Several services are available. Thus, depending on the selected service some variations could emerge, such as different method signatures in the interface of the services or different access protocols).

(6) **Variation in technology**

Variation in technology refers to the possibility of deriving two specific products from the same family of products which are constructed using different software technologies or abstractions. By software technology or abstraction we mean different programming languages, different programming techniques (e.g, recursion vs iteration) and so forth.

We have identified two potential sources for variation in technology at the implementation level:

- Changes the programming language.
- Changes in the set of abstractions, primitives, techniques or guidelines that are used to construct the software (e.g. synchronous messages or events).

3.2 AOP/MDD Mechanism for Variability Management

This section gives a brief explanation of the main Aspect-Oriented and Model-Driven mechanisms for variability management in Software Product Lines. Each mechanism is described below.

(1) **Joinpoint interception**

Joinpoint interception can be considered the most basic Aspect-Oriented mechanism, which is present in all of Aspect-Oriented languages, although under different forms and with some slight differences.

An aspect is a special module for encapsulating crosscutting concerns. An aspect encapsulates a crosscutting concern and provides its functionality through a set of *advices* (similar to object methods). These advices are not explicitly invoked by the software modules, instead they are triggered automatically. How and when these advices require being executed is specified by means of special composition rules, called *pointcuts*, which designate logic (in program code) or instants (in program execution) at which advices must be executed. The set of valid points of a program code which can be designated by a pointcut are called *joinpoints*. Finally, a kind of compiler/pre-processor is the responsible of composing all these pieces of code together as specified by the pointcuts. This composition process, called *weaving*, can be performed at compile time (static weaving), load-time or even run-time (dynamic weaving).

We have selected AspectJ (Kiczales et al, 2001) for illustrating this Aspect-Oriented mechanism because AspectJ can be considered the most mature and well-known Aspect-Oriented language.

(2) **Intertype declarations**

An intertype declaration is a mechanism that combined with *joinpoint interception* allows modifying the structure of the code. An *intertype declaration* has to be encapsulated inside an aspect. With *intertype declarations* is possible to add methods or

attributes to a class, to change class inheritance, modify existing methods, override existing attributes, etc.
AspectJ supports this mechanism, so we have selected AspectJ as language for evaluating this mechanism for the same reasons as in the previous point.

(3) **Family polymorphism plus mixin composition**

A family class[2] is a large-scale piece of functionality which involves a group of related classes. Abstraction, late binding, and subtype polymorphism is supported at the level of family classes. A family class is a special class which can contain inner classes called "virtual classes". Just like methods and fields, they are also members of instances of their enclosing family class, called family object. Hence, at any time during execution their meaning is relative to the dynamic type of the family object. Subclasses of a family class can refine inherited inner classes (further-binding). In such further binding, we can override inherited methods, add new methods or new state, as well as add additional superinterfaces and superclasses.
Mixin composition is a composition mechanism that allows a family class to inherit from more than one super class, with the constraint that super classes must have a common super type. A special linearization method is used to avoid conflicts in the composition.

We have selected CaesarJ (Aracid et al, 2006) for evaluating this variability management mechanism because CaesarJ can be considered the most mature and well-known language implementing family polymorphism plus mixin composition. Moreover, CaesarJ, for internal constraints of the AMPLE project, must be the target language of the TENTE approach.

(4) **Code Generation**

Code generators are tools that allow the automatic generation of source code from some sort of model. The rules for the code generation are defined in one or more code generator template files, which specify how the code must be produced according to the contents of one or more input models. The input model can vary from a simple list of parameters to a UML model. These rules are interpreted by a code generator engine.

[2] Other terms used in literature for family classes are collaborations, layers, teams, feature classes…

When used for variability management, the goal of code generation to automatically generate the variable code of a Software Product Line, using as input some model that indicates what features have been selected or unselected. For our evaluation, we will consider that the generated code is common Object-Oriented code. Figure 3-1 shows the general schema of a code generator tool in which several template files are applied to an input model file, generating several source code files as a result.

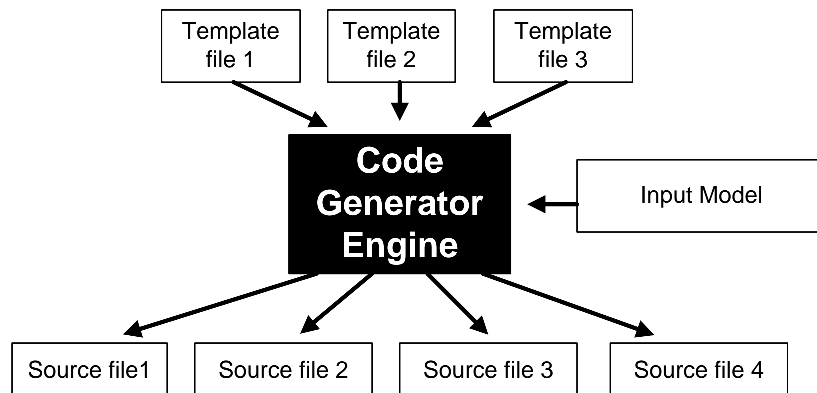


Figure 3-1 Code generation general schema

We have selected Java Emitter Template[3] (JET) as a representative of the code generation variability mechanism for the evaluation. JET is a subproject of Eclipse Modeling Framework (EMF) (Budinsky et al, 2003). A JET template includes fixed code that is written directly on the output source code language. Inside the source code special tags are used to insert code depending of the input parameters. We have selected JET because it is a well-known and exemplar code generator and we have previous experience using it.

[3] http://www.eclipse.org/modeling/m2t/?project=jet

3.3 Results of the evaluation

This section summarizes the results of our evaluation, outlining benefits and drawbacks of each variation mechanisms as compared to each other. Further details about this evaluation can be found in Nebrera et al (AMPLE D3.2, 2007).

The results of the evaluation are presented through tables 3-1 to 3-6. Each column refers to one of the Aspect-Oriented or Model-Driven variation mechanism analyzed. Each row refers to a particular variability scenario (e.g. changing the coordination protocol) inside a specific kind of variation (e.g. variation in structure). Each cell contains a short sentence that outlines if the corresponding variation mechanism improves or not the current state-of-art and brief justification for such a sentence. If there is improvement, the variation mechanism is considered as suitable for dealing with that kind of variation, if there is not such an improvement, it is considered as not suitable. For the cases where the mechanism can be used but without convincing benefits, the variation mechanism is considered just as *usable*.

Variation in structure	Joinpoint interceptions	Intertype declarations	Feature classes	JET code generator
The problem of the variable constructor	**Not suitable** (the separation does not provide clear benefits)	**Not suitable** (no improvement over joinpoint interception)	**Not suitable**, the solution is the same one than for object orientation	**Suitable**, automatically generation of the initialization code
The problem of behavior structurally dependent	**Suitable,** especially if there is crosscutting. It separate the variation from the original code	**Not suitable**, no improvement over joinpoint interception	**Suitable** if there is no crosscutting, helps to encapsulate dependencies avoiding class castings	**Suitable** if there is no crosscutting, avoid manual modifications to select between variants

Table 3-1 Variation in structure results

Variation in data	Joinpoint interceptions	Intertype declarations	Feature classes	JET code generator
Different data types	**Suitable**, avoid the use of castings and modifications of the source code, quantification for dealing with crosscutting	**Not suitable**, no improvement over joinpoint interception	**Suitable** but achieving only encapsulation benefits and reusability	**Suitable** if is solved by parameterization of references. Suitable by parameterization of methods if there is no crosscutting
Semantically different data types	**Suitable**, same solution than for different data types	**Not suitable**, no improvement over joinpoint interception	**Suitable**, same solution than for different data types	**Suitable**, same solution than for different data types

Table 3-2 Variation in data results

Variation in behaviour	Joinpoint interceptions	Intertype declarations	Feature classes	JET code generator
Addition of new components with new functionalities	**Not suitable**, (the separation does not provide clear benefits) Not clear mechanisms for adding new interfaces.	**Not suitable**, no improvement over joinpoint interception	**No suitable**, the solution is the same one than for object orientation	**Suitable**, automatically generation of the initialization code and new interfaces.
Variation in the implementation of one service	**Not suitable**, no improvements against object-oriented techniques	**Not suitable**, no improvement over joinpoint interception	**Suitable** but achieving only encapsulation benefits and reusability	**Suitable**, avoid manual code modifications
Variation that affects several services or interfaces	**Suitable**, due to the crosscutting nature of the problem	**Not suitable**, no improvement over joinpoint interception	**No suitable**, no mechanism to solve the crosscutting	**No suitable**, no mechanism to solve the crosscutting
Variation in how the different services are coordinated	**Suitable**, decouple coordination from computation and encapsulate the crosscutting coordination	**Not suitable**, no improvement over joinpoint interception	**No suitable**, no mechanism to decouple coordination from computation, no solution for the crosscutting	**No suitable**, no mechanism to decouple coordination from computation, no solution for the crosscutting

Table 3-3 Variation in behavior results

Variation in quality	Joinpoint interceptions	Intertype declarations	Feature classes	JET code generator
Internal method implementation	**Not suitable**, no improvements as compared with object orientation	**Not suitable**, no improvement over joinpoint interception	**Suitable** but achieving only encapsulation benefits and reusability	**Suitable** in absence of crosscutting, simplifies variant selection
Quality attributes	**Suitable**, mainly due to the crosscutting nature of quality attributes	**No suitable**, no improvement over joinpoint interception	**No suitable**, no solution for the crosscutting	**No suitable**, no solution for the crosscutting

Table 3-4 Variation in quality results

Variation in environment	Joinpoint interceptions	Intertype declarations	Feature classes	JET code generator
Node type:_ implementing several components	**Not suitable**, (there is not clear benefits for the separation of the code)	**Not suitable**, no improvement over joinpoint interception	**Suitable** but achieving only encapsulation benefits and reusability	**Suitable**, no modifications needed to select between variants
Node type: managing alternative references	**Suitable** to externally manage the alternative references	**Not suitable**, no improvement over joinpoint interception	**Suitable**, no modifications of the original code, encapsulation of dependencies	**Suitable**, no modifications needed to select between variants
Deployment configuration	**Usable**, helps isolate the remote communication from the original code and solve the crosscutting. Not clear benefits over current middleware technologies.	**Not suitable**, no improvement over joinpoint interception	**No suitable**, no solution for the crosscutting	**No suitable**, no solution for the crosscutting
External services	**Suitable**, facilitate the implementation of adapters for the external services	**Not suitable**, no improvement over joinpoint interception	**Suitable** in absence of crosscutting	**Suitable** in absence of crosscutting

Table 3-5 Variation in environment results

Variation in technology	Joinpoint interceptions	Intertype declarations	Feature classes	JET code generator
Programming language	**Not suitable**, no mechanism to solve the problem	**Not suitable**, no mechanism to solve the problem	**Not suitable**, no mechanism to solve the problem	**Not suitable**, no mechanism to solve the problem at implementation level
Set of abstractions, primitives and guidelines	**Suitable** in cases where there is crosscutting in the solution and the solution can be encapsulated into an aspect.	**Not suitable**, no improvement over joinpoint interception	**Suitable** if there is no crosscutting, and the solution can be encapsulated into an aspect.	**Suitable** if there is no crosscutting, and the solution can be encapsulated into an aspect.

Table 3-6 Variation in technology results

3.4 Conclusions

This section comments on the results described in tables 3-1 to 3-6:

- *Joinpoint interception* has demonstrated to be an interesting mechanism for dealing with certain kind of variability problems, particularly when dealing with these problems implies dealing with crosscutting pieces of code. This is not surprising, since joinpoint interception was precisely created for solving the lack of modularization of crosscutting concerns (variable or not). From the 17 kinds of variability analyzed, 7 of them could be solved with an improvement as compared with the current state-of-art using *joinpoint interceptions*. Other important benefit of this mechanism is that is able to modify a previously existing code without any manual modification on it (although depending on the language, access to the source code could be required in order to perform the weaving). This characteristic makes joinpoint interception particularly suitable for implementing component adapters and coordinators (Fuentes and Sánchez, 2005. Fuentes and Sánchez, 2007).
- Family classes provide some benefits related to reusability and in some degree to scalability, since dependencies between variants are well-encapsulated. The main variation mechanism behind family classes is inheritance between them. This inheritance affects the virtual classes defined inside the family class. So, virtual class of parent family classes can be extended and/or overridden in child family classes. So, dependencies and relationships between classes are encapsulated into a family class. This simplifies the management of such a dependency, which are now automatically managed by the language compiler.
- Code generation offers the best solution, from the analyzed ones, for dealing with *the problem of the variable constructor,* since it allows the generation of large amounts of initialization code in an easy and manageable way.
- In any Aspect-Oriented case, variability management depends on if a certain aspect is introduced or not into a compilation unit. These mean a specific make or ant file need to written for each specific product inside a SPL. Code generation can be used to construct a template make file, which generates a specific make file for a specific configuration of variants. Additionally, some parts of these aspects need to be manually written at the application engineering level. These parts often follow a well-defined pattern than can be written as a template code generator that, with the

adequate input parameters, generates the corresponding aspect for a specific product and variants configuration.

There is no mechanism able to solve all kinds of variations itself. Each one of them offers different advantages and disadvantages for each particular case. It seems that the combination of code generation and family classes plus mixin composition allows solving almost all variability types that can be found in the context of a Software Product Line. Thus, we have opted for creating a Feature-Oriented Model-Driven process, which uses family classes plus mixin composition, both at the domain and application engineering levels. Code generators are used to generate code skeletons at the domain engineering level and the complete code for specific products at the application engineering level. CaesarJ was selected as target language, since it is the most well-known and mature language providing family polymorphism plus mixin composition. The xPand language of the openArchitectureWare Model-Driven suite was selected as code generation language by internal constraints of the AMPLE project.

CHAPTER 4: Model-Driven Feature Oriented Software Product Lines: The TENTE Approach

This chapter provides a general overview of TENTE[4], our Feature-Oriented Model-Driven process for Software Product Line Engineering with advanced mechanisms for separation of concerns. TENTE uses advanced mechanisms, such family polymorphism pus mixin composition, for separation of concerns both at the architectural and implementation levels.

4.1 TENTE OVERVIEW

Although some processes currently exists that applies advanced techniques for separation of concerns and/or Model-Driven techniques to Software Product Line engineering (Trujillo et al, 2007; Völter and Groher, 2007; Laguna et al, 2007), there is a general lack of processes that integrate all them. Advanced techniques for separation of concerns at the implementation level, often do not have a corresponding counterpart at the modeling level. Similarly, processes that provide advanced separation of concerns through the complete software lifecycle (Laguna et al, 2007), often do not take advantage of Model-Driven techniques.

In order to overcome these shortcomings, the TENTE approach presents an innovative process for Software Product Line architectural design and implementation that integrates relevant advances, from a SPL point of view, for separation of concerns, such as family polymorphism and mixin composition, and MDD technologies.

[4] TENTE is the Spanish name for Lego. We have selected this name because we view an SPL as a Lego game: it is about constructing specific products from prebuilt blocks.

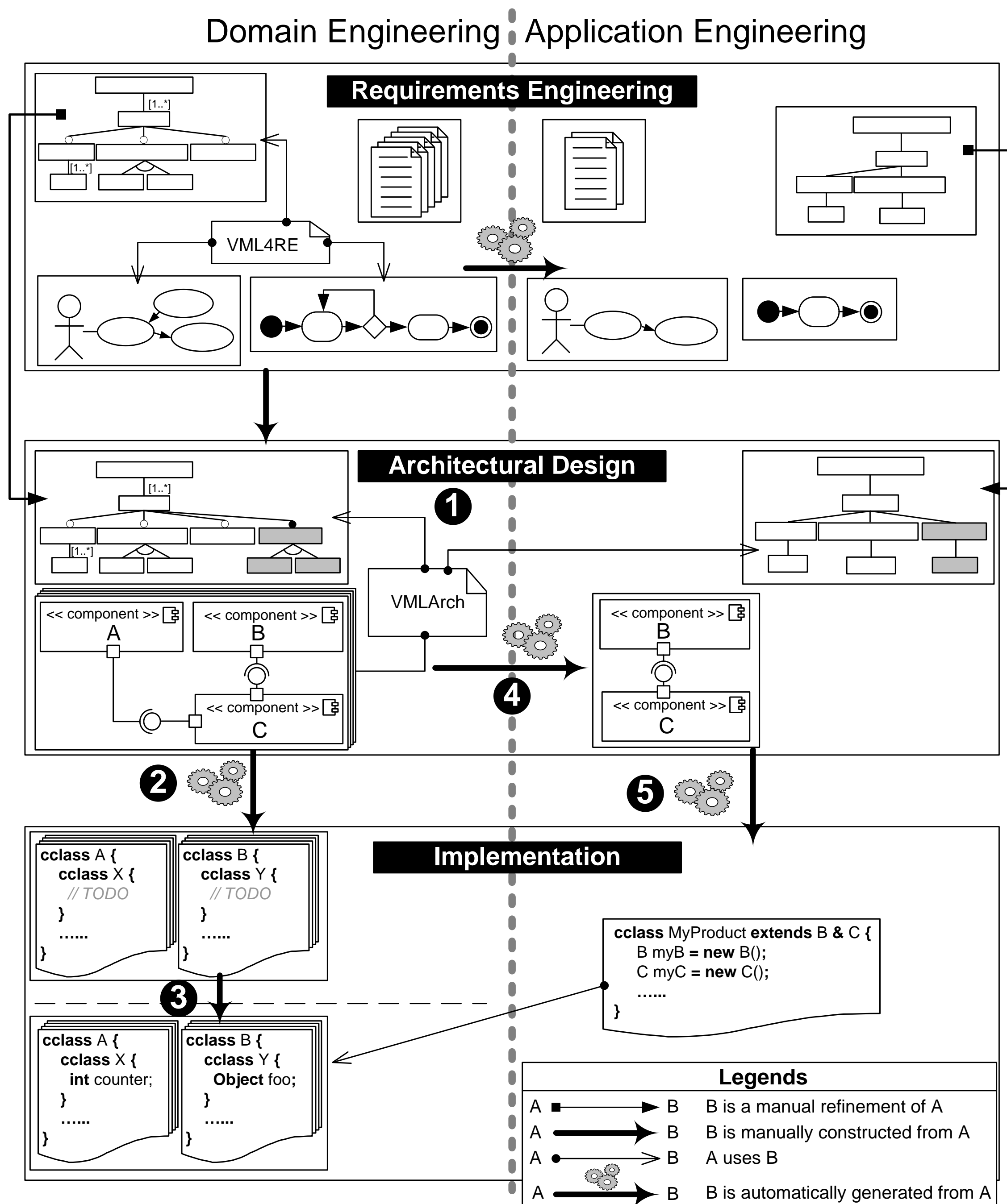


Figure 4-1 General overview of TENTE

TENTE is comprised of five steps, as depicted in Figure 4-1. The process covers the architectural design and implementation software development stages, both at the domain and application engineering levels. Architectures are expressed in UML 2.0, according to the AMPLE architectural modeling language (AMPLE D2.2, 2007). The implementation language selected is CaesarJ (Aracic et al, 2006), a subset of the

upcoming AMPLE implementation language (AMPLE D3.2, 2007). The first three steps correspond to the Domain Engineering level. They serve to create the infrastructure from which specific products will be derived. The last two steps correspond to the Application Engineering level and they serve to create specific products inside a Software Product Line. The whole process is described as follows:

1. **Architectural Design.** First of all, an architectural model for the SPL is constructed (Figure 4-1, label 1). This model, which we have named *reference architecture*, contains the architectural design of both the commonalities and the variabilities of a complete family of products. Variability specification, i.e. the declaration of which parts of the architecture are variable and why they are variable, is expressed by means of cardinality-based feature models (Czarnecki et al, 2005). The architectural model is designed in UML 2.0. Coarse-grained variants are separated in different UML packages, which are combined by means of the UML merge operator, similarly to Laguna et al (Laguna et al, 2007). The link between the feature model and the architecture expressed in UML 2.0 is established using VML (Variability Modelling Language) (Loughran et al, 2008, Sánchez et al, 2008), an innovative language for connecting variability specification (i.e. problem space) with variability realization (i.e. solution space). A VML specification also contains all the information required for automatically deriving the architectural model of a specific product from the family model.

2. **Transformation of architectural models into implementation**. Using model transformations, part of the implementation is automatically generated from the architectural model (Figure 4-1, label 2). More specifically, the skeleton of components and the logic for connecting them are generated. The part corresponding to the behavior of each method is left empty for being completed at the implementation level. Separation of variants achieved at modeling level by using the merge operator is preserved at the implementation level using CaesarJ family classes plus mixin composition.

3. **Domain engineering implementation**. Each component skeleton previously generated is completed with their corresponding business logic (Figure 4-1, label 3). This step completes the domain engineering level. As a result, the entire infrastructure for the automatic derivation of software products is obtained.

4. **Derivation of a specific architectural model**. This is the first step in our process for deriving specific products inside the SPL. First, a configuration of the

feature model, i.e. a valid selection of variants to be included in a specific product, is created. This configuration specifies what variants must be included in a specific product. Using this configuration, the architectural model of the desired product is automatically derived from the family model, using model transformations (Figure 4-1, label 4).

5. **Derivation of a specific implementation**. Based on the configuration created in the previous step and the architectural model generated from that configuration, the complete implementation of a specific product is automatically generated by code generation templates (Figure 4-1, label 5). This implementation uses the components created in step 3, which were partially generated in step 2.

Advanced mechanisms for separation of coarse grained variants enable the encapsulation of variable elements in separate units. By coarse-grained variant, we mean a variable feature of a Software Product Line, related to a coherent set of functionality, which implies the addition of new components/classes, or non-trivial modifications to a set of components/classes that are part of the Software Product Line architectural design or implementation. For instance, considering the Smart Home case study from Chapter 2, the automatic light management is considered a coarse-grained variant. This separation of coarse grained variabilities simplifies variability management and composition, therefore facilitating product derivation. Separation of variants is kept during all the process, both at the architectural design and at the implementation level. Moreover, MDD techniques automate part of this process, such as the generation of the implementation skeletons or the product derivation process, avoiding repetitive and tedious tasks to be performed manually.

TENTE, as already mentioned, is an innovative process for Feature-Oriented Model-Driven architecture design and implementation of Software Product Lines. TENTE combines advanced techniques for the separation of concerns, such as Feature-Oriented decomposition by means of *family polymorphism* plus *mixin composition* (Herrman, 2002; Aracic et al, 2006) with MDD techniques, both at the domain and application engineering levels. The use of advanced techniques for the separation of concerns facilitates the separation of reusable software assets through the software lifecycle, whereas Model-Driven techniques enable the automation of repetitive tasks of SPL engineering, such as the engineering of specific products.

Throughout all the process steps traceability information is gathered and stored in a traceability repository developed in the context of the AMPLE project (AMPLE D4.1,

2008, Anquetil et al, 2009). Each one of the steps of the TENTE process are described in detail in the next subsections as well as traceability information is gathered and stored.

4.2 Domain Engineering

This section describes the first part of a the Software Product Line engineering process, called Domain Engineering, which is the creation of the reusable software assets that will be used for the creation of specific products.

4.2.1 Architectural Design

This subsection describes the starting point of the TENTE process, which is the design of an architectural model for a complete family of products. This architectural model plays the role of reference architecture from which the software architecture of specific products will be derived. This architecture is modeled according to the techniques and methods developed in the context of the AMPLE project (AMPLE D2.2, 2007). Reference architecture in TENTE is comprised of three related models (see Figures 4-2 to 4-4):

1. A cardinality-based feature model (Czarnecki et al, 2005);
2. A UML 2.0 model (UML, 2005);
3. A VML (*Variability Modelling Language*) specification (Loughran et al, 2008; Sanchez et al, 2008).

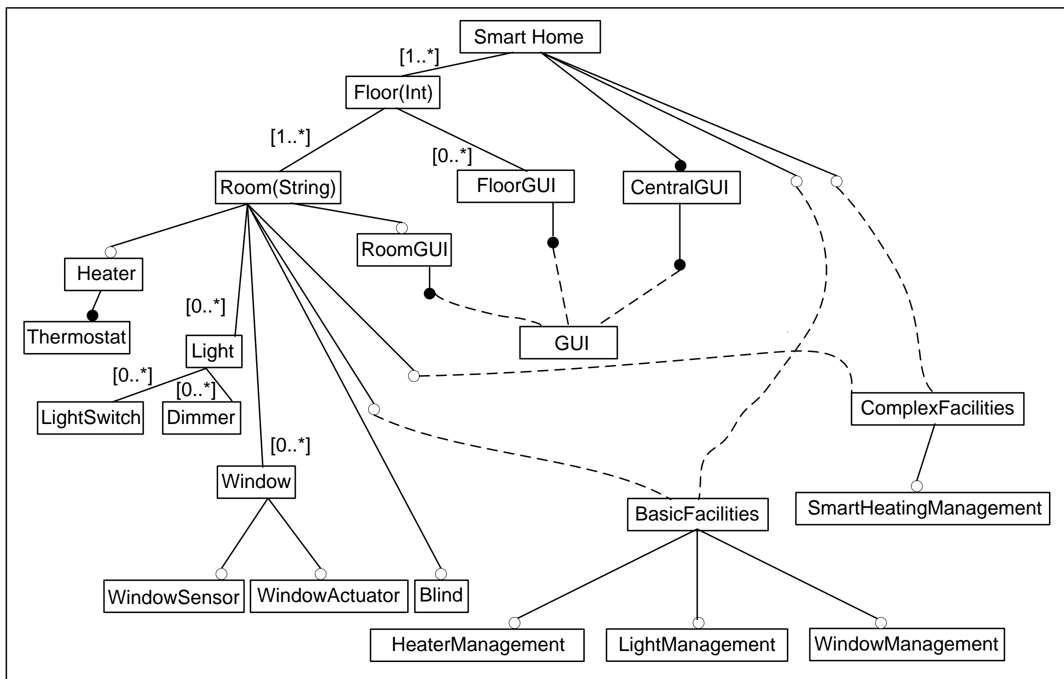


Figure 4-2 Simplified SmartHome Cardinality based Feature Model

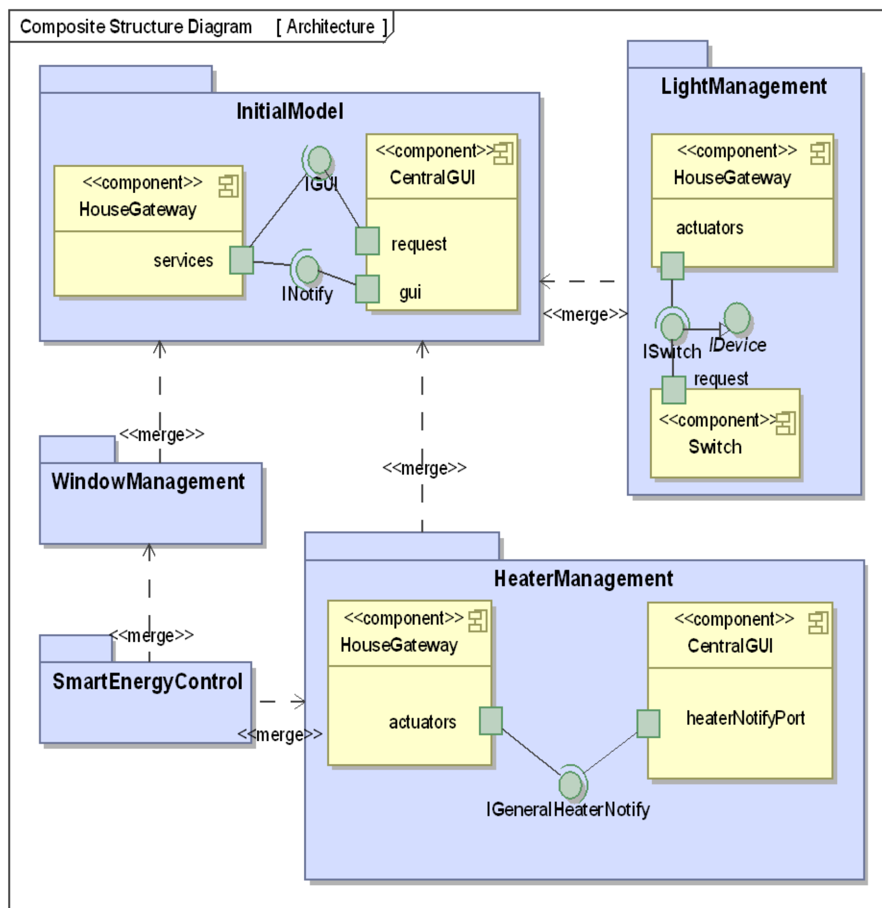


Figure 4-3 Simplified SmartHome Component View

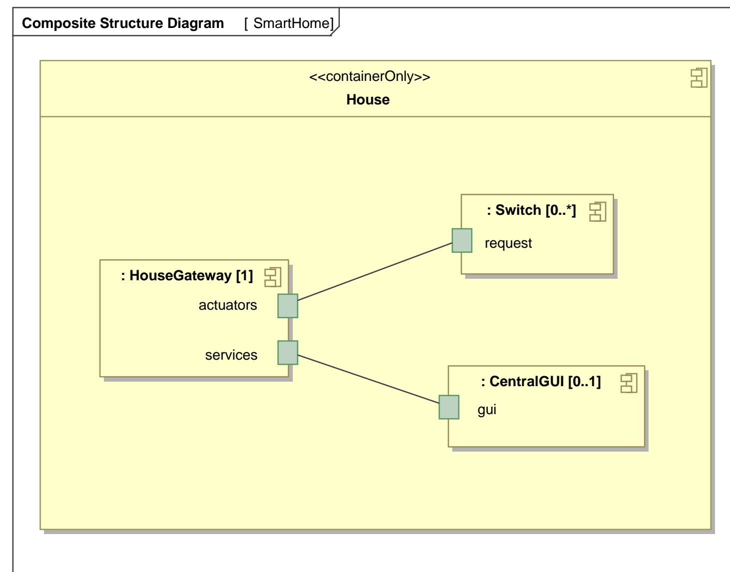


Figure 4-4 Simplified SmartHome Composite Structure View

The cardinality-based feature model specifies which parts of the architecture are variable and why they are variable. For instance, Figure 4-2 specifies a Smart Home can have several rooms. Each room can have a different number of optional facilities, such as automatic light (`LightManagement`), heater (`HeaterManagement`) or window (`WindowMnagement`) management. `SmartEnergyManagement` is an advance optional facility that ensures windows and heaters are managed coordinately in order to save energy. Therefore, `SmartEnergyManagement` is a feature that requires and extends the `HeaterManagement` and `WindowManagement` features. This feature model represents variability specification or *problem space*.

Then, a variable architecture that satisfies this specification is designed using UML 2.0 model (Figure 4-3 and 4-4). This model represents variability realization or *solution space*. The architectural design in UML 2.0 must incorporate mechanisms for making the architecture variable. Coarse-grained variations, such as the incorporation of automatic light management, are encapsulated at the design level into UML packages. These packages are composed by means of UML merge relationships, using the technique presented by Laguna et al (Laguna et al, 2007) and also adopted in the AMPLE project (AMPLE D2.2, 2007). Each UML package represents an architectural increment (Hendrickson and van der Hoek, 2007), which adds new components,

interfaces and so forth to an existing architecture, extending the architecture with new functionalities.

Software architectures are modeled using two views. Each package contains these two views, which represent the architectural design of an architectural increment or coarse-grained feature. These views are described below:

1. **Component View**. It specifies the decomposition of a software system into a set of interconnected component types. For instance, Figure 4-3 (package `InitialModel`) specifies the software architecture of a Smart Home has as component types `HouseGateway` and `CentralGUI`, amongst others, and that these component types are connected through the interfaces `IGUI` and `INotify`. This view is modeled using UML 2.0 component/class diagrams and it is typically constructed during domain engineering, for specifying which component types (e.g. `HouseGateway`) are available for constructing specific products.
2. **Composite Structure View**. It specifies how specific component instances are connected and composed. For instance, Figure 4-4 specifies that each instance of the `House` component has nested an instance of the `HouseGateway` component and that this instance is mandatory. It also specifies that this instance is connected to a variable number of `Switch` instances. This view is modeled using UML 2.0 composite structure diagrams. This view is partially specified at domain engineering and completed at application engineering, when the specific number of component instances to be included in a specific product is known, i.e. when structural variability is solved. For instance, depending on the specific number of lights to be included in a specific house, a different number of `Switch` instances would be created.

```
00 import features <"/SmartHome.fmp">;
01 import core <"/SmartHome.uml">;
02
03 variant SmartHome {
04 SELECT:
05    createPackage("MyHome");
06    merge("MyHome","InitialModel");
07 } //SmartHome
08
09 variant LightManagement {
10 SELECT:
11    merge("MyHome","LightManagement");
12 UNSELECT:
13    remove("LightManagement");
14 } //LightMng
15
16 variant WindowManagement {
17 SELECT:
18    merge("MyHome","WindowManagement");
19 UNSELECT:
20    remove("WindowManagement");
21 } // Celsius
```

Figure 4-5 VML specification of the Smart Home case study

Finally, a VML specification (Figure 4-5) links the feature model and the UML 2.0 architectural model. VML (Loughran et al, 2008, Sánchez et al, 2008) is an innovative language for facilitating variability management in architectural models. VML describes For each variant, which actions must be carried out if a variant is selected or unselected. Thus, the VML connects feature models and architectural models, specifying which effect has over architectural models the decisions adopted over a feature model. This is necessary as the mapping between decisions on feature models and their effects in architectural models is rarely a simple one to one relationship.

This specification determines how to obtain the architecture of a specific product given a configuration of the architectural feature model. This product derivation process is as follows: First of all, a new UML package representing the final product being derived is created. This package is called `MyHome` and it is initially empty. This empty package would merge those packages that correspond to selected features, e.g. `LightManagement`. The piece of code for creating this package (Figure 4-5, line 05) is placed into the SELECT clause of the SmartHome feature, which is the root feature of the architectural feature model depicted in Figure 4-2. In this case, this package merges the `InitialModel` package (Figure 4-5, line 06), which represents the minimum and core functionality that any SmartHome must have. Since this feature is always selected,

this piece of code is always executed and the `MyHome` package is always created and a merge relationship is initially created between this package and the InitialModel package.

When coarse-grained features encapsulated into UML packages, such as `LightManagement`, are selected (Figure 4-5, line 10-11), a new merge relationship is added between the package representing the final product, `MyHome`, and the package representing the coarse-grained feature (e.g. `LightManagement`). This means that the contents of this package will be included in the final product. In case a coarse-grained feature represented by a UML package is not selected, e.g. `LightManagement` (Figure 4-5, line 12-13), the corresponding package is removed. We could let this package be there, since if no merge relationship exists between this package and the package representing the whole product, i.e. `MyHome`, the contents of this coarse-grained feature will not be included in the final product anyway. Nevertheless, in order to reduce the size and complexity of the models, we have opted for removing it.

Fine-grained variations can also be managed through different VML4Arch operators. We refer the interested reader to Loughran et al (Lughran et al, 2008) for a more comprehensive list of such operators.

As it can be noticed, the separation of coarse-grained variants into separate UML packages improves feature traceability, since coarse-grained variations, such as automatic light or window management, is most likely to be encapsulated into a single UML package. This also simplifies VML specifications because large sequences of operators, representing the addition of new components with new functionalities to a base architecture, can be reduced to a simply merge between UML packages, simplifying variability management. We refer the interested reader on the benefits of this approach for modeling software architectures of SPL to a previous report (AMPLE D2.2, 2007), where these benefits are discussed in-depth.

4.2.2 Code Generation

This subsection describes how the previous architecture model at the domain engineering level is transformed into the skeleton for an implementation. The implementation platform chosen is CaesarJ (Aracic et al, 2006), which is Java-based Aspect-Oriented language that supports Feature-Oriented decompositions through *family polymorphism* plus *mixin composition* (Gasiunas and Aracic, 2007).

This subsection details the model transformation process for generating an implementation, in CaesarJ, corresponding to the architectural model, expressed in UML 2.0, defined at the domain engineering level.

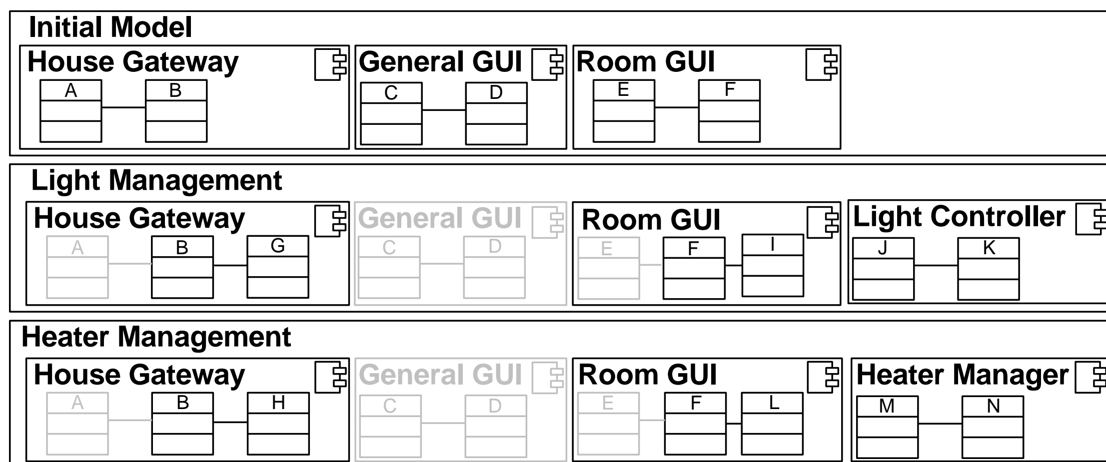


Figure 4-6 Two-level family classes schema

This transformation process generates two-levels of family classes, such as illustrated in Figure 4-6. Each architectural increment, represented by a UML package, generates a *first-level* family class. For instance, the `LightManagement` and `HeaterManagement` architectural increments would lead to the creation of two family classes (see Table 4-1). Each architectural increment is comprised of a set of interconnected architectural components. Each architectural component (e.g. `HouseGateway`) is implemented as a bundle of classes. Thus, each architectural component is implemented as a family class, where the classes that implement a component are virtual classes of that family class. This family class is nested in the family class corresponding to the package where the architectural component is nested. Thus, each component is a *second-level* family class contained inside a first-level family class.

This allows components can be refined in different architectural increments. For instance, the `LightManagement` architectural increment can refine the `HouseGateway` component defined in the `InitialModel` architectural increment, in order to add to the `HouseGateway` the functionality regarding light manipulation. Internal classes of architectural components can also be refined, taking advantage of the strong and advanced type system provided by CaesarJ. This two-level family classes schema also enables components that belong to a same architectural increment can be developed, tested and compiled independently from the rest of components of the family of products. The main benefit of making components independent one of each other is that is possible to change the business logic of one component without modifying the other components of the application.

The correspondence between architectural elements of UML 2.0 models and implementation artifacts on CaesarJ is outlined in Tables 4-1 to 4-2 and then explained more in-depth.

Architecture	Implementation	Rationale
Package	First-level family class	A package is mapped into a family class, which will contain the result of transforming the elements the package contains. A new Java package is also created for each architectural package in order to preserve the implementation files corresponding to different packages logically and physically separated and avoid name conflicts.
Merge relationship	Inheritance between first-level family classes.	A merge relationship between packages is mapped into an inheritance relationship between family classes. The merged package will act as the parent and the merging package as the child in the inheritance relationship. Merging of multiple packages is supported by CaesarJ, since CaesarJ allows multiple inheritance by means of propagating mixin composition.
Component	Second-level family class	Components are mapped into second-level family classes which are contained in the first-level family class that results of transforming the architectural package where the component is contained. A new Java package, contained in the Java package that results of transforming the architectural package where the component is contained, is also created in order to maintain the implementation files of the component physically and logically separated and to avoid name conflicts.
Component inner class	Virtual class contained in a second-level family class.	A component inner class is mapped into a virtual inner class, which is contained in the family class result of transforming the architectural component that contains the inner class. Component inner classes which serve as type of a port are not transformed following this rule.
Interface	Interface	An architectural interface is mapped to a common Java interface. CaesarJ family classes do not support the definition of interfaces inside them, thus, a generated Java interface is placed in the Java package corresponding to the transformation of the architectural package that contains this interface, but it is placed outside the family class corresponding to this architectural package.
Port	Inner virtual class contained in a second-level family class plus and attributed contained in that second-level family class	A port is an attribute of a component with a specific type, which is a component inner class. Thus, the inner class is transformed into a virtual inner class, which is contained in the second-level family class that results of transforming the architectural component that contains the inner class. The port itself is mapped into an attribute contained in the second-level family class that results of transforming the architectural component that contains the port. This attribute has as type the virtual inner class previously generated. In addition, a getter method is generated in the second-level family class that results of transforming the architectural component that contains the port for retrieving the attribute corresponding to the port.

Table 4-1 Correspondence between architectural elements in UML 2.0 and implementation artifacts in CaesarJ (1)

Architecture	Implementation	Rationale
Required relationship between a port and an interface	A list of interfaces, plus a connect method.	A required relationship between a port and an interface is transformed into an attribute which type is a list. The type of the elements of the list is the Java interface that corresponds to the transformation of the architectural interface that the port requires. Moreover, a `connect` method for connecting the implementation of two components through a port is generated.
Provided relationship between a port and an interface	Interface implementation	A provided relationship between a port and an interface is transformed into an implements relationship between the inner virtual class which corresponds to the transformation of the inner class that serves as type for the port and the Java interface result of transforming the architectural interface.
Attribute	Java attribute plus getters and setters	Attributes of architectural components and classes are mapped into a Java attribute plus the corresponding pair of getter and setter methods (depending upon attribute visibility and kind, e.g. read-only attributes do not generate a setter method). Attributes of architectural interfaces only generates the pair of getter and setter method, since Java does not allow the declaration of attributes in interfaces. An attribute is contained, in the implementation, in the result of transforming the architectural classifier that owns this attribute.
Method	Java method	Methods of architectural components, interfaces and classes are mapped into Java methods. An method is contained, in the implementation, in the result of transforming the architectural classifier that owns this method.

Table 4-2 Correspondence between architectural elements in UML 2.0 and implementation artifacts in CaesarJ (2)

Package to first-level family class

As already commented in Table 4-1, each architectural increment or coarse-grained reusable software asset, represented by a package in UML 2.0 is transformed into first-level family class in CaesarJ, with the same name as the architectural package (See Figure 4-7 and Table 4-1). This first-level family class will serve as container for the result of transforming the architectural elements contained in this package. Moreover, in order to keep the files corresponding to the implementation of an architectural increment logically and physically separated, a new Java package is created, with the same name as the architectural package.

Merge to inheritance between family classes

A merge relationship between two packages at the architectural level is transformed into an inheritance relationship between the first-level family classes corresponding to the result of transforming the packages participating in the merge relationship. The family class corresponding to the merging package will act as child family class and the family class corresponding to the merged package will be the parent family class.

Moreover, a Java package import clause is added to the beginning of each implementation file of the merging package, in order to make the elements of the parent family-class, corresponding to the merged package, visible by the implementation files of the merging package.

Figure 4-7 shows an excerpt of the architectural design of the Smart Home case study, in which only the packages representing architectural increments and relationships between these packages are illustrated. Table 4-3 shows the implementation artifacts generated when transforming the model depicted in Figure 4-7.

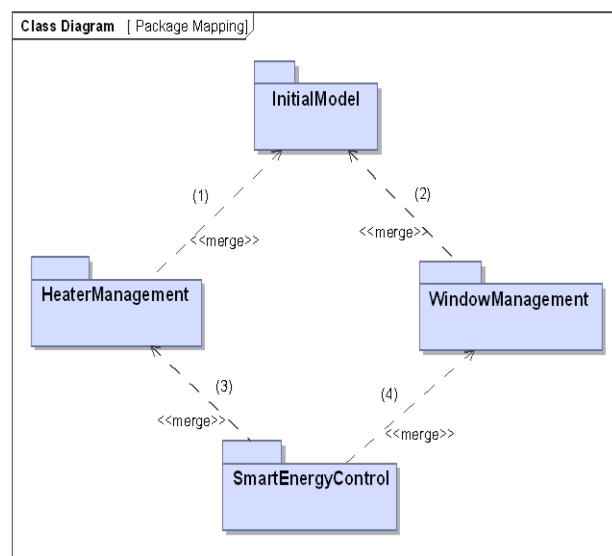


Figure 4-7 Package mapping, model example

Table 4-3, third row shows the result of transforming the `HeaterManagement` package. First, a family class with name `HeaterManagement` is created (Table 4-3, third row, line 03). The `HeaterManagement` package merges the `InitialModel` package, (Figure 4-7, label 1). Thus, firstly, a Java import clause is added at the

beginning of the implementation file (Table 4-3, third row, line 01), in order to make the implementation artefacts resulting of transforming the `InitialModel` package, visible in this implementation file. Then, we make that the `HeaterManagement` inherits from the `InitialModel`, extending it with new components and features (Table 4-3, third row, line 03). A similar strategy is applied to the `WindowManagement` package. In the case of the `SmartEnergyControl` package, this package merges two packages at the same time (Figure 4-7, labels 3 and 4). This means that the `SmartEnergyControl` family class resulting of transforming this architectural package must inherit from two family classes, i.e. from `HeaterManagement` and `WindowManagement` at the same time. Fortunately, CaesarJ supports multiple inheritance between family classes due to propagating mixin composition (Table 4-3, fifth row, lines 04 and 05)

Architectural Package	**Implementation code generated**

InitialModel	`00 package initialModel;`
	`01`
	`02 cclass InitialModel{`
	`03    …`
	`04 }`

HeaterManagement	`00 package HeaterManagement;`
	`01 import InitialModel.*;`
	`02`
	`03 cclass HeaterManagement extends InitialModel{`
	`04  …`
	`05 }`

WindowManagement	`00 package windowManagement;`
	`01 import InitialModel.*;`
	`02`
	`03 cclass WindowManagement extends InitialModel{`
	`04  …`
	`05 }`

SmartEnergyControl	`00 package smartEnergyControl;`
	`01 import HeaterManagement.*;`
	`02 import WindowManagement.*;`
	`03`
	`04 cclass SmartEnergyControl extends WindowManagement &`
	`05                               HeaterManagement{`
	`06   …`
	`07 }`

Table 4-3 Code generated for the architectural model depicted in Figure 4-7

Architectural component to second-level family class

An architectural component is transformed into a family class, which contains set of classes. These classes implement the functionality of the component. This family class is placed inside the family class that results of transforming the architectural package where the architectural component is contained. Thus, the family class resulting of transforming an architectural component is a virtual class of the family class corresponding to the architectural package that contains the component. Because of this reason, we call the family classes resulting of transforming components, *second-level family classes*. The use of family classes enables the implementation of a component declared in one package can be refined in successive family classes representing different architectural increments that extend the former one.

In addition, all components inherit from a core family class, called *Component*, which provides some infrastructure methods and functionality for component management. For instance, this infrastructure adds an `ID` attribute to all components in the implementation in order to be able to uniquely identify a component instance.

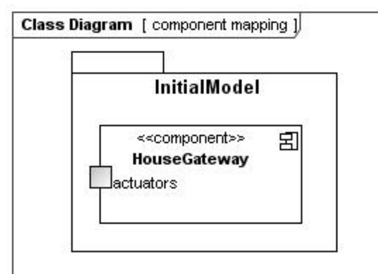


Figure 4-8 `HouseGateway` architectural component

Figures 4-8 and 4-9 shows the transformation of the `HouseGateway` component, which is included in the `InitialModel` package into implementation artifacts. The transformation of the port is not explained here, since it will be covered later, in the corresponding section. Using the package to first-level family class transformation rule, code of lines 00-02 is obtained. For the `HouseGateway` component a second-level family class is generated (line 04). As commented, this family class extends `Component`. Some infrastructure methods, such as standard constructors, are also generated.

```
00   package InitialModel;
01
02   public cclass InitialModel {
03     ...
04     public cclass HouseGateway extends Component{
05             ...
06     }
07   ...
08   }
```

Figure 4-9 Code generated when transforming the `HouseGateway` component

CaesarJ supports the definition of virtual classes of family classes in separate files. This feature has been demonstrated to be useful for separating the implementation of the different components contained in a same first-order family class in separate files, facilitating the management of these implementation files and the independent development of components.

Figure 4-10 shows this technique applied to the generation of the code for the `HouseGateway` component. In this case, an empty `InitialModel` family class, and a Java package with the same name, has already been created. In this case, line 00 of Figure 4-10 is used to declare that all classes and family classes defined in this file are virtual classes of the `InitialModel` family class, which is placed in the `InitialModel` Java package. In this way, the family class declared in line 03 is a virtual class, or *second-level family class*, nested in the `InitialModel` family class.

```
00   cclass InitialModel.InitialModel;
01
03   public cclass HouseGateway extends Component {
04          ...
05   }
```

Figure 4-10 Separation of implementation files for components

Component inner class to virtual class of a second-level family class

Component inner classes are implemented as inner virtual classes of the second-level family class which results of transforming the component that contains the class. This enables these inner classes can be refined in successive family classes extending this second-level family class. There is not any need of inheriting from any special infrastructure class, such as in the component case. A virtual class is declared inside the

same file that the second-level family class that contains this class. In addition, since components are managed as black-box entities, the generated virtual class is declared as protected. Classes that are types of ports are excluded of this rule, because of the reasons that will be exposed when commenting the *Port to inner class plus attribute* transformation rule.

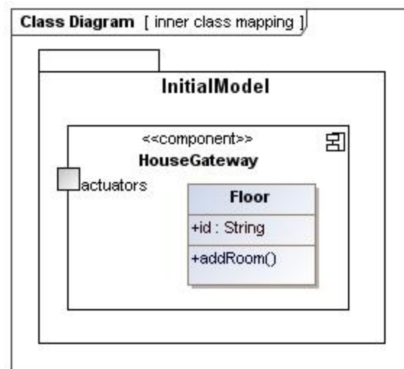


Figure 4-11 Floor inner class

Figure 4-11 shows an excerpt of the architectural design of the Smart Home case study, where an inner virtual class called `Floor` is added to the `HouseGateway` component of the `InitialModel` package. Figure 4-12 contains the code generated when transforming this architectural model. We suppose the first-level family class for the package and the second-level family class for the component has already been created as a result of applying the *Package to first-level family class* and *Architectural component to second-level* family class transformation rules. Then, a virtual class named `Floor` (Figure 4-12, line 05) is generated as result of transforming the `Floor` inner class. This virtual class is contained in the second-level family class corresponding to the `HouseGateway` architectural component. Lines 07-16 corresponds to the transformation of the `id` attribute, which is explained in the next subsection.

```
00   cclass InitialModel.InitialModel;
01
03    public cclass HouseGateway extends Component{
04           ...
05           protected cclass Floor {
07                  protected String id;
08                  public String getID(){
09                         return id;
10                  }
11                  public void setId(String value){
12                         id=value;
13                  }
14                  public void addRoom(){
15                  // TODO: add logic to this method
16                  }
17    }
```

Figure 4-12 Code generated when transforming the `Floor` inner class

Attribute to Attribute plus getter and setter methods

Each attribute in the architectural model is mapped into an attribute at the implementation level. This attribute will be contained in the classifier at the implementation level that results of transforming the classifier that contains the attribute at the architectural level. For instance, if the attribute is contained in a component, it would be placed in the second-level family class corresponding to the transformation of the component. If the attribute is placed in a component inner class, it would be placed in the virtual class corresponding to the transformation of the inner class.

Public attributes are converted to protected attributes in the implementation, and a pair of getter and setter methods, and their corresponding logic is generated. If the attribute is read-only, the setter method would be not generated. Protected and private attributes keep the same visibility as at the architectural level, but getters and setters methods are not generated.

Figure 4-12 lines 07-13 illustrate the code generated when transforming the `id` attribute of the `Floor` inner class, depicted in Figure 4-12.

UML method to Java method

A method at the architectural level is transformed into a method at the implementation level. The generated method will be contained in the classifier, at the implementation level, that results of transforming the classifier that contains the method at the architectural level. For instance, if the method is contained in a component, it

would be placed in the second-level family class corresponding to the transformation of the component. If the method is placed in a component inner class, it would be placed in the virtual class corresponding to the transformation of the inner class. The body of the method is left empty for being completed manually after generation.

Figure 4-12 lines 14-16 illustrate the code generated when transforming the `addRoom()` method of the `Floor` inner class, depicted in Figure 4-11.

UML Interface to Java Interface

Architectural interfaces are transformed into common Java interfaces. CaesarJ does not support the declaration of interfaces inside family classes. Therefore, an interface is placed in the Java package resulting of transforming the architectural package where the interface is contained, but outside the first-level family class corresponding to that package. A separate file is generated for each interface.

Figure 4-13 shows an example of the architectural design of the SmartHome case study, where an interface `INotify` is declared inside the `InitialModel` package. Figure 4-14 illustrated the code generated as a result of transforming this interface.

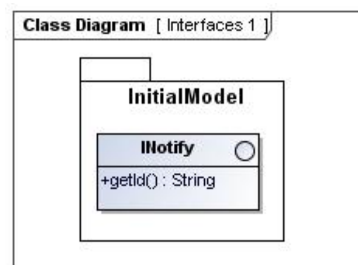


Figure 4-13 `INotify` interface

```
00   package initialModel;
01   public interface INotify{
02       public String getID();
03   }
```

Figure 4-14 Code generated when transforming the `INotify` interface

Interfaces are declared outside the CaesarJ family class hierarchy, so it is not possible to implement the merge relationships using inheritance between family classes,

as we did for components. Nevertheless, an interface declared in one package can be merged with an interface declared in other package by means of using inheritance between interfaces.

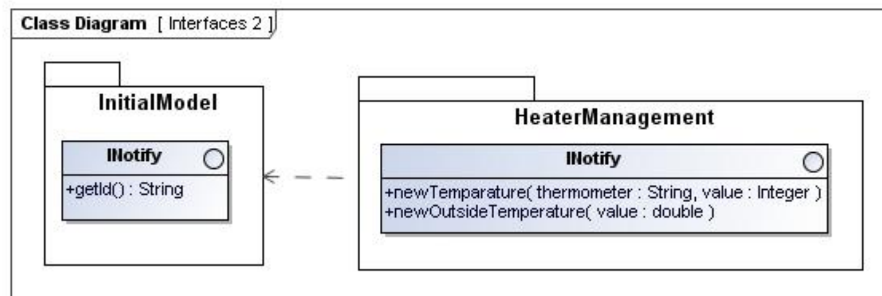


Figure 4-15 `INotify` interface declared in two different packages related by a merge.

Figure 4-15 illustrates an example of the architectural design for the Smart Home case study where an interface `INotify` is contained in the `InitialModel` and `HeaterManagement` packages. The `HeaterManagement` merges the `InitialModel` package, and the interface `INotify` of the `HeaterManagement` adds new methods to the `INotify` interface of the `InitialModel` package. The generated code result of transforming the `INotify` interface of the `HeaterManagement` package is depicted in Figure 4-16. Line 01 shows the declaration of the interface at the implementation level. The full name of the `INotify` interface is used in order to avoid name conflicts.

```
00   package HeaterManagement;
01   public interface INotify extends InitialModel.INotify {
02      public void newTemperature(String thermometer,int value);
03      public void newOutsideTemperature(double value);
04   }
```

Figure 4-16 Code generated when transforming the `INotify` interface of the `HeaterManagement` package

Hence, every time an architectural interface is mapped into implementation and the package that contains this interface merges other package, we must check if an interface with the same name exists in other packages which are reachable through the hierarchy of merge relationships. If one interface with the same name is reachable, an inheritance relationship between these interfaces must be created. The interface being mapped will

be the child interface and the reachable interface the parent interface in the inheritance relationship. If new reachable interfaces are discovered, we must check if this new reachable interface can be reached from an interface from which the interface being mapped inherits. If so, nothing happens, otherwise, a new inheritance relationship must be created between the interface being transformed and the new reachable interface. Again, the interface being mapped will be the child interface and the new reachable interface the parent interface in the inheritance relationship. Multiple inheritance between interfaces is allowed in Java, therefore, this technique is feasible.

Port to inner class plus attribute

An architectural port is comprised of: (1) a port declaration, which is like an attribute of the component that contains the port, plus (2) a class that provides the type for a port. This is due to the fact that ports in UML 2.0 can contain some behavior and, for instance, ports are able to filter and redirect messages arriving or leaving the port. Thus, the class that defines the type of the port is transformed into a virtual class that is contained in the second-level family class that results of transforming the architectural component that owns the port. The port itself is transformed into an attribute of the second-level family class that results of transforming the architectural component that owns the port. These attributes corresponding to ports are instantiated inside the constructor of the second-level family class which corresponds to the transformation of the component that contains the port. A getter method is generated by each attribute corresponding to a port.

The mapping of the classes that serves as type for a port differs from the mapping of common inner classes in that the generated classes must inherit from an infrastructure class, called `Port`, in the same way, generated second-family classes that correspond to components must inherit from `Component`. The infrastructure class `Port` provides functionality for the management of unique identifiers of ports, linking of ports to component instances and connections between ports of different components.

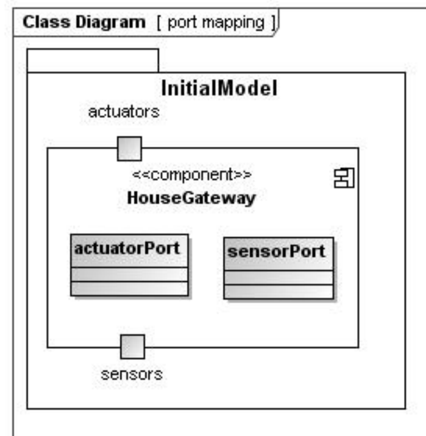


Figure 4-17 `HouseGateway` component with two ports

Figure 4-17 shows an example of the architectural design of the SmartHome case study, which contains a `HouseGateway` component with two ports, called `actuators` and `sensors`, which types are the classes `actuatorPort` and `sensorPort`, respectively. Figure 4-18 shows the code generated when transforming this architectural model. First, the classes that serve as type for the ports are transformed (Figure 4-18, lines 10-19). It should be noticed that these classes, unlike common inner classes, inherit from the `Port` class. Some infrastructure methods are also created. Then, two new attributes are generated inside the `HouseGateway` second-level family class, as a result of transforming the `actuators` and `sensors` ports themselves (Figure 4-18, lines 03-04). These attributes are instantiated in the constructor of the second-level family class result of transforming the architectural component (Figure 4-18, lines 05-09).

```
00 cclass InitialModel.InitialModel;
01 public cclass HouseGateway extends Component{
02     ...
03     protected ActuatorPort actuators;
04     protected SensorPort sensors;
05     public HouseGateway(String ID){
06            super(ID);
07            actuators = new ActuatorPort(this);
08            sensorts  = new SensorPort(this);
09     }
10     public cclass ActuatorPort extends Port{
11            public ActuatorPort(Component value){
12                   super(value);
13            }
14     }
15     public cclass SensorPort extends Port{
16            public SensorPort(Component value){
17                   super(value);
18            }
19     }
20 }
```

Figure 4-18 Code generated when transforming the `HouseGateway` component with two ports

Provided interface relationship to implements relationship

Provides relationships at the architectural level can be only established between a port and an interface. A *provides relationship* between a port and an interface at the architectural level is transformed into an `implements` relationship between the inner virtual class corresponding to the transformation of the class which is the type of the port and the interface generated as result of transforming the interface the port provides. The methods of the interface are automatically copied in the inner virtual class and the body of the method is left empty, for being completed manually after generation.

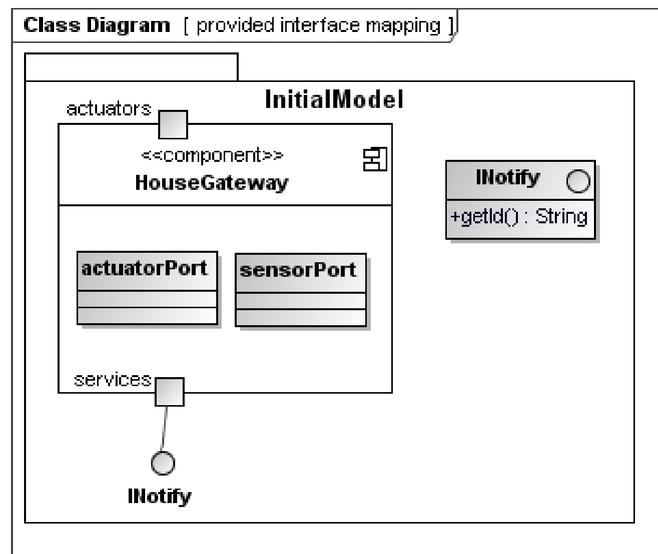


Figure 4-19 Provides relationship between the `services` port and the `INotify` interface

Figure 4-19 shows an example of the architectural design of the Smart Home case study. In this case, the `services` port provides the `INotify` interface. Figure 4-20 illustrates the code result of transforming this architectural model. Line 03 shows how the inner virtual class `SensortPort`, which correspond to the transformation of the inner class `sensorPort`, which is the type of the services port, *implements* the `INotify` interface. Lines 05-07 show the empty body for the unique method of this interface.

```
00 cclass initialModel.initialModel;
01 public cclass HouseGateway extends Component{
02     ...
03     public cclass SensorPort extends Port implements INotify{
04             ...
05             public String getId(){
06
07             }
08     }
09 }
```

Figure 4-20 Code generated when transforming the provides relationship between the `services` port and the `INotify` interface

Required interface relationship to list of interfaces plus a connect method

Required relationships at the architectural level can be only established between a port and an interface. A requires relationship at the architectural level means that a port can be connected with other port, which provides the interface the former port requires.

Thus, a *requires* relationship between a port and an interface is firstly transformed into an attribute with a list of the required interface as type. We are creating a list because a component instance can connect to a variable number of component instances through a port. This attribute is contained in the inner virtual class that corresponds to the transformation of the inner virtual class that is the type of the port. Thus, a port that requires an interface can store references to other ports providing that interface. Moreover, the code for initializing this attribute is added to the constructor of the inner virtual class which corresponds to the transformation of the inner class which is the type of the port. Finally, a `connect` method is generated. This method serves to connect the port which requires the interface with ports that provide that interface.

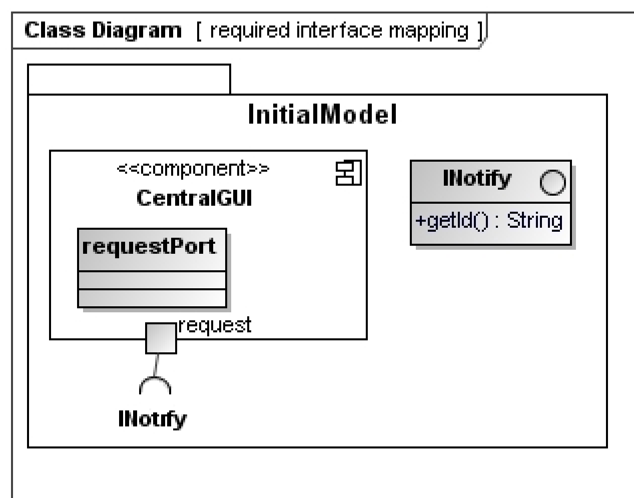


Figure 4-21 Requires relationship between the `request` port and the `INotify` interface

Figure 4-21 shows an excerpt of the architectural design of the SmartHome case study, where a request port, of type `requestPort`, requires an interface `INotify`. Figure 4-22 is the resulting code of transforming this architectural model. As explained, an attribute is created for holding the list of connected ports which provides the interface that the port requires (Figure 4-22, line 04). This list has a type of its elements the interface which is required by the port. The attribute is contained in the inner virtual class that corresponds to the transformation of the inner virtual class which is the type of the port. This list is a simple Java `ArrayList` because CaesarJ does not support Java generics currently. In addition, the code for initializing this list is added to the constructor of the inner virtual class where the list is contained (Figure 4-22, line 07). Finally, a `connect` method is generated (Figure 4-22, line 09-11). This method serves

for connecting this inner virtual class, which is the result of transforming an inner class which is the type of a port, with other inner virtual classes, which are result of transforming inner virtual class which are types of port that provides the `INotify` interface, i.e. with other inner virtual class of type Port, which, in addition, implements the `INotify` interface.

```
00 cclass InitialModel.InitialModel;
01 public cclass CentralGUI extends Component{
02     ...
03     public cclass RequestPort extends Port{
04             public ArrayList INotifyList;
05             public RequestPort(Component comp){
06                     super(comp);
07                     INotifyList=new ArrayList();
08             }
09             public void connect(INotify port){
10                     INotifyList.add(port);
11             }
12     }
13 }
```

Figure 4-22 Code generated when transforming the requires relationship between the `request` port and the `INotify` interface

A port can provide and require several interfaces at the same time. If a port provides several interfaces, the inner virtual class corresponding to the type of the port will implement several interfaces, which is allowed in Java. If a port requires several interfaces, the transformation rule would create several list of required interfaces and a connect method for each interface the port requires. An example of the connection between ports can be found in the subsection 4.5.

4.2.2 Component Implementation

Once the skeletons of an implementation corresponding to the architectural design of the SPL have been generated, these skeletons are completed manually in order to get a set of reusable components we can use to assemble specific applications at the application engineering level.

Some implementation level variabilities, such as supporting two different versions of a same API, could still be addressed using implementation-level techniques, such as conditional compilation. The reader interested on which kind of variability can be

solved at the architectural level and which kind of variability must be solved at the implementation level can read the AMPLE report (AMPLE D2.2, 2007). The reader interested on how some kind of variability can be solved at the implementation level using Aspect-Oriented techniques can read the AMPLE report (AMPLE D3.2, 2007) about this topic.

At the end of this step, a set of components implementing the family of products is obtained. We only need to appropriately instantiate and connect these components in order to obtain a specific product. This is addressed by the application engineering phase.

4.3 Application Engineering

At the domain engineering level, the entire infrastructure for the automatic construction of specific products is created. This section describes the application engineering level, i.e. the engineering of specific products as automatically as possible using the previously created infrastructure.

4.3.1 Configuration of a Specific Architecture

At the application engineering level, a specific product is configured by selecting those features that must be included in that product and instantiating and connecting components according to that selection of features. Thus, the application engineering phase starts with a valid selection of variants.

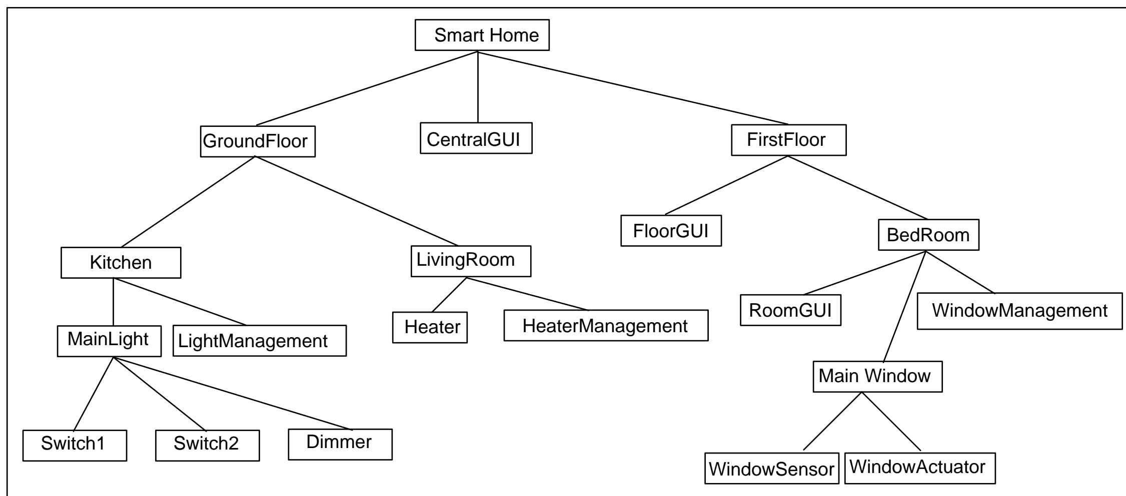


Figure 4-23 Final product configuration

For instance, for the Smart home case study, a customer could decide to buy a Smart Home with two floors, a ground floor and a first floor. The ground floor will have two automated rooms, a kitchen and a living room. The kitchen will have light management, controlling one light with two switches. The living room will have heater management with one heater being controlled. The first floor would have a dedicated GUI and an automated room, the bedroom. This room will also have a dedicated GUI and window management controlling the main window of the bedroom. These selections serve to configure the feature model of Figure 4-2, obtaining the configured feature model of Figure 4-23.

The selection of features can be done using directly a feature model, dedicated wizards, special tools for this purpose such as DecisionKing (Rabiser et al, 2007), or by means of defining a metamodel compliant with the cardinality-based feature model and transforming a model instance of this metamodel into a configuration of the cardinality-based feature model that express the architectural variability (Stephan and Antkiewicz, 2008).

With a feature configuration, such as depicted in Figure 4-23, an architectural model of the specific product is automatically generated from the architectural design of the family of products, defined at the domain engineering level. This generation process is carried out by executing the VML specification created on the domain engineering level (see Figure 4-6). A VML specification is compiled into a set of low-level model transformations, which implements the derivation of software architectures for specific products given a selection of variants (Sánchez et al, 2008). These model

transformations are expressed in a general purpose model transformation language. The execution of this automatically generates model transformations, with a valid configuration model as input, generates automatically the software architecture for the selection of variants specified by the configuration model. Thus, software architects can benefit from the automation provided by model transformation languages, but they do not need to learn any model transformation language, which is usually a non trivial task, since these model transformations are automatically generated as result of compiling a VML specification. A complete description of the VML implementation is beyond the scoped o this work. We refer the interested reader to Sánchez et al (Sánchez et al, 2008).

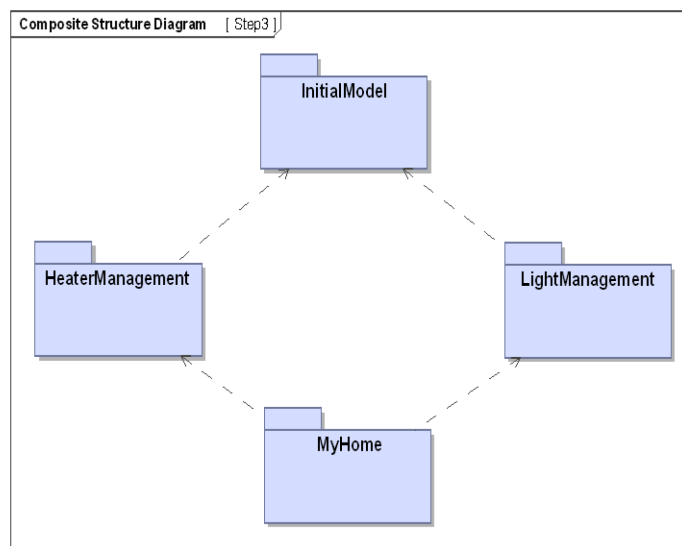


Figure 4-24 Package structure of the architectural model of a specific product.

As a result of executing VML with the previous configuration model and the reference architecture model as input, the specific architectural model depicted in Figures 4-24 and 4-25 is obtained. This architecture contains only those packages and components corresponding to selected features. For instance, the package corresponding to the `WindowManagement` option has been removed, since no `WindowManagement` has been selected. Finally, a new package `MyHouse`, representing the specific product, is created. This package is empty and inherits from all leaf packages in the package hierarchy, i.e. from all those packages that are not merged with other packages. The goal of this package is to combine all selected features. This package represents the complete product.

For each package, new component instances are also created and appropriately connected according to the user configuration. For instance, Figure 4-25 shows a composite structure diagram that specifies the structure of the house. New component instances are created, and appropriately connected, for the different GUIs to be placed along the house.

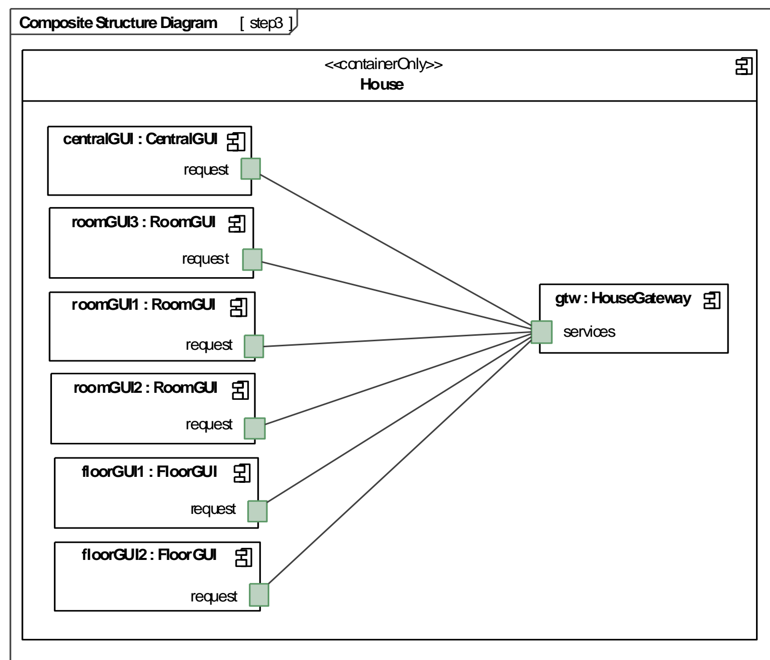


Figure 4-25 Application model composite structure diagram

At the end of this step we obtain the architecture of a specific product inside the product line. This architectural model serves as input for automatically obtaining a complete implementation of the product, which is explained in the next section.

4.3.2 Code Generation

Once the architectural model for a specific product has been obtained, this architectural model is used to generate, by means of model transformations, a complete implementation of this product, in CaesarJ. These transformations use and instantiate the CaesarJ family classes created at the domain engineering level. The task of these transformations is basically to instantiate, initialize and appropriately assemble the first-

level and second-level family classes, corresponding to architectural increments and components, according to the customer selection.

This transformation process is carried out following the correspondences between architectural elements, expressed in UML 2.0, and implementation elements, expressed in CaesarJ, is outlined in Table 3 and commented with more detail below.

Architecture	Implementation	Rationale
Leaf package	Final family class	The new architectural package added to the architectural model at the application engineering level, which is a leaf package, is transformed into a new first-level family class, which will serve to instantiate the final product. This generated first-level family class will inherit from the first-level family classes corresponding to the transformation of the architectural packages this leaf package merges.
Component Instance	Instantiated attribute in the final family class	Each component instance created in the architectural model at the application engineering level is transformed into an instantiated attribute, which is contained in the final family class. This attribute has as type the second-level family class corresponding to transformation of the architectural component, which is the type of the component instance.
Inner class Instance	Instantiated attribute in the final family class.	Each component inner class instance created in the architectural model at the application engineering level is transformed into an instantiated attribute, which is contained in the final family class. This attribute has as type the virtual inner class corresponding to transformation of the component inner class, which is the type of this instance.
Attribute initialization	Call to a setter method	Each initialization of an attribute of an instance, at the architectural level, is transformed into a call to the setter method for updating the corresponding attribute of the family-class result of transforming the architectural classifier that contains the attribute.
Component Instances connection	Call to a `connect` method	A connection between two component instances is transformed into a call to the connect method of the inner virtual class result of transforming the architectural port that requires the interface through these components are connected. As parameter of the `connect` method, the attribute corresponding to the transformation of the port that provides the interface through these components are connected is used.

Table 4-4 Correspondence between architectural elements and implementation artifacts, at the application engineering level.

Leaf package to final family class

The first step in the transformation process at the application engineering level consists on generating the code for a final family class. This final family class is the family class that will be used to instantiate the final product. This generated first-level family class will inherit from the first-level family classes corresponding to the

transformation of the architectural packages the leaf architectural package merges. Family classes not reachable from this final family class in the inheritance hierarchy are excluded of the compilation process.

Figure 4-26 illustrates part of the code generated as a result of applying this transformation rule to the architectural model of Figure 4-24. The code for creating a final family class is generated. This family class inherits from the family classes implementing coarse-grained reusable software assets selected by the customer, in this case, `LightManagement` and `HeaterManagement`.

```
00  cclass MyHome extends LightManagement & HeaterManagement {
01    ...
```

Figure 4-26 Code generated when transforming the leaf package of Figure 4-24

Component instance to instantiated attribute

In order to generate a final specific product we must also create all the component instances that will comprise that specific product. This is achieved by means of transforming all component instances, created in the specific architectural model at the application engineering, into instantiations of the second-level family classes which result of transforming the component which serves of type for the instances. These instantiations are placed in the final family class that represents the specific product.

Figure 4-27 shows the result of applying this rule to the architectural model of Figures 4-24 and 4-25. The code for appropriately instantiating the component instances that will be part of the final product is generated (Figure 4-27, lines 02-09). This also implies the generation of unique identifiers for each component instance.

```
00  cclass MyHome extends LightManagement & HeaterManagement {
01
02   // Component declaration and instantiation
03   HouseGateway gtw       = new HouseGateway("0");
04   CentralGUI centralGUI  = new CentralGUI("1");
05   FloorGUI floorGUI1     = new FloorGUI("2");
06   FloorGUI floorGUI2     = new FloorGUI("3");
07   RoomGUI roomGUI1       = new RoomGUI("4");
08   RoomGUI roomGUI2       = new RoomGUI("5");
09   RoomGUI roomGUI3       = new RoomGUI("6");
10
11   ...
12  }
```

Figure 4-27 Generated initialization code for a specific product

Inner class instance to instantiated attribute

In order to adequately instantiate a component, we must also create the inner classes that implement that component. This is achieved by means of transforming all instances of a component inner class, created in the specific architectural model at application engineering, into instantiations of the inner virtual classes which result of transforming the component inner classes which serves of type for these instances. These instantiations are placed in the final family class that represents the specific product. Virtual classes of second-level family classes cannot be created directly due to CaesarJ constraints on the type system. Therefore, these instances must be created calling to a `createInstance` method, generated for this purpose, in the second-order family classes. To ensure that everything is instantiated in the right order TENTE internally creates an instantiation sequence with all the components and classes to be instantiated and order it correctly to avoid inconsistencies.

Figure 4-28 shows the result of applying this rule to the architectural model corresponding to the configuration of Figure 4-25. The internal data structure of the `gtw` component instance is appropriately initialized. The methods `createXXXInstance` (lines 03-08) returns an instance of inner class of the component.

```
00 cclass MyHome extends LightManagement & HeaterManagement {
01     ...
02 // Inner classes definition and instantiation
03 HouseGateway.House houseData   = gtw.createHouseInstance("House");
04 HouseGateway.Floor groundFloor = gtw.createFloorInstance("groundFloor");
05 HouseGateway.Floor firstFloor  = gtw.createFloorInstance("firstFloor");
06 HouseGateway.Room  kitchen     = gtw.createRoomInstance("kitchen");
07 HouseGateway.Room  bedRoom     = gtw.createRoomInstance("bedRoom");
08 HouseGateway.Room  livingRoom  = gtw.createRoomInstance("livingRoom");
10 ...
11 }
```

Figure 4-28 Code generated when transforming the inner classes corresponding to the configuration of Figure 4-25

Attribute initialization to setter method call.

Each initialization of an attribute of an instance to a certain value, at the architectural level, is transformed into a call to a setter method. The method updates the corresponding attribute of the family-class, result of transforming the architectural classifier that contains the attribute.

Figure 4-28 shows the result of applying this rule to the architectural model corresponding to the configuration of Figure 4-24. Lines 03-06 set the floors and rooms identifiers for the components to know which part of the house controls each component.

```
00  cclass MyHome extends LightManagement & HeaterManagement {
01     ...
02  // Atribute initialization
03  groundFloor.setFloorId("groundFloor");
04  firstFloor.setFloorId("firstFloor");
05  livingRoom.setFloorId("groundFloor");
06  livingRoom.setRoomId("livingRoom");
08   ...
09  }
```

Figure 4-29 Code generated when transforming the attribute initialization corresponding to the configuration of Figure 4-24

Component Instances connection to call to a connect method

Finally, we must connect the generated component instances in order they can communicate among them. This is achieved by means of transforming the connections between component instances specified in the composite structure diagrams (Figure 4-24), into calls to connect methods. These methods are the `connect` method of the inner virtual class result of transforming the architectural port that requires the interface. As parameter of the `connect` method, the attribute corresponding to the transformation of the port that provides the interface through these components are connected, is used.

For instance, in Figure 4-29 it is specified that the `request` port of the `centralGUI` component instance must be connected to the `services` port of the `gtw` instance of the component `HouseGateway`. In this case, the `services` port is which requires an interface that is provided by the `request` port. Therefore the statement of Line 02 of Figure 4-29 is generated. It consist on a call to the `connect` method of the services port of the `gtw` instance with the `request` port of the `centralGUI` instance as parameter.

```
00  cclass MyHome extends LightManagement & HeaterManagement {
01      ...
02    //Components interfaces interconnection
03    gtw.getServicesPort().connect(centralGUI.getRequest());
04    gtw.getServicesPort().connect(floorGUI1.getRequest());
05    gtw.getServicesPort().connect(floorGUI2.getRequest());
06    gtw.getServicesPort().connect(roomGUI1.getRequest());
07    gtw.getServicesPort().connect(roomGUI2.getRequest());
08      ...
09  }
```

Figure 4-30 Code generated when transforming the port connection corresponding to the model of Figure 4-24

After generating this final family class and the instantiation of components, the final product is implemented and appropriately initialized. Nevertheless, some implementation-level variabilities, such as selecting between two versions of a same API might need still to be bound. For this task, tools such as *pure:variants* (Beuche, 2003) or *Gears* (Krueger, 2007) can help. This subject is beyond the scope of this work.

4.4 Traceability information gathering

The code generation steps described in previous sections generates traceability information that should be stored in a repository in order to maintain the links between the different artifacts of software development process. These links are useful for traceability tasks such as change impact analysis, orphan analysis or trace visualization (Anquetil et al, 2009).

As already commented, the model-to-text transformations explained in the previous sections are implemented in xPand, the model-to-text transformation language of openArchitectureWare. xPand is a template-based language. Traceability information is collected superimposing *aspectual templates* (that play the role of aspects) on the templates that implement the code generation steps of the TENTE approach. This traceability information is stored in ATF (AMPLE Traceability Repository) (AMPLE D4.1, 200, Anquetil et al, 2008), a traceability repository and framework, created in the context of the AMPLE project. Aspectual templates generate an XML file with the traceability information. This XML file is then processed by a special *plug-in* added to the ATF[5].

For each trace link between a source element of the architectural model and an implementation artifact, a new trace link is added to the XML file collecting traceability information. Each trace line contains the following information:

1. The metatype of the source artifact.
2. The qualified name, i.e. the complete name, including the path until the element, of the architectural artifact.
3. The kind of element of the CaesarJ language, e.g. a class or an attribute, which the target artifact is.
4. The full path to the file where the target artifact is placed.
5. The name of the target element.

[5] ATF only supports the input of traceability data through special Eclipse plugins called extractors. We have developed our own extractor for the TENTE approach.

```
00 «AROUND ComponentTemplate::componentTemplate FOR uml::Component»
01 «targetDef.proceed()»
02 «LET this.getNearestPackage().nestingPackage.name AS packageName»
03 «writeTraceabilityFile("Component",name,
   "Model."+packageName+".ComponentView."+name,
   "Component Virtual Class",name,""+packageName+"/"+name+".java")»
05 «ENDLET»
06 «ENDAROUND»
```

Figure 4-31 Gathering traceability information with aspectual templates

Figure 4-31 shows one of the aspectual templates for collecting traceability information. This aspectual template intercepts the execution of a common template and adds some behavior before and after the intercepted template execution. The execution of the intercepted template execution can even be skipped.

Specifically, the aspectual template depicted in Figure 4-31 intercepts the execution of the template for transforming architectural components into second-order family classes (line 00). This aspectual template, firstly, invokes the intercepted template as a result of executing the `proceed` action (line 01). Then, traceability information is collected (line 02) and stored in the corresponding XML file, using some helper functions (e.g. `writeTraceabilityFile`).

The use of aspectual templates allows the separation of the logic for gathering traceability information from the logic for transforming architectural models into implementation artifacts, avoiding the tangling and scattering of both ones. Thus, it is possible to modify the logic for gathering traceability information without updating the logic for code generation.

CHAPTER 5: Related Work

This section comments on related work related to the topic of this master thesis.

First, there are several commercial tools, such as *pure::variants* (Beuche, 2003) or *Gears* (Krueger, 2007), which target the automation of SPL product derivation processes. However, they focus on the implementation level, deriving code for a specific product from family or reference implementations. The architectural stage is not considered. These tools would need to be significantly extended for dealing with architectural models. These extensions would be not trivial as they are often based on XMI-based manipulations. Moreover, these tools could not be used for creating model transformations from architecture to implementation.

Ziadi and Jézéquel, (Ziadi and Jézéquel, 2006) and Czarnecki and Antkiewicz (Czarnecki and Antkiewicz, 2005) address the automation, by means of model transformations, of the derivation of architectural/detailed design models for specific products from models that represent the complete family of products. However, these works do not deal with the transformation of these models into an implementation and the separation of coarse-grained variants for facilitating variability management.

Laguna et al (Laguna et al, 2007) present a seamlessly process for SPL engineering where variants are separated in UML packages combined by *merge* operators at the architectural level. These UML packages are managed at the implementation level as separated projects, where classes are implemented as partial C# classes. Laguna et al (Laguna et al, 2007) use UML packages and UML merge as the unique mechanism for dealing with variability. We support, by using VML (Loughran et al, 2008), other variability mechanisms, such as selecting between different implementations of an interface.

Trujillo et al (Trujillo et al, 2007) presents a MDD process for *portlet* development. A portlet is a third-party component for the development of web applications. Trujillo et al creates a set of model transformations for automating the development of portlets. Similarly to us, Trujillo et al also generate the implementation skeletons of the portlet models. These skeletons are manually completed, adding the business logic to

the implementation of the methods. This MDD process is extended for the development of family or SPLs of portlets. Variable features or variants of a portlet are separated from the core part of the portlet. A portlet is mainly specified by XML documents. Variants are specified as refinements, or separate XML documents, which are later composed by *xak*, a tool for XML artifacts refinement and composition (Trujillo et al, 2006). Trujillo et al applies the previously developed model transformations to the variant, obtaining the transformation of the variant transformed for each modeling level. In order to synthesize a product, the lowest level expression of the selected variants is combined with the core. Thus, composition is carried out at the implementation level. Trujillo et al argue the same result should be obtained if variants were composed with the core at any modeling level, and then the result transformed into an implementation. This equation creates a mechanism for validating both models as model transformations. If two applications obtained after composing variants at different modeling levels are different, something is not right either in the constructed models or in the developed model transformations. Our approach is similar to the work of Trujillo et al, but not focused on web engineering. We use UML 2.0, a general purpose modeling language, which is supposed to cover a wider range of applications.

Our approach, compared to the work of Laguna et al (Laguna et al, 2007) and Trujillo et al (Trujillo et al, 2007), benefits of a more powerful type system provided by CaesarJ, which eases the management of dependencies between features, support direct feature instantiation and the polymorphic use of features, among other issues (Aracic et al 2006, Mezini and Ostermann, 2004).

CHAPTER 6: Conclusions and Future Work

This section summarizes this work, provides a critical discussion on the TENTE benefits and comments on future work.

6.1 Discussion

This master thesis has presented TENTE, a Feature-Oriented Model-Driven process for architectural design and implementation of Software Product Lines. After the introduction, Chapter 2 provided some background on the techniques, tools and technologies used throughout this work. Then, an analysis of different Aspect-Oriented and model-driven development mechanism for variability management is presented. Based on the results of this analysis, we defined TENTE, which was described in Chapter 4. Chapter 5 commented on related work. The main contributions of TENTE, as compared to state-of-art of Software Product Line engineering are discussed below.

Separation of coarse-grained variants at architectural design and implementation

Coarse-grained variants are separated both at the architectural and at the implementation levels. At the architectural level UML packages combined by means of merge relationships are used. At the implementation level, family classes plus mixin compositions are applied. The separation of variants is therefore kept through the architectural and implementation stages.

This Feature-Oriented decomposition allows an incremental development of the reference architecture and implementation. New features can be added on an existing architecture or implementation by simply adding a new package or family class and relating it with previously existing packages or family classes.

The encapsulation of coarse-grained variants in well-defined modularization units eases variability management. For instance, the selection a coarse-grained feature, such as `LightManagement` results on simple operation on the architectural model, such as the addition of a merge relationship between packages. Without this decomposition, we would need to add/remove all the components, and their relationships, related to `LightManagement` as a consequence of selecting this feature.

Moreover, this encapsulation also helps to make the dependencies between features more explicit, with ease dependency management. For instance, the selection of the `WindowManagement` and the `HeaterManagement` features as a consequence of the `SmartEnergetControl` feature in the `SmartEnergyControl` in the SmartHome case study is automatically enforced by the semantics of the merge relationships and the inheritance between family classes.

Thus, the encapsulation of the coarse-grained variants in packages and family classes contributes to a better modularization, and therefore, to a better evolution and maintenance.

Support for negative and positive variability

TENTE is able to deal with positive and negative variability at the same time. For positive variability, TENTE uses mainly uses UML packages and merge relationships at the modeling level. Features that cannot be adequately separated using UML packages and merge relationships are managed by the VML language (Loughran et al, 2008; Sánchez et al, 2008). VML is able to add, remove or modify fine-grained elements of domain engineering models.

Explicit models at domain engineering level

Models are used both at the domain and engineering level, contrarily to processes such as presented by Völter and Groher (Völter and Groher, 2007), where models are considered only at the application engineering level. This approach encodes most architecture information implicitly in a set of model transformations. Thus, software engineers need to extract information about the reference software architecture and implementation from the model transformation or code generation templates, which can

be a cumbersome task, in case software architects are not Model-Driven experts, which is the common case.

In TENTE, the reference architecture is an explicit model at the domain engineering level. This model represents the architecture of the family of products covered by the Software Product Line. The existence of this reference architecture enables tasks such as architectural reasoning, architecture assessment and evaluation. This task can be hardly made in the case of the approach proposed by Völter and Groher (Völter and Groher, 2007). The same argumentation can is applied to the reference implementation existing at the implementation level.

Automation of repetitive and error-prone tasks

Several parts of the process are automated by means of model transformations. The application engineering level and product derivation processes are fully automated. Furthermore, we also address the automation of part of the domain engineering level.

The transformation process at the domain engineering level automates the generation of implementation skeletons corresponding to an architectural design. This automation helps to save some development effort, since this generation process does not need to be performed manually. Using model transformations at the domain engineering level a 30%-70% of the implementation code of the Smart Home components was automatically generated. These model transformations also avoid unavoidable mistakes that might be introduced in a manual process. It also helps to ensure consistency because all architectural elements are transformed in the same way, following the same rules. Since these rules are interpreted by a computer, they cannot be misunderstood, as it might happen with traditional mapping processes, described textually through a set of tables and textual rules, which can be misinterpret by human actors. Moreover, since it is supposed these rules implements the best solution for transforming architectural elements into an implementation, this automation also helps to ensure quality.

TENTE automates completely application engineering, i.e. the development of specific products. For starting this process, no knowledge about modeling techniques, UML or even programming is required. It is enough with knowing how to create configurations. This process can be highly simplified by means of creating the proper user interfaces. This means no specialized professionals are required to generate final products of a SPL developed using TENTE.

No expertise on Model-Driven tools or languages is required

Code generators are domain-independent, being reusable for multiple Software Product Lines, contrarily to other approaches, such as Völter and Groher (Völter and Groher, 2007). Indeed, these code generators have been successfully applied to the development of the Sales Scenario Software Product Line, a case study provided by SAP in the context of the AMPLE project.

The reusability of the code generators, plus the use of VML for specifying product derivation processes at the architectural level, avoids software architects and developers need to learn general purpose transformation languages, such as xTend. Thus, software architects and developers can benefit from the automation provided by model transformation techniques at a low cost.

Therefore, neither a new metamodel nor a model transformations must be created for developing a new Software Product Line using the TENTE approach. Nevertheless, no knowledge about Model-Driven languages, such as model transformation languages or code generation templates, is required from software architects and product line engineers.

As counterpart, the domain-independence of the TENTE code generators make them not as optimal as code generators developed specifically for a certain Software Product Line or domain, where the knowledge about the domain can help to implement some code optimizations in the code generation templates.

Traceability information gathering and adaptability

Code generators of the TENTE approach collects traceability information for being stored in a traceability repository. Gathering of traceability information is made by means of aspectual templates. This separation enables the logic for traceability information gathering can be updated without modifying the templates for code generation. Aspectual templates intercept templates for code generation based on some kind of signatures. Due to this syntactical coupling between aspectual templates and templates for code generation, small changes in the templates for code generation might leave the aspectual templates obsolete. Thus, small changes in the base templates could lead to undesirable ripple effects in the aspectual templates.

Usability and industrial applicability

TENTE uses UML as modeling. UML is a standard with a noticeable industrial adoption and well-known by software designers. There is a wide range of mature tools available in the market to create and manage UML models. Many of them are currently used in industrial environments (e.g. IBM Rational or Telelogic TAU G2 (Baker et al, 2005). For feature modeling, there is also a number enough of available tools, such as pure::variants or fmp (Czarnecki et al, 2005b).

For VML specifications, the corresponding tools are provided by the AMPLE project (Sánchez et al, 2008). These tools comprise the VML editor and the VML compiler. These tools have been successfully applied to the SmartHome and Sales Scenario case studies, provided by SAP and Siemens respectively.

CaesarJ is an innovative language, which supports family classes and mixin composition. Nevertheless, the compiler generates plain Java code as output. This means the products developed using TENTE and CaesarJ can be run in any computer with the Java Virtual Machine installed on it. Since Java is one of the most spread languages nowadays, and the generated code made no use of the most complex parts of the Java API, the code can run either in a standard computer as well as a mobile or an embedded device.

On the other hand, the CaesarJ compiles needs to introduce some extra code in the result of the compilation. This extra code might result in certain overhead sometimes, which decreases performance. Moreover, this current extra code is not compatible with technologies such as J2EE or Java Beans. CaesarJ also presents some shortcomings related to the remote deployment of components, although a promising Caesar RMI compiler already exists.

Applicability

We have selected cardinality-based feature models (Czarnecki et al, 2005) for specifying variability in *the problem space*. This notation is easy to understand, concise and allows the creation of tools to simplify the selection between variants for any kind of SPLs. Nevertheless, feature modeling has a wide number of detractors, which argues not all kind of variability can be captured in feature models and consider arbitrary metamodels and DSLs as a more suitable alternative. However, we have not found any

case during the development of the Smart Home and the Sales Scenario case studies that demonstrate that feature models are not able to capture some kind of variability.

The TENTE approach is currently tailored to UML as architectural design language. This means that expressivity of TENTE for architectural design is tailored to expressivity of UML for architectural design. If the reference architecture of a SPL can be successfully modeled using UML 2.0, the TENTE approach will work well, otherwise, it will fail. Certain specialized Software Product Lines, such as embedded systems with hard real-time constraints, might have difficulties related to expressiveness of the UML.

The TENTE approach imposes some constraints about how architectural models must be designed, which must be followed by software architects designing the software architecture. For example components are compulsory connected through ports. These ports have to be modeled as classes inside the components. Other restriction is that two ports cannot be connected two times through different interfaces because this creates conflicts in the code generation.

Finally, we would like to point out that the transformation rules described in this work, and used by the TENTE approach, are independent of the CaesarJ language. They can also be used to implement code generators for other programming languages. The unique requirement is that the target must support family classes plus mixin composition. A language with these characteristics is, for instance, ObjectTeams (Heerman, 2002). Nevertheless, there are not too much languages of this kind currently available.

6.2 Evaluation

Using the TENTE approach, a Smart Home SPL case study, provided by Siemens in the context of the AMPLE project, has been fully developed, including: the architectural design; (2) the generation of the code skeletons for the reference implementation; (3) the manual implementation of the logic of the components for the reference implementation; (4) the creation of configuration models for three different products; (5) the derivation the specific software architectures for these three products; (6) the generation of the complete implementation of these specific products; and (7) the testing of the three generated products.

Then, TENTE was applied to the Sales Scenario case study, provided by SAP in the context of the AMPLE project. The main purpose of the Sales Scenario is the holistic management of business data related to Sales processes, including central storage and access controlled retrieval. A full description of the case study can be found in Morganho et al (AMPLE D5.2, 2007).

For the Sales Scenario case study, the manual implementation of the logic for the reference implementation was skipped, since this would only serve to demonstrate we have skills enough about Java programming, but nothing interesting about TENTE. The architectural design, the generation of the component skeletons, the derivation of software architectures for specific products and the generation of implementation code for specific products was enough to test that TENTE could be successfully applied to different SPLs. For the Sales Scenario, a feature model, an UML 2.0 reference architecture, a VML specification and several configurations were created. The code generators required being updated only for fixing minor bugs, not detected in the Smart Home case study. So, these code generators were successfully applied to a domain different from the SmartHome.

Nevertheless, during the development of the Sales Scenario case study, we found that the extension of the TENTE code generator might be a desirable in order to consider new elements of the UML metamodel that were not used for the Smart Home case study, such as import relationships between packages. These relationships enables that a package can use the contents of the other package, but without extending them, as it would happen with the merge relationship. The semantics of the merge relationships include implicitly an import, but in the case of the merge, we can not avoid designers extend a model element from the merged package unintentionally. This can be prevented by means of using import relationships.

6.3 Future work

TENTE code generators work currently only with component, class and composite structure diagrams, but UML offers a wider range of diagrams, which also provide useful information. It is out intention to add more UML diagrams to the architectural design and develop the corresponding code generators. We have in our plans to incorporate state machines for describing protocol state machines, deployment diagrams and sequence diagrams, which would serve to specify test cases and they will be transformed into JUnit code. The code generator for deployment diagrams will make use of the CaesarJ RMI compiler, as the new features related to remote deployment that might emerge in the near future.

At the current moment, traceability information is simply generated, but not used at all. Moreover, the information generated is very trivial. It is our intention to investigate as future work what kind of information must be generated for more interesting purposes than for the sake of generating trace information, such as, for instance, estimating the cost of a design change. We would also like to add features to the TENTE approach, such as highlight regions of code that correspond to a certain model element, which should be easily implementable using the right traceability information.

We would also like to integrate requirements engineering techniques for Software Product Line engineering, in order to cover all the stages of the software lifecycle. This work is currently being done in the context of the AMPLE project.

The creation of configuration models from feature models can be simplified if the end-user is assisted with the appropriate user interfaces. These user interfaces can vary from simple forms or wizards to very complex graphical user interfaces, such as a graphical tool for designing house layouts. It is our intention to investigate how part of these tools can be more easily developed with the help of Model-Driven techniques.

The template code generation language used for the development of the TENTE approach is xPand, the model-to-text transformation language of the openArchitectureWare model-driven suite. Nevertheless, we have found xPand, and openArchitectureWare, few usable, with important shortcomings, such as code regeneration, preservation of manually introduced code, protected regions and

important and recurrent bugs when applied to the UML metamodel[6], and a serious lack of readable documentation. These shortcomings make almost impossible round-trip engineering, which hampers scalability, maintenance and evolution. The selection of xPand was mainly due to business alliances of the AMPLE project, rather than sound technical reasons. Therefore, we would also like to reimplement the code generators in a more robust code generation language, such as MOFScript (Oldevik et al, 2005).

[6] For instance, setter functions related to attributes of UML metaclasses does often not work. The "name" property and its associated "setName" function is a clear exemplar of this bug.

References

(Alférez et al, 2008)

M. Alferez, U. Kulesza, A. Sousa, J. Santos, A. Moreira, J. Araujo, V. Amaral, "A Model-Driven Approach for Software Product Lines Requirements Engineering", Proc. of the 20th Int. Conference on Software Engineering and Knowledge Engineering (SEKE), Redwood City (San Francisco Bay, USA), July 2008.

(AMPLE D2.2, 2007)

P. Sánchez, N. Gámez, L. Fuentes, N. Loughran and Alessandro Garcia. "*A Metamodel for Designing Software Architectures of Aspect-Oriented Software Product Lines*". AMPLE Project Deliverable D2.2, September 2007.

(AMPLE D2.4, 2008)

C. Nebrera, P. Sánchez, L. Fuentes, C. Schwanninger, L. Fiege, M.Jäger. *"Description of model transformations from architecture to design*".AMPLE Project Deliverable D2.2, September 2008.

(AMPLE D3.1, 2007)

V. Gasiunas, P. Sánchez, C. Nebrera, N. Gámez, L. Fuentes, J. Noyé, M. Südholdt, A. Núñez, C. Pohl, A. Rummler, I. Groher, C. Schwanninger and M. Völter. "*Overview of Extensions/Improvements to Existing Implementation Technologies*". AMPLE Project Deliverable D3.1, December 2007.

(AMPLE D3.2, 2007)

C. Nebrera, P. Sánchez, N. Gámez, L. Fuentes. "*Evaluation of existing AOP and MDD technologies as compared to other implementation technologies for variability management with respect to the requirements of SPLs*". AMPLE Project Deliverable D3.2, December 2007.

(AMPLE D4.1, 2008)

I. Galvão, S. Shakil Khan, J. Noppen, J.C. Royer, A. Rummler, P. Sánchez, R. Mitschke, N. Anquetil, B. Grammel, H. Morganho, C. Pohl, C. Schwanninger, L. Fuentes, A. Rashid and A. Garcia. "*Definition of a traceability framework (including the metamodel and the modelling of processes and artefacts to allow traceability in the presence of uncertainty) for SPLs*". AMPLE Project Deliverable D4.1, December 2007.

(AMPLE D5.2, 2008)

H. Morganho, C.Gomes, J. P.Pimentão, R. Ribeiro, B. Grammel, C. Pohl, A. Rummler, C. Schwanninger, L Fiege, M. Jaeger. "*Requirement specifications for industrial case studies*" AMPLE Project Deliverable D5.1, March 2008.

(Anquetil et al, 2009)

N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J. C. Royer, A. Rummler, A. Sousa. "*A Model-Driven Traceability Framework for Software Product Lines*". Journal on Software and Systems Modeling (SoSym). Accepted for publication.

(Aracic et al. 2006)

I. Aracic, V. Gasiunas, M., Mezini and K. Ostermann. *"An Overview of CaesarJ"*. Transactions on Aspect-Oriented Software Development, A. Rashid and M. Aksit (Eds), LNCS 3880:135-173, February 2006.

(Baker et al, 2005)

P. Baker, S. Loh and F. Weil. "*Model-Driven Engineering in a Large Industrial Context - Motorola Case Study*". Proc. of the 8th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS), L. C. Briand and C. Williams (Eds.), LNCS 3713: 476-491, Montego Bay (Jamaica), October 2005.

(Bayer et al, 2006)

J. Bayer, S. Gerard, Ø. Haugen, J. Mansell, B. Møller-Pendersen, J. Oldevik, P. Tessier, J. P. Thibault and T. Widen. "*Consolidated Product Line Variability Modelling*". In: T. Käkölä and J. C. Dueñas (Eds). Software Product Lines: Research Issues in Engineering and Management: 195-241. Springer, October 2006.

(Beuche, 2003)

D. Beuche. "*Variant management with pure::variants*". Technical report, pure-systems GmbH, 2003.

(Beydeda et al, 2005)

S. Beydeda, M. Book and V. Gruhn, "*Model-driven software development*", Springer, September 2005

(Budinsky et al, 2003)

F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. "*Eclipse Modeling Framework*", Addison-Wesley, July 2003.

(Czarnecki et al, 2005)

K. Czarnecki , S. Helsen and U. W. Eisenecker. "*Staged Configuration through Specialization and Multilevel Configuration of Feature Models*". Software Process: Improvement and Practice 10(2):143-169, January-March 2005.

(Czarnecki et al, 2005b)

K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, K. Pietroszek. "*fmp and fmp2rsm: Eclipse plug-ins for Modeling Features using Model Templates*". Proc. of the Companion to the 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA): 200-201, San Francisco (California, USA), October 2005.

(Deelstra et al, 2006)

S. Deelstra, M. Sinnema and J. Bosch. "*Product Derivation in Software Product Families: a Case Study*". Journal of Systems and Software 74(2): 173-194, January 2005.

(Gamma et all, 1995)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). "*Design Patterns - Elements of Reusable Object-Oriented Software*." Addison-Wesley.

(Gasiunas and Aracic, 2007)

Vaidas Gasiunas and Ivica Aracic. “*Dungeon: A Case Study of Feature-Oriented Programming with Virtual Classes*”. 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE), 6th Int. Conference on Generative Programming and Component Engineering (GPCE), Salzburg (Austria), October 2007.

(Groher and Völter, 2007)

I. Groher and M. Völter. “*XWeave: Models and Aspects in Concert*”. Proc. of the 10th Int. Workshop on Aspect-Oriented Modeling (AOM), 6th Int. Conference on Aspect-Oriented Software Development (AOSD), Vancouver (British Columbia, Canada), March 2007.

(Ernst, 1999)

E. Ernst. *“gbeta - A Language with Virtual attributes, Block Structure, and Propagating, Dynamic Inheritance”*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

(Fuentes and Sánchez, 2005)

L. Fuentes and P. Sánchez. “*AO Approaches for Component Adaptation”*. Proc. of the 2nd Workshop on Coordination and Adaption Techniques (WCAT), 19th European Conference on Object Oriented Programming (ECOOP), 79-86, Glasgow (Scotland), July 2005.

(Fuentes and Sánchez, 2007)

Lidia Fuentes and Pablo Sánchez. "Aspect-Oriented Coordination". Proc. of the 3rd Int. Workshop on Coordination and Adaption Techniques for Software Entities (WCAT), Electronic Notes in Theoretical Computer Science (ENTCS) 189:87-103, July 2007. Elsevier.

(Fuentes et al, 2009)

L. Fuentes, N. Gámez and P. Sánchez. "*Managing Variability of Ambient Intelligence Middleware*". Int. Journal of Ambient Computing and Intelligence (IJACI). 1(1):64-74, January-March 2009.

(Haugen et al, 2005)

O. Haugen, B. Moller-Pedersen and J. Oldevik. "*Comparison of System Family Modeling Approaches*". Proc. of the 9th Int. Software Product Line Conference (SPLC): J. H. Obbink and K. Pohl (Eds.): LNCS 3714:102-112, Rennes (France), September 2005.

(Hallsteinsen et al, 2006)

S. Hallsteinsen , G. Schouten, G. Jan Boot, T. Erlen. *"Dealing with Architectural Variation in Product Populations",* In: T. Käkölä, , J. C. Dueñas. Software Product Lines: Research Issues in Engineering and Management*:*245-273*,* Springer, October 2006.

(Herrman, 2002)

S. Herrmann. "*Object Teams: Improving Modularity for Crosscutting Collaborations*". Revised Papers of the 3rd Int. Conference NetObjectDays (NODe), M. Aksit, M. Mezini and R. Unland (Eds), LNCS 2591:248-264, Erfurt (Germany), October 2002.

(Kaköla and Dueñas, 2006)

T. Kakola and J. C. Dueñas. *"Software Product Lines: Research Issues in Engineering and Management"*. Springer-Verlag Berlin, 2006

(Kiczales et al, 1997)

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, .J.M. Loingtier, J. Irwin. "*Aspect-Oriented Programming*". Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP), M. Aksit and S. Matsuoka (Eds.), LNCS 1241: 220-242, Jyväskylä (Finland), June 1993.

(Kiczales et al, 2001)

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. "*An Overview of AspectJ*". Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP), J. L. Knudsen (Ed.), LNCS 2072:327-353, Budapest (Hungary), June 2001.

(Krueger, 2007)

C. W. Krueger. "*Biglever Software Gears and the 3-tiered SPL Methodology*". Proc. of the Companion to the 22nd Int. Conference on Object-Oriented Programming Systems and Applications (OOPSLA): 844–845, Montreal, (Quebec, Canada), October 2007.

(Laguna et al, 2007)

M. A. Laguna "*Seamless Development of Software Product Lines*". Proc. of the 6th Int. Conference on Generative Programming and Component Engineering (GPCE): 85-94, Salzburg (Austria), October 2007.

(Loughran et al, 2008)

N. Loughran, P. Sánchez, A. Garcia and L. Fuentes. "*Language Support for Managing Variability in Architectural Models"* Proc. of the 7th Int. Symposium on Software Composition (SC), C. Pautasso and É. Tanter (Eds), LNCS 4954:36-51, Budapest (Hungary), April 2008.

(Madsen and Moller, 1989)

O. L. Madsen and B. Moller-Pedersen. "*Virtual classes: A powerful mechanism in object-oriented programming"*. Proc. of the 4th Int. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA):397–406, New Orleans, (Louisiana, USA), October 1989.

(Meyer, 2001)

B. Meyer, "*Concurrent Object-Oriented Programming",* Proc. of the 38th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS): 203, Zurich (Switzerland), March 2001.

(Mezini and Ostermann, 2004)

M. Mezini and K. Ostermann. "*Variability management with feature-oriented programming and aspects*", Proc. of the 12th Int. Symposium on Foundations of Software Engineering (FSE):127-136, Newport Beach (California, USA), October-November 2004.

(Noopen et al, 2009)

J. Noppen, P. van den Broek, N. Weston, A. Rashid. "*Modelling Imperfect Product Line Requirements with Fuzzy Feature Diagrams*". Proc. of the 3rd Int. Workshop on Variability Modeling of Software-Intensive Systems (VaMOS): 93-102, Seville (Spain), January 2009.

(Oldevik et al, 2005)

J. Oldevik, T. Neple, R., Grønmo, J. Ø, Aagedal and A. J. Berre. "*Toward Standardised Model to Text Transformations*". Proc. of the 1st European Conference on Model Driven Architecture (ECMDA), A. Hartman and D. Kreische (Eds), LNCS: 3748: 239-253, Nuremberg (Germany), November 2005.

(Pastor and Molina, 2007)

O. Pastor and J. C. Molina. "*Model-driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*". Springer, July 2007.

(Pohl et al, 2005)

K. Pohl, G. Böckle and F. van der Linden, "*Software Product Line Engineering - Foundations, Principles, and Techniques*", Springer, September 2005.

(Prehofer, 2001)

C. Prehofer. "*Feature-Oriented Programming: A NewWay of Object Composition*". Concurrency and Computation: Practice and Experience 13(6): 465-501, May 2001.

(Rabiser et al, 2007)

R. Rabiser, P. Grünbacher and D., Dhungana. "*Supporting Product Derivation by Adapting and Augmenting Variability Models*". In Proc. of the 11th Int. Conference on Software Product Lines (SPLC): 141-150, Kyoto (Japan), September 2007.

(Sánchez et al, 2007)

P. Sánchez, L. Fuentes, A. Jackson and S. Clarke. "*Aspects at the right time*". Transactions on Aspect-Oriented Software Development IV (TAOSD), Special issue on Early Aspects, Awais Rashid, Mehmet Aksit, João Araújo and Elissa Banassiad (Eds), LNCS 4640:54–113, November 2007.

(Sánchez et al, 2008)

P. Sánchez, N. Loughran, L. Fuentes and A. Garcia. "*Engineering Languages for Specifying Product-derivation Processes in Software Product Lines"* Proc. of the 1st Int. Conference on Software Language Engineering (SLE), D. Gašević , R. Lämmel, and E. Van Wyk (Eds). LNCS 5452: 188-297 Toulouse (France), September 2008.

(Santos et al, 2008)

A. L. Santos, K. Koskimies and A. Lopes. "*Automated Domain-Specific Modelling Languages for Generating Framework-Based Applications*". Proc. of the 12th Int. Software Product Line Conference (SPLC): 149-158, Limerick (Ireland), October 2008.

(Stephan and Antkiewicz, 2008)

M. Stephan and M. Antkiewicz. "*Ecore.fmp: A tool for editing and instantiating class models as feature models*". ECE, University of Waterloo, Technical Report #2008-08, May 2008.

(Trujillo et al, 2007)

S. Trujillo, D. Batory and O. Díaz. "*Feature-Oriented Model Driven Development: A Case Study for Portlets*". Proc. of the 29th Int. Conference on Software Engineering (ICSE): 44-53, Minneapolis, (Minnesota, USA), May 2007

(UML, 2005)

Object Management Group (OMG), "Unified Modeling Language: Superstructure version 2.0 (formal/05-07-04)", August 2005.

(Völter and Groher, 2007)

M. Völter and I. Groher. "*Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*". Proc. of the 11th Int. Software Product Line Conference (SPLC): 233-242, Kyoto (Japan), September 2007.

(Ziadi and Jézéquel, 2006)

T. Ziadi and J. M. Jézéquel. "*Software Product Line Engineering with the UML: Deriving Products*". In: T. Käkölä, , J. C. Dueñas. Software Product Lines: Research Issues in Engineering and Management, 557-588, Springer, October 2006.

APENDIX A. TENTE Plug-in User Manual

This first appendix explains the basic concepts to start using TENTE. Firstly, how to install and uninstall the TENTE Eclipse Plug-in is commented. Then, how to generate code skeletons is described. Finally, how to generate complete products at application engineering level is explained. The models of the Smart Home Case Study will be used in the examples.

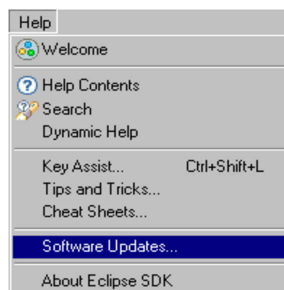
A.1 Install and Uninstall the TENTE Eclipse plug-in

The TENTE code generators are provided as en Eclipse plug-in. Therefore, the first step to use is to get an Eclipse installation. Eclipse can be downloaded the program from the web page http://www.eclipse.org/downloads/. After downloading it, uncompress it inside the desired folder. Eclipse version 3.4 or higher is required. Moreover, the following plug-ins must also be installed in Eclipse:

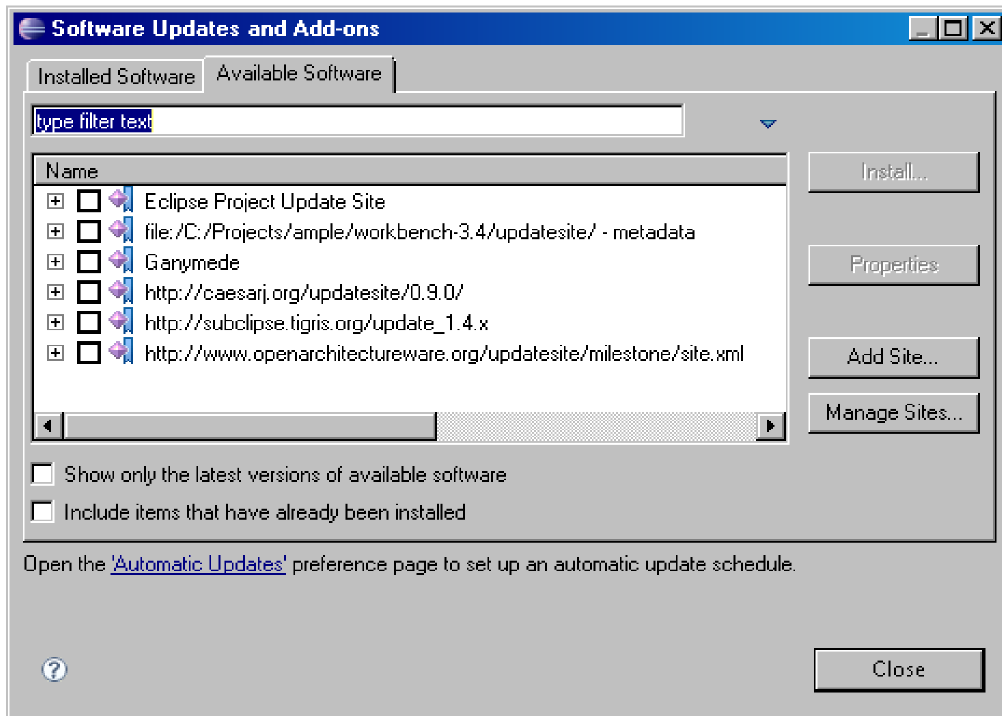
- OpenArchitectureWare 4.3 or higher, which can be obtained from http://www.openarchitectureware.org.
- CaesarJ and the CaesarJ development tools, which can be downloaded from http://caesarj.org

Installation

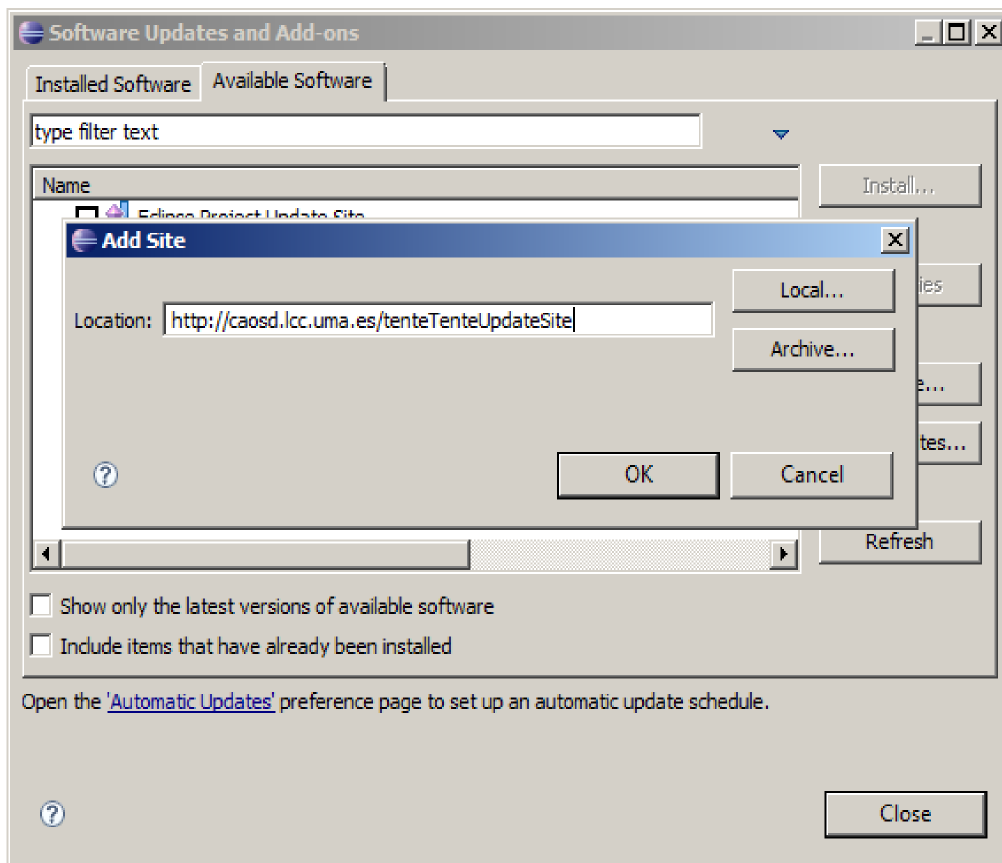
1) Once Eclipse has been installed, go to Help -> Software Updates



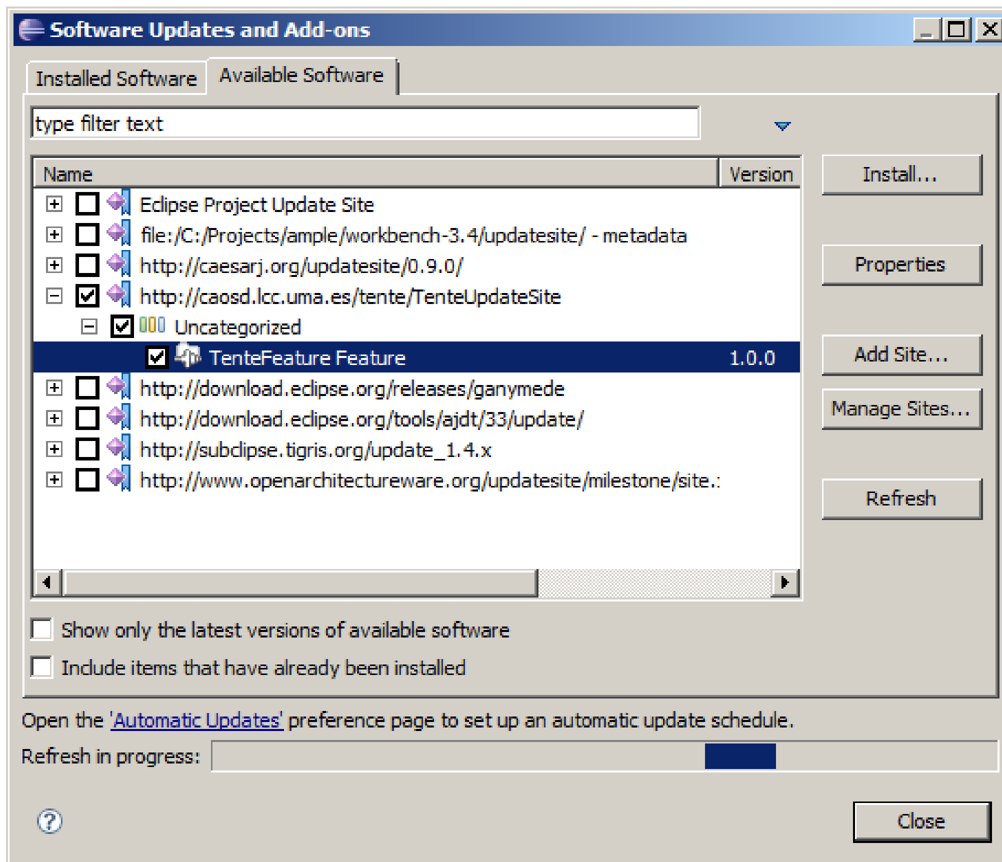
2) Click in Available Software-> Add Site



3) Write in location http://caosd.lcc.uma.es/tente/TenteUpdateSite and press OK

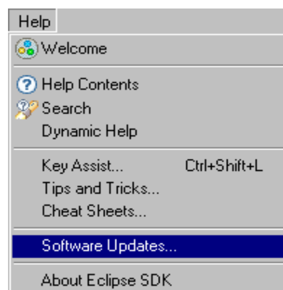


4) Select TenteFeature and press Install

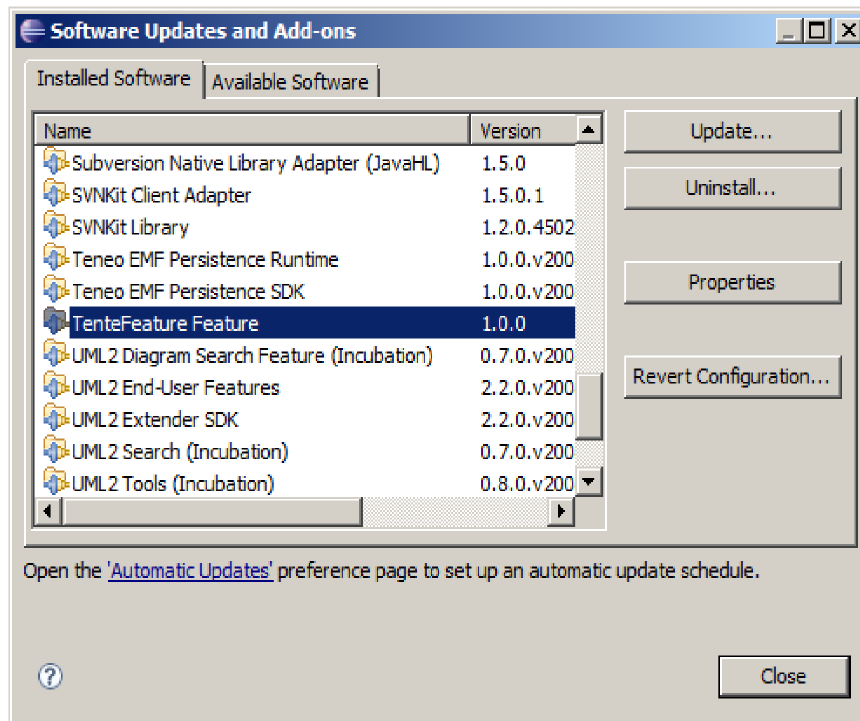


A.2 Uninstallation

1) Open Eclipse, go to Help -> Software Updates

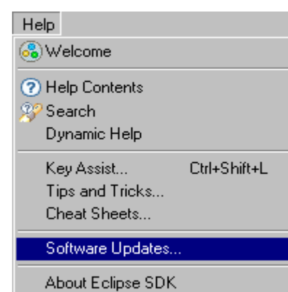


2) Click in the InstalledSoftware, search for TenteFeature and press Uninstall

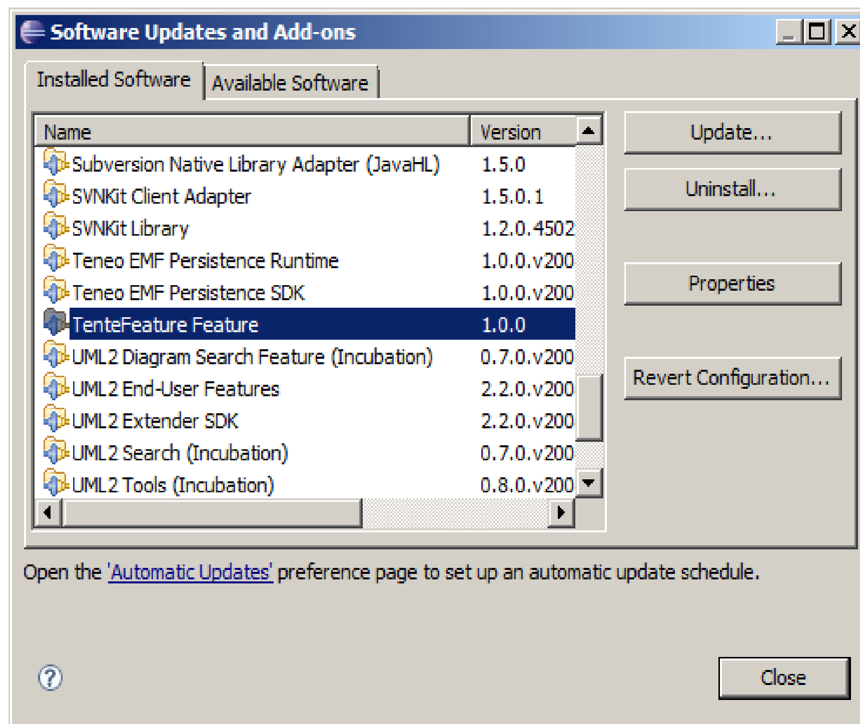


A.3 Updating

1) Open Eclipse, go to Help -> Software Updates



2) Click in the InstalledSoftware, search for TenteFeature and press Update.



A.4 Generation of Code Skeletons

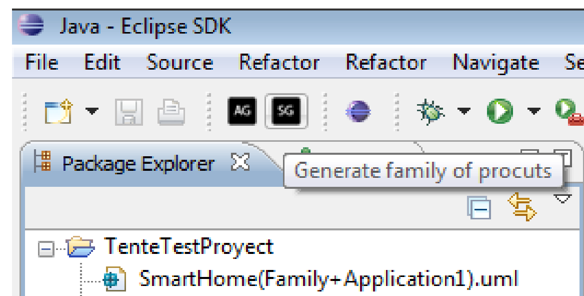
Creating the Family Model.

In order to generate code skeletons, the first step is to create a *reference architecture*. There are several tools that work with UML models like MagicDraw, EnterpriseArchitecture or even Eclipse. Any UML tool been able to work with component diagrams and able to export models to XMI version of the UML2 tools can be used for this purpose.

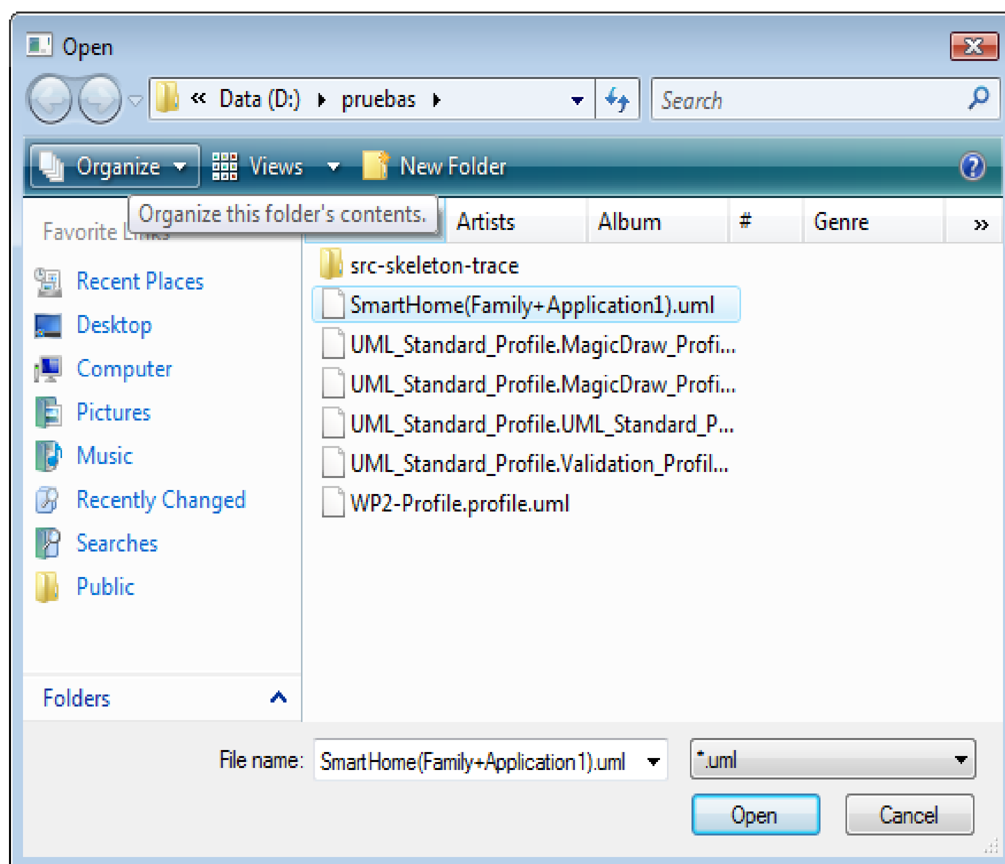
Generating code outside the Eclipse workbench.

The TENTE plug-in allows code generation using a model file that is outside the eclipse workbench. When the code is generated using this option, it is also possible to select the directory in which the code will be generated. We describe the steps for performing this task:

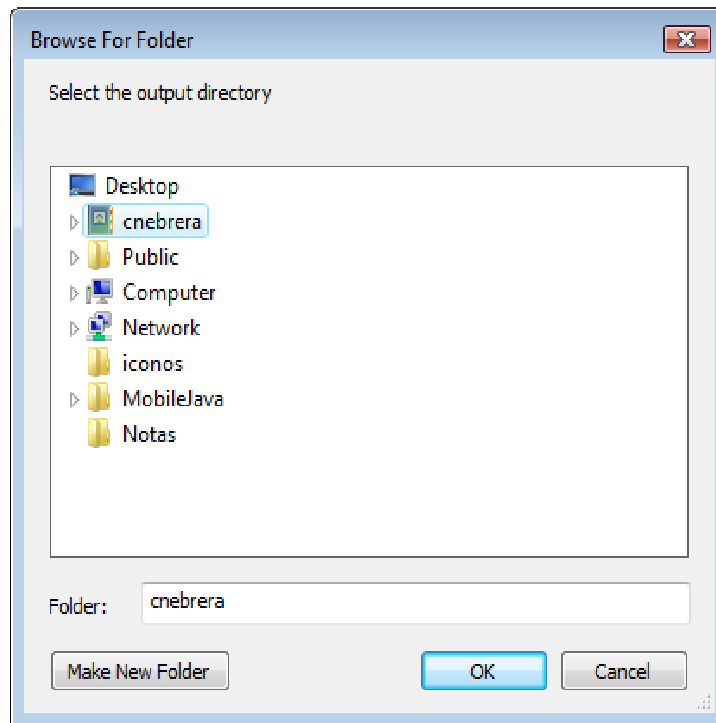
1. Press the SG black button in the eclipse tool bar.



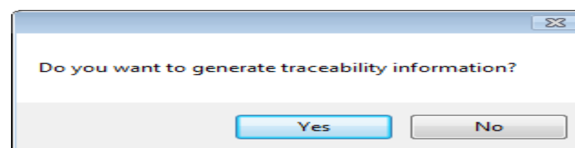
2. Select the model file inside the file system and press Open.



3. Select the output folder for the generated code. Inside this folder, a folder called scr-skeleton-gen will be created. The generated code will be placed in this latter folder.



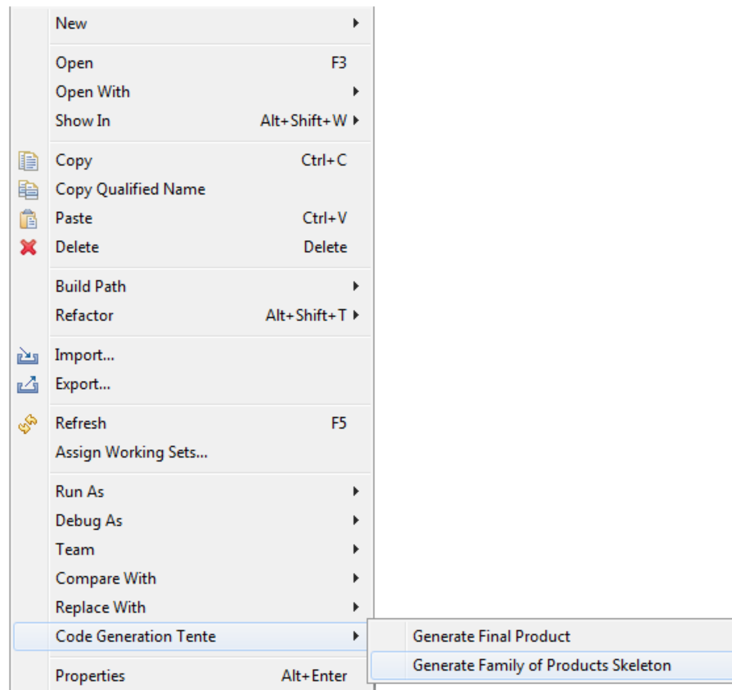
4. Once the output folder has been selected, the plug-in will ask if we want to generate traceability information. If so, a new folder called scr-skeleton-trace will be created. The traceability information is stored as a XML file inside this folder.



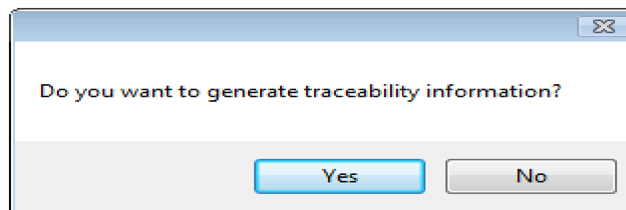
Generating code inside the workbench.

TENTE also allows the code generation directly from a model that is contained in the Eclipse workbench. In this case, the code is generated inside the same directory in which the model is stored. We describe the steps for performing this task:

1. The first step is to select the model file and right click in order to show the context menu. Select TENTE code generation -> Generate Family of Products Skeleton.



2. We are asked if we want to generate traceability information. If so, a folder scr-skeleton-trace for the storing the XML traceability file is created. A new folder called scr-skeleton-gen is created for the code skeletons. Both directories are created in the same project where the model is stored.



A.5 Code Generation of Specific Products

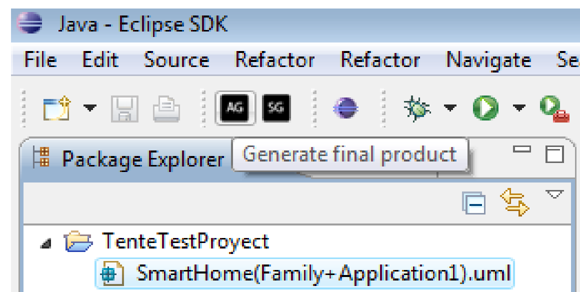
Creating the Application Model.

In order to generate a final product, the first step is to obtaining the software architecture for a specific product. This could be done automatically by selecting the product features and using VML or it could be done manually by means of creating with a model tool like MagicDraw, EnterpriseArchitecture or even Eclipse. As in the domain engineering level, the model must be in the XMI version of the UML2 tool.

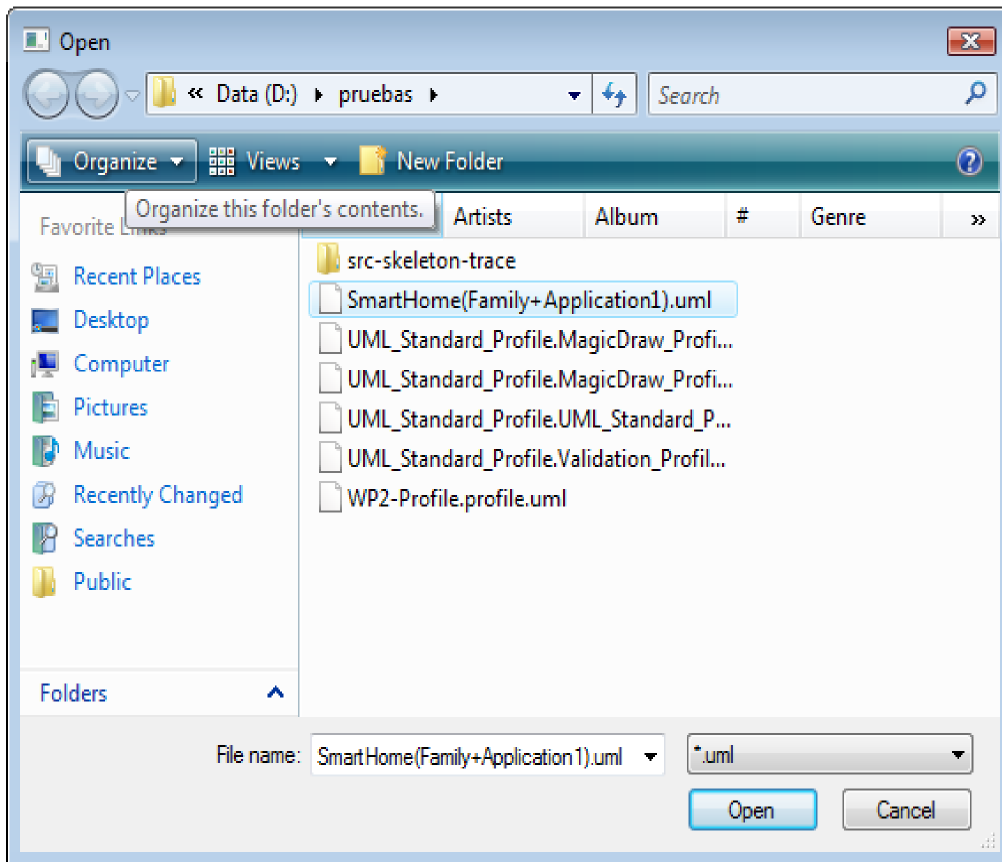
Code Generation code outside the Eclipse workbench.

The TENTE plug-in allows code generation using a model file that is outside the Eclipse workbench. When the code is generated using this option, it is also possible to select the directory in which the generated code will be placed. We describe the steps for performing this task:

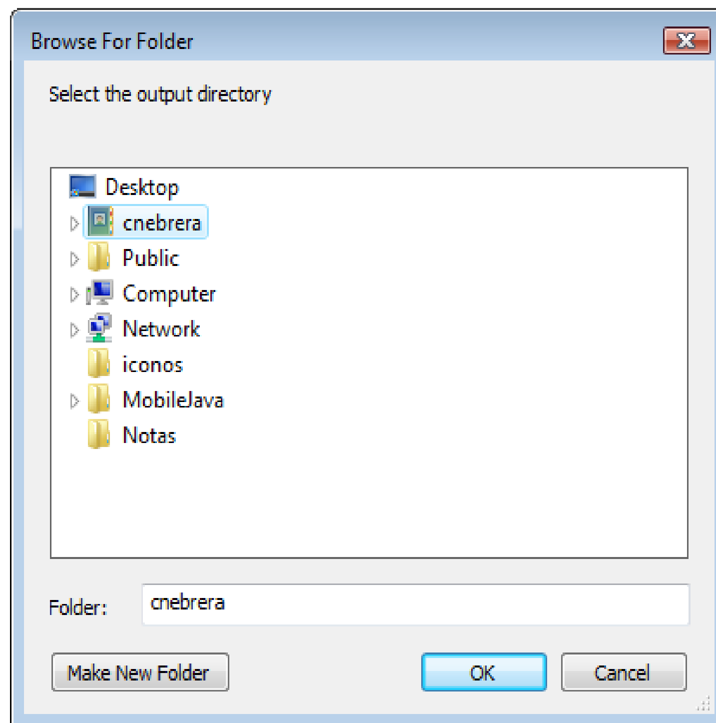
1. Press the AG black button in the Eclipse tool bar.



2. Select the family model file inside the file system and press Open.



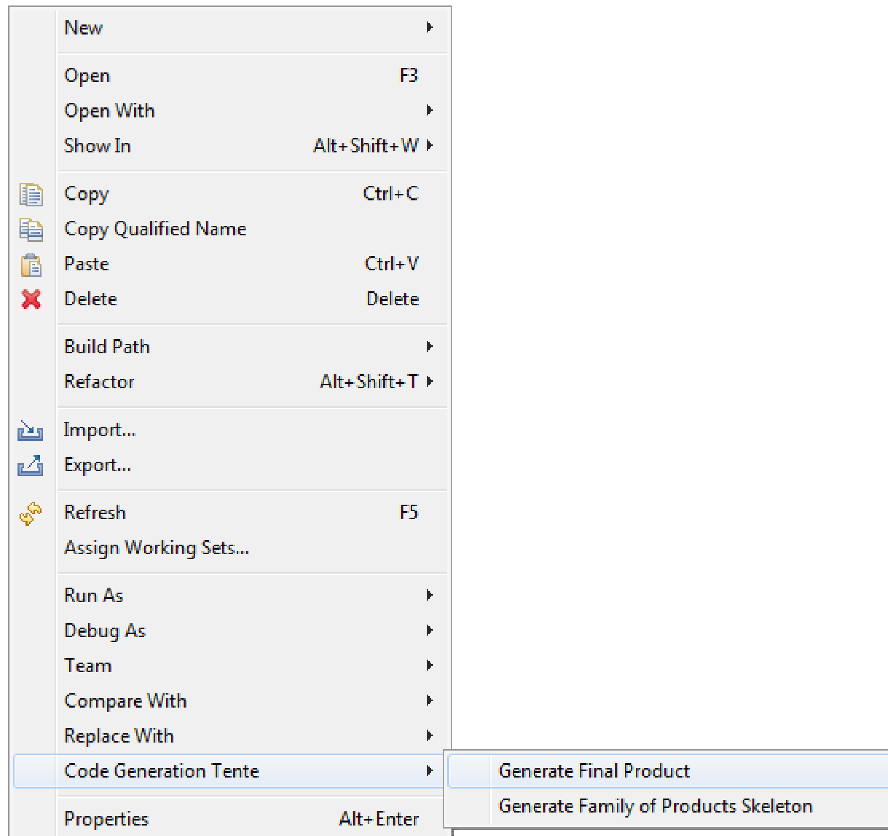
3. Select the output folder for the generated code. Inside this folder, a new folder called scr-application-gen will be created. The generated source code will be placed in this folder.



Generating code inside the workbench.

The TENTE plug-in also allows code generation directly from a model that is visible from the workbench. In this case, the generated code is placed inside the same project in which the model is stored. We describe the steps for performing this task:

1. Select the model file and right click on the model file to show the context menu. Go to TENTE code generation->Generate Final Product.



2. A new folder called scr-skeleton-gen is created for the final product code. The directory is created in the same place where the model is stored.

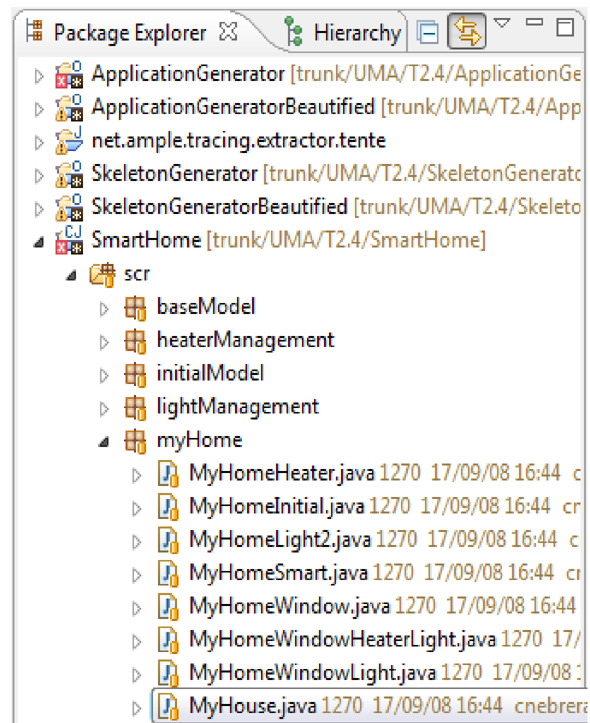
APPENDIX B. Smart Home User Manual

In the first section of this appendix it will be explained how to generate a new Smart Home Final Product and how to set the generated product inside the SPL to work with it. We will also explain how to execute an instance of generated product. In the second section the visual elements of the SmartHome will be explained. The third section will explain with more detail the different modules of the product, the logic rules introduced by each module, and how to interact with them using the GUI.

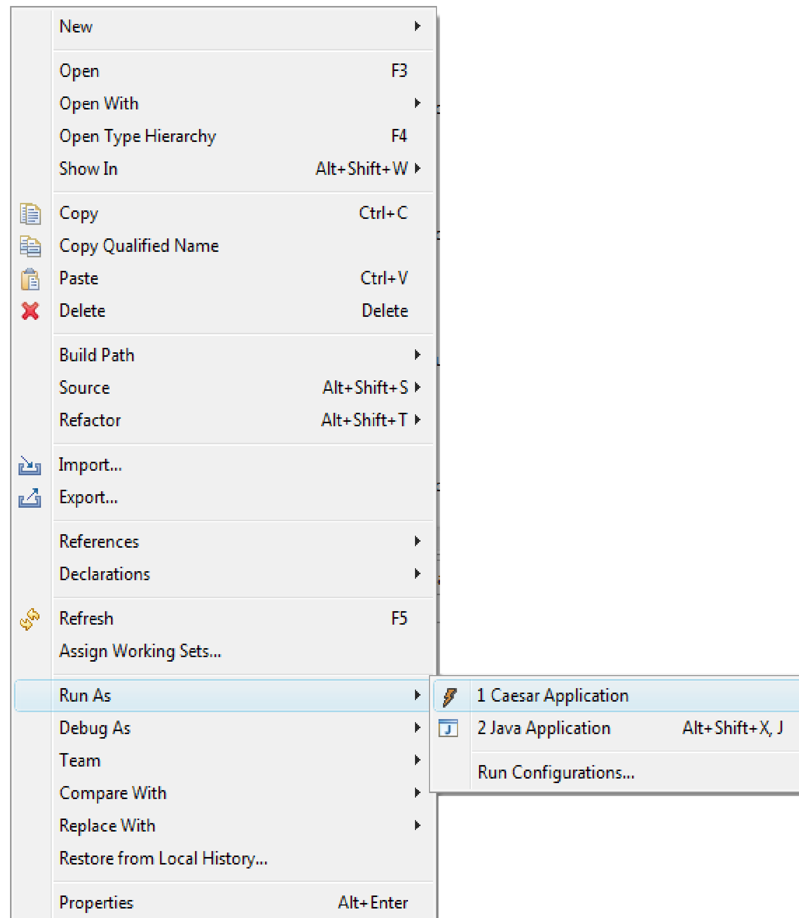
B.1 Generating and Executing

To generate a Smart Home product we have to follow the steps described in section A.7. Inside the Eclipse project "ApplicationGeneration" we could found 3 different Application Models for the SmartHome that are already in UML2 format. Once the code is generated, go to the folder where we have generated the code, there will be a file called MyHome.java inside the finalProduct directory. This file is the final *family class* that instantiate and interconnect all the components.

The Smart Home SPL can be found in the Eclipse project "SmartHome". To run a final product first we have to copy the file MyHome.java that we have generated and contain the final product information, inside the package MyHome in the project "SmartHome". This package contains final products for the SPL. There are other test final products that were manually generated to test the Smart Home functionality, like MyHomeHeater.java that corresponds with a Smart Home with only heaters.



To run a final product we have to do right click over the java file that correspond to the product and select Run As->Caesar Application. In this case to run the final product we have generated we have to do right click over MyHouse.java and select Run As-> CaesarJ Application.

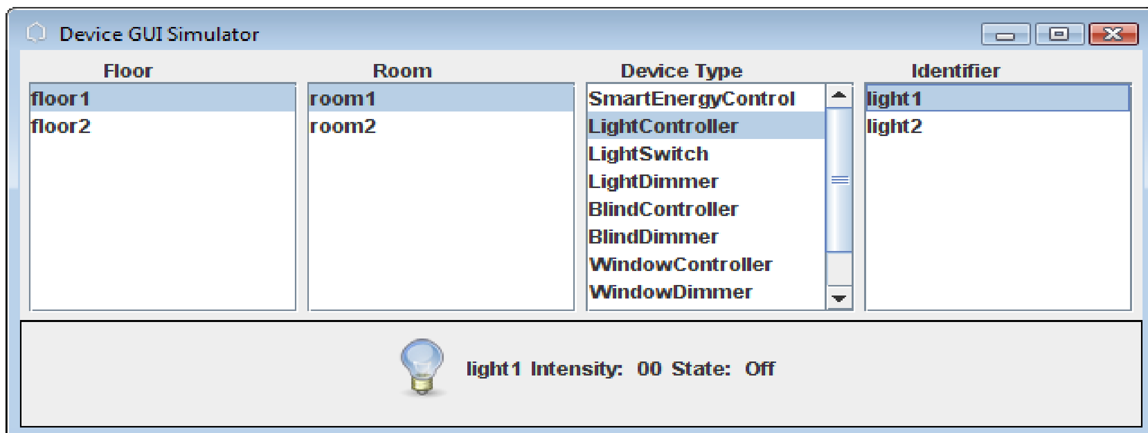


B.2 UI Description

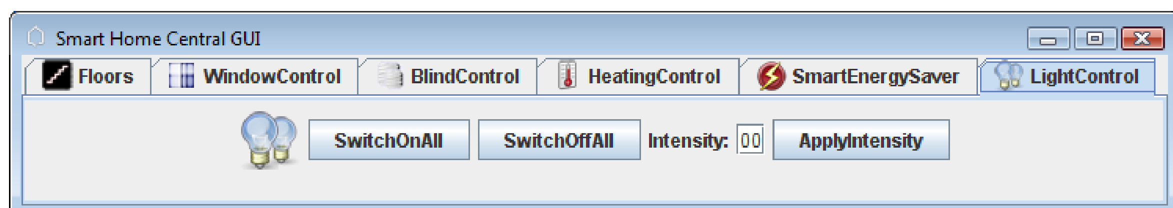
Independently of the Smart Home product configuration, two visual windows will be opened when we run the product. One is called “Device GUI Simulator” and the other “Smart Home Central GUI”.

Since there are no physical devices to test the product, a device simulator has been created. Normally for each component that corresponds to a physical device, a new element is added to the simulator. The simulated devices communicate directly with the components, simulating a physical interaction between them and a human. The white panel situated in the left part shows the floors of the house. Once a floor is selected, the next panel will show the rooms that we could find inside that floor. Once a room is selected, the next panel shows the different device types that can be found in the room. And finally the last panel shows the devices of the selected types that can be found in the room. Basically it is a way of put some order in the simulator to search for a concrete device. Once a device is selected, the simulated controls will appear in the

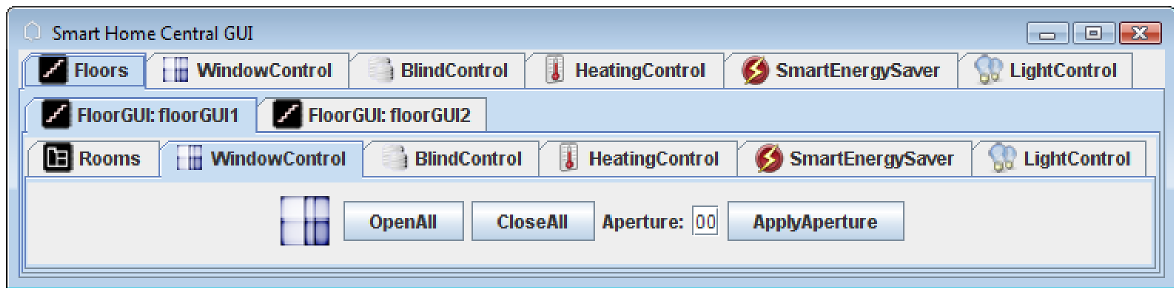
bottom window. Next figure shows an example using the final product of B.1 step, when a light controller is selected.



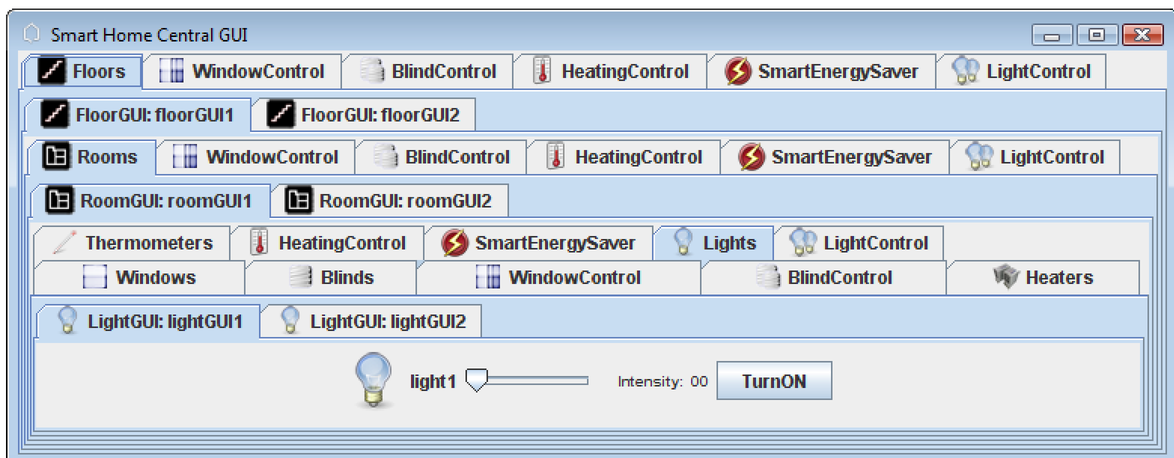
The "Smart Home Central GUI" windows are the real GUI that is shown to the Smart Home user. Using the GUI the user can interact with the Smart Home. The GUI has been structured in tabs at different levels. The first level show the general controls that affect the full house. For example if the house has light control, there is a tab "LightControl" that allows to modify the state and intensity of all the lights of the house in the same time when it is selected. Next figure shows this example:



In the first tab level there is a tab called Floors, once it is selected it shows a second level tab with all the floors of the house. Once a floor is selected a new tab with the controls for that floor is shown. This tab contains the controls that affect to the whole floor. For example, if we select the first floor and the WindowControl tab, we can open, close or change the aperture for all the windows inside that floor. The next figure shows this example:



If we press the tab called Rooms, it shows the rooms that we could found in the selected floor. Once a room is selected the controls for that room are shown. This time it shows the general control for the room, like we saw when a floor was selected and the general control for the floor appear, and there are also tabs for individual elements. For instance, if we select the first room of the first floor, we can see the general controls like LightControl or BlindControl and also the tab that corresponds to individual controls, like Lights or Blinds. If one of this tabs, like Lights is selected, it shows a tab for each light inside the room and selecting them we could individually control each light. Next figure shows this example:



In the next section the different features that can be selected for the Smart Home will be explained. We will explain also the different visual elements that are added for each feature and the logic rules of the product.

B.3 Functionality

The Smart Home is divided in different modules, each one of them adds a concrete functionality. Most of the modules are independent; they don't require other modules to

work. For example, it is possible to configure a house with light management or window management or none of them. The only module that is not independent is the smart energy saver module that requires window management and heater management to work.

In this section we will explain what is added when a module is selected in the GUI, what new devices can be are controlled and the logic rules for the control of the devices.

1) Light Management

Light Management module allows the control of lights, it doesn't require other modules. It allows switching on and off individual, or groups of lights, and controlling their intensity individually or by groups. Initially all the lights are switched off and the intensity is set to 0.

For each controlled light there should be 3 devices in the Device Simulator:

- LightController: Represents the device that controls electrically the light. It shows the actual state and intensity. The intensity could vary from 0 to 100.
- Switch: Represent the device that allows switching on and off a light manually. It shows a button to do this operation.
- Dimmer: Represent the device that allows changing the intensity of a light manually. It shows the actual intensity and a scroll bar to change it.

The next elements are added to the Smart Home GUI when the light management module is included:

- LightControl tabs: In the general tabs a new tab called LightControl is added. When it is pressed a panel is shown. This panel allows switching on all the lights of the house, switching off the lights or modifying their intensity. The same panel is added to the tabs of each floor and each room, allowing controlling all the lights inside a floor or a room respectively.
- Lights tabs: In the tabs of each room a new tab called Lights is added. When this tab is selected, a tab for each light inside the room is shown. This tab opens a panel allowing individual control of the selected light. A scroll bar allows changing the intensity and a button to modify the state of the light.

2) Window Management

Window Management module allows the control of windows and blinds, it doesn't require other modules. The aperture of a window or a group of windows can be selected. The aperture can vary from 0 to 100. This module also controls the aperture of the blinds of the house with the same rules than the windows. Initially the windows and blinds are closed; therefore their aperture is set to 0.

For each controlled window there should be 2 devices in the Device Simulator:

- WindowController: Represents the device that controls electrically the window aperture. It shows the actual aperture of the window.
- WindowDimmer: Represent the device that allows changing the aperture of a window manually. It shows a scroll bar to perform this operation.

For each controlled blind there should be 2 devices in the Device Simulator:

- BlindController: Represents the device that controls electrically the blind aperture. It shows the actual aperture of the blind.
- BlindDimmer: Represent the device that allows changing the aperture of a blind manually. It shows a scroll bar to perform this operation.

The next elements are added to the Smart Home GUI when the window management module is included:

- WindowControl tabs: In the general tabs a new tab called WindowControl is added. When it is pressed a panel is shown. This panel allows full opening and closing on all the windows of the house or modifying their aperture to the desired value. The same panel is added to the tabs of each floor and each room, allowing controlling all the windows inside a floor or a room respectively.
- Windows tabs: In the tabs of each room a new tab called Windows is added. When this tab is selected, a tab for each window inside the room is shown. This tab opens a panel allowing individual control of the selected window. A scroll bar allows changing the aperture of the window.
- BlindControl tabs: In the general tabs a new tab called BlindControl is added. When it is pressed a panel is shown. This panel allows full opening and closing on all the blinds of the house or modifying their aperture to the desired value.

The same panel is added to the tabs of each floor and each room, allowing controlling all the blinds inside a floor or a room respectively.

- Blinds tabs: In the tabs of each room a new tab called Blinds is added. When this tab is selected, a tab for each blind inside the room is shown. This tab opens a panel allowing individual control of the selected blind. A scroll bar allows changing the aperture of the blind.

3) Heater Management

Heater Management module allows temperature control inside the house, it doesn't require other modules. For this purpose thermometers and heaters devices are added. There could be any number of thermometers in a room. Each thermometer controls the temperature inside the room in the position it is placed, and the temperature outside the room in the outside of the house. Like thermometers, any number of heaters can be placed in a room. The heaters can work in two modes, heating or cooling and the power of them can be set between 0 and 100. Each heater is related with a thermometer.
The house user cannot control the heaters directly like it can be done for windows or lights. The heaters have to be controlled through the Smart Home GUI, selecting the desired temperature. Initially all the heaters are switched off and theirs temperature set to 25 degrees in the GUIs. The program will set the power and mode of the heater depending of the temperature inside the room and the selected temperature for the heater. The rule is simple, if the heater is on and the temperature selected is lower than the inside temperature given by the thermometer associated, the heater mode is set to cooling. In other case is set to heating. For the power the difference of selected and inside temperature is calculated, a 1% of power is given by each 0.1 degree of difference. If the difference is bigger than 10 degrees the heater is set to full power.

For each controlled heater there should be a device in the Device Simulator:

- HeaterController: Represents the device that controls electrically the heater aperture. It shows if the heater is active, if it is cooling or heating and the power it is set to.

For each controlled thermometer there should be a device in the Device Simulator:

- Thermometer: Represents the device that measures the temperature. Since there are no real thermometers and in order to do accurate simulations, the panel of this device allows us to manually modify the external and internal temperature given by the thermometer. Initially the thermometers are set with an internal temperature of 25 degrees and an external temperature of 30 degrees.

The next elements are added to the Smart Home GUI when the heater management module is included:

- HeatingControl tabs: In the general tabs a new tab called HeatingControl is added. When it is pressed a panel is shown. This panel allows switching on and off all the heaters of the house or modifying their temperature to the desired value. The same panel is added to the tabs of each floor and each room, allowing controlling all the heaters inside a floor or a room respectively.
- Heaters tabs: In the tabs of each room a new tab called Heaters is added. When this tab is selected, a tab for each heater inside the room is shown. This tab opens a panel allowing individual control of the selected heater. The power, state, mode and selected temperature of the heater are shown. It also allows modifying the temperature or switching on and off the heater.
- Thermometers tabs: In the tabs of each room a new tab called Thermometers is added. When this tab is selected, a tab for each thermometer inside the room is shown. This tab opens a panel showing the temperatures given by the thermometer.

4) Smart Energy Control

This module allows energy saving in the heating system of the house; it requires the heating management and window management modules. For each room of the house it is possible to activate or deactivate the smart energy mode. When this mode is active, the program calculates the average difference between the internal and external temperature of the room. It also calculates the average selected temperature in the active heaters. If the selected temperature could be reached by opening the windows, the windows of the room are opened and the power of the heaters is set to 0 in order to save energy. The calculations are done each time the temperature of a

thermometer changes, the selected temperature in a heater changes, the state of a heater is changed or the smart energy saver is switched on in the room.

For each room there should be a device in the Device Simulator:

- SmartControl: It is not really a device, it shown if the smart energy is set on or off for the selected room. It is added for testing purpose only.

The next elements are added to the Smart Home GUI when the smart energy control module is included:

- SmartEnergySaver tabs: In the general tabs a new tab called SmartEnergySaver is added. When it is pressed a panel is shown. This panel allows switching on and off the energy saver system for all the rooms of the house. The same panel is added to the tabs of each floor and each room, allowing controlling the save energy system inside a floor or a room respectively.