protoBOM

Framework that semi-automatically generates Decision Support Systems based on Software Product Lines

Master’s Thesis in Computer Systems Engineering

María Gómez Lacruz
ABSTRACT

This thesis presents the development of a prototype of the Baseline Oriented Modeling (BOM) approach, called protoBOM.

BOM is a framework that semi-automatically generates Decision Support Systems in a specific domain, based on Software Product Lines.

protoBOM semi-automatically generates applications as PRISMA architectural models by using Model-Driven Architecture and Software Product Line techniques. These models are automatically compiled and the object code (C#, in .NET) is generated obtaining an executable application.

In protoBOM, the user constructs Decision Support Systems in a simpler way by using the ontologies of the diagnosis and the application domains by means of Domain Specific Languages. The interfaces will be closer to the problem domain, which will facilitate user interaction in a manner simple and intuitive.

KEYWORDS

TABLE OF CONTENTS

List of figures...........................................................................................................................................v

List of tables .............................................................................................................................................vii

1. Introduction...............................................................................................................................................1
   1.1 Position of the problem and justification of the work................................................................. 1
   1.2 Thesis objective................................................................................................................................. 2
   1.3 Thesis structure................................................................................................................................. 2

2. Foundations.............................................................................................................................................3
   2.1 Model Driven Architecture.............................................................................................................. 3
   2.2 Software Product Lines.................................................................................................................... 3
      2.2.1 Examples of Software Product Lines....................................................................................... 5
      2.2.2 Domain Engineering Process and Application Engineering Process ................................... 7
      2.2.3 Creation of the Software Product Line..................................................................................... 10
      2.2.4 Benefits of Software Product Lines......................................................................................... 11
      2.2.5 Repositories of Software Product Lines.................................................................................. 12
      2.2.6 Feature Oriented Programming................................................................................................. 13
   2.3 The PRISMA Model......................................................................................................................... 13
   2.4 Decision Support Systems.............................................................................................................. 14

3. BOM approach....................................................................................................................................... 15
   3.1 Modeling the development process of the SPL .............................................................................. 15
      3.1.1 Domain Engineering ............................................................................................................... 19
      3.1.2 Application Engineering ......................................................................................................... 29

4. Prototype of BOM: protoBOM.............................................................................................................32
   4.1 Software Requirements Specification (IEEE Std. 830-1998)....................................................... 32
      4.1.1 Introduction................................................................................................................................. 32
         4.1.1.1 Purpose............................................................................................................................... 32
         4.1.1.2 Scope................................................................................................................................ 32
         4.1.1.3 Definitions, acronyms, and abbreviations ......................................................................... 32
      4.1.2 Overall description...................................................................................................................... 32
         4.1.2.1 Product perspective.............................................................................................................. 32
4.1.2.2 Product functions .......................................................................................... 33
4.1.2.3 User characteristics ...................................................................................... 34
4.1.2.4 Constraints .................................................................................................. 34
4.1.2.5 Assumptions and dependencies .................................................................... 34

4.1.3 Specific requirements .................................................................................... 34
4.1.3.1 External interface requirements ................................................................. 34
4.1.3.1.1 User interfaces ....................................................................................... 34
4.1.3.1.2 Software interfaces ................................................................................ 34
4.1.3.1.3 Hardware interfaces .............................................................................. 34

4.2 Development of protoBOM ............................................................................. 35
4.2.1 protoBOM in the Domain Engineering ......................................................... 37
4.2.1.1 Creation of the Baseline ........................................................................... 38
4.2.1.1.1 Add Kit Box ......................................................................................... 42
4.2.1.1.2 Add Production Plan ........................................................................... 49
4.2.1.2 Create Feature Model .............................................................................. 50
4.2.1.3 Create Decision Tree .............................................................................. 52

4.2.2 protoBOM in the Application Engineering .................................................... 54
4.2.2.1 Obtain Domain Features .......................................................................... 56
4.2.2.2 Obtain Application Domain Features ...................................................... 57
4.2.2.3 Create PRISMA types ............................................................................. 61
4.2.2.4 Create executable systems ..................................................................... 63

5. Conclusions ....................................................................................................... 64

References ............................................................................................................ 66

Annexes:
- Annex A: The insertion process implementation ............................................. 68
- Annex B: Example of an aspect PRISMA type in XML document generated by protoBOM .......................................................... 72
- Annex C: Source code class specification (Methods corresponding to the list Features) ............................................................... 76
LIST OF FIGURES

Figure 2-1. Software Product Lines
Figure 2-2. MERGER
Figure 2-3. Nokia Mobile Phones
Figure 2-4. Product line development
Figure 2-5. Essential Product Line activities
Figure 3-1. Visual metaphor of PRISMA skeleton
Figure 3-2. Visual metaphor of PRISMA types
Figure 3-3. Create Feature Model
Figure 3-4: Create Decision Tree
Figure 3-5: Create Asset Selecting Process
Figure 3-6: Create Domain Conceptual Model
Figure 3-7: Create Application Domain Conceptual Model
Figure 3-8: Create Skeletons
Figure 3-9: Visual metaphor of an interface
Figure 3-10: Create Interface and Architectural Element PRISMA types.
Figure 3-11: Create Feature Insertion Main Process
Figure 3-12: Create Feature Insertion Processes
Figure 3-13: Create Aspect-Process Skeletons
Figure 3-14: Create Packaged Hybrids
Figure 3-15: Create Architectural Model Configurations
Figure 3-16: Create RAS Models
Figure 3-17: Create Kit Boxes
Figure 3-18: Kit box with its content
Figure 3-19: Create Baseline
Figure 3-20: Baseline
Figure 3-21: Create Production Plan of the SPL
Figure 3-22: Production Plan
Figure 3-23: Obtain domain features
Figure 3-24: Select assets
Figure 3-25: Unpack box
Figure 3-26: Obtain Application Domain Features
Figure 3-27: Obtain PRISMA types
Figure 3-28: Compile Architectural Model
Figure 3-29: Create executable DSS
Figure 4-1. Use case Domain Engineering.
Figure 4-2. Use case Application Engineering.
Figure 4-3. Initial GUI
Figure 4-4. Activities offered in the Domain Engineering.
Figure 4-5. BOM’s architecture.
Figure 4-6. Database’s tables
Figure 4-9. Add Kit box
Figure 4-10. Error message.
Figure 4-11. Information message.
Figure 4-12. File System
Figure 4-13. Add Artifacts
Figure 4-14. File dialog.
Figure 4-15. Add Hybrid
Figure 4-16. Aspect Process Skeleton
Figure 4-17. Add Interface
Figure 4-18. Add Production Plan
Figure 4-19. Feature Model GUI
Figure 4-20. Feature Model options
Figure 4-21. Create Decision Tree
Figure 4-22. Create path
Figure 4-23. Add Kit box
Figure 4-25. Options
Figure 4-26. Activities offered in the Application Engineering.
Figure 4-27. Execute Production Plan.
Figure 4-28. Error messages.
Figure 4-29. Obtain Domain Feature.
Figure 4-30. Obtain Application Domain Features.
Figure 4-31. Specification class.
Figure 4-32. Add properties.
Figure 4-33. Add hypotheses.
Figure 4-34. Add derivation rules.
Figure 4-35. Create PRISMA types.
Figure 4-36. Create executable
Figure A-1. Insertion process metaphor.
LIST OF TABLES

Table 2-1. Processes of the engineering in the SPL.
Table 2-2. Domain Engineering Activities.
Table 2-3. Application Engineering Activities.
Table 3-1. SPEM icons
Table 3-2. SPEM icons developed in this thesis.
1. INTRODUCTION

This thesis presents the development of a prototype for Baseline Oriented Modeling (BOM) approach, called protoBOM.

BOM is a framework that automatically generates Decision Support Systems based on software product lines, to achieve the following goals:

- to decrease production costs by reusing software,
- to generate automatic code to increase the productivity and quality of software and to decrease the time to market,
- to create new diagnostic systems in different domains,
- to construct systems in a simpler way by using the ontologies of the diagnosis and the application domains. The models will be closer to the problem domain, which will facilitate user interaction,
- to develop platform independent systems from the problem perspective and not the solution perspective, this will provide generality in the development approach and applicability in different domains.

Through protoBOM an user without knowledge in computer science or programming languages is able to create an executable application of Decision Support Systems in a simple way.

1.1 Position of the problem and justification of the work

In the last few years, there has been an increase in interest in Decision Support Systems that perform diagnostic tasks. The main objective in this kind of systems is to capture the state of an entity from a series of data (observation variables) and produce a diagnosis. The domain of Decision Support Systems for diagnosis includes systems for many application fields, for example: medical and education diagnosis, among others.

The development of Decision Support Systems is complex, since the elements that form their software architecture variant. These architectural elements change in their behavior, and in their structure.

In these kinds of systems, the variability management cannot be performed through an unique Feature Model, and it is necessary to treat the variability in two phases: The first phase through different Base Architectures derived from a unique Generic Architecture; and the second phase by means of a more classic treatment of the Feature Model, decorating these base architectures with different features.
Different technological spaces are used in order to cope with the complexity of the problem. These are the following:

- The **Model Driven Architecture** as the abstract modeling level (PIM).
- The **Software Product Line** as a systematic reuse in software products.
- The **PRISMA Architectural Framework** as the target software level (PSM).
- The **Decision Support Systems**, to capture the knowledge of experts and try to imitate their reasoning processes when they solve problems in a certain domain.

1.2 Thesis objective

The main objective of this thesis is the implementation of a prototype for the Baseline Oriented Modeling approach called protoBOM. protoBOM is a framework that automatically generates executable applications based on BOM approach. However, it is not able to automate at all, it is needed an user that interacts with the tool providing information about the domain variability.

protoBOM will generate semi-automatically Decision Support Systems as PRISMA architectures, by using Model Driven Architecture techniques and Software Product Lines techniques.

1.3 Thesis structure

The thesis is structured as follows:

- **Section 2: Foundations.** The different technological spaces that the work integrates are presented.

- **Section 3: BOM approach.** The BOM approach is presented.

- **Section 3: Prototype of BOM: protoBOM.** It is presented the prototype for BOM.

- **Section 4: Conclusions.** This section presents the conclusions and provides some ideas for future work.
2.- FOUNDATIONS

2.1 Model Driven Architecture

Model Driven Architecture (MDA) [MDA] is an initiative of the Model Driven Engineering (MDE) approach which is promoted by the Object Management Group (OMG) [OMG], for software system development. MDA is based on the separation of the description of the functionality of the system from its implementation on specific software platforms. MDA increases the abstraction level of the software production process by emphasizing the importance of conceptual models – Computer Independent Models (CIM) or Platform Independent Models (PIM) – and their role in the software development process. In MDA the models are first class citizens in the software development process. MDA proposes defining and using models at different abstraction levels. From these models, it is able to automatically generate code by means of mappings or by applying transformation rules and obtaining executable Platform Specific Models (PSM).

2.2 Software Product Line

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single individual product. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, and so software reuse must be increased. Software Product Line Engineering (SPLE) offers a solution to these problems [Clements et al., 2002]. The basis of SPLE is the explicit modeling of what is common and what differs between product variants.

Software Product Lines (SPL) [SPL] arose from an effort to control and minimize the costs of software development. As programs grow in complexity, the one-of-a-kind software development is becoming no longer economical; a better approach is the creation of a design that all members of a program family (Product Line) share.

A Software Product Line (SPL) is a set of software systems that share a common, managed set of features satisfying 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.

[Clements et al., 2002]

SPL are rapidly emerging as a viable and important software development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers. SPLE can also enable rapid market entry and flexible response, and provide a capability for mass customization.
Summarizing, we can say that a product line is a group of products that have a set of features in common and vary only in some specific features [Chetana et al., 2002]. Products of a product line are differentiated by their features. The key objectives of software product lines are: to capitalize on commonality and manage variation in order to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems.

Developing a group of products that have a majority of features in common supports a great deal of reuse in all the phases of system development, and it increases the productivity and quality of software products.

**Features** are an abstract concept for describing commonalities and variabilities. In this sense a feature is a relevant characteristic of a system. Features can be used to distinguish members of a Software Product Line.

A feature is an increment in product functionality.

Developing a group of products that have a majority of features in common supports a great deal of reuse in all the phases of system development, and it increases the productivity and quality of software products.

SPL approach, from a practical point of view, is one of the most successful approaches to software reuse, which focuses on developing a family of systems. Products are different in some features but share a basic architecture. SPL provides an industrial approach to the software development process. The main goal is to develop a framework that represents the family, which is then customized to develop individual products. A product family is a collection of similar products that share most of the features. Hence there are numerous requirements that are common across the family but others are unique to individual products. This implies changing the existing engineering process by introducing a distinction between the Domain Engineering Process and Application Engineering Process.
2.2.1 Examples of Software Product Lines

Successful software product lines have been built for families of:

- Mobile phones
- Command and control ship systems
- Ground-based spacecraft systems
- Avionics systems
- Pagers
- Engine control systems
- Billing systems
- Web-based retail systems
- Printers
- Consumer electronic products
- Acquisition management enterprise systems
Next, some examples will be shown:

- **Example 1. Celsius Tech: Ship System 2000 ([SEI])**
  - A family of 55 ship systems
  - Integration test of 1-1.5 million SLOC requires 1-2 people
  - Rehosting to a new platform/OS takes 3 months
  - Cost and schedule targets are predictably met
  - Performance/distribution behavior are known in advance
  - Customer satisfaction is high Hardware-to-software cost ratio changed from 35:65 to 80:20.

- **Example 2. Cummins Inc.: Diesel Engine Control Systems ([SEI])**
  - Over 20 product groups with over 1000 separate engine applications.
  - Product cycle time was slashed from 250 person-months to a few person-months
  - Build and integration time was reduced from one year to one week
  - Quality goals are exceeded
  - Customer satisfaction is high
  - Product schedules are met

- **Example 3. Market Maker GmbH: MERGER ([SEI])**
  - Internet-based stock market software.
  - Each product “uniquely” configured
  - Three days to put up a customized system

- **Example 4. National Reconnaissance Office/Raytheon: Control Channel Toolkit ([SEI])**
  - Ground-based spacecraft command and control systems
  - Increased quality by 10 X
  - Incremental build time reduced from months to weeks
  - Software productivity increased by 7X
  - Development time and costs decreased by 50 %
  - Decreased product risk
**Example 5. Nokia Mobile Phones** ([SEI])

- Product lines with 25-30 new products per year.
- Across products there are:
  - Varying number of keys
  - Varying display sizes
  - Varying sets of features
  - 58 languages supported
  - 130 countries served
  - Multiple protocols
  - Needs for backwards compatibility
  - Configurable features
  - Needs for product behaviour change after release

![Figure 2-3. Nokia Mobile Phones ([SEI])](image)

### 2.2.2 Domain Engineering Process and Application Engineering Process

The software reuse inside an application domain goes by the discovery of common elements to the systems belonging to this domain [García et al., 2002].

This focus produces a change of a development guided for an only product software to a development of several products that contain some common characteristics, forming a family of products. This implies a restructuring in the development of the software, arising two different processes: the Domain Engineering and the Application Engineering.

In the first one, the problem space will be modeled, it is to say, it will determine what is common to, and what differs between, the different product variants. In the second one, the problem and solution space models are used to create a product variant.

<table>
<thead>
<tr>
<th>Domain Engineering</th>
<th>Domain Analysis</th>
<th>Development of reusable basic components</th>
<th>Production planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Engineering</td>
<td>Characterization of the product</td>
<td>Synthesis of the product</td>
<td>Construction of the product</td>
</tr>
</tbody>
</table>

*Table 2-1. Processes of the engineering in the SPL [Clements et al, 2002]*
The division between the Domain Engineering and the Application Engineering is fundamental in the approach of the SPL [Clements et al., 2002].

This partition is the base of any reuse intent and automation in software processes [Czarnecki et al., 2000]. In the Domain Engineering, a group of assets are developed in order to be reused in a specific application domain, and allowing automation; in the Application Engineering, these assets will be used to obtain software products of bigger quality, using less cost and time.

- **Domain Engineering Process**

In general, the domain engineering process defines the shared architecture and the variability of the SPL. More than creating products, it is a question of putting assets together in a Baseline warehouse. The construction of the assets and their variability (domain engineering process) is separate from the application production (application engineering process).

In the domain engineering phase, a set of assets and processes are created. During this process, the commonality and variability in the product line are analyzed in light of the overall requirements of the product line. This activity consists of developing a product line use case model, product line analysis model, software product line architecture, and reusable components. The artifacts produced during this phase are stored in a software product line repository.

<table>
<thead>
<tr>
<th>Domain Engineering Activities</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain Analysis</strong></td>
<td>It studies the variability of the domain. Frequently this study is given in terms of characteristic of the domain and it is represented using a model of characteristic.</td>
</tr>
<tr>
<td><strong>Development of reusable basic components</strong></td>
<td>It conceives, designs and implements the basic reusable components. This does not only involve the development of domain functionality, but it also defines how the basic reusable components should be extended.</td>
</tr>
<tr>
<td><strong>Planning of the production</strong></td>
<td>It defines how singular products are created. In general it implies the capacity of production of the LPS.</td>
</tr>
</tbody>
</table>

*Table 2-2. Domain Engineering Activities*
• Application Engineering Process

Application Engineering is guided toward the construction or development of individual products, belonging to the family of products, and that they satisfy a group of requirements and restrictions expressed by a specific user, reusing, adapting and integrating the reusable elements existent and produced in the Domain Engineering. [García et al., 2002].

For each SPL there is a well defined production plan that specifies the process to obtain each of the individual products. After the Domain Engineering Process, the logical next step is the automated production of solution variants.

In the application engineering phase, the assets are used to produce software products of high quality with a minimal cost and time by executing the stored processes. During this process, an individual application that is a member of the software product line is developed. Instead of starting from scratch, as is usually done with single systems, the application developers make full use of all the artifacts developed during the domain engineering. Given the overall requirements of the individual application, the product line use case model is adapted to derive the application use case model; the product line analysis model is adapted to derive the application analysis model; and the software product line architecture is adapted to derive the architecture of the software application. Given the application architecture and the appropriate components from the product line repository, the executable application is deployed.

<table>
<thead>
<tr>
<th>Application Engineering Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characterization of the product</td>
</tr>
<tr>
<td>It chooses the characteristics that differentiate a selected product. This process begins with the selection of the characteristics.</td>
</tr>
<tr>
<td>Synthesis of the product</td>
</tr>
<tr>
<td>It gathers reusable basic components to obtain the “matter prevails” (a product is made up of). The techniques of variability are used in this process.</td>
</tr>
<tr>
<td>Construction of the product</td>
</tr>
<tr>
<td>It processes the “matter prevails” following the construction process (e.g. to compile, to generate code, to execute, etc.), to obtain a final product.</td>
</tr>
</tbody>
</table>

*Table 2-3. Application Engineering Activities*
2.2.3 Creation of the Software Product Line

This section describes the process for the creation a Software Product Line.

Next picture illustrates that product line development consists of cooperation among three different constituencies: core asset development, product development, and management.

![Diagram of product line development](image)

*Figure 2-4. Product line development ([Clements et al., 2002]).*

The process for creation of a Software Product Line consists of three essential activities:

- Core Asset Development
- Product Development
- Management

There is no “first” activity; all these activities are interrelated and highly iterative. In some contexts, existing products are mined for core assets; in others, core assets may be developed or procured for future use.

There is a strong feedback loop between the core assets and the products.
Figure 2-5. Essential Product Line activities (From Carnegie Mellon Software Engineering Institute)

- Core Asset Development

Core asset development is the creation and maintenance of the artifacts or core assets in the product line. These core assets are used to create systems that match the quality criteria of the product line. The goal of the core asset development activity is to create a capability within the organization to produce a particular type of application and will thus yield the same or similar software architecture.

- Product Development

The second constituency is product development. Product development involves the creation of products or systems from the core assets of the product line. If a system requires an asset that is not included in the core assets, the core asset must be created if the asset can be shared across multiple products in the product line. It is a strategic decision whether or not to build a new core asset or to create a product-specific feature to the project under development. Also, if the core asset that exists in the product line does not match the quality requirements of the product under development, the core asset may be enhanced or modified.

- Management

It is also important to identify which assets are part of the product line and which ones is part of the development of the individual products of the system. Management consists of the management of individual projects within the product line, as well as overall product line managers.
2.2.4 Benefits of Software Product Lines

The benefits of the software product line approach come in the form of improvements in software engineering, deploying software products faster, cheaper, and better.

The following engineering benefits can be gained:

- Reduction in the average time to create and deploy a new product
- Reduction in the average number of defects per product
- Reduction in the average engineering effort to deploy and maintain a product, and therefore reduction in the average engineering cost per product
- Increase in the total number of products that can be effectively deployed and managed
- Reduced time-to-market and time-to-revenue for new products
- Improved competitive product value
- Higher profit margins
- Reduced risk in product deployments.

The benefits offered by software product lines can be attributed to strategic software reuse. Software product line techniques explicitly consolidate and capitalize on commonality throughout the product line. They formally manage and control the variations among the products in the product line. They eliminate all duplication of effort in the engineering processes.

As a result, the only unique engineering effort required for any product in the product line is for the product variations that are truly unique to the product.

2.2.5 Repositories of Software Product Lines

Software Product Lines require to store its software assets in repositories.

A SPL repository is a specialized database that stores software assets and it facilitates the recovery and maintenance of the software assets.

Their objective is to assure the readiness of assets to support the development of products of the Software Product Line.

In BOM approach the repository is called Baseline.
2.2.6 Feature Oriented Programming

Feature Oriented Programming (FOP) is the study of the modularity of the features of a domain and their use. This technique is used in the construction of the SPL.

The features are considered as first-class citizens in the design process. FOP is an approach to SPL where the programs are built by means of feature composition. These features are considered as building blocks of the programs. They are units that increase monotonically the functionality of the application by providing different products. Each one of the features can be included in the different software artifacts. In general, a SPL is characterized by the set of features being used, which is called the feature model of the SPL.

2.3 The PRISMA Model

PRISMA provides a model for the definition of complex software systems [Pérez, 2006].

The PRISMA architectural model integrates two approaches: Component-Based Software Development (CBSD) [Szyperski, 1998], and Aspect-Oriented Software Development (AOSD) [AOSD]. This integration is obtained by defining the architectural elements through their aspects.

The PRISMA model consists of three types of architectural elements: components, connectors, and systems. A component captures the functionality of the system, whereas a connector acts as a coordinator among other architectural elements. A system is an architectural element of great granularity that allows the encapsulation of a set of components, connectors, and other systems. This, in turn, allows the system to correctly connect them.

PRISMA defines the architectures in two abstraction levels: the type level and the configuration level. In the type level, the types of the architectural artifacts are defined (all of which can be reused): interfaces, aspects, components, connectors, and systems. In the configuration level, the types are instantiated and the topology of the model (its configuration) is specified.
2.4 Decision Support Systems

Decision Support Systems capture the knowledge of experts and try to imitate their reasoning processes when the experts solve problems in a certain domain.

These systems usually have a basic architecture that makes up a knowledge base, an inference motor and the user interface. These are the main components of the architecture of a Decision Support System based in rules.

These components are independent and are composed of separates units. The data are grouped into the working memory (temporary storage of dynamic information). The representation of the knowledge is captured by means of rules of the type: IF <antecedent> THEN <conclusion>, which makes up the knowledge base. The control aspect is independent and is performed by the inference motor during the inference processes using different reasoning strategies. The input and output of the information of the systems are done through the user interface.
3. BOM APPROACH

protoBOM follows the BOM approach [Cabello, 2007] for the development of a DSS.

BOM is based on SPL techniques. This approach is based on the production of families of software products starting from a group of assets; these assets represent the generic architecture of the domain, and it is shared for all the members of a SPL. For example, the family of the Diagnosis.

This implies the existence of particular features for each member of the SPL. These particular features are represented as variability points, and their choice configure the final product. Therefore, the variability is treated through variability points in a Decision Tree (DT). This DT selects the right assets for a concrete member of the family. For example, through the DT will be selected Medical Diagnosis, Educational Diagnosis…

Furthermore, in the development process of a specific application (a final product), a second variability emerges. This variability is managed by decorating the base assets with the features of the application domain. For example, the final product will be a Medical Diagnosis System, therefore the concrete features for the system have to be provided: fever, cough…

In addition to this, the developmental process involves building a Baseline, a repository containing all the assets needed to build a product of the SPL: the Kit Boxes of the SPL, and to model the production plan of the SPL. The production plan specifies the process for obtaining each product.

3.1 Modeling the development process of the Software Product Line

BOM approach is proposed for automating the development processes, through the creation of SPL in specific application domains. BOM offers technologies to carry out the reuse and the automation in the software processes.

protoBOM following this approach derives in semi-automatic way, a specific product in a software product line starting from a series of previous models. In this way, SPLs can be encapsulated in reusable assets, until it is possible to automate its development [Clements et al., 2002].

protoBOM uses the PRISMA-MODEL-COMPILER tool [Cabe do et al., 2005] to automatically generate the executable code in (.NET, C#) [.NET] of the generated model. protoBOM executes the object code over the PRISMA-NET Middleware [Costa et al., 2005].

Using the previous approach, the engineering processes of the SPL, with the tasks that will be executed in each one of the parts, are described:
Domain Engineering

- Domain analysis tasks:
  - to create feature model
  - to create decision tree
  - to create asset selection process
  - to create domain conceptual model
  - to create application domain conceptual model

- Development of reusable core assets tasks:
  - to create skeletons
  - to create PRISMA types interfaces
  - to create PRISMA types Architectural Elements
  - to create the Feature Insertion Main Process
  - to create the Feature Insertion Processes
  - to create Aspect- Process Skeleton
  - to create packaged Hybrids
  - to create information of the Base Architectural Model Configuration
  - to create the RAS models
  - to create the Kit boxes
  - to create the Baseline

- Planning of the production tasks:
  - to model production plan

Application Engineering

- to obtain Domain Features
- to obtain Application Domain Features
- to select assets
- to unpack the Kit box
- to create PRISMA types
- to compile model (to generate code)
- to create executable application

Summarizing the domain engineer creates the production plan and the software artifacts; the application engineer executes the production plan for the SPL.

This work is based in a standard for modeling the processes for the creation of the SPL, reusing the same group of resources to create the family of software products: the Software Process Engineering Metamodel (SPEM) [SPEM], which defines a language that models processes of software development using a common and standard terminology. SPEM is defined by the OMG.
Next, the standard icons of SPEM used in this thesis are presented, as well as their meaning.

<table>
<thead>
<tr>
<th>ICON</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Activity Icon]</td>
<td>Activity</td>
</tr>
<tr>
<td>![Process Role Icon]</td>
<td>Process Role</td>
</tr>
<tr>
<td>![Guidance Icon]</td>
<td>Guidance</td>
</tr>
<tr>
<td>![Process Icon]</td>
<td>Process</td>
</tr>
<tr>
<td>![Work Product Icon]</td>
<td>Work Product</td>
</tr>
<tr>
<td>![UML model Icon]</td>
<td>UML model</td>
</tr>
<tr>
<td>![Document Icon]</td>
<td>Document</td>
</tr>
</tbody>
</table>

*Table 3-1. SPEM icons*
Furthermore some icons have been created on purpose for this thesis. They will be used in the Domain Engineering and Application Engineering.

<table>
<thead>
<tr>
<th>ICON</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Aspect-Process Skeleton Icon]</td>
<td>Aspect-Process Skeleton = Aspect Skeleton + Features Insertion Process</td>
</tr>
<tr>
<td>![Packaged Hybrid Icon]</td>
<td>Packaged Hybrid = Aspect Process Skeleton + Interface (PRISMA type) + Architectural Element (PRISMA type)</td>
</tr>
<tr>
<td>![Kit box Icon]</td>
<td>Kit box</td>
</tr>
<tr>
<td>![Baseline Icon]</td>
<td>Baseline</td>
</tr>
</tbody>
</table>

**Table 3-2. SPEM icons developed in this thesis.**

The domain engineer should be sufficiently expert to determine the common platform that it shares the whole family of products.

A platform of a SPL is the group of software artifacts that form a common structure. The parts that compose the platform are called assets [Clements et al., 2002]. The platform is the base on which the products can be created adding characteristics (variability). From a structure a group of products can be produced and developed efficiently [Meyer et al., 1997].

The platform developed is this thesis is specified in the ADL of PRISMA. The platform consists on a group of architectural elements represented in the model PRISMA. These elements are classified in: aspects, interfaces, components and connectors.

Two kinds of elements are considered:

- **Skeletons**: The skeletons are specification templates, represented as XML documents. These templates contain gaps; later on, they will be stuffed with the characteristics of the specific application domain. Only the aspects can be considered as skeletons (see figure 3-1).

- **Types**: The types are XML documents containing the information of the specific application domain, it is to say characteristics or “features” in the study case. Therefore, the types are skeletons stuffed with the information of a concrete study case. For it, the visual metaphor of the types is represented as icons with color (representing the characteristics inserted), as it is shown in the figure 3-2. There are three additional PRISMA types: interface, component and connector.
3.1.1 Domain Engineering

In this first phase the assets are created and the production plan of the SPL is described.

Next, all the activities carried out by the domain engineer are shown in SPEM.

All these tasks are carried out in three parts: domain analysis, development of reusable core assets and planning of the production.

The following software artifacts are created by the diagnostic domain engineer. These artifacts are necessary to generate the product plan of the software product line: the artifacts 1 and 2 are Computation Independent Models (CIMs), and the other artifacts are Platform Independent Models (PIM).

**Artefact1. The Feature Model (CIM):** the Feature Model identifies the SPL in terms of the variability in the domain. The features as variability sources of this model are represented in a Decision Tree (see artefact 2).
Artefact 2. **The Decision Tree (CIM):** the variability sources observed in the Feature Model are shown as the variation points of the Decision Tree, where the leaves represent the assets, families of the SPL. The Decision Tree is used to create the Asset Selection Process (see artefact 3).

![Create Decision Tree](image)

*Figure 3-4: Create Decision Tree (Adapted from [Cabello, 2007]*)

Artefact 3. **The Asset Selection Process (PIM):** this process computes the paths of the Decision Tree, which is used to select the assets. Each leaf of the Decision Tree corresponds with one Kit Box asset.

![Create Asset Selecting Process](image)

*Figure 3-5: Create Asset Selecting Process (Adapted from [Cabello, 2007]*)

Artefact 4. **DCM: Domain Conceptual Model (PIM):** the features as variability points present in the Feature Model and The Decision Tree are represented in the Domain Conceptual Model, which captures the variability of the diagnostic domain. In the application engineering, this artefact is configured when the engineer input to the system, through protoBOM, information of the specific domain, in order to select assets from the Baseline.

![Create Domain Conceptual Model](image)

*Figure 3-6: Create Domain Conceptual Model (Adapted from [Cabello, 2007]*)
**Artefact 5.** **ADCM: Application Domain Conceptual Models (PIM):** these models capture the variability of the application domain. This software artefact is used by the application engineer to capture through protoBOM the features of the application domain, in order to decorate the base architecture of this application domain.

![Create Application Domain Conceptual Model](adapted-from-Cabello-2007)

*Figure 3-7: Create Application Domain Conceptual Model (Adapted from [Cabello, 2007])*

**Artefact 6.** **The Skeletons (PIM):** the skeletons of the SPL are represented by means of the aspects. The features are inserted in the aspects. These skeletons are used in the application engineering, when protoBOM full them with the features of the application domain. These features are input to the system by the engineer.

The skeletons filled are the PRISMA types.

![Create Skeletons](adapted-from-Cabello-2007)

*Figure 3-8: Create Skeletons (Adapted from [Cabello, 2007])*
Artefact7. The PRISMA type Interfaces (PIM): An interface contains the group of services that receives/sends through a port of an architectural element. The interfaces don't involve "features", therefore they are not considered it as skeletons, but they are PRISMA types.

Figure 3-9: Visual metaphor of an interface

Artefact8. The PRISMA type Architectural Elements (PIM): there are different classes of skeletons or templates for: components, connectors, aspects, and interfaces; they follow the PRISMA Model. The aspects that are necessary for the definition of these architectural elements are: the functional aspects of each one of the components, and the coordination aspect of the connector. These aspects use interface services.

Figure 3-10: Create Interface and Architectural Element PRISMA types. (Adapted from [Cabello, 2007])

Artefact9. The Feature Insertion Main Process (PIM): this process contains the feature insertion general process, which it is divided in several subprocesses or feature insertion individually processes associated to the skeletons of aspects.

Figure 3-11: Create Feature Insertion Main Process (Adapted from [Cabello, 2007])
**Artefact10: The Feature Insertion Processes (PIM):** These processes insert the features into the software artifacts.

![Figure 3-12: Create Feature Insertion Processes (Adapted from [Cabello, 2007])](image)

**Artefact11. The Aspect-Process Skeleton (PIM):** Each skeleton with a feature insertion process are packaged following the RAS technology to form assets (to be executed in the application construction phase).

![Figure 3-13: Create Aspect-Process Skeletons](image)

**Artefact12. The Packaged Hybrids (PIM):** a packaged hybrid is composed by its aspect-process skeleton, its interface PRISMA type, and its component/connector PRISMA type. These assets are deposited in the Baseline (after to be packaged and to form part of one new asset (see artefact 13). In the application domain, the packaged skeletons selected are full with the features of the specific application.

![Figure 3-14: Create Packaged Hybrids (Adapted from [Cabello, 2007])](image)
Artefact13. The Architectural Model Configurations (PIM): this asset contained the manner of to configure the base architecture formed by its respective skeletons. This asset also is deposited in the Baseline.

![Create Architectural Model Configurations](image)

*Figure 3-15: Create Architectural Model Configurations (Adapted from [Cabello, 2007])*

Artefact14. The RAS Models of the Assets (PIM): this models store information from each one of the assets: ID asset identifier, asset classification, description of the different asset artefacts, and variability points of the asset artefacts.

![Create RAS Models](image)

*Figure 3-16: Create RAS Models (Adapted from [Cabello, 2007])*
**Artefact 15.** The Kit boxes (PIM): The assets selected by the domain engineer, to form families of SPL with common characteristic (from the first variability points), are packed as an asset. These packed assets form kit boxes whose content it is built in the same phase.

![Create Kit Boxes (Adapted from Cabello, 2007)](image1)

*Figure 3-17: Create Kit Boxes (Adapted from [Cabello, 2007])*

The figure 3-18 shows a visual metaphor of a Kit Box with its content.

![Kit box with its content](image2)

*Figure 3-18: Kit box with its content (Adapted from [Cabello, 2007])*
**Artefact 16. The Baseline (PIM):** The Baseline is the repository that contains all the assets and all the application domain conceptual models, which are used to capture the specific application features. The baseline is the last asset built in the domain engineering, and it is need to obtain a product from the SPL.

The Baseline consists of a set of packaged assets forming boxes. Each kit box contains: the Packaged Hybrids, the Application Domain Conceptual Model, the Main Feature Insertion Process, the Information Architecture of the Base Configuration, and the RAS Model of the Assets. In addition to these cases, the Baseline contains the Production Plan of the SPL. This implies that our content Baseline is both a repository of all assets as of the knowledge necessary to produce the SPL.

Baseline is used in the application engineering to choose assets (kit boxes) for the specific domain.

*Figure 3-19: Create Baseline (Adapted from [Cabello, 2007])*
A visual metaphor of the Baseline is shown in Figure 3-20.

![Figure 3-20: Baseline](image)

**Artefact 17. The Production Plan of SPL (PIM):** This process shows the life cycle of production of the SPL. This process, at the stage of engineering domain is specified in SPEM and is deposited in the Baseline as an asset. In the application engineering, the Production Plan is done to generate a DSS as a result of the SPL.

![Figure 3-21: Create Production Plan of the SPL (Adapted from [Cabello, 2007])](image)

**PRODUCTION PLAN**

The production plan describes the process through tasks or activities that are carried out to obtain a final product of the SPL. This process is shown in the figure 3-22, making use of the standard SPEM.

Our production plan involves 8 tasks, all them carried out by BOM through the interaction with the user with the role application engineer.
Figure 3-22: Production Plan (Adapted from [Cabello, 2007])
3.1.2 Application Engineering

The production plan of the Software Product Line is described using SPEM. The tasks performed by the application engineer consume input artifacts and produce output artifacts.

The tasks are described below:

- **Task 1. To obtain the Domain Features:** The engineer introduces the domain information of the case study. BOM captures the variability information by means of the domain features detailed in the Domain Conceptual Model. This model allows the engineer (by means of a GUI) to introduce the information of the Model.

  ![Figure 3-23: Obtain domain features (Adapted from [Cabello, 2007])](image1)

- **Task 2. To select Assets and the Application Domain Conceptual Model:** BOM selects the assets and the Application Domain Conceptual Model from the Baseline (by means of the decision tree).

  ![Figure 3-24: Select assets (Adapted from [Cabello, 2007])](image2)
- **Task 3. To unpack the Box**: The box selected by the engineer has to be unpackaged to use its assets individually.

  ![Figure 3-25: Unpack box (Adapted from [Cabello, 2007])](image1)

- **Task 4. To obtain the Application Domain Features**: The engineer introduces the application domain information of the case study. protoBOM captures the variability information by means of the application domain features contained in the Application Domain Conceptual Model.

  ![Figure 3-26: Obtain Application Domain Features (Adapted from [Cabello, 2007])](image2)

- **Task 5. To obtain the PRISMA Types**: protoBOM fills the selected skeletons (aspects) with the data of the specific features of the case study that were defined by the engineer, thereby creating the PRISMA software artifact types. Furthermore, protoBOM recovers the PRISMA types (interface and architectural element) stored in the hybrid.

  ![Figure 3-27: Obtain PRISMA types (Adapted from [Cabello, 2007])](image3)
- **Task 6. To compile the Architectural Model**: BOM uses the PRISMA-MODEL-COMPLIER tool [Cabedo et al., 2005] to automatically generate the code (in .NET, C#) of the software architecture of the preceding task.

![Figure 3-28: Compile Architectural Model (Adapted from [Cabello, 2007])](image)

- **Task 7. To create the executable system**: BOM creates the final diagnostic system as application executable, i.e. an instance of the SPL. This is executed on top of the PRISMANET Middleware [Costa et al., 2005].

![Figure 3-29: Create executable DSS (Adapted from [Cabello, 2007])](image)
4. - *PROTOTYPE OF BOM: protoBOM*

4.1 SOFTWARE REQUIREMENTS SPECIFICATION (IEEE Std. 830-1998)

This section describes the software requirements of protoBOM following the standard IEEE 830-1998 [IEEE].

4.1.1. Introduction

4.1.1.1 Purpose

This document describes the software requirements of a framework that semi-automatically generates applications based on Software Product Lines.

4.1.1.2 Scope

The product is called protoBOM. protoBOM is a framework that semi-automatically generates Decision Support Systems based on Software Product Lines, to achieve the following goals: to decrease production costs by reusing software, to generate automatic code to increase the productivity and quality of software, to decrease the time to market, to create new diagnostic systems in different domains and to construct systems in a simpler way.

4.1.1.3 Definitions, acronyms, and abbreviations

**IEEE**: Institute of Electrical & Electronics Engineers.

**SPL**: Software Product Line.

4.1.2 Overall description

4.1.2.1 Product perspective

protoBOM is an independent product; its main objective is to assist the process of generation of applications based on Software Product Lines. However, it is not able to automate at all, it is needed an user that interacts with the framework providing information.
4.1.2.2 Product functions

The functions that protoBOM can carry out are classified in blocks:

a) **Domain Engineering**
   
   - **Feature Model**
     - To create the feature model
   
   - **Decision Tree**
     - To create the decision tree
     - To create the decision tree’s paths to select assets
   
   - **Baseline**
     - Inert kit box into the baseline
     - Insert Production Plan to the baseline
     - Visualize baseline’s content
     - Add artifacts to a kit box
     - Insert Base Architectural Model Configuration
     - Insert Feature Insertion Main Process
     - Insert RAS model of assets
     - Insert Application Domain Conceptual Model
     - Insert package Hybrid
     - Insert Aspect-Process Skeleton
     - Insert Interface
     - Insert Architectural element

b) **Application Engineering**
   
   - To obtain Domain Features
   - To select assets
   - To unpack kit box
   - **to obtain Application Domain Features**
     - Insert properties by level
     - Insert hypotheses by level
     - Insert derivation rules by level
   
   - **To create PRISMA types**
     - To recover PRISMA types from the kit box
     - Execute Insertion Process
   
   - **Create executable**
4.1.2.3 User characteristics

protoBOM is an application for the operating system Windows, therefore the user should be familiarized with environments of windows.

Two kinds of users are distinguished: the Domain Engineer and the Application Engineer. The Domain Engineer creates the software artifacts needed for the creation of the SPL. On the other hand the Application Engineer executes the production plan and a product of the SPL is developed.

4.1.2.4 Constraints

The application is developed using an Object Oriented Programming language: C#, and it is an application for the operative system Windows. The access to the database is done with SQL Server 2005, and for the access via web has been employed ASP.Net.

It is not need any specific hardware.

4.1.2.5 Assumptions and dependencies

The software to carry out will be compatible with the operating system Windows, for that the computer in which the product will work will have the operating system Windows in some of its versions (superior versions to Windows 3.1).

The application does not require any specific hardware, the limitations hardware will come determined by the computer in which the application works.

4.1.3 Specific requirements

4.1.3.1 External interface requirements

4.1.3.1.1 User interfaces

protoBOM is an application with windows, very intuitive. The interfaces are simple with a lot of buttons with pictures to make easy the work. The user will familiarize very fast with the application.

4.1.3.1.2 Software interfaces

The product is developed in Windows XP, but it will be compatible with previous versions.

4.1.3.1.3 Hardware interfaces

It is not relevant for this Project.
4.2 DEVELOPMENT OF protoBOM

protoBOM integrates several technological spaces, offering an approach to build applications of the DSS, in a simple way.

protoBOM captures the information of the domain and the information of the specific application domain, to automatically generate Decision Support Systems, with PRISMA architectures, based on product lines.

Two different roles are involved in the framework: the domain engineer and the application engineer.

The use of protoBOM, by the domain engineer, to build the software artifacts needed to obtain a product of the SPL implies:
  o to build the feature model
  o to build the decision tree
  o to build the decision tree’s paths to select assets
  o to pack the assets to form the Baseline

The use of protoBOM, by the application engineer, to generate a product of the SPL, it integrates the use of several tools.

Some of these tools have been developing on purpose for protoBOM and others were created before for other environments.

The tools developed in this thesis allow:
  o through a GUI, to recover domain conceptual models and application domain conceptual models.
  o to select assets of the Baseline.
  o to create the PRISMA types, through transformations (insert processes of characteristic) in XML documents (skeletons).
  o to incorporate in the tools of PRISMA the needed information to configure the architecture of the DSS, to generate their code and to create the executable.
  o to integrate all the tools mentioned in an alone one: protoBOM

The tools used in protoBOM but created in other environments can:
  o to use the compiler of models PRISMA-MODEL-COMpiler to automatically generate the code C # of the software architecture of the previous step [Cabedo et al., 2005].
  o to create the executable DSS (an instance of the SPL) on the middleware PRISMA-NET [Cabedo et al., 2005].
In order to capture the functional requirements of protoBOM, it has been used a visual modeling language, UML [UML]; specifically the use case technique. Next, the use cases corresponding to protoBOM are presented (Figure 3-1, Figure 3-2).

Figure 4-1. Use case Domain Engineering.

Figure 4-2. Use case Application Engineering.
protoBOM integrates all the tools mentioned, so that the initial GUI allows the user to select, according to his role, the activities to carry out of the domain engineering or of the application engineering, as the following figure shows:

![Initial GUI](image)

*Figure 4-3. Initial GUI*

### 4.2.1 protoBOM in the Domain Engineering

In this phase all the assets are created, and the configuration process and the production plan are described. But a SPL require storing the assets and the production plan in a repository. In this thesis, all the assets are deposited in the Baseline, for that the domain engineering phase consists in the creation of the Baseline.

BOM offers to the domain engineer, besides building the Baseline (as specific task of this stage of development of the SPL), creating the Feature Model, the Decision Tree and the trajectories of this tree whose leaves point to the group of selected assets (i.e. the selected kit box).
The following software artifacts are created by the domain engineer:

- Feature model
- Decision tree
- Decision tree’s paths to select assets
- Baseline

4.2.1.1 Creation of the Baseline

The Baseline is a specialized repository that stores assets and it facilitates the recovery and the maintenance of the assets. Its objective is to assure the readiness of assets to support the development of products of the SPL.

When the domain engineer wants to create an asset, he will inspect the Baseline searching one that covers the requirements. If none of the members of the family satisfies their necessities, it will be created an asset for that case, starting from the member most similar to the requirements. When it finishes the creation task, the domain engineer will carry out to include the new variant in the family of products inside the Baseline. The configuration process of the new asset will generate the new variant, to be applied whenever the original conditions appear. It is important to consider that the domain engineer should have a good knowledge of the domain; he will be able to follow patterns and to establish similarities.

In protoBOM, the Baseline has been implemented like a database that resides in a web service.
In the broadest sense, a Web service is a method or set of methods that are publicly available to remote applications over Hypertext Transfer Protocol (HTTP). The implementation of a Web service is completely shielded from the consuming application (that is, the application that is invoking the Web service). The consuming application only knows the rules for interfacing with the Web service.

This opens the door to reuse software product lines, in this way the software product lines developed can be easily shared and distributed.

The database is composed for the following tables:

```
Box *
- Path
- Bmc
- Hmp
- Res
- Addm
- Hybrid

Hybrid *
- Path
- APSkeleton
- Interf
- ArchEth

AP_Skeleton *
- Path
- Aspectdk
- Health_proc

ProdPlan *
- Path
```

Next, part of the code from the web service is shown:
using System;
using System.Data;
using System.IO;
using System.Web;
using System.Web.Services;

/// <summary>
/// Description of Service1.
/// </summary>
{
    public Service()
    {
    }

    [WebMethod]
    public int InsertBox(string name)
    {
        if (!Directory.Exists("c:\baseline\" + name))
        {
            Directory.CreateDirectory("c:\baseline\" + name);
            SqlConnection con = new SqlConnection("Data Source=.\SQLEXPRESS;Initial Catalog=Baseline;Integrated Security=SSPI;"); 
            SqlCommand cmd = new SqlCommand("INSERT INTO Box (Path) VALUES (@Path)", con);
            cmd.Parameters.Add(new SqlParameter("@Path", "c:\baseline\" + name));
            SqlCommand cmd1 = new SqlCommand("UPDATE Box SET Box.Hybrid='" + cajaPath + '\' + artifactName + '\" WHERE Box.Path='" + cajaPath + '\", con);
            con.Open(); cmd.ExecuteNonQuery(); cmd1.ExecuteNonQuery(); con.Close();
            return 0;
        }
        else return -1;
    }

    [WebMethod]
    public void InsertBamc(string cajaPath, string artifactName, byte[] content)
    {
        FileStream str = new FileStream(cajaPath + "\" + artifactName, FileMode.CreateNew);
        BinaryWriter wri = new BinaryWriter(str);
        wri.Write(content); wri.Flush(); wri.Close();
        SqlConnection con = new SqlConnection("Data Source=.\SQLEXPRESS;Initial Catalog=Baseline;Integrated Security=SSPI;"); 
        SqlCommand cmd = new SqlCommand("UPDATE Box SET Box.Bamc='" + cajaPath + '\' + artifactName + '\" WHERE Box.Path='" + cajaPath + '\", con);
        con.Open(); cmd.ExecuteNonQuery(); con.Close();
    }
protoBOM offers a friendly and easy interface for the creation of the baseline (Figure 3-7). In the interface we can see two buttons: the first one for the addition of a kit box to the baseline; and the second one for the addition of the production plan. Furthermore, we can see in the middle a picture representing the baseline. Doing double click on the central picture, the content of the baseline will be shown (see Figure 3-8).

![Create Baseline](image)

*Figure 4-7. Create Baseline*

The table situated on the left side shows the Kit Boxes available in the Baseline; when a Kit Box is selected, all its artifacts are shown in the table situated on the right side.

![Baseline content](image)

*Figure 4-8. Baseline content*

Furthermore, the button “Delete” allows to delete a Kit Box from the Baseline.
4.2.1.1.1 Add Kit Box

For the creation of the Baseline, the domain engineer will pack all the assets (previously created). All the assets will be deposited (one to one) in the Baseline, that is to say, all the Kit Boxes (with their assets) will conform the Baseline. When the button “Add Box” is clicked the next GUI is shown:

![Add Kit Box GUI](image)

**Figure 4-9. Add Kit box**

A new Kit Box will be created and added to the Baseline introducing its name and doing click in the button “Add”. The new Kit Box is an empty Kit Box. This operation is carried out by the web service. protoBOM requests a service of the web service, invoking one of its methods (in this case “InsertBox”), and the web service will carry out the task. If an error occurs during the process, an error message will be shown; if the task can be finalized successfully, the user will be informed too.
Next the source code in protoBOM is shown:

```csharp
box = textBoxNameBox.Text;

BaselineServer.Service ser = new BOM.BaselineServer.Service();
int result = ser.InsertBox(box);

if (result == 0)
    System.Windows.Forms.MessageBox.Show(this, "The box was successfully inserted.", "Information",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
else
    System.Windows.Forms.MessageBox.Show(this, "The box already exists. Introduce a different name.", "Warning",
        MessageBoxButtons.OK, MessageBoxIcon.Warning);
```

![Figure 4-10. Error message.](image1)

![Figure 4-11. Information message.](image2)

At the same time that the Kit Boxes are added to the baseline a file system representing the baseline is created in the computer where protoBOM is running. The baseline is created in the following path: “C:\baseline”.

43
Once the new Kit Box has been created, the box’s content has to be added. Doing click in the button “Add Artifacts” the following GUI it is shown:
The assets contained in each box are:
- Base Architectural Model Configuration
- Feature Insertion Main Process
- RAS Model of Assets
- Architectural Domain Model Configuration
- Package Hybrid

The first four assets will be added to the box introducing a name, and the path where the content of the asset is stored. protoBOM facilitate this task showing a File Dialog for the selection of the file (Figure 3-14).

![File Dialog](image)

**Figure 4-14. File dialog.**

These tasks will be developed in the same way that the insertion of the box. BOM will request one service for each asset to the web service.

The heads of the web methods for each asset are the following:

```
[WebMethod]
public void InsertBamc(string boxPath, string artifactName, byte[] content)

[WebMethod]
public void InsertFimp(string boxPath, string artifactName, byte[] content)

[WebMethod]
public void InsertRasma(string boxPath, string artifactName, byte[] content)
```
[WebMethod]
public void InsertAdcm (string boxPath, string artifactName, byte[] content)

protoBOM will invoke the adequate web method for the asset that it is storing. When the user does click in the button “Add” to add the asset the following method is invoked by protoBOM:

/************* BUTTON “ADD ARTIFACT” *************/
private void buttonAdd_Click(object sender, EventArgs e) {

    string caja = "c:\baseline\" + labelBox.Text;
    FileStream str = new FileStream(textBoxPath.Text, FileMode.Open);
    BinaryReader rea = new BinaryReader(str);

    BaselineServer.Service ser = new BOM.BaselineServer.Service();

    switch (numArtifact) {
        case 1:
            break;

        case 2:
            break;

        case 3:
            break;

        case 4:
            break;

        default:
            break;
    }

    rea.Close();

    System.Windows.Forms.MessageBox.Show(this, "The artifact was successfully inserted.", "Information", MessageBoxButtons.OK, MessageBoxIcon.Information);
}
There is an artifact that has a different treatment than the others, the hybrid. This artifact is not deposited directly in the Baseline; the hybrid has to be packaged before to be added.

The hybrid is composed by:
- An Aspect-Process Skeleton
- Interface
- Architectural Element

First of all, the user will introduce a name for the hybrid, and then he will add its elements.

![Figure 4-15. Add Hybrid](image)

This task, the same as the previous ones, will be carried out by the web service.

In the GUI there are three buttons for the addition of: Aspect-Process Skeleton, Interface and Architectural element. Doing click in the first one (corresponding to the Aspect-Process Skeleton) a new interface it is shown (see figure 4-16).
The user will give a name for the Aspect-Process Skeleton, and it will be added to the kit box (with the button "Add"). Later he will give a name for the Skeleton, and it will select the path for loading the content to the skeleton. Next he will repeat the process commented for the asset "Feature Insertion Process". The insertion process fills the skeleton with the features of the application domain; it will be executed in the application engineering. The insertion process has been implemented on purpose for this thesis.

Once the AP-Skeleton has been added, the user will add the Interface and Architectural element to the hybrid doing click in the corresponding button.
The user has to indicate the name and the content’s path. The same process will be repeated for the architectural element.

4.2.1.1.2 Add Production Plan

When the button “Add Prod Plan” is clicked the next GUI is shown:

![Add Production Plan GUI](image)

Figure 4-18. Add Production Plan

The user will give a name for the production plan, and he will select the file where the contents are. Doing click in the button “Add”, protoBOM will request a service to the Web service and the production plan will be added to the Baseline.
4.2.1.2 Create Feature Model

protoBOM offers an interface for the creation of the Feature Model. The GUI corresponding to this activity is shown in the figure 4-19:

![Figure 4-19. Feature Model GUI](image)

With this tool the different variability points of the domain and its possible values are introduced forming the feature model.

The user will introduce the point’s name and the possible values to the point. Finally, the user has to press the button “Add” to add the point in the Feature Model. In the right part, the Feature Model is shown.

Besides, protoBOM offers three options: Initialize the feature model, load points stored in a file, and save points from the feature model into a file.

![Figure 4-20. Feature Model options](image)

To facilitate the task, it has been implemented the class “VariabilityPoint”, the source code is shown:
using System;
using System.Collections;

namespace BOM
{
    public class VariabilityPoint
    {
        private string nombre;
        private ArrayList possibleValues;
        private string actualValue;

        public VariabilityPoint()
        {
            this.nombre="";
            this.possibleValues=new ArrayList();
            this.actualValue="";
        }

        public VariabilityPoint(string nombre)
        {
            this.nombre=nombre;
            this.possibleValues=new ArrayList();
            this.actualValue="";
        }

        public void setName(string nombre)
        {
            this.nombre=nombre;
        }

        public void setPossiblevalues(ArrayList pv)
        {
            this.possibleValues=pv;
        }

        public void setAcual(string av)
        {
            this.actualValue=av;
        }

        public string getName()
        {
            return this.nombre;
        }

        public ArrayList getPossibleV()
        {
            return this.possibleValues;
        }

        public string getActualV()
        {
            return this.actualValue;
        }

        public void addPossibleValue(string v)
        {
            this.possibleValues.Add(v);
        }
    }
}
4.2.1.3 Create Decision Tree

protoBOM offers an interface for the creation of the Decision Tree. The Decision Tree’s leaves point to the group of selected assets (the selected kit box).

The GUI corresponding to this activity is shown in the figure 4-21.

![Create Decision Tree](image)

*Figure 4-21. Create Decision Tree*

The points forming the feature model are automatically loaded. When a point is selected its possible values are loaded too. Doing click in the button “Add” (“+”) the variant of the variability point will be added to the decision tree to form the path that will point to a kit box.

![Create path](image)

*Figure 4-22. Create path*
The available kit boxes in the baseline are automatically loaded. The path will point to a kit box; the kit box will be selected and added to the decision tree by means of this option.

![Add Kit Box]

*Figure 4-23. Add Kit box*

Once the path is created, there are two possibilities: Save the path in a new file or add the path in an existing file (doing click in the buttons “Save tree” or “Save path”). The application will show a FileDialog to select the file where the Decision Tree will be saved.

- **Save Tree**
  - This button will save the path in a new file.

- **Save path**
  - This button will add the path in a file containing more paths to conform the decision tree.

In the right side, the paths created will be shown (Figure 4-24).

![Path]

*Figure 4-24. Path*
protoBOM offers an additional option: load points created in a previous feature model.

Figure 4-25. Options

4.2.2 protoBOM in the Application Engineering

protoBOM executes the Production Plan in the application engineering phase. The baseline and the assets created in the Domain Engineering phase are used in this phase.

The Application Engineer inputs to the system data of specific domains. This data are the variants of the variability points (first variability) and the application domain features (second variability).

The Production Plan describes the process by means of tasks or activities to obtain a final product of the SPL.

The following activities are carried out by the Application Engineer for the creation of a product of the SPL:

- Obtain Domain Features
- Obtain Application Domain Features
- Create PRISMA types
- Create executable

Figure 4-26. Activities offered in the Application Engineering.
The production plan process is shown in an interface using the SPEM notation (Figure 4-27). The Production Plan is composed by 8 activities:

1. Obtain Domain Features
2. Select Assets
3. Unpack Kit box
4. Obtain Application Domain Features
5. Create PRISMA types
6. Compile Architectural Model
7. Create executable

The Application engineer only takes part in the activities: 1, 4 and 5. The rest of activities will be carried out automatically by protoBOM.

![Production Plan of SPL](image)

**Figure 4-27. Execute Production Plan.**

There are some tasks that have to be done before than others, for that protoBOM will show error messages to guide the order between tasks. For example, “Obtain Domain Features” has to be done before that “Create PRISMA types”; and create “PRISMA types” has to be done before to “Create executable”. Some examples of these errors are shown:
4.2.2.1 Obtain Domain Feature

The Production Plan starts when the application engineer obtains the domain's characteristics represented as variability points. This task is carried out by protoBOM by means of the GUI showed in the figure 4-29.

The decision tree created in the Domain Engineering phase is loaded, and the user will select the variability points until a Kit box arrives. The box will be selected by clicking on the button “Select Box”. The Kit box selected, it is automatically unpackaged by BOM, to be able to use each one of the assets of their content.
4.2.2.2 Obtain Application Domain Features

protoBOM offers an interface to introduce the characteristics of the application domain considered as "features" of the second variability. The application engineer introduces the application domain information of the case study.

This task is divided into three parts:
- Add properties
- Add Hypotheses
- Add Derivation rules

Doing click in the buttons corresponding to each part, the corresponding interface will be shown.

The interface offers three additional options:
- Initialize the specification empty.
- Save the specification in a file.
- Load a specification stored in a file.

For the implementation of this part a new class has been create, the class: “Property”. The Properties and Hypotheses of the case study will be represented as objects from the Property class.

Next the source code of the Property class is shown:
public class Propertie {

    private int level;
    private string name;
    private string type;
    private string actualValue;
    private ArrayList possibleValues;

    public Propertie() {
        this.level = -1;
        this.name = "";
        this.type = "";
        this.actualValue = "";
        this.possibleValues = new ArrayList();
    }

    public Propertie(int level, string name, string type) {
        this.level = level;
        this.name = name;
        this.type = type;
        this.actualValue = "";
        this.possibleValues = new ArrayList();
    }

    public int getLevel() { return this.level; }
    public void setLevel(int level) { this.level = level; }

    public string getName() { return this.name; }
    public void setName(string name) { this.name = name; }

    public string getType() { return this.type; }
    public void setType(string type) { this.type = type; }

    public string getActualValue() { return this.actualValue; }
    public void setActualValue(string value) { this.actualValue = value; }

    public ArrayList getPossibleValues() { return this.possibleValues; }
    public void addPossibleValue(Object pValue) { this.possibleValues.Add(pValue); }

    public string getPossibleValuesString() {
        string s = "";
        for (int i = 0; i < this.possibleValues.Count; i++)
            s += this.possibleValues[i].ToString() + " ";
        return s;
    }

    public void setPossibleValues(ArrayList values) { this.possibleValues = values; }
    public void setPossibleValues(string[] values) {
        ArrayList pVal = new ArrayList();
        foreach (string v in values)
            pVal.Add(v);
        setPossibleValues(pVal);
    }
}
For storing the Application Domain Features, it has been implemented another class called “Specification”.

This class has three lists, implemented as ArrayLists, for storing features, hypotheses and derivation rules. In each position of the list is stored a list with the properties from the level corresponding with the position in the list. For example, in the Features list, in the position [0] it is stored a list with all the features of level 0.

Next a visual metaphor of the Specification class is shown.

![Figure 4-31. Specification class.](image)

Part of the source code of Specification class is shown in Annex C. The treatment and methods for the three lists are similar, for that only will be shown the methods corresponding to the list Features.

- **Add properties**

The introduction of features will be carried out by protoBOM very easily by means of the following interface (see Figure 4-32). For each property, the user will indicate the level, name, type and possible values. The new property will be add doing click in the button add (represented with “+”). In the right side, there is a list with all the properties added in a level. When a level is selected, all the features stored in that level will be automatically loaded in the list. protoBOM offers the possibility to delete (button represented with “-”) or modify (button represented with a pencil) a property.
• **Add hypotheses**

The interface corresponding to “Add hypotheses” is identical to the previous interface, for that the operation will be the same that was commented in the previous point.
• **Add derivation rules**

protoBOM offers an interface to create derivation rules. First of all, the level has to be selected. protoBOM automatically loads the properties and hypotheses corresponding to that level, to facilitate the task to the user. The user will select different values to form derivation rules of different level. In the middle of the interface, there is a list where the rules created are shown. BOM offers the possibility to delete a rule selected. When a level is selected, the rules created will be load in the list.

![Add Derivation Rules](image)

**Figure 4-34. Add derivation rules.**

### 4.2.2.3 Create PRISMA types

The Kit box selected by the engineer in the first task of this phase is unpackaged in order to use each asset independent. In the top of the interface the user is informed about the unpacked box (see Figure 3-35). Also all the assets contained in the Kit box are shown in a list.

One of the assets recovered from the Box is the Base Architectural Model Configuration (BAMC), which is used by BOM in order to obtain (from the application engineer) the features of the application domain considered as variants of the second variability.
Other assets recovered from the Kit box are the Packaged Hybrids. These assets are in turn unpacked (aspect skeleton, feature insertion process, interface and architectural element) to produce the PRISMA types in two steps:

- In the first step, protoBOM invokes the features insertion processes to fill the selected skeletons (aspects) with the specific features of the case study defined by the engineer, thereby creating the PRISMA software artifact types. The software artifacts are represented as XML documents, therefore the process consists in fill XML documents with the features of the specific application domain. The insertion process has been implemented on purpose for this thesis (the insertion process is detailed in Annex A).

- In the second step, protoBOM recovers the PRISMA type (interfaces and architectural elements) stored in the package hybrid.

Next the interface offered by protoBOM in this task is shown (Figure 3-35).

![Create PRISMA Types](image)

**Figure 4-35. Create PRISMA types.**
protoBOM requests to the user the folder where the PRISMA types will be stored. Once the path has been selected, the user will invoke the Feature Insertion Process doing click in the button “Create Prisma types”.

On the other hand the BAMC has to be selected by the user.

In Annex B, the PRISMA type created by BOM is shown.

4.2.2.4 Create executable system

BOM shows an GUI (see Figure 4-33) with information about the PRISMA types available, and the Base Architectural Model Configuration.

In this step finishes the work of protoBOM. protoBOM has created a PRISMA type with the application domain features through the insertion process, this type is joined to the other PRISMA types recovered from the selected Kit Box.

In this point, all the PRISMA types, represented as XML documents, are stored in a concrete folder. Now all is ready to create the executable application.

The PRISMA types are introduced to the PRIMA-Model Compiler [Cabedo et al., 2005] and the PRISMANET Middleware [Costa et al., 2005] will be used to automatically generate an executable application.

![Create executable](image)

*Figure 4-36. Create executable*
5.- CONCLUSIONS

In this thesis has been presented the development of a prototype of the Baseline Oriented Modeling (BOM) approach. The BOM prototype is called protoBOM.

protoBOM is a framework that semi-automatically generates Decision Support Systems, based on Software Product Lines.

protoBOM offers numerous advantages in the development of Decision Support Systems. protoBOM has been designed to improve the development of diagnostic systems in following ways:

- Various technological spaces are integrated to cope with the complexity of the problem. They are current trends in Software Engineering.

- It integrates the use of several tools, some of these tools have been develop on purpose for this thesis and others were created before in other environments.

- To apply techniques from the field Software Product Lines by building a design that shares all the members of a program family. In this way, a specific design can be used in different products. Since a specific product is obtained from a series of previous models, and the costs, time, effort, and complexity can be reduced.

- To construct Product Line Architectures in the PRISMA framework, by reusing software components and integrating components and aspects, in order to facilitate the management of complexity.

- To apply MDA techniques in order to automatically generated code and to obtain an executable DSS.

- To construct Decision Support Systems in a simpler way by using the ontologies of the diagnosis and the application domains. The interfaces will be closer to the problem domain, which will facilitate user interaction,

- To development an executable application in a simple way through protoBOM, by means of simple and intuitive interfaces.

- To use ProtoBOM in order to create a Decision Support System without knowledge in programming languages or Computer Science. An executable application is obtained only doing click in some buttons, and providing some values to the tool.
protoBOM implements a generic approach to SPL development that can be applied to different domains, application domains and systems. As future work, protoBOM will be applied in different case studies and validated in real life cases. Additionally, it will uses benchmarks in order to compare protoBOM with other approaches.
REFERENCES


ANNEX A
The feature insertion process implementation

The software artifacts are represented as XML documents, therefore the insertion process consists in fill XML documents with the features of the specific application domain.

protoBOM recovers skeletons (XML documents) from the package hybrid and it fills them with the Application Domain Features, obtaining a PRISMA type. A visual metaphor of the insertion process is shown:

![Figure A-1. Insertion process metaphor.](image)

The insertion process has been implemented on purpose for this thesis. Next, part of the source code is shown:

```csharp
private void buttonXML_Click(object sender, System.EventArgs e) {

    XmlDocument doc = new XmlDocument();
    XMLElement skeletonElement = (XMLElement)doc.AppendChild(doc.CreateElement("SKELETON"));
    XMLElement aspectElement = (XMLElement)skeletonElement.AppendChild(doc.CreateElement("ASPECT"));
    XMLElement typeElement = (XMLElement)aspectElement.AppendChild(doc.CreateElement("TYPE"));
    XmlText typeTextElement = (XmlText)typeElement.AppendChild(doc.CreateTextNode("TYPE"));
    typeTextElement.Value = "Functional Aspect";
```
XmlElement nameElement = (XmlElement)aspectElement.AppendChild(doc.CreateElement("NAME"));
XmlText nameTextElement = (XmlText)nameElement.AppendChild(doc.CreateTextNode("NAME"));
nameTextElement.Value = "FBaseDPE";

XmlElement interfaceElement = (XmlElement)aspectElement.AppendChild(doc.CreateElement("INTERFACE"));
XmlText interfaceTextElement = (XmlText)interfaceElement.AppendChild(doc.CreateTextNode("INTERFACE"));
interfaceTextElement.Value = "using IDomainDPEDT";

XmlElement attributesElement = (XmlElement)skeletonElement.AppendChild(doc.CreateElement("ATTRIBUTES"));
XmlElement variablesElement = (XmlElement)attributesElement.AppendChild(doc.CreateElement("VARIABLES"));

//Add properties level 0
ArrayList featuresLevel0 = this.specification.getPropertiesLevel(0);
for (int i = 0; i < featuresLevel0.Count; i++) {
    Propertie p = (Propertie)featuresLevel0[i];
    XmlElement featuresFP0Element = (XmlElement)variablesElement.AppendChild(doc.CreateElement("FP0"));
    XmlText featuresFP0TextElement = (XmlText)featuresFP0Element.AppendChild(doc.CreateTextNode("FP0"));
    featuresFP0TextElement.Value = p.getName() + ":" + p.getType();
}

XmlElement derivedsElement = (XmlElement)attributesElement.AppendChild(doc.CreateElement("DERIVEDS"));

//Add properties level 1
ArrayList featuresLevel1 = this.specification.getPropertiesLevel(1);
for (int i = 0; i < featuresLevel1.Count; i++) {
    Propertie p = (Propertie)featuresLevel1[i];
    XmlElement featuresFP1Element = (XmlElement)derivedsElement.AppendChild(doc.CreateElement("FP1"));
    XmlText featuresFP1TextElement = (XmlText)featuresFP1Element.AppendChild(doc.CreateTextNode("FP1"));
    featuresFP1TextElement.Value = p.getName() + ":" + p.getType();
}

//Add hypotheses
XmlElement hypothesisElement = (XmlElement)derivedsElement.AppendChild(doc.CreateElement("FH"));
XmlText hypothesisTextElement = (XmlText)hypothesisElement.AppendChild(doc.CreateTextNode("FH"));

ArrayList hypotheses = this.specification.getHypothesesLevel(1);
ArrayList hypotheses = this.specification.getHypothesesLevel(1);
hypothesisTextElement.Value = hypotheses[0].ToString();
///Add DERIVATIONS
XmlElement derivationsElement = (XmlElement)skeletonElement.AppendChild(doc.CreateElement("DERIVATIONS"));
XmlText derivationsTextElement = (XmlText)derivationsElement.AppendChild(doc.CreateTextNode("DERIVATIONS"));
derivationsTextElement.Value = "Derivations";

///Add Rules Level1
ArrayList rulesL1 = this.specification.getRulesLevel(1);
for (int i = 0; i < rulesL1.Count; i++) {
XmlElement rulesL1Element = (XmlElement)derivationsElement.AppendChild(doc.CreateElement("FR1"));
XmlText rulesL1TextElement = (XmlText)rulesL1Element.AppendChild(doc.CreateTextNode("FR1"));
rulesL1TextElement.Value = rulesL1[i].ToString();
}

///Add Rules Level2
ArrayList rulesL2 = this.specification.getRulesLevel(2);
for (int i = 0; i < rulesL2.Count; i++) {
XmlElement rulesL2Element = (XmlElement)derivationsElement.AppendChild(doc.CreateElement("FR2"));
XmlText rulesL2TextElement = (XmlText)rulesL2Element.AppendChild(doc.CreateTextNode("FR2"));
rulesL2TextElement.Value = rulesL2[i].ToString();
}

///Add SERVICES
XmlElement servicesElement = (XmlElement)skeletonElement.AppendChild(doc.CreateElement("SERVICES"));
XmlText servicesTextElement = (XmlText)servicesElement.AppendChild(doc.CreateTextNode("SERVICES"));
servicesTextElement.Value = "Services";

///Add etiqueta BEGIN
XmlElement beginElement = (XmlElement)servicesElement.AppendChild(doc.CreateElement("BEGIN"));
XmlText beginTextElement = (XmlText)beginElement.AppendChild(doc.CreateTextNode("BEGIN"));
beginTextElement.Value = "Begin";

..................

///Add NILFP0
ArrayList featuresL0 = this.specification.getPropertiesLevel(0);
for (int i = 0; i < featuresL0.Count; i++) {
XmlElement nilFP0Element = (XmlElement)incleanElement.AppendChild(doc.CreateElement("NILFP0"));
XmlText nilFP0TextElement = (XmlText)nilFP0Element.AppendChild(doc.CreateTextNode("NILFP0"));
nilFP0TextElement.Value = featuresL0[i] + " := nil, ";
}

..................
//Añado etiqueta ASIGNAFP0
ArrayList propertiesL0 = this.specification.getPropertiesLevel(0);
for (int i = 0; i < propertiesL0.Count; i++){
(Property) p = propertiesL0[i];
XmlElement asignaFP0Element =
(XmlElement) inferP1Element.AppendChild(doc.CreateElement("ASIGNAFP0"));
XmlNode asignaFP0TextElement =
(XmlNode) asignaFP0Element.AppendChild(doc.CreateTextNode("ASIGNAFP0"));
asignaFP0TextElement.Value = p.getName() + "::=PROPERTI0_" + i.ToString() + ";";
}

//Add ROLES
XmlElement rolesElement =
(XmlElement) playedElement.AppendChild(doc.CreateElement("ROLES"));
XmlNode rolesTextElement =
(XmlNode) rolesElement.AppendChild(doc.CreateTextNode("ROLES"));
rolesTextElement.Value = "KNOWLEDGE for IDomainDPEDT::=limpiarBD

//Add TYPEEND
XmlElement typeEndElement =
(XmlElement) aspectEndElement.AppendChild(doc.CreateElement("TYPEEND"));
XmlNode typeEndTextElement =
(XmlNode) typeEndElement.AppendChild(doc.CreateTextNodes("TYPEEND"));
typeEndTextElement.Value = "End_Functional Aspect";

doc.Save("FBaseDPE.xml");
System.Windows.Forms.MessageBox.Show(this,"El XML se ha generado correctamente.","Information",MessageBoxButtons.OK,MessageBoxIcon.Information);
Example of an Aspect PRISMA type in XML document generated by protoBOM

<?xml version="1.0" encoding="UTF-8"?>
<SKELETON>
  <ASPECT>
    <TYPE>Functional Aspect</TYPE>
    <NAME>FBaseDPE</NAME>
    <INTERFACE>using IDomainDPEDT</INTERFACE>
  </ASPECT>
  <ATTRIBUTES>
    <VARIABLES>
      <FP0>library: String</FP0>
      <FP0>computerEquipment: String</FP0>
      <FP0>laboratories: String</FP0>
      <FP0>facilities: String</FP0>
    </VARIABLES>
    <DERIVEDS>
      <FP1>infrastructure: String</FP1>
      <FP1>studyPlan: String</FP1>
      <FH>developmentalStage: String</FH>
    </DERIVEDS>
  </ATTRIBUTES>
  <DERIVATIONS>
<FR1>(library='good' and computerEquipment='good') infrastructure='good'</FR1>
<FR1>(laboratories='good' and facilities='good') studyPlan='good'</FR1>
<FR1>(library='bad' and computerEquipment='bad') infrastructure='bad'</FR1>
<FR1>(laboratories='bad' and facilities='bad') studyPlan='bad'</FR1>
<FR2>(infrastructure='good' and studyPlan='good') developmentalStage='good'</FR2>
<FR2>(infrastructure='bad' and studyPlan='bad') developmentalStage='bad'</FR2>
</DERIVATIONS>

<SERVICES>
<BEGIN>Begin()</BEGIN>
<CLEAN>in cleanDB();</CLEAN>

<VALUATIONCLEAN>Valuations</VALUATIONCLEAN>

<INCLEAN>[in cleanDB ()]
<NILFP0>library:=nil</NILFP0> <NILFP0>computerEquipment:=nil</NILFP0> <NILFP0>laboratories:=nil</NILFP0> ... <NILFP1>infrastructure:=nil</NILFP1> <NILFP1>studyPlan:=nil</NILFP1> <NILFH>developmentalStage:=nil</NILFH></INCLEAN>

<INFERPROP1>in inferPropertiesN1 (input PROPERTIESN0:string, output PROPERTIESN1: string)</INFERPROP1>

<VALUATIONINFERPROP1>Valuations</VALUATIONINFERPROP1>

<ININFERPROP1>[in inferPropertiesN1 ()]
<ASIGNAFP0>library:=PROPERTY0_0</ASIGNAFP0> <ASIGNAFP0>computerEquipment:=PROPERTY0_1</ASIGNAFP0> <ASIGNAFP0>laboratories:=PROPERTY0_2</ASIGNAFP0>
<ASIGNAFP0>facilities:=PROPERTY0_3</ASIGNAFP0>  
<ASIGNAFP1>PROPERTY1_0:=infrastructure</ASIGNAFP1>  
<ASIGNAFP1>PROPERTY1_1:=studyPlan</ASIGNAFP1>  
</ININFERPROP1>  
<INFERHYP>in inferHypotheses (input PROPERTIESN1: string, output HYPOTHESES: string)</INFERHYP>  
<VALUATIONINFERHYP>Valuations</VALUATIONINFERHYP>  
<ININFERHYP>[in inferHypotheses () ] <ASIGNAFP1>infrastructure:=PROPERTY1_0</ASIGNAFP1>  
<ASIGNAFP1>studyPlan:=PROPERTY1_1</ASIGNAFP1>  
<ASIGNAFH>HYPOTHESES:=developmentalStage</ASIGNAFH>  
</ININFERHYP>  
<END>end;</END>  
</SERVICES>  
<PLAYEDROLES>Played_Roles  
<ROLE>KNOWLEDGE for IDomainEP-TV::=limpiarBD?()</ROLE>  
<ROLE>inferProperties1?(PROPERTY0_0,PROPERTY0_1,PROPERTY0_2,PROPERTY0_3,PROPERTY1_0,PROPERTY1_1)</ROLE>  
<ROLE>inferProperties1!(PROPERTY0_0,PROPERTY0_1,PROPERTY0_2,PROPERTY0_3,PROPERTY1_0,PROPERTY1_1)</ROLE>  
<ROLE>inferHypotheses?(PROPERTY1_0,PROPERTY1_1HYPOTHESES)</ROLE>  
<ROLE>inferHypotheses!(PROPERTY1_0,PROPERTY1_1HYPOTHESES)</ROLE>  
</PLAYEDROLES>  
<PROTOCOLS>  
<FBASE>FASE::=begin():1-&gt;P0</FBASE>  
<P0>P0::=KNOWLEDGE_cleanDB?():1-&gt;P1</P0>  
<P1>PROPERTY0_3,PROPERTY1_0,PROPERTY1_1-&gt;KNOWLEDGE_inferProperties1!(PROPERTY0_3,PROPERTY1_0,PROPERTY1_1):1-&gt;P2</P1>
<P2>PROPERTY1_0,PROPERTY1_1HYPOTHESES)\rightarrow KNOWLEDGE_inferHypotheses!(PROPERTY1_0,PROPERTY1_1HYPOTHESES):1-
&\rightarrow P3</P2>

\<P3\>P3::=end():1;</P3>
</PROTOCOLS>

<NAMEEND>

<ASPECTEND>

\<TYPEEND\>End_Functional Aspect</TYPEEND>

<NAME>FBaseDPE</NAME>

</ASPECTEND>

</NAMEEND>

</SKELETON>
ANNEX C
Source code class specification (Methods corresponding to the list Features)

```java
public class Specification {
    private ArrayList Features;
    private ArrayList Hypotheses;
    private ArrayList rules;

    public Specification() {
        this.Features=new ArrayList();
        this.Hypotheses=new ArrayList();
        this.rules=new ArrayList();
        for(int i=0;i<5;i++) {
            this.Features.Insert(i, new ArrayList());
            this.Hypotheses.Insert(i, new ArrayList());
            this.rules.Insert(i, new ArrayList());
        }
    }

    public void addFeat(Propertie f) {
        int level=f.getLevel();
        ArrayList PropertiesLevel=this.getPropertiesLevel(level);
        PropertiesLevel.Add(f);
    }

    public ArrayList getPropertiesLevel(int level) {
        ArrayList featLevel=(ArrayList)this.Features[level];
        return featLevel;
    }

    public Propertie getPropertieByName(string name, int level) {
        Propertie f_aux=new Propertie();
        ArrayList PropertiesLevel=(ArrayList)this.Features[level];
        for(int i=0;i<PropertiesLevel.Count;i++) {
            Propertie f=(Propertie)PropertiesLevel[i];
            if(f.getName().Equals(name))
            {
                f_aux=f;
                break;
            }
        }
        return f_aux;
    }

    public int numLevelsProps() {
        int i = 0,count=0;
        ArrayList aux;
        while (i > -1) {
            aux = (ArrayList)Features[i];
            if (aux.Count > 0)
            {
                count++;
            }
        }
    }
```
    i++;
    }
    else   i = -1;
}
return count;  }

public void deletePropertie(int level, Propertie p)
{
    ArrayList featuresLevel = getPropertiesLevel(level);
    featuresLevel.Remove(p);
}

private void escribePropiedadesLevel(int level, StreamWriter wri) {
    ArrayList props = getPropertiesLevel(level);
    for (int i = 0; i < props.Count; i++)     {
        wri.WriteLine("<P" + level + ">");
        Propertie prop = (Propertie)props[i];
        wri.WriteLine(prop.getName());
        wri.WriteLine(prop.getType());
        wri.WriteLine(prop.getPossibleValuesString());
        wri.WriteLine();
    }
    wri.Flush();
}

//Save the specification in a file
public void saveSpecification(string path) {
    FileStream str = new FileStream(path, FileMode.Create);
    StreamWriter wri = new StreamWriter(str);
    for (int i = 0; i < numLevelsProps(); i++)
        escribePropiedadesLevel(i, wri);
    for (int j = 1; j <= numLevelsHyp(); j++)
        escribeHypothesesLevel(j, wri);
    for (int k = 1; k <= numLevelsRules(); k++)
        escribeRulesLevel(k, wri);
    wri.WriteLine("PATH:");
    wri.WriteLine(this.getPathPrismas());
    wri.Close();
}