AMPLE
Aspect –Oriented, Model-Driven, Product Line Engineering
Specific Targeted Research Project: IST- 33710

Description of feasible industrial case studies

ABSTRACT

This document describes a collection of feasible case studies that can be taken as test-bed for results of AMPLE research work packages.

Document ID:	AMPLE D5.1
Deliverable/ Milestone No:	D5.1
Work-package No:	WP5
Type:	Deliverable
Dissemination:	CO
Status:	final
Version:	1.0
Date:	2007-09-27
Author(s):	Morganho, Hugo; Pimentão, João Paulo; Ribeiro, Rita (Holos), Pohl, Christoph; Rummler, Andreas (SAP); Schwanninger, Christa, Fiege, Ludger (Siemens AG)

Project Start Date: 01 October 2006, Duration: 3 years

History of Changes

Version	Date	Changes
0.1	2007-07-08	Document creation
0.2	2007-08-09	Smart Home by Siemens
0.3	2007-08-10	Sales Scenario by SAP
0.4	2007-08-30	Added Holos contribution, minor changes
0.5	2007-08-31	Integrated feedback
0.6	2007-09-05	WP4 work-package description
0.7	2007-09-07	Added WP1 description by UNL/FCT
0.8	2007-09-12	Integrated feedback
0.9	2007-09-15	WP 2 work-package description
0.10	2007-09-21	Integrated feedback
1.0	2007-09-27	Integrated final feedback

Table of Contents

1. Introduction

This document describes feasible case studies from industry and SME partners. They are used to

- find the challenges in current software product line development,
- test the suitability of existing tools and technologies and detect their weaknesses and deficiencies concerning product line engineering support,
- serve as test-bed for proof of concept implementations of WP1 to WP4.

For this purpose we aim at conducting a series of case studies from small experiments to whole demonstrators that cover all life cycle steps. A small case study covering product line lifecycle steps from domain analysis to product implementation was already implemented in the first year. It is used to clearly identify challenges and existing support. Further small experiments will investigate research challenges that are addressed in the other work packages. In the large demonstrator, all the contributions of AMPLE are shown in own setting.

In the following several case studies are described. A selection of these case studies will be implemented within the next two project years.

2. Case Study: Smart Home

The Smart Home case study investigates an intelligent home, its configuration, the services necessary for its automation, and the technological platform for its realization.

2.1 Domain and Scope

In the homes of European citizens you typically find a wide range of electrical and electronic devices: home equipment like lights, thermostats, electrical blinds, fire and glass break sensors, white goods like washing machines, dryers and dishwashers, entertainment equipment like TVs, radio and devices to play music or movies, and communication devices like (smart) phones and PCs. Sensors are devices that measure physical properties of the environment and make them available to Smart Home. Actuators activate devices whose state can be monitored and changed. Varying types of houses, different customer demands, the need for short time-to-market and saving of costs drive the need for a Smart Home product line, which is characterized by a wide range of variants.

The goal of projects in the Smart Home domain is to network those devices and enable the inhabitants of a home to monitor and control the home from various user interfaces. A rudimental solution allows controlling of devices from certain technical areas inside the house, which also execute home-centric applications. A more ambitious solution integrates more kinds of devices and includes an external platform to enable remote access and services from other providers. Tasks like billing, logging and platform management are involved in this case.

The home network also allows the devices to coordinate their behaviour in order to fulfil complex tasks without human intervention. The status of devices can either be changed by inhabitants via the user interface or by the system using predefined policies. Policies let the system react autonomously to certain events. For example, in case of smoke detection, windows get closed, doors get unlocked and the fire brigade is called.

Irrespective of how advanced the smart home application is, personalisation is an issue. A user should be able to define her/his preferences and the system should make it easy for her/him to achieve a preferred status of certain devices and use services in her/his preferred way. It may be desirable to offer certain services only to certain persons. Therefore, authentication and authorization come into play. Authentication becomes even more important when the system contains an external platform and external services.

The most important *functional requirements* for a smart home platform are:

- Monitor and change the status of devices (for the end user)
- Monitor the changes made manually (by the end user)
- Installation of new devices (for the end user or an operator)
- Installation of new kinds of devices (for an operator)
- Let the system act autonomously according to defined policies
- Personalization
- Authorization
- Authentication

The following *non-functional requirements* are predominant:

- Stability/Reliability: If security relevant services like fire alarm, babysitter functions or emergency help for aged people are provided, the stability and reliability of the system are very important.
- Short feedback times to GUI: For the acceptance of the system an instant reaction of the system to user input is indispensable.
- Scalability: A smart home platform should be usable for small homes with few technical devices as well as for large homes with many devices and a high diversity of devices.
- Security: The smart home platform may contain security relevant information like information if a person is at home, if children are alone at home or where alarm devices are. Also personal information like documents or pictures may be contained. Therefore unauthorized access to this information has to be barred.
- Variability: For different users and different situations different views on the devices are necessary.
- Extensibility: The inclusion of new services and extension of already available services should be easy.

The domain consists of the following key entities:

- House
- Floors
- Rooms
- Controlled Devices
- Smart Home Service Platform

- Remote Control GUI Devices (inside house)

Optionally, the following key entities are involved:

- Service Enabling Platform (outside house)
- Remote Control GUI Devices (outside house)
- 3rd party services

The following lists typical devices, communication media and protocols:

- PC, embedded real time systems, small sensors and actuators
- Communication: Ethernet, WLAN, Instabus
- Middleware: OSGi, UPnP Protocol

The following key roles are defined:

- Inhabitant: lives in the home and is interacting with the GUI devices and partially directly with the smart home devices.
- Smart Home Provider
- External Service Provider
- Service Platform Administrator: installs smart home platform in the home, adds new devices or new kinds of devices and administrates users and their rights.
- Service Enabling Platform administrator: enables external services.

The entities and roles are visualized in the following figure; the boxes represent domain hardware entities, while the lines represent dependencies.

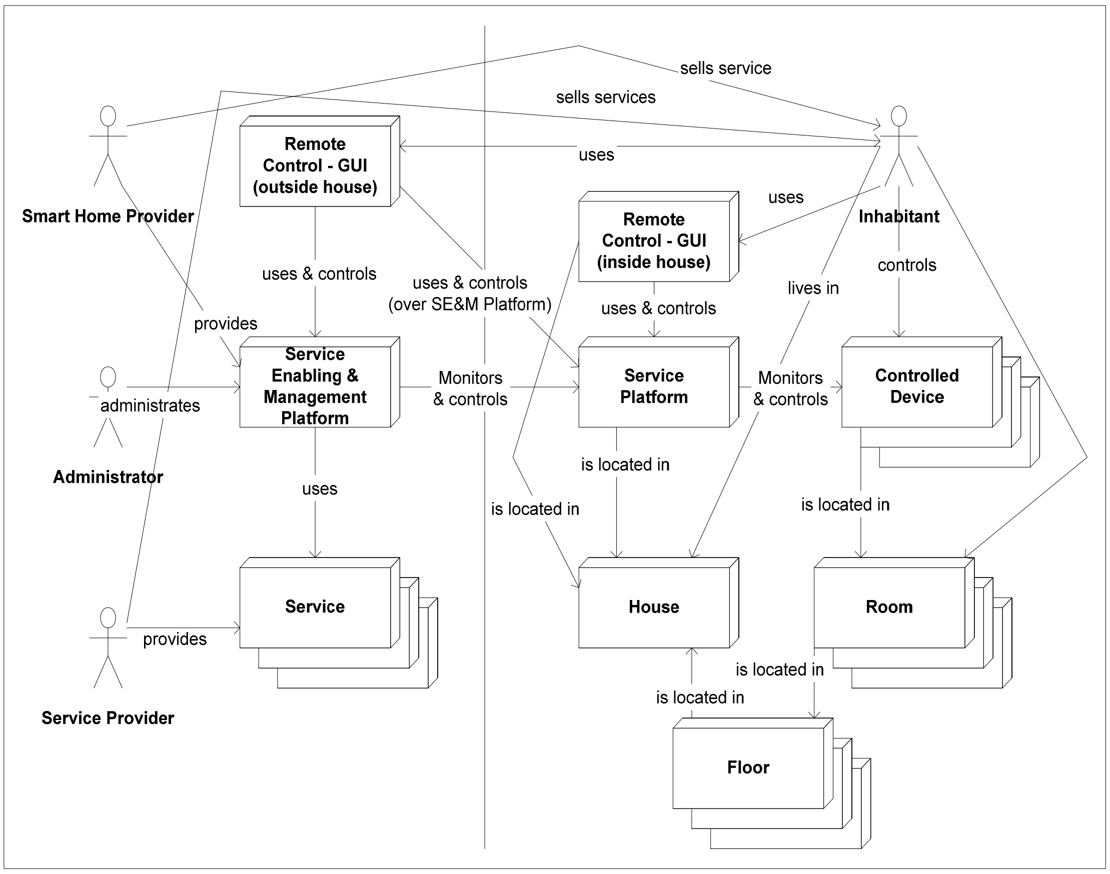

Figure 1 Entities and Roles in the Smart Home domain

2.2 Challenges

2.2.1 Variability

Every home is different. A smart home product line needs to accommodate a considerable amount of variability.

The first and major source of variability is due to the fact that the layout of a house – the number of floors, rooms, connecting doors, windows – is typically unique for every house. The (building's) architectural components need to be equipped with devices and connected amongst each other.

Second sources for variability are the services a home owner wants to have installed in his house. Services orchestrate a number of devices to fulfil complex tasks without user intervention. Intelligent climate control, power saving or burglar detection and intrusion prevention are examples of such services. Depending on the customer, these services have to be added to the home automation system and use the given infrastructure to perform as intended. So, there is a dependency between this kind of service variation and the devices that are installed in the house.

A smart home product line has to manage this dependency.

2.2.2 Support for Problem Domain Expert Instantiation

Ideally, a smart home distribution should not require a "solution domain expert". The stakeholders for specifying and installing a Smart Home are typically not software experts, therefore specifying the properties of a concrete Smart Home has to be intuitive for stakeholders like building architects.

Specifying a concrete Smart Home should be possible in a specific graphical or textual language that is intuitive to use for a building architect. Icons and terms in this language should be expressive to prevent errors in describing the home. The architect is typically not familiar with the IT-domain. The automation software should be generated, as well as a plan which devices should be installed where.

The kinds of devices that should be supported are at least:

- Lights, light switches
- Window openers, window sensors, blinds, glass break sensors
- Radiators and thermostats
- Door sensors and door openers
- Cameras
- Motion and light detectors
- Presence sensors
- Fire and smoke detectors, sprinkler system
- Alarm devices
- Home entertainment systems, e.g. TV, Audio
- Communication systems, e.g. phones, PCs or other internet enabled devices
- White goods like washing machines

The Smart Home description language should make it easy to specify which devices should be situated in which rooms and should also allow connecting sensors and actuators as necessary, e.g. mapping a light switch to a light or a blind to a window.

It should be easy to integrate devices from different vendors by simply adding the software that is shipped with the device. Device vendors are typically the choice of the home owner.

Once the home is specified, the automation software should be generated automatically.

The Smart Home system shall offer higher level functionality in which several sensors and actuators are working together. This high level functions should be optional, but if selected, they require certain devices to be installed in the house. These functions should be but are not limited to:

- Energy saving
  Thermostats, heating, window openers, door openers and white goods should be orchestrated to spend as few energy as possible and when white goods are involved, they should be operated such that the cost for energy is as low as possible, e.g. in some countries power is cheaper during the night.
- Climate control system
  Heating, thermostats, blinds and windows should be orchestrated to keep a preferred temperature in the rooms of the house.
- Security system
  Glass break sensors, door sensors and motion detectors should be used to detect if persons who are not allowed to enter the house try to do so. If the house detects intrusion, it should give alarm with either sirens or bells or inform the police or a security company.
- Fire and smoke handling systems
  Fire and smoke detectors, sprinkler systems, window and door sensors and

openers/closers, alarm devices and communication devices should work together to prevent foremost human damage in case of fire and smoke. Furthermore the fires should be extinguished and the fire brigade should be called. The system should be intelligent enough to not start extinguishing fire if there are only a couple of people smoking in a room.

- Leaving the house / alarm system
  The security system is controlled from a panel in the hallway. This panel also acts as the answering machine for the house. There are a series of voice prompts on the alarm panel that will lead you through its operation. Setting the alarm will activate the central locking system. This will close all windows and doors and can switch off any devices. It may turn off any lights left on during the daytime. The security system can also be operated through the internal telephone or by dialling into the house from an outside line. In these instances, the voice commands work in the same way as using the panel directly. Any of the remote control devices can also be used to operate the alarm system.

2.2.3 Evolution

The innovation rate in home automation systems is expected to be high. This means the product line has to care for accommodating evolution, e.g. by being able to easily integrate new devices and new kinds of services.

For example for expanding the smart home domain to secure homes, a whole set of device types is necessary, like

- Fire and smoke detectors
- Sprinklers
- Intrusion detectors
- Motion detectors
- Cameras
- Alarm devices

These devices make new services possible, e.g. intrusion detection and raising alarms under certain conditions when the inhabitants are not at home. Another service would deal with fire in the house, were it is necessary to inform all inhabitants, unlock doors, shut windows and call help from the fire brigade.

2.2.4 Traceability

Though intuitive, a smart home product line is complex. Therefore it is necessary to keep the information about the connection between the different lifecycle artefacts accessible to be able to instantiate products and to evolve the product line. Important directions of traceability are

- From domain requirements to domain implementation to be able to
  - Know what artefacts to change when requirements change (e.g. where new variations have to be built)
  - Map application requirements to reusable domain artefacts after comparing application requirements with domain requirements
- From domain implementation back to domain requirements to be able to judge
  - If changes in the implementation violate domain requirements which typically also affects the applications' behaviour

- From applications (products) to artefacts reused from the domain and vice versa to
    - Benefit from error corrections or enhancements in the domain

2.2.5 Unanticipated or Customer Specific Changes

It is likely that for a given set of requirements for a smart home, the product line architecture does not provide the required configuration or customization hooks. There has to be support for easily adapting either the product line to support the requirements or to make adaptations to a product, if the requirements are too specific to be integrated into the product line.

2.3 Research Areas and Solution Proposals

Providing a solution for a Smart Home product family requires supporting an enormous amount of variability in the software base assets. AOSD and MDD help us to solve the challenges imposed by the above described requirements. In particular, the demonstrator investigates the following concepts for solving typical product line engineering challenges:

- Model to model transformations (Problem Space to Solution Space)
  The problem space specification of a Smart Home consists of devices and their relations to achieve intelligent compound behaviour. In the solution space (also called software domain) those entities are implemented in software components.
  Example: For a thermostat device and a radiator device in the problem space model, the solution space model will consist of
    - A radiator driver component for each radiator
    - A thermostat component or each thermostat
    - And a service component per floor that coordinates and manages thermostats and radiators using some kind of control algorithm
- AO to allow for fine granular tracing:
  Aspects should reduce the granularity of assets such that traceability of requirements to code on a per-file-level becomes possible.
  Example: For integrating security aspects into a home, we will weave monitoring functionality into component implementations. Monitoring functionality is modularized in an aspect.
- Architecture Level AO:
  AO is not just useful on the level of the implementation language, it can also be used on the level of architectural building blocks, i.e. components (see invasive vs. non-invasive AO languages). For OSGi, we will have to provide a means to deploy interceptors. AO component middleware like CAM/DAOP supports this concept already.
  Example: An encryption interceptor encrypts communication among components.
- Model weaving (AO modelling):
  Model weaving is used to implement variability on the modelling and the meta modelling level.
  Examples: Depending on the Edition (Economy, Secure, Luxury …) the domain meta model must be changed. Advanced editions offer more kinds of devices and therefore also more advanced compound functionality.

- Combining AO and Generators:
  Aspects on templates and transformations are used to adapt model transformation and code generation.
  Examples:
  - In order to generate code for different target platforms, we will use AO techniques for templates to adapt the code generation.
  - If we change the domain meta model (because of various editions), and if we keep the solution space meta model similar, we need to adapt the problem space to solution space transformation.
- Combining feature models and DSLs for
  - Varying models (DSL sentences, e.g. specifying an alarm device with a feature model that offers choices for the alarm signal (audio or visual or sending an alarm message to the police)
  - Varying the DSL itself (meta models, syntax, editors)
  - Weaving transformations and transformation steps
- Unanticipated Changes & AO:
  Using AO languages one can hook into generated (or manually written library) code at places where the product line architects did not identify a hook.
- Traceability starting from Requirements:
  Representing requirements as models and establish traceability links to models created in subsequent phases.

3. Case Study: Sales Scenario

The Sales Scenario case study proposed by SAP investigates a software product line driven business application, its feature-based variability, constraints and interactions, certain integrated change scenarios and high-level architecture for its implementation.

3.1 Domain and Scope

3.1.1 Introduction

In the following example we describe a software-product line in the context of business applications. Considering this rather huge application domain quickly indicates the plethora of parts that could be treated in such an example, e.g., Product Life Cycle Management (PLM), Supply Chain Management (SCM) or Supplier Relationship Management (SRM), to mention only a few. Therefore, we focus on one specific part of the business domain, the Customer Relationship Management (CRM), and combine it with different subsets of some of the aforementioned parts. It has to be stated that although we focus on CRM, the example does not claim to be complete at all. We rather choose those CRM parts that seem to be relevant and meaningful enough to integrate into a variability-driven context and adjust them to our needs. Therefore, the presented example is called “Sales Scenario” to clearly distinguish from CRM as such. The whole example is mainly based on explanations of Buck-Emden and Zencke in [BZ04].

3.1.2 Description

The key message for the illustrated example is *„management is all”*. Core concept is the acquisition and exploitation of business process data (i.e., the holistic management). Central storage and access controlled retrieval shall be provided. It is irrelevant for the example, whether data is only stored and the system provides only a user interface for I/O, or if other communication channels are used to communicate directly or indirectly with other parties involved in a business process (printing invoices, quotations, dunnings, etc. is considered as indirect communication in this regard).

The Sales Scenario example focuses on sales processes of enterprises selling one or more products. This involves different things, ranging from Opportunity Management (currently not in the feature model, as depicted in Figure 2: Feature Model) to quotations to customers, sales order and invoice processing. Concrete building blocks can be found in the feature model.

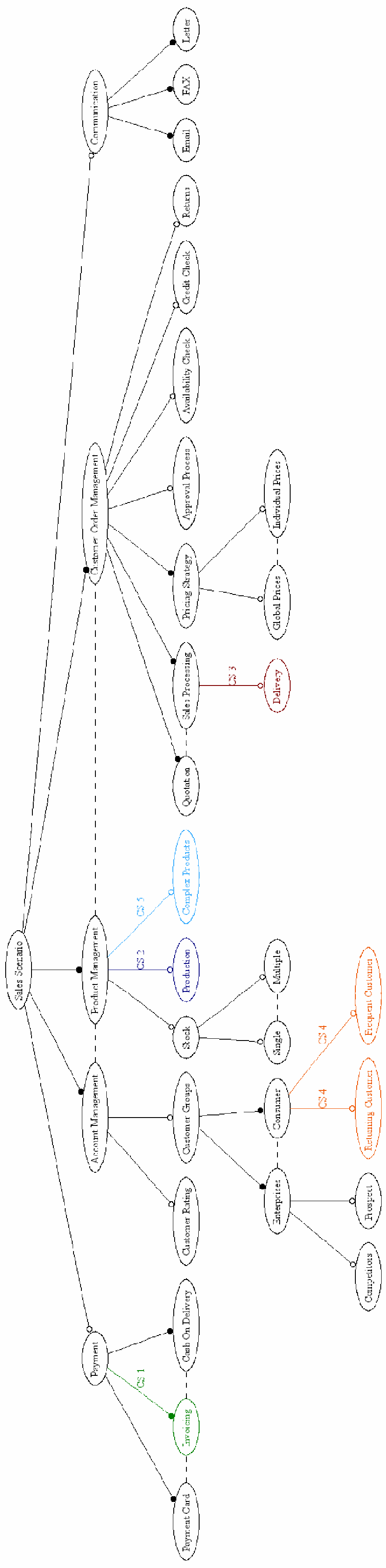

Figure 2: Feature Model

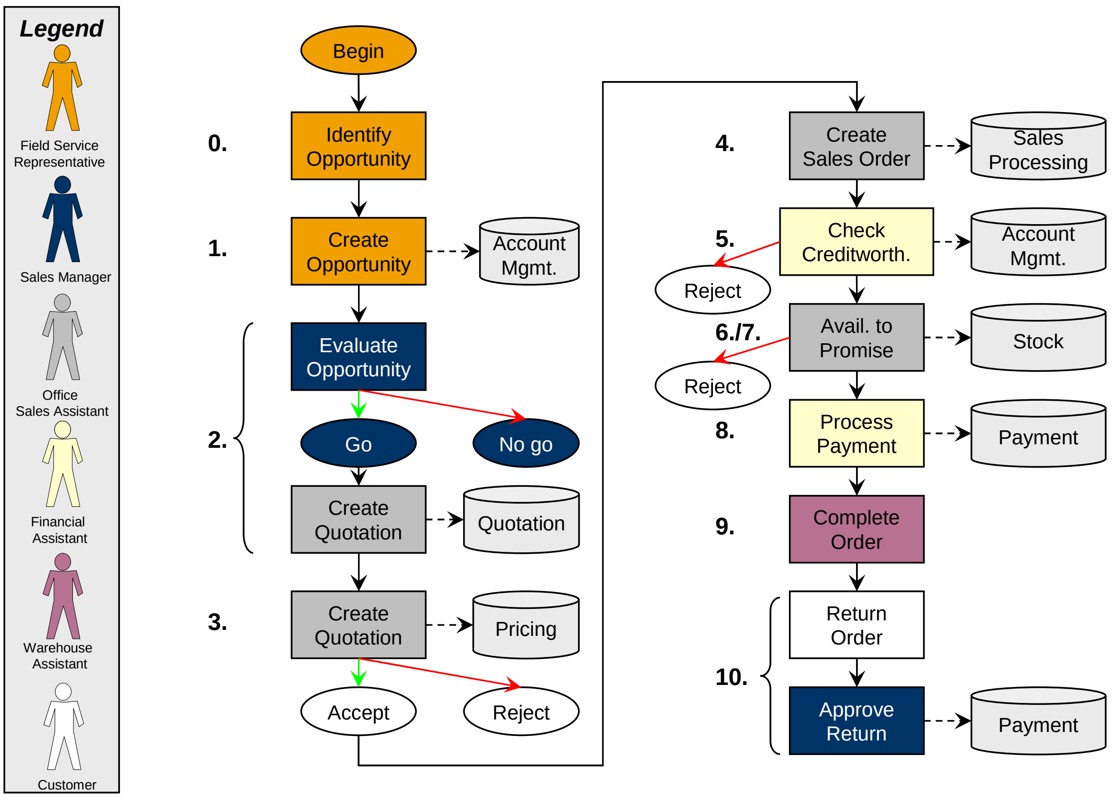

Figure 3: Schematic Process Flow of the Sales Scenario

In order to clarify the interplay between certain features and to give an understanding of what is covered with the given example, consider the following case study of the overall sales process and its individual components. To better mark down certain features, this process was clearly divided into separate steps, as shown in Figure 3 and by the corresponding numbering in the following text block.
Remark: The first part of the Case Study involves Opportunity Management features that are not included in the feature model and should be elaborated separately. The correspondent content is nonetheless included for explanatory reasons.

- A field representative of a manufacturer of computer hardware receives a message on his PDA, telling him that *company X* is planning to replace their complete system in the next quarter for which they have budgeted substantial financial resources.
- He immediately enters this information in the system, i.e., master data of the potential customer (prospect), including budget estimation, description of sales opportunity, sales volume, and timeframe of the opportunity. In this respect he uses a service of the *Sales Scenario* application that provides functionality to enter information for what ever kind of business partner data (***Account Management***) and ensures that each business partner is held only once in the system. Hence, the employee is sure not to enter already contained data.
- After subsequent evaluation and go/no go decision by sales management, another employee of the sales office creates an offer using the ***Quotation*** functionalities, rates a quotation template based on the sales opportunity.
- Based on categorisation of the prospect in a ***Customer Group***, estimated sales volume, and sales probability, resulting in an overall ***Customer Rating***, an ***Individual Pricing*** strategy is used to calculate a discount for the customer, which is included in the quotation.
- After the sales office has contacted the customer and received an order, the system automatically converts the quotation into an order upon mouse click using the correspondent ***Sales Processing*** functionality.
- To check the creditworthiness of the customer, an (optional) ***Credit Check*** is performed during sales processing by interacting with the ***Account Management*** module.
- An ***Availability Check*** is performed to check warehouse stock for required capacities.
- The availability to promise check requires interaction with warehouse management (***Stock***). In case of ***Multiple Stocks*** only those warehouses sufficiently close to the shipping address are included.
- If ***Payment*** is to be integrated into the process, it would be activated automatically upon creating a binding sales order. Depending on the method of payment offered by the system and selected by the customer, an automatic debit transfer from the customer's account can be triggered (***Payment Card***) or an invoicing document can be attached to the delivery (***Cash On Delivery***).
- The order status is set to "completed" by an employee as soon as it is delivered to the customer.
- In case an open or already completed sales order is returned by the customer (***Returns***), an ***Approval Process*** will be triggered, which involves the sales management for commitment.

All described process steps where concrete documents are involved are influenced by available communication channels. This determines, e.g., whether an invoice is created separately using Microsoft Word, printed from the system, or automatically sent via e-mail.

3.1.3 Use cases

A use case diagram illustrating the Sales Scenario, which is directly related to the user roles identified above, is shown in Figure 4: Use Case Diagram for the Sales Scenario. It gives an overview about common use cases that are derived from the process flow shown in Figure 4. In addition to the use cases a systematic partitioning into logical packages is denoted.

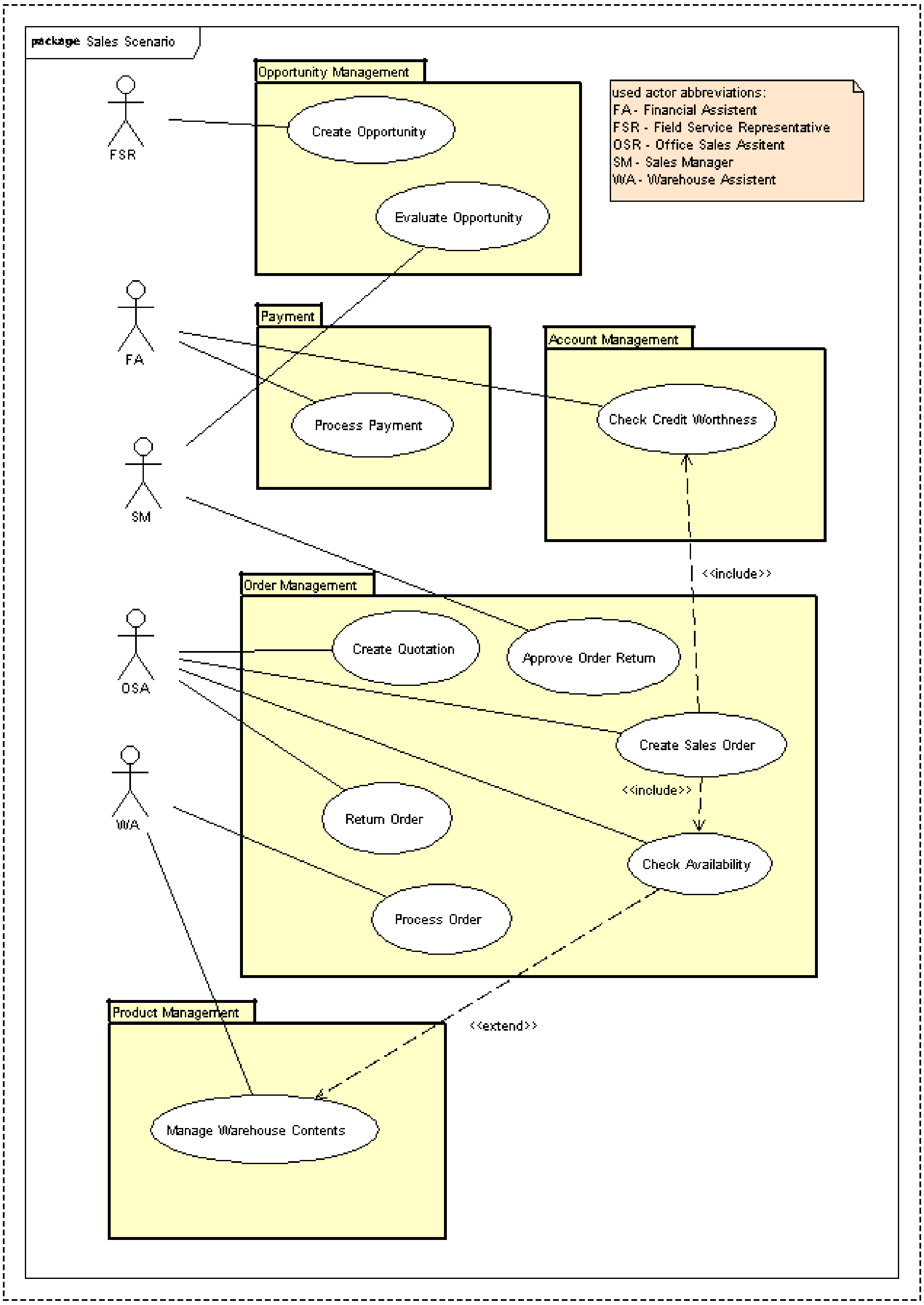

Figure 4: Use Case Diagram for the Sales Scenario

3.1.4 Features and Feature/Family Model

In this Section, a short description of some of the significant features of the feature model illustrated in Figure 2: Feature Model, is given. It should help to complete the understanding for the described use case.

Opportunity

The Opportunity feature manages the evolution from an initial customer contact (in the example for replacing the computer system) towards a decision to send a Quotation to the customer. This includes an action log of various steps of customer engagement (e.g., mail, appointments, etc.) that should develop an Opportunity so a decision by sales management is possible. Typical behaviour for an Opportunity is, for example, its evolving state (active, stalled, failed) and predicted value.

Payment

This feature covers payment processing. It manages receipts of incoming payments and outgoing payments. A direct connection to with banking systems is appropriate.

Customer Groups

This feature enables categorisation of customers in groups. Current division distinguishes between enterprise customers and end customers. Customer groups influence ***Pricing*** and offered product portfolio ***(Product Management)***. For instance ***Returning Customers*** or ***Long Term Frequent Customers*** may be offered certain discounts. For ***Competitors*** certain articles are either not offered at all or at special rates. ***Prospects*** subsume prospective (potential) customers for offered products.

Product Management

Includes management of product portfolio and possibly covers one or more product lines. Products are registered in the system and augmented with meta-data. In the absence of warehouse management (***Stock***), inventory/capacity is kept as additional attribute per product.

Stock (Single)

A stock manages product inventories in a warehouse. Here exists a direct relation to inventory/capacity attribute of products described in Product Management. Additionally, metadata concerning information about storage location (aisle, shelf, etc.) of individual products is managed here.

Stock (Multiple)

Multiple warehouses are not trivial with respect to calculating the total inventory. Delivery processes and storage of different goods often depend on several warehouses rather than on only one single.

Customer Order Management

Sales Processing manages processing of sales orders from customers and hence includes the whole life cycle of such an order, beginning at the point of entry over approving processes (always tracking status indicators) up to the completion of orders after arrival at customer's location. If available, this feature strongly interacts with ***Delivery*** and ***Payment*** as well as other features on the same level (e.g. ***Availability Check***).

Quotation interacts with ***Pricing Strategy***, ***Customer Groups*** and ***Customer Rating***.

The feature ***Approval Process*** covers the indicated process and is triggered by ***Quotation***, ***Sales Processing***, and ***Returns***. It hence extends these features and is only useful if at least one of these features is activated.

The ***Availability Check*** is triggered by ***Quotation*** and ***Sales Processing*** and requires Stock for checking availability to promise.
The ***Credit Check*** is triggered by ***Quotation*** and ***Sales Processing*** and proves the creditworthiness before creating a quotation as well as before accepting a Sales Order.
The ***Returns*** feature depends on the ***Approval*** feature.

3.1.5 Feature constraints and interaction in detail

The features introduced in the above sections, contain several constraints and interact with each other. Besides the textual description of these feature relations in the preceding sections, this section serves with a detailed and more technical specification of them. The syntax used in the constraint descriptions, conforms to the Prolog dialect *pvProlog*. Further information to pvProlog can be found in pure::variants documentation on [PS@ps].
As short explanation for the purposes of this document, it has to be stated that the relation `hasFeature(fN)` describes a usage relation of the defining feature to the feature `fN`, e.g. `a: hasFeature('b')` explains, that the implementation of the feature `a` may use implementation parts of feature `b` as far as `b` is present in the current variant. The relation `requiresFeature(fN)` states that a feature `a` depends on the presence of a feature `b` for functioning.

1) Availability Check

```
Availability Check: requiresFeature('Stock') and
(hasFeature('Production'))
```

Remark: The second part of the constraint references CS2.

2) Cash on Delivery

```
Cash on Delivery: hasFeature('Communication')
```

3) Complex Products

```
Complex Products: requiresFeature('Stock')
```

Remark: This constraint references CS5.

4) Delivery

```
Delivery: requiresFeature('Stock')
```

Remark: This constraint references CS3.

5) Frequent Customer

```
Frequent Customer: hasFeature('Production')
```

Remark: This constraint references CS4.

6) Individual Price

```
Individual Price: requiresFeature('CustomerGroups')
```

7) Invoicing

```
Invoicing: hasFeature('Communication')
```

8) Payment Card

```
Payment Card: hasFeature('Communication')
```

9) Quotation

```
Quotation: hasFeature('PricingStrategy') and
(hasFeature('CustomerGroups') and (hasFeature('CustomerRating') and
(hasFeature('ApprovalProcess') and (hasFeature('CreditCheck') and
(hasFeature('Communication'))))))
```

10) Returning Customer

```
Returning Customer: hasFeature('Production')
```

Remark: This constraint references CS4.

11) Returns

```
Returns:            requiresFeature('ApprovalProcess')          and
(hasFeature('Communication'))
```

12) Sales Processing

```
Sales      Processing:      hasFeature('ApprovalProcess')       and
(hasFeature('Delivery')     and      (hasFeature('Payment')     and
(hasFeature('AvailabilityCheck') and (hasFeature('CreditCheck') and
(hasFeature('Communication'))))))
```

3.1.6 Change Scenarios (CS)

Change scenarios describe situations and/or conditions under which the initial functionality of the system is changed. Such situations may or may not be predicted at the time of the design of the system.

13) CS1: New payment option (Invoicing)

The company, that uses the described system, allows their customers new payment freedom by establishing the option to pay by invoice**.** In this case, invoices are sent separately from the sales orders, allowing later settlement. Hence, such a feature introduction extends the sales process. It is visible to users of the system and changes the requirements on the system.

14) CS2: Integration of production management

The production management is meant to act like a connection to the production process. In this scenario such a management does not claim to cover any specific details, but provides triggering and monitoring functionality. This feature has an impact e.g. on the ***Availability Check***, in such a way, that products, which are not anymore in stock, are newly requested with an appropriate quantity. The monitoring functionality is highly crosscutting, the feature itself changes the requirements and is externally visible.

15) CS3: Integration of delivery management

A delivery processing feature beneath ***Sales Processing*** strongly interacts with ***Stock*.** Multiple stocks might make the delivery process very complex e.g. when different parts of a sales order come from different stocks. In cooperation with the ***Delivery***, the order would be split into separate orders for each involved warehouse, which have to be scheduled appropriately. This feature can be understood as an internal variation, invisible to users. Delivery processing with multiple stocks may be completely transparent to a user of the system. This feature also changes the requirements on the system.

16) CS4: Additional customer groups

A finer categorisation of customer groups, in order to encapsulate additional functionality, would be another change scenario. E.g. the distinction into ***Returning Customers*** and ***Frequent Customers*** would have effects on ***Product Management***, by triggering an increase or decrease of production. In addition the definition of customer groups may affect the selection of pricing strategies, depending on the group a customer is assigned to. This feature is a combination of internal and external variation. The definition of customer groups must be performed by the user of the system, while processes like pricing strategy selection relying on customer groups may work transparently to the user.

17) CS5: Complex products

The management of complex products, e.g. product bundles as special offers or even real complex products consisting of different parts (cf. computer hardware shop), has an impact on warehouse management, i.e. stock of certain products is composed of stock of included products. This variation is externally visible and changes the requirements.

18) CS6: Pricing Strategy Selection

According to properties of customers (i.e. affiliation to a certain customer group) or regional settings an overall pricing strategy for price calculations can be selected. This pricing strategy may be fixed for a certain product instance of the system and selected at the start time. The variation is transparent to users and affects mainly the implementation of the system instance. Technically the selection of a pricing strategy may be handled via special plug-in classes that are loaded into the system at start-up.

19) CS7: System Architecture Change

The system may be implemented based on different architectures. Two possible software platforms may be J2EE and Spring. In addition the user interface may come in various flavours. Conceivable are a web front-end and a tailored SWT-based client application. Variations of this kind affect the development process on architectural level and are visible externally.

3.1.7 Additional Requirements

Behaviour of business processes is **variable**. Some means to model/configure this variability are required. A special challenge is the **crosscutting** nature of this behaviour variability across different modules.

For example, in *Payment* / Invoice Processing must guarantee that the invoice will not be issued to a customer unless all products with the required number are in stock. Action semantics could be used as an enabling technology for specifying these actions at PIM level. Since some of these issues are rather technical whereas others have a business background, special attention must be paid on the **abstraction level** to model the variability.

3.1.8 High-level Architecture

A first high-level architecture is depicted in Figure 5 in Fundamental Modeling Concepts notation [FMC]. It outlines the key entities and their interactions / interfaces according to the features identified above. It is meant as a basis for elaborating the design and development of the example application.

The shown block diagram clarifies the two main feature types of the example product line: well modularisable, coherent features on the one hand and wide spread, crosscutting features on the other. Exemplary for the first feature type are *Product Management* and *Payment*, while the *Communication* feature belongs to the second feature type. By considering the latter, a certain point-of-view problem arises. While the *Communication* feature itself is well coherent from a functional viewpoint – necessary functionality can be implemented in few classes that are independent from the rest of the implementation (as shown below) – it is rather wide spread concerning the invocation of this functionality. Each communication layer call has to be introduced in proper places of other modules, well distinguishing the kind of communication. This cross cutting characterization is illustrated with the many links that connect other modules with the communication layer.

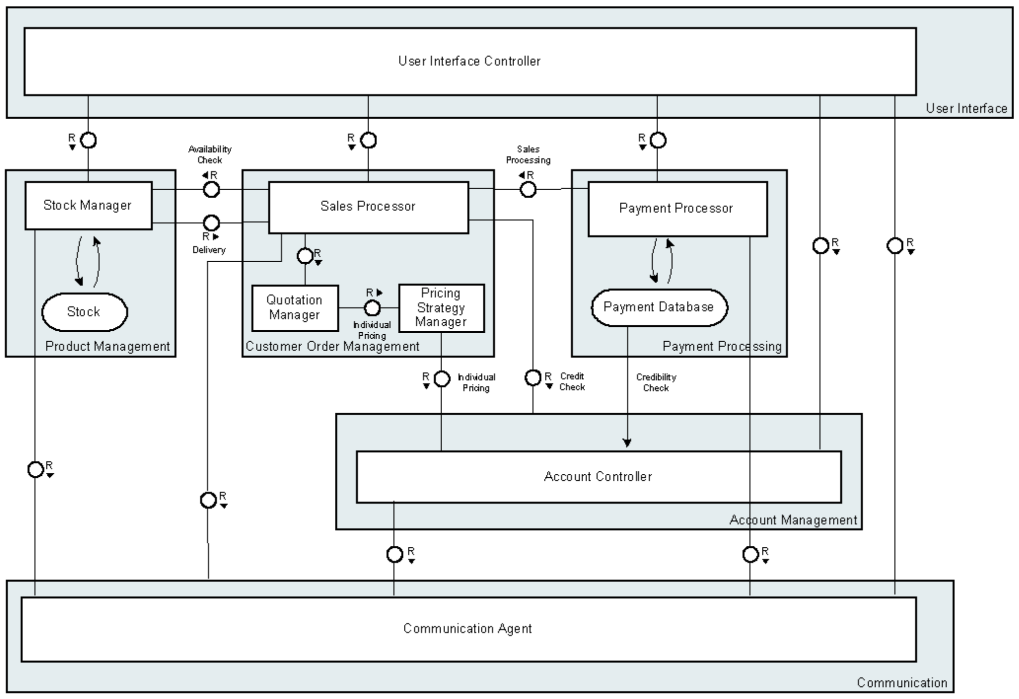

Figure 5: Architectural overview

Immediately noticeable in the above illustration are also the many connections of *all* feature implementations to the user interface (UI). Each feature has to be represented (if desired) for interaction with the user that shall handle the resulting application in a proper way. Due to the direct and indirect dependency of the user interface to the rest of the application, the term *crosscut* gets another meaning within this consideration. From the UI perspective it describes a feature's impact to UI components, such as screens, widgets or interaction methods. While functional interaction of feature implementations on a non-UI level is solvable from a strict technical perspective, a semantic consideration of this impact to the UI is indispensable. The reason for this semantic treatment is the necessity of getting sensible interaction possibilities in each product line variant.

3.2 Challenges

Based upon the definition of change scenarios above it is possible to define the following research challenges. Because of the same project scope the challenges itself have similar appearances in all case studies, while the actual details for each of the challenges is different.

3.2.1 Traceability from Requirements to Code

The development process of a software system in the scope of AMPLE reaches from the definition of requirements in an appropriate way over model creation and transformation to actual code generation. A methodology is needed to track artefacts that are created and/or transformed throughout the whole process Each time a requirement is fulfilled, artefacts of different abstraction levels are created and must be bound together to form trace paths that provide enough information to enable tracing. Every trace record should be annotated with information on the decisions that led to the tracked transition. Artefacts might be abstract concepts like packages, activities or compilation units or things from the lowest abstraction level like methods, fields or declarations.

All tracing information can be understood as directed, annotated graphs. These graphs may grow very large and complex. In this context answers are needed about how large such graphs can grow in real-world systems and up to which complexity these graphs can be handled by today's computational hardware.

3.2.2 Traceability from Code to Requirements

The previous section describes the challenge of forward tracing requirements to actual code fragments. Vice versa, the tracing from changes of code fragments or other artefacts generated during the development process back to the requirements which were the origin for the creation of these artefacts is interesting likewise. Usually not all artefacts created by some kind of automatic generator are ready to use, either because they cannot be generated entirely or the output of certain generator does not accomplish certain expectations of a developer. In each case manual intervention is necessary to eliminate existing shortcomings. In this context it is interesting what implications such changes cause and to which degree they affect the coverage of all requirements. In the worst case such changes lead to unexpected system behaviour that may render it unusable or at least unacceptable for deployment to a customer.

3.2.3 Tool Integration of Traceability Concepts

Currently, traceability information is often managed in isolation from the systems that actually create this information. Different tools have to be handled in parallel to make meaningful use of this information. Ideally, the presentation of this information should be integrated in the UI of its originating tools. In an Integrated Development Environment (IDE) covering several development phases, traceability information could be leveraged to such an extent that all related artefacts of each and every trace record can be immediately followed and examined. This would greatly improve user experience in comparison to current solutions. An interesting question related to backtracking of trace paths is how to visualize to human users certain problems caused by manual changes.

3.2.4 Feature Tree to Model Transformations

During the development feature tree must be bound to models to achieve a reduction of the abstraction level. Elements in a feature tree are mapped to model elements, e.g., in a UML model. To perform such a mapping, components must be defined that can be used as mapping targets for feature tree elements. In the described scenario above entities like *products*, *orders* or *stocks* could be understood as components. Following such an approach sub trees from a feature tree could be transformed into model elements by "collecting" appropriate components. It is to examine if such an approach really works in practice and if so, if components can be defined and described in UML to enable reuse of elements in model transformations.

3.2.5 Mapping of Solutions to Several Platforms

A common use case when utilizing software products (either based on dedicated software product lines or not) is the deployment to different platforms. The product itself or respectively its feature tree can be understood as staying constant, while internal changes must be made to allow the functioning of the product on a certain platform. Examples for such changes are the complete recreation of the binary code via a cross compiler or the adaptation, in- or exclusion of certain code fragments to face constraints of the target platform. Other parts of the product may be realized in a way specifically dedicated to each target platform. As an example the Sales Scenario application may be implemented based on different enterprise platforms like

JBoss/Seam or the Spring Framework or purely based on client technologies providing user interfaces realized in Swing or SWT.
Especially interesting in this challenges is the way how mappings to different target platforms can be specified in models and how the characteristics of different platforms can/should/must be handled.
An extension to the already mentioned issues is the case where certain target platform does not allow the inclusion of certain features because of some missing technical prerequisites. The information that enables the treatment of such issues must also be specified in the appropriate models.

3.2.6 AO to simplify Tracing Granularity

The concept of aspect orientation offers the chance to reassemble cross-cutting concerns into separate modules. Examples for such cross-cutting concerns would be system monitoring or the introduction of access rights to certain system parts. Introducing such modules for encapsulating the appropriate functionality has two important consequences for enabling tracing. First, a new dimension in the tracing space is introduced; in addition to objects now aspects have to be covered too, which increases the effort when tracking changes. In contrast changes to the functionality of one of the mentioned examples for cross-cutting concerns need not to be traced to various fragments scattered over the whole project, but to one single module, which simplifies the task of tracing. Both consequences are of contrary nature, advantages and disadvantages of both must be examined.

3.2.7 Architectural Level AO

In aspect-oriented programming code fragments that scattered over the whole code base are collected and integrated into one separate module. In this context the question for benefits of raising the idea of aspect-orientation to the architectural level is interesting. Is it possible to extract functionality from models on architectural level into separate modules? What interfaces on architectural level must be defined to be able to weave this functionality back into the model by a model weaver before performing model transformations?

3.2.8 Anticipated Changes vs. Unanticipated Changes

When adding features to existing systems, a differentiation between anticipated features and unanticipated features must be made. The basic question that needs to be answered here is the question whether the additional integration of a feature was planned or not. This is sometimes also referred to as Variation in Time vs. Variation in Space. For the former case of anticipated changes conventional object-oriented techniques seem to be sufficient. APIs for user-defined extensions or complete plug-in interfaces may serve as examples for the state-of-the-art. For the latter case object-oriented techniques may not be flexible enough, here AO techniques may come into play. So far cases for possible applications of both techniques seems to be clear, more interesting in this context is the real delimitation between both. Is it possible to state in which cases aspect-orientation could/should/must be favoured to object-orientation? In principle this question may be answered based on the availability of hooks for extensions, but is a sheer delimitation based upon these hooks always reasonable or are there other factors that must taken care of?

3.2.9 Binding Time

During the systematic development of product lines, a designer has to cope with variability. Variability describes the whole solution space in which all possible products are contained in. Variability in certain product features must be resolved at a certain point in time.

At a first glance there seem to be four different moments that can be used for variability resolution:

- **Design time**: a feature variant is bound by design in applicable models, modules etc.
- **Compile-time**: a feature is included when building a product. This feature cannot be changed or deactivated later. As a consequence each product of a product line does have a separate code base.
- **Deployment-time**: a feature is included when a product is deployed to a stakeholder. In this case the code base for all products is the same, but a different set of compiled features are deployed.
- **Start-up time**: all products of the product line are delivered in the form of a *master product*. A set of features is selected at start-up time (i.e. by activation/deactivation in a configuration file) and cannot be altered at runtime. The actual instance of a product line is formed during the start-up phase of the system and is changeable only once in its lifetime.
- **Runtime**: a set of features is selected at runtime of a system. In this case a “product” exists at a certain point in the lifetime of a system and be changed multiple times.

Ideally the actual binding time (the point in time at which some variability is resolved) is user configurable. Therefore a system supporting the design of product line should offer the possibility to configure the binding time of a certain feature. Creating a system the supports the handling of resolved and unresolved variability is a big challenge, a suitable approach to this is missing. Is it possible to create techniques for model transformations and/or code generators that leave a user more freedom in defining binding times? In addition, is it possible to even generate “recommendations” for choosing binding times?

3.2.10 UI Integration

The introduction of new features to the scenario described above implies the development of new components for the user interface. It is obvious that the necessity of an examination of the influence of feature additions to the design of the user interface arises. Up to which degree may a user interface be extended to cover feature additions/changes and to maintain its usability? At what point in time must a user interface be redesigned to reflect the changes to a system and to reconstitute usability? In conjunction to these questions the possibility of generating user interfaces for products in a product line should be examined.

3.3 Research Areas and Solution Proposals

The “Sales Scenario” case study provides a number of access points for applying Software Product Line technologies with both, Aspect-Oriented and Model-Driven extensions.

3.3.1 Bridge the Gap between Feature Models and Variability in Architecture Models

When implementing the Sales Scenario in a classical model-driven approach, one challenge is to transform the variability from rather high-level feature models to relatively concrete architectural artefacts exposed using architecture description languages (ADL) or arbitrary domain-specific languages (DSL). An interesting approach for tackling this issue is currently investigated in WP2 and is called “Variability Pointcut Language”. The Sales Scenario would be an ideal test case for a proof of concept.

3.3.2 Looser Coupling between Aspects and Base Code

Controlled extensions to basic platform functionality are a key requirement, not only for business software, in order to keep application suites manageable and evolvable. One road block for the large-scale industrial adoption of AOSD techniques is the strong coupling between aspect and base code, which complicates upgrade releases and allows customers to create unsolicited extensions to core components. Research in explicit aspect interfaces could address this problem. The general idea would be to abandon the strict obliviousness property of AOSD to some degree and to come to a more constrained "grey-box" model of exposing join points.

3.3.3 Flexible binding time of variability

Another key challenge is the flexibility to arbitrarily change the binding time of some variability. This would allow developers, for instance, to arbitrarily decide during product line development to bind the variability regarding certain features earlier or later if requirements for flexibility or performance change. Currently, the choice for a certain variation mechanism ultimately determines the binding time. Further research on Model-Driven techniques seems promising since MDD allows binding variability to code at various points; it could even produce runtime-interpretable DSLs out of some models to defer binding of variability to runtime.

4. Case Study: Space Weather Decision Supporting System (SWDSS)

4.1 Domain and scope

Space Weather (SW) is the combination of conditions (occurrences/events) on the sun and in the solar wind, magnetosphere, ionosphere and thermosphere, which can influence the performance, integrity and reliability of space-borne and ground-based technological systems. For example, the degradation of sensors and solar arrays or unpredicted changes in the on-board memories, can often be associated with SW event occurrences. Moreover, it is being acknowledged by scientists, that SW influence on earth is capable of affecting human life and health.

The availability of an integrated solution containing Space Weather and specific S/C (Spacecraft) onboard measurements' data, allows performing online and post-event analysis, thus increasing the S/C Flight Controllers' ability to react to unexpected critical situations and indirectly, to enhance the knowledge about the dynamics of the S/C itself.

To provide such capabilities, a decision support system was envisaged – the Space Environment Support System for Navigation and Telecom Missions (SESS). The main objective of SESS is to provide the FCT (Flight Control Team) with accurate real-time information about the ongoing space weather conditions, spacecraft onboard measurements as well as Space Weather predictions. This decision tool supports the FCT decision-making process on how to react to ongoing space weather conditions and event occurrences. Furthermore, the system increases awareness and understanding of space weather and its effects on spacecraft performance. All together, it leads to operations that are more efficient and to a quality increase of services as well as improving the safety of the payloads.

SESS is a multi-mission system in the sense that, with the same infrastructure, while Spacecraft data is only available to the FCT that monitors that Spacecraft, Space Weather data is centrally collected and made available to all FCT.

A similar product was developed SEIS (Space Environment Information System) based on the same architecture that was a mono-mission system; i.e. collecting SW data and S/C for a single mission and FCT.

These two systems (SESS and SEIS) are what ESA calls Operational Prototypes; i.e. prototypes that are operational to be used by the FCT, but do not need to comply with ESA's qualification processes for operational systems. They are a proof of concept.

ESA has now requested the development of a fully operational product (SEISOP) for which a proposal is being prepared.

The next sections will present an overview of these SWDSS focusing on the SESS version.

4.1.1 SESS Main Features

SESS system is a multi-mission decision support system capable of providing near real-time monitoring and visualization, and offline historical analysis of space weather and spacecraft data, events and alarms to Flight Control Teams.

SESS core services are:

- Space Weather and Spacecraft data integration reliability;
- Inclusion of Space Weather and Space Weather effects estimations generated by a widely accepted collection of physical Space Weather models;

- Near real-time alarm triggered events, based on rules extracted from the Flight Operations' Plan (FOP) which capture users' domain knowledge;
- Near real-time visualization of ongoing Space Weather and Spacecraft conditions through the SESS Monitoring Tool;
- Historical data visualization and correlation analysis (including automatic report design, generation and browsing) using state-of-art Online Analytical Processing (OLAP) client/server technology - SESS Reporting and Analysis Tool.
- Metadata repository which provides the "glue" for all concepts, methods and denominations used in the system.

The general architecture for the SESS is shown in Figure 1.

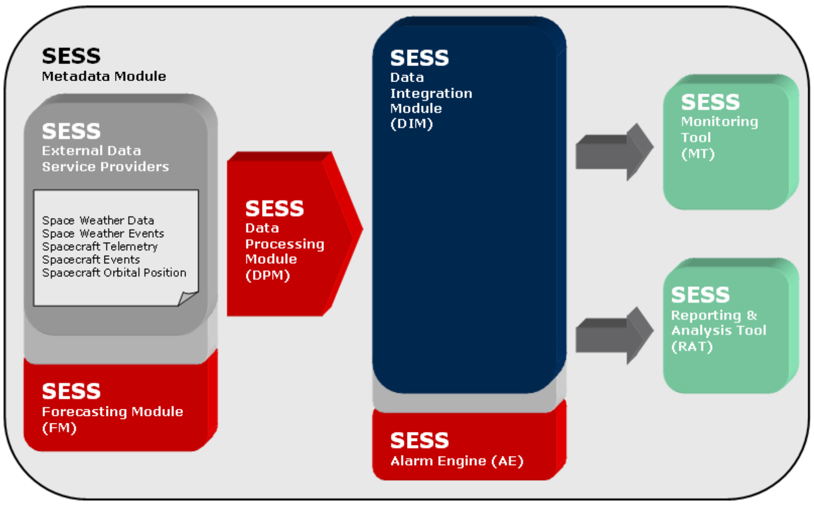

Figure 6 – The SESS general architecture

4.1.2 SWDSS in Ample

Due to the high complexity of SWDSS, in Ample we will focus on the DPM (a complex system per se), which is connected with the metadata repository that provides the general "glue" for the whole decision support system.

4.1.3 Data Processing Module (DPM)

4.1.3.1 Concept

The DPM - Data Processing Module, is responsible for all file retrieval, parameter extraction and further transformations applied to all identified data, and validation mechanism ensuring that all online and offline availability constraints, are met whilst having reusability and maintainability issues in mind.
In Figure 2 we can observe the sub-modules of the DPM and its interaction with the metadata repository and the other components.

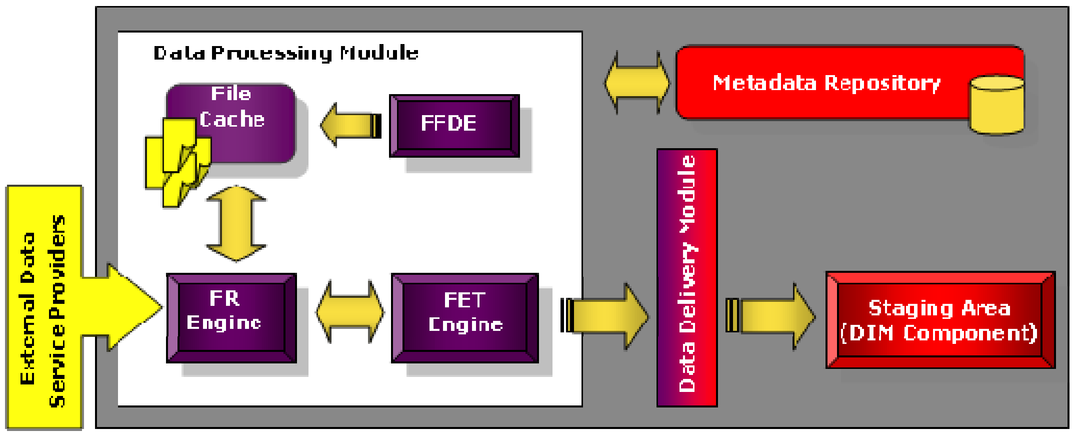

Figure 7 – The Data Processing Module of the SESS system

The features of the DPM are:

- The FR (File Retriever) Engine acquires near real-time data from multiple external Data Service Providers: the files to be downloaded, the location and all configuration needed for this purposes is available from the Metadata Repository;
- The FR Engine caches locally the retrieved files;
- The files are sent for extraction and transformation into the DPM FET (File Extract and Transform);
- The FET's processed data is then sent to the Data Delivery Module and integrated in the system's Data Warehouse so others SESS modules can analyse the normalized data;
- Since the retrieved files can evolve (i.e. format changes), the processing tasks definition can be defined / updated in the FFDE (File Format Definition Editor) application.

The data is retrieved from external service providers (scientific and industrial communities) isolated across different locations, in a variety of file formats (usually codified as ASCII data files, each with its own data structure or file format) accessible via common web protocols (HTTP, FTP).

The data files' distributing service, for the majority of the institutions involved, may not be reliable enough for the SESS system's near real-time demands and/or for historical data needs (on most situations and mainly for data with high sampling rates, historical data files are kept online only for the past 30-60 days), so a local caching mechanism was implemented.

The FR component relies on XML based configurations and scheduling definitions available on meta-information. It provides also a HMI (Human Machine Interface) front-end, which allow users to visualize the log of all activities and FET configuration options.

A thorough analysis of retrieved files indicated that different files have their own format for representing data (e.g. column oriented, table oriented, different data and date representations). Moreover, the problem becomes even more complex when a single file provided by a given Data Service Provider can be found with distinct file formats).

As maintainability, reusability and ease of use are some of the primary goals of the entire Data Processing Module, each file has an associated File Format Definition file, represented as an XML file, which contains all required information to process the data file retrieved by the FR engine. In this way, the FET Engine is the next

component to be found in the DPM data pipeline. Just like the FR application and to allow better tracing of its processing activities, FET also performs logging of all main tasks into a text file.
The FET application is responsible for applying a user predefined set of ETL operations to each class/type of Provided Files. The specification of the ETL operations is held in a File Format Definition (FFD) file in a declarative form. Each FFD contains the following specifications:

- Sectioning information;
- Definition of Single Field and Table Fields;
- Sequence definition of transformation operations;
- Specification of the delivery data.

The Data Processing Module is also a Metadata driven module (since SESS uses shared meta-information), i.e. whenever required, its components perform transparent read/write operations on the shared Metadata, stored on a centralized Metadata Repository.

4.1.3.2 Solution Proposal for DPM

4.1.3.2.1 The FR component

The File Retriever constitutes the interface with all external data distributing services in the SESS system. This component is responsible for the retrieval[1] of all identified files, dealing with both space weather and spacecraft data at the different Data Service Providers'[2] locations, being able to cope with remote service availability failures and recover lost data due to access unavailability. For minimizing this last issue, all retrieved data are afterwards stored into a local file cache repository. Once stored, the data files are immediately sent for processing – to the DPM FET component.
Figure 8 focuses on FR's interactions:

- DPM HMI (Human-Machine Interface) and FR Metadata configurability of the Data Service Providers;
- Data acquisition;
- FR file caching;
- Data forwarding to DPM FET.

The DPM File Retriever is a stand-alone full multi-threaded application whose sub-components are detailed in following sections.

[1] Communication with these components is achieved using Web Services interfacing layers.
[2] Data Service providers are all entities (research institutes, universities, international organizations), which freely distribute space weather or spacecraft data through their servers.

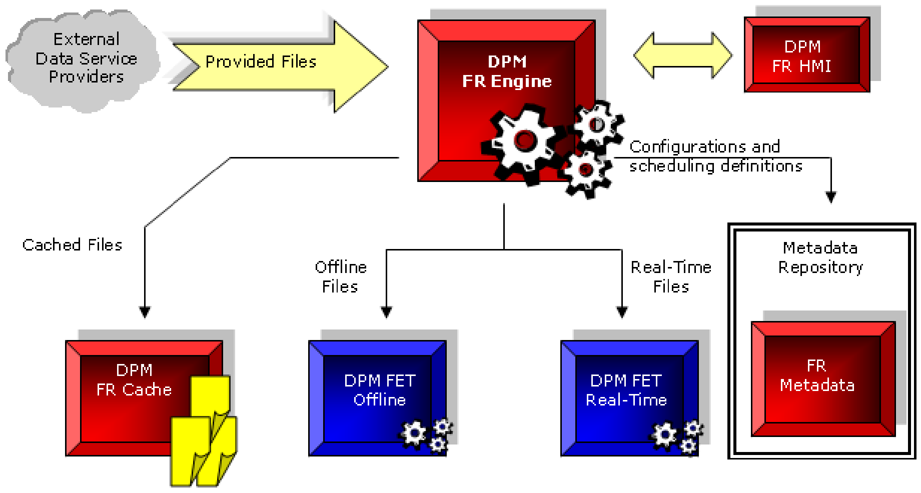

Figure 8: The Data Processing Module's File Retriever sub-component data fluxes

4.1.3.2.2 DPM FR Engine

The File Retriever Engine acts as a server component and is a retrieval scheduler, capable of using standard protocols (such as FTP and HTTP) and web services[3]. This component allows communication among different heterogeneous components, thus easing any integration task to be performed. The primary goal of using such infrastructure is to transparently wrap custom interfacing mechanisms with each specific data generators, so that the File retriever Engine is able to use a single common request-retrieve API common to all components.

To improve performance, the FR Engine contains as many schedulers as the number of data source providers[4]. Provided files are classified as real-time or summary (both types contain a temporal sliding window of data). While real-time files (e.g. files available every 5 minutes) offer near real-time/estimation data for a very limited time window, summary files (that can be available daily) offer a summary of all measures registered during that day (discrepancies between real-time and summary files contents are possible to find).

The FR Engine is prepared for the possibility of recovering previously unavailable data from the Data Service Providers. Different approaches are made, depending on the data type:

1. For real-time data, the FR will try to retrieve the file in the next 2/3 minutes: if it is not possible, the recover attempt will fail and be discarded, since other new near real-time data is available;
2. If the unavailable data refers to a summary file, the recovery time window is extended to a maximum of 2 days; if this time has elapsed the file will be obsolete so the recover attempt will fail and be discarded.

For performance and improvement purposes, the processing weight of all these functionalities is reasonably reduced.

4.1.3.2.3 DPM HMI

[3] The DPM File Retriever Web Service is a Server component for interaction with external data providers.

[4] Each file to be retrieved (and its attributes, as defined by the user) will be assigned to its own producer thread.

The Data Processing Module HMI acts as a client component and provides front-end user functionalities. It constitutes the primary means of interfacing with the user, such as:

- Logging information (beside the usual preserved logging information, all active threads and their current status are also stored);
- Management operations for addition, deletion, and update of data service provider and provided files definitions.
- General configuration such as:
    - Commands for the FR Engine start/stop Data Service Providers and file retrieval and processing;
    - Commands for the FET Engine start/stop data processing;
    - Retrieval files caching.
- Status information about current FET servers being used,
- Configuration of the FET servers to be used in file processing, identifying for every file which FET engine shall be used and determines if a Round-Robin (Figure 9) or Specific Routing (Figure 10) policy shall be used.

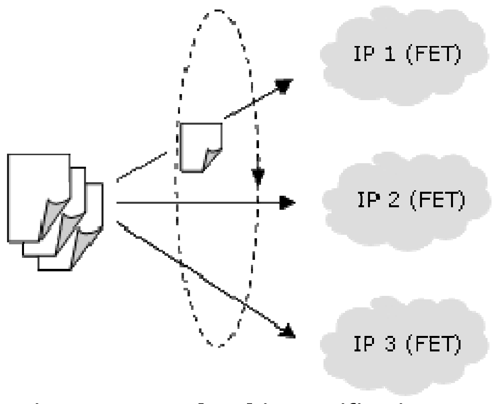

Figure 9: Round-Robin specification

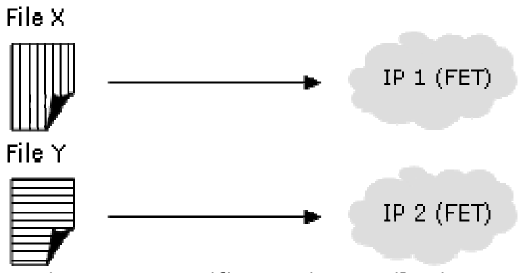

Figure 10: Specific Routing attribution

4.1.3.2.4 DPM FR Cache

The DPM File Retriever Cache component is a local directory responsible for storing all relevant data files. To ease management and decrease space allocated by cached files, a simple MS Windows NTFS compressed file system is used. The cache mechanism is used for better traceability, system integration and data backup.

4.1.3.2.5 The FET component

The FET component is mainly responsible for the processing of the data arrived from the FR Module. This process includes performing extraction, transformation and data loading according to defined ETL scripts – File Format Definitions (FFD) for online/offline and summary data files.

The engine performs data transformation services on received unstructured data files from the FR component and applies the respective ETL definitions, thus ensuring all relevant parameters are made available to any data reader client.

The FET Engine is deployed as a Web Service and always a minimum of two instances should co-exist: one for near real-time processing of near real-time text files (each of the instance are capable to handle very larges text files) and an additional one for dealing with the offline archive processing. FET processes different types of data with different types of priority.

The FET service has been implemented as a fully multi-threaded Web Service, whose core functionalities lie in data transformation and delivery libraries. Figure 11 depicts the File Extractor and Transformer detailed architecture.

As Figure 11 shows, the FET component receives processing requests from the File Retriever, always under the form of pairs (unstructured input data file; File Format Definition). The process starts by creating a new thread to handle the request[5] improving overall performance. Whenever the FET detects an error due to the change on a provided file format, the system administrator is notified by email with the correspondent input and FFD. To process a file, the FET engine firstly uses the transformation library to apply the extraction and transformations defined in the FFD file and afterwards, the outputs are sent into the DIM's Data Delivery component.

In addition, all FET engine's actions are logged into a log file with the same format as all other components (by using a common console viewer it is possible to correlate and trace all FR and FET activities). The interaction between components is depicted in Figure 12:

- The FET real-time and offline engines receive FR's retrieved data associated with the own FFD's file identifier;
- FET interacts with the FET Metadata (FFD loading and caching) and then processes the data;
- Finally, the FET delivers the data to be stored internally at the system's DW.

[5] It is assumed that using multiple threads increases the number of simultaneous processed files.

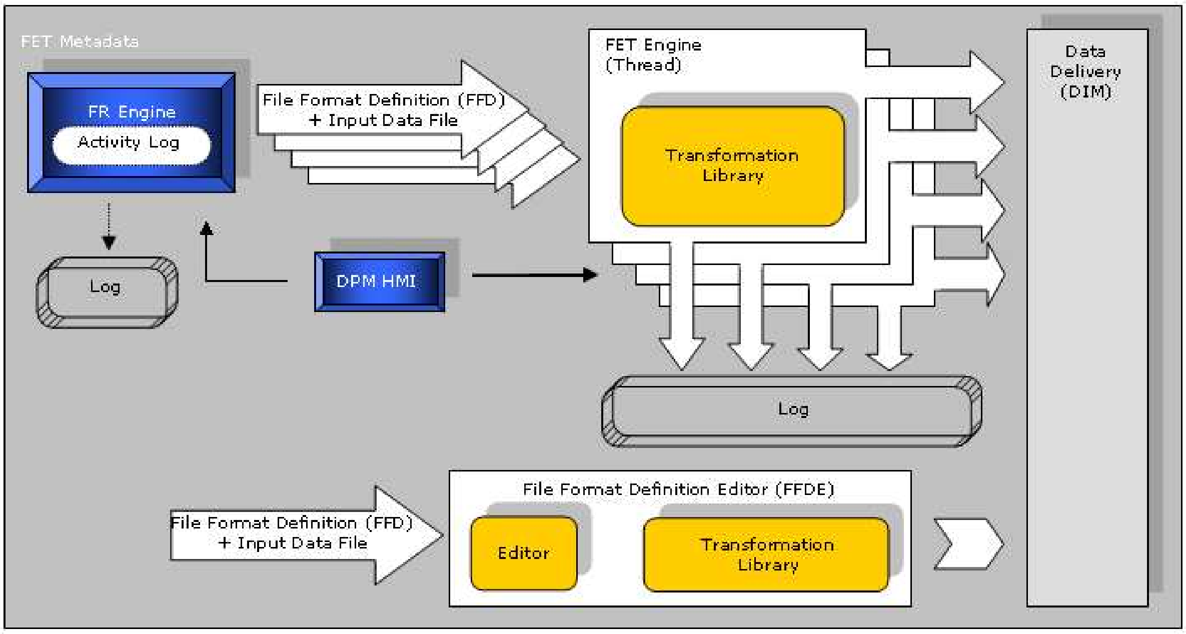

Figure 11: The FET detailed architecture view

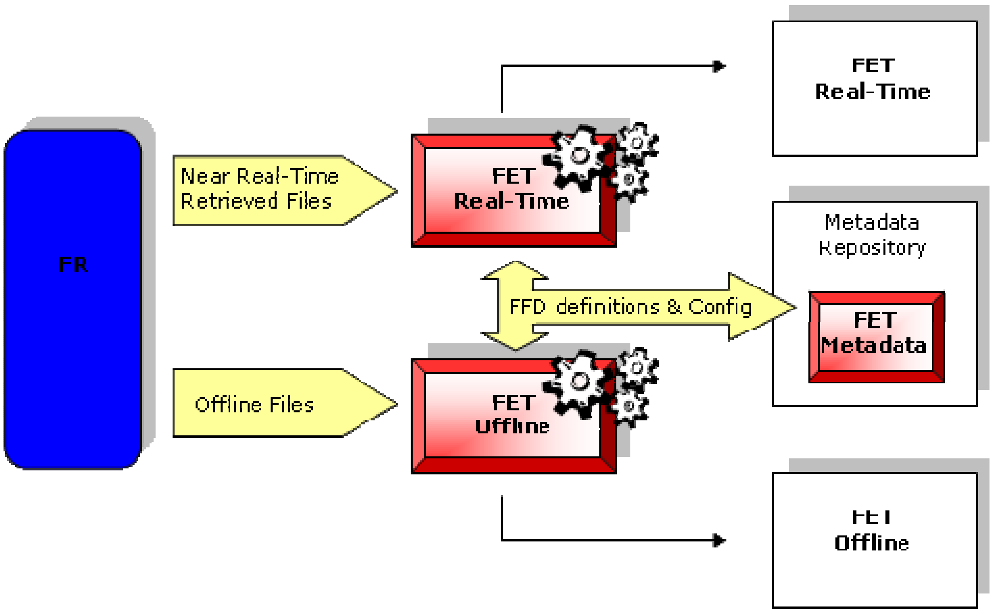

Figure 12: The File Extractor and Transformer (FET) sub components' interaction

4.1.3.2.6 FFD

The FET engine performs a set of high-level operations, which can be grouped together:

- Data extraction – This includes splitting the text file into *Sections* and performing extraction of relevant field values that are to be transformed and further loaded;
- Data transformation – The engine implements a series of transformation functions, ensuring that data is reformatted according to the output's needs;
- Data Loading – FET processing job ends when data is finally delivered to the Data Delivery Web Service.

FFD are also easier to maintain since they are not directly coded by the user, but defined in a client application (FFDE).

4.1.3.2.7 FET Metadata

The File Extractor and Transformer is also highly dependent on shared Metadata. Once again, this shared metadata is stored on the SESS Metadata repository.
This Metadata comprises:

[1] File Format Definitions – Collection of extraction, transforming and loading definitions for each of the files to be processed;

a) FET Supporting configuration files – General configuration data.

4.1.3.3 The Log component

The Log component (please refer to Figure 11) registers all the task activities for the DPM component. A common log format is respected by all application requiring logging activities. This way it is possible to open FET's and FR's log with an independent application log tool/component.
The Log is divided in server/client components:

1. Server: handles all the logic;
2. Client: provide a Human Machine interface that manages the graphical representation of the log files, allowing partial analysis of log files using multiple filters and querying mechanism.

4.1.3.4 The FFDE Component

The File Format Definition Editor, deployed as a stand-alone desktop tool application, provides File Format Definition management, allowing creation, editing and validation of FFD definitions stored in the Metadata repository or cached locally as XML files. To improve simplicity – and always based on an example file - the FFDE application supports the following concepts:

1. Section definition: can be contiguous or delimited;
2. Field definition (within each section): single field or tabular fields;
3. Data typing and validation rules for single and table columns;
4. A set of various Transformations is available;
    5. Developed as individual plugins for future extensibility.
    6. Enables graph visualization of the currently defined environment – for example allow the user to understand (visualizing it) in what output the input data will be transformed.
7. For Data Delivery purpose, the FFDE uses templates to easily (re-) use an already existing matching pattern and allow visualizing the Data Delivery simulation;
8. For determining the correctness of the FFD, the FFDE is able to process a set of sample files.

4.1.4 Metadata repository (MR)

In the SESS project, there are a significant number of components and sub-components that must deal with several types of data, particularly the DPM module. Besides the actual data in the databases, metadata – the data about data is also present. This metadata appears in various forms, namely: domain metadata and technical metadata.

Since every component will use its own formats, the repository must store the metadata in a generic format, being XML the natural choice. A validation process must be executed prior to the storage of the metadata. While the XML syntax is easily validated (several libraries support these functionalities), the XML documents structure and semantics must be validated. The standard way to do so is through a specific type of XML documents – the XML Schemas (and to obtain certain validations, SchemaTron is contained within the XML Schemas). This way, the two main document types of the repository are Concepts (XML Schemas) and the Instances (remaining XML documents). Follows some examples of domain and technical metadata concepts and correspondent Instances.
Domain metadata concepts:

- Space Agencies: “ESA”, “NASA”;
- Spacecrafts: “INTEGRAL”, “ENVISAT”;
- Space Weather Events: “Coronal Mass Ejection”, “Cosmic Ray Even”.

Technical metadata concepts:

- Data Source Providers: “KYOTO”, “NOAA”;
- Configurations: “Report and Analysis Tool”, “Data Integration Module”.

4.1.4.1 Solution proposal for Metadata Repository

Since the Metadata repository is an integral part of the DPM we need to address this module in more detail.

4.1.4.1.1 Metadata Repository architecture

The next figure depicts the internal layered architecture of the Metadata Repository, according to the features presented above:

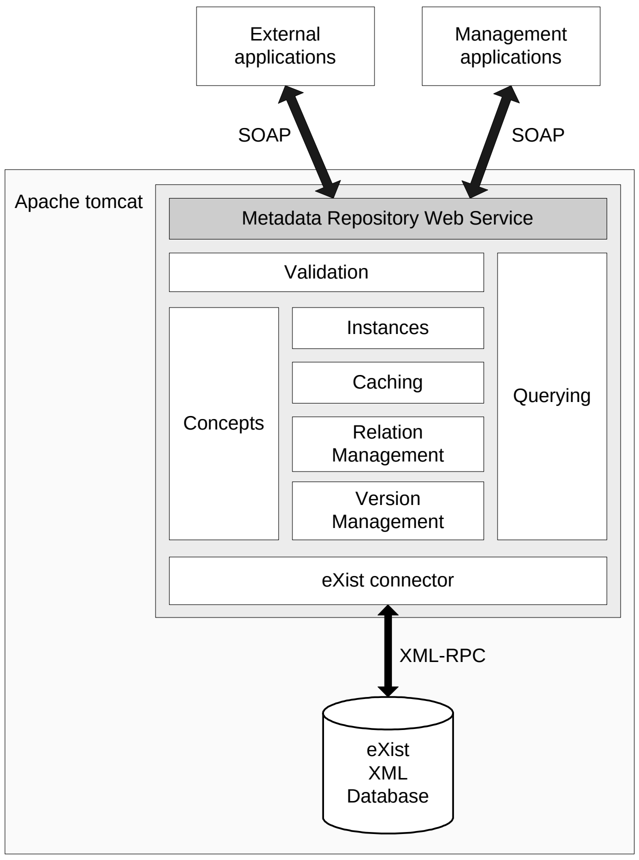

Figure 13 - Metadata Repository internal architecture

4.1.4.1.2 Functionalities

As explained before, the Metadata Repository shall support a set of basic functionalities for storing and querying metadata information, and a simple interface for easy integration among Information Systems. Additional features like instance relations and versioning provide added functionality and capabilities to the repository. In this chapter those main features will be briefly presented and explained.

4.1.4.1.2.1 Storage

The Metadata Repository must store both instances and concepts. As the repository uses eXist as the supporting XML database, a set of system collections are created in a database instance, for holding each of these information model resources. Upon storage the resources are indexed by the eXist database, for speeding up future queries to be performed using XQuery.
Besides the information model resources, other management resources are stored in the database. These are internal structures represented in XML for supporting concept and instance validation, management, versions and relations.

4.1.4.1.2.2 Instance Versioning

The Metadata Repository supports instance versioning. The versioning system is fully managed by the Repository, in a way that version numbers, creation and modification dates are assigned automatically to each version. Instance Relations
The metadata repository supports explicitly relating instances between each other's. This is done by defining in the concept schema an element of a predefined relation

type, with a special annotation to control relation cardinality (minimum and maximum) and identification of the concept for the target instances of the relation. Specific versions of instances can be referenced, or if the version information is omitted, the last version is referenced. The metadata repository guarantees the relations integrity: ensures that the target instance version of one relation always exists. This is achieved by a validation mechanism before storage, and by restricting that instances being referenced by others cannot be deleted from the Metadata Repository.

4.1.4.1.2.3 Validation

Validation is an important task for maintaining coherency in the repository. Before being stored, both concepts and instances are validated in various aspects.
Concepts definitions are validated in their syntax, and checked if the XML Schema and optional SchemaTron fragments are valid, in their definition and structure in the XML Schema complies with the concept rules (defined for all concepts, as explained above). Additional concept validations are also performed for specific elements control such as instance relation elements, where the existence of the target concept is checked.
Instances are validated in their XML syntax, and by their corresponding concept definition (the concept XML Schema and SchemaTron definitions are used to validate instances). Instance relations are also validated, checking if the target instance version exists in the repository, ensuring the reference integrity.

4.1.4.1.2.4 Querying

Once metadata is stored in the repository, querying mechanisms are provided to obtain metadata in various formats. By using eXist XQuery support, it is possible to write **queries** to be performed over the existing instances and/or concepts. XQuery provides the flexibility in which full XML documents or fragments can be fetched and outputted, featuring support for variables, control structures, cycles, etc. Queries can output XML documents, HTML documents, or other text documents. The recommended output is XML, as it provides greater flexibility for applications to consume or for further transformation.

In AMPLE queries will be created to return all the metadata required for the client tools to run, and transforms / transform pipelines will be created to generate automatic documentation in HTML from the metadata stored in the repository.

4.1.4.1.2.5 Caching

For speeding up the operations, the Metadata Repository contains an internal cache that maintains instance state in memory. Instead of querying all the time the eXist database, if the results to be obtained are available in the memory cache, they are promptly returned. However caching mechanisms add a data redundancy overhead, so the cache must be handled with care to avoid inconsistencies between the cached information and the real database information.

4.1.4.1.2.6 Web Service

The Metadata Repository is available in a form of a web service, for managing and client applications to connect to. By using web services, integration efforts of the Metadata Repository in an Information System are reduced.

4.2 Challenges

4.2.1 Traceability

During the system development, the project stakeholders need to have a fast and accurate process to check that requirements have been met during all phases of development and the relations of all produced artefacts during the different phases. A reliable mechanism to check all the consistency is also needed. The development methodology should collect information for any of the produced artefacts, to allow the possibility to check/preview the consequence of a feature changing on the whole system. The intervenients on all steps of the development cycles should be cleared identified.

4.2.2 Variability

The existence of different software sub-modules on the DPM arises the problem to manage all the possible different system architectures and the different system configurations. The system deployment should be able to be a flexible and easy process without a major effort on the need to adapt and customize the different modules.

4.2.3 Evolution

In spite of the fact that Space Weather Decision Supporting System has achieved a stable performance, the evolution of the system is always a requirement present to the development team. The different file retrieval mechanisms or the creation of an access control level, are some of the new functionalities that may appear. These possibilities lead to the need of flexible and strong methodologies to sustain the previous development process and to plan the future work.

4.2.4 Aspects

The usage of Aspects should bring benefits to the implementation of existing crosscutting concerns on the system. All the techniques and methodologies offered by the paradigm should be taken on attention to maximize the development cycle.

4.3 Research areas

Traditional software development methods of large software systems are not flexible enough to cope with the need for addressing rapid changes and growth of requirements to be satisfied. This dictates the need to improve the reusability of development assets during software systems engineering. Thus, Product Line Engineering can contribute to minimise costs and shorten time to market, if identification, representation and composition of variations is supported by software development practices in a semi-automatic or automatic way. Therefore, we believe it is important to exploit the synergies between aspect-orientation and software product lines to achieve this goal. While aspects form natural candidates to manage the crosscutting nature of variations, composition provides a promising means to trace, forward and backward, variations identified in the requirements of a product line to their implementation.

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Core assets are produced and reused in a number of products that form a family. These core assets may be documents, models etc. comprising product portfolios, requirements, project plans, architecture, design models and, of course, software components.

In this case study we use a Data Processing Module system as a software module that can be used in any Space Weather Decision System. Its aim is to exploit the synergies between two emerging software development techniques – aspect-oriented software development (AOSD) and software product lines (SPL). Although software product lines aim to simplify software development and improve reuse by managing variations of software across different operational contexts, they themselves tend to become quite complex. This is due to the fact that variations tend to be crosscutting in nature. For instance, variations in the persistence or security policies in a product line can have a wide ranging impact across the assets and the product spectrum. These variations and their subsequent evolution need to be managed effectively.

5. Integration of Research Work Packages

This section describes how WP1 to WP4 are involved in finding solutions for the identified challenges and how the respective results are applied in the case studies.

WP1 investigates how features, commonalities and variations can be found semi-automatically from requirements documents with early aspects methodologies. Smart Home requirements and scenario documents will be used to explore how current tools provided by ULANC and UNL need to be extended to support the mining of common and variable features.

Additionally, WP1 investigates how to express requirements as a set of models without technical details and how to support traceability of high-level requirements and variations through model transformations. Features models and use case models from the Sales Scenario case study will be used to explore how to deal with traceability and evolution towards later SPL development phases as well as among requirements artefacts.

Unanticipated changes management is also part of the investigation in WP1. It explores how current aspect-oriented modelling and volatile requirements modelling techniques can be extended and used in SPLE. High-level, untangled requirement models used together with aspect-oriented techniques, which provide improved composition support of unanticipated behavior and structure changes into a running system, constitute the base of our approach to improve evolution. The Sales Scenario case study will provide potential scenarios where unanticipated requirements in the sales process, like business rules, must be integrated in a running system.

The tool support developed in WP1 will be applied and evaluated within the large demonstrator. Feature models and variant descriptions derived from requirements are traced to the variability modelled with the tooling developed in WP1 and WP2.

WP2 investigates how AOSD modelling can improve the documentation of variation in architectural models. This includes a language for the first class representation of variability in architectural views and methods for binding the variants in derived architectures. The *Variability Pointcut Language* (VPL) is a language with the means to *identify* and *affect* (i.e. connect, merge, remove) points of variability in architectural elements. In effect, the VPL provides the link between feature model and architectural elements, it thus integrates results from WP1 and WP2 and supports the traceability work done in WP4. An orthogonal approach to variability ensures that variability is not tangled within architectural models thus facilitating understandability and changeability. The case studies will provide variability scenarios across different views and thus act as a test bed for the VPL. Transformations developed in WP2 bind the variability in the architecture and design of derived products.

The approaches and tools developed in WP2 will be applied and evaluated within the case study. The transformations to design will be integrated with manually provided code libraries and code generators implemented in this work package, both of which will use implementation technology developed in WP3.

The goal of WP3 is to improve the technology for implementation of product lines. The specifics of product line implementation are mostly related with high requirements to variability support. The case studies help WP3 to identify different types of variability and to evaluate how good they are supported by existing technologies. In this way WP3 can better understand the strengths and weaknesses of existing implementation technologies, and see how they can be combined to solve the problems at hand. Besides, by evaluating existing technology the case studies can also identify unsolved challenges, which is a valuable input for further development of implementation techniques in the context of the project.
In the later phase of the project, the case studies will be used to experiment with the new improvements to languages and tools developed in WP3. The case studies will serve as a basis for evaluation of the new techniques and development of guidelines for their application.

The objective of the WP 4 is the definition and implementation of a software platform supporting general traceability of assets and decisions in software product line development. Based on concepts of model-driven engineering, variants, alternatives and aspects, and their relation are explicitly represented. Collecting the necessary information involves WP1, WP2, and WP3. Their tools and technologies bridge between the artefacts of the respective work packages and have to provide means to record the tracing information, either manually, where decisions are made, or automatically where transformations are used to derive or configure the next artefacts.

During the first year the main task was to establish a precise state of the art about traceability, configuration management and software evolution. Analysis of the first small case study and of existing tools, practices and experiments from industrial partners revealed important deficiencies in support for traceability. The case studies have been used to collect requirements on traceability, for a trace taxonomy and thus to get a clearer understanding of the needs and the software support we will provide. In the first small case study developed in the first year, preliminary tracing data is generated during automatic transformations to support the elaboration of the base abstract meta-model for traceability.

Finally the case studies are also the practical basis for investigating software configuration and evolution. Starting from the feature models, the tracing information will be leveraged in the large case study to select assets and to configure derived products. Furthermore, the interaction of features, e.g., in the Sales scenario, will be assessed based on their relation as described by tracing. Thus, support is offered for explicitly handling open decisions and uncertainty, and of course for making decisions for variants during development and configuration.

-

References

[BZ04] Rüdiger Buck-Emden, Peter Zencke; *mySAP CRM: The Official Guidebook to SAP CRM 4.0*; ISBN 1-59229-029-9, Galileo Press, 2004
[PS@ps] pure-systems GmbH; http://www.pure-systems.com
[FMC] Fundamental Modeling Concepts; http://www.fmc-modeling.org/